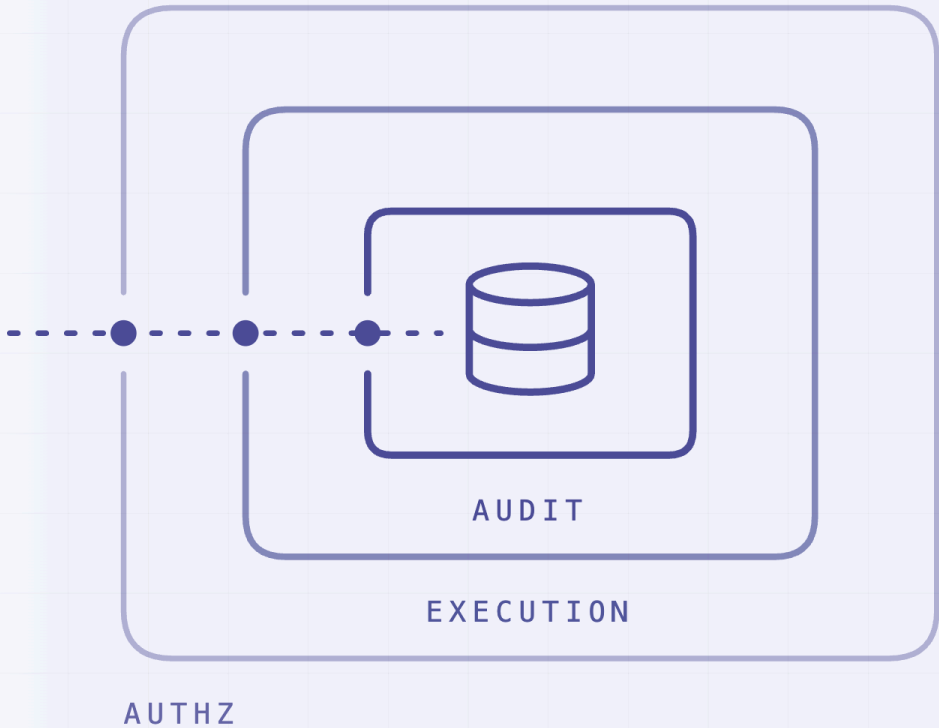




The Production Database Access Guide

The Production Database Access Guide

Identities, credentials, just-in-time access, mediated execution and audit

Muhammet Şafak

Contents

Preface	6
PART I — FOUNDATIONS	13
Chapter 1 — The Door Nobody Designed	14
Chapter 2 — A Threat Model for Production Data	28
Chapter 3 — Identities, Roles and Least Privilege	45
PART II — APPLICATION ACCESS	67
Chapter 4 — Credentials and Secrets	68
Chapter 5 — Connections as Policy	86
Chapter 6 — Reads, Replicas and Data Copies	103
PART III — HUMAN ACCESS	116
Chapter 7 — Paths In: Bastions, Proxies and Access Gateways	117
Chapter 8 — Just-in-Time and Break-Glass Access	138
Chapter 9 — Mediated Execution: Query Analysis and Dangerous Operations	162
Chapter 10 — Ad-hoc Queries and Data Fixes	177
Chapter 11 — Schema Changes and Migration Access	201
PART IV — PROTECTING THE DATA	221
Chapter 12 — Sensitive Data: Classification, Masking and Minimization	222
Chapter 13 — Backups, Exports and Other Side Doors	243
PART V — PROVING AND OPERATING	264
Chapter 14 — Auditing and Accountability	265
Chapter 15 — Access During Incidents	284
Chapter 16 — An Access Checklist and Closing Thoughts	301

Preface

Imagine a Thursday evening at Ledgerly, an invoicing and payments company that does not exist. Robin is on call. A customer's invoice has been stuck in a "sending" state since the afternoon, and the fix looks small: one row, one status column. The team's desktop database client remembers the shared admin password, connects to the primary, and commits automatically, as these clients often do. A one-row fix goes through that client, under a role that can change every table, and its `WHERE` clause matches a whole batch of invoices instead of one.

Robin spends the rest of the evening on the repair. What takes longer is the question that follows it: who ran that statement? The database log shows the shared admin role, which every service and, say, twenty people use. The password manager shows who opened the entry, not who ran what. The team can reconstruct the change from the application's side effects, and eventually it does. But for a day or so, nobody can say with confidence what exactly happened, to which rows, and whether anything else used the same door at the same time.

Nothing in this story was unreasonable on its own. The password was shared because sharing it was the quickest way to get the first service running. The client was auto-committing because that is its default. The role could change every table because, early on, it was easier than deciding which tables each service needed. Each step was reasonable on its own day. Together they built a door that nobody designed.

That door is what this book is about.

Why this book exists

Access to a production database tends to be treated as plumbing. It gets set up when the first service is deployed, and then it is touched again only when something breaks or someone leaves. Meanwhile the database be-

comes the most valuable thing the company runs: the customers, the money, the history. The gap between how much a company depends on it and how little design goes into its access is the subject of this book.

A good deal has been written about each piece. Database documentation explains how to create roles and grants. Secrets tools explain how to issue credentials. Security guidance explains why standing access is a risk. What is harder to find is the whole picture from an engineer's side: how the pieces fit together, what each one costs the people who build and operate the system, and in what order a team with limited time might adopt them.

This book tries to be that picture, and it starts from one claim. Production database access is not a credential-distribution problem; it is an authorization, execution, and accountability problem.

The default view treats access as a matter of who holds a password and how the password is stored, rotated and handed out. The Thursday evening above shows where that view stops. Robin held a legitimate credential, and the credential did exactly what it was issued to do. What failed was everything around it: nothing decided that this person should run a write against that table, nothing inspected the statement or bounded its effect before it committed, and nothing tied the result to a person afterward. Authorization decides what may happen, and where. Execution is how each statement reaches the data. Accountability is the evidence that remains. Each chapter names the pillar it serves, and Part I builds the vocabulary for all three. The book treats every control as a decision with a price. A control that is right for the table of payment tokens may be pointless for a table of invoice templates, and the book will say so when it matters.

I should be plain about the stance. The book is written from the defender's side. Threats are described only as far as they explain why a control exists; nothing here is a guide to getting in. It does not sell fear and it does not quote breach statistics, because a calm design review is more useful than an alarming one. And it counts friction honestly. Every control asks something of engineers and of whoever is on call, and a control that adds friction without reducing exposure is security theater. When I think a popular practice is theater, I will say so and explain why.

Who this book is for

The book is written for **backend and platform engineers** who build services that talk to a production database, for **SRE and DBA roles** who operate that database, for **tech leads** who have to decide how access should work, and for **engineers who have newly become responsible for security** in a team that never had a dedicated function. If you have ever wondered whether the shared password is still the shared password, hesitated before granting a colleague write access, or been asked "who can see this table?" and realized the honest answer needs a day of digging, you are the reader I have in mind.

You do not need a security background. You should be comfortable with a few things:

- Relational databases: tables, roles and grants, transactions, and what a replica is.
- Services that connect to a database over a network, and the vocabulary of connection pools and timeouts.
- On-call work: being paged, repairing something under pressure, and writing down afterward what happened.

Familiarity with a particular database engine helps with examples, but is not required, as a later section explains.

What this book is not

It is not a database administration book. It will not teach you to tune a planner, size a buffer or choose a replication topology. Where administration matters for access, as with replicas and migrations, the book covers that part and leaves the rest to your database's own documentation.

It is not a compliance handbook. Regimes such as GDPR, KVKK, PCI DSS, SOC 2 and HIPAA come up because they shape what auditors and customers ask, and the book describes them only in that spirit: regimes of this kind tend to ask who can see what, how access is approved, and what record exists afterward. It does not interpret any regime, cite its articles or tell you what is required of you. Your legal and compliance colleagues own that interpretation, and the book's aim is to give them an architecture they can reason about.

And it is not an attack book. You will not find techniques for breaking in. When a chapter needs a threat, it names the risk in a sentence, such as a credential leaking, a query landing on the wrong database or a vendor session left open, and moves on to the control. If describing the threat more precisely would not change the design, the detail is left out.

How the book is organized

The book has five parts and sixteen chapters, followed by an appendix.

Part I, Foundations (Chapters 1 to 3) argues the thesis above and builds the vocabulary for the rest. Chapter 1 looks at how doors to a database accumulate. Chapter 2 builds a threat model for production data that includes mistakes as well as misuse. Chapter 3 covers identities, roles and least privilege, which every later chapter depends on.

Part II, Application Access (Chapters 4 to 6) is about the services. Chapter 4 covers credentials and secrets, from static passwords to dynamic credentials and workload identity. Chapter 5 treats connections as policy: pooling, limits and timeouts. Chapter 6 covers reads, replicas and the other copies of data that production quietly creates.

Part III, Human Access (Chapters 7 to 11) is about people. Chapter 7 compares bastions, proxies and access gateways. Chapter 8 covers just-in-time and break-glass access. Chapter 9 covers execution itself: how a gateway inspects each statement, bounds its effect and commits it. Chapter 10 covers ad-hoc queries and data fixes. Chapter 11 covers schema changes and who is allowed to make them.

Part IV, Protecting the Data (Chapters 12 and 13) shifts from who can connect to what they can see. Chapter 12 covers classification, masking and minimization. Chapter 13 covers backups, exports and other side doors that bypass all the controls built so far.

Part V, Proving and Operating (Chapters 14 to 16) covers how a team knows its controls work and keeps them working. Chapter 14 covers audit logs and access reviews. Chapter 15 covers access during incidents, when pressure is highest and shortcuts are most tempting. Chapter 16 offers a checklist, a maturity ladder with exit tests and some closing thoughts on culture.

The appendix collects a glossary, a quick reference of the patterns, a matrix of dangerous operations, a compliance crosswalk and templates for access reviews and post-incident reviews.

Every chapter has the same shape. It opens with a short, openly fictional scenario. It then explains the problem and why the obvious approach falls short, presents the control, and states its costs and when not to use it. It lists common mistakes, treated as traps in the design and never as personal failings, and ends with five to seven takeaways and a bridge to the next chapter. The shared shape means you can read the chapters in order or jump to the one that matches this week's problem, and still find the cost section where you expect it.

Meet Ledgerly

Ledgerly is a fictional business-to-business invoicing and payments service. Companies issue invoices through it, collect payments and keep a ledger of what moved. It runs five services: `billing-api`, `invoice-worker`, `payments-service`, `reporting-job` and `admin-console`. Its primary database has two replicas and an analytics copy for the data team. It stores customer contact details, bank account information and payment tokens, which is why its access design matters.

The people around it fill familiar roles. There are on-call backend engineers, a support team, a data team, a small platform group that runs the databases, and an external accounting vendor that receives exports.

At the start of the book Ledgerly has a shared admin password, standing access for most of the backend team, desktop clients on laptops and no record of who did what. By the end it has a role for each service, short-lived credentials, a gateway for humans, a reviewed way to fix data, statements that are checked and bounded before they run, masked copies and a log that cannot be quietly rewritten. It gets there one step at a time, and each chapter says which step it adds and what that step costs the team.

Ledgerly is made up, and so is everything that happens to it. Every chapter opens with a scenario marked as fiction, and any number in it is hypothetical. Names such as Robin, Sam, Jordan and Alex are placeholders for roles on a team. When a scenario goes wrong, the lesson is about the design that let it go wrong, and never about the person in the story.

How to read the code

No example in this book is written in a real programming language. Procedures are written in plain pseudocode, and roles, grants and tables use a SQL-like notation that belongs to no particular engine. Nothing in the book is meant to be pasted into a terminal. You will not find configuration syntax for a specific proxy, vault or cloud product, because that syntax changes and the pattern behind it does not.

Diagrams use Mermaid. Each is preceded by a sentence telling you what to look for, and denied or failing paths are labeled. Examples never contain realistic secrets, account numbers or email addresses; placeholders such as `<secret>` and `IBAN-PLACEHOLDER` stand in for them.

Why English, and why engine-independent

The book is written in English because most of the vocabulary of this field, from least privilege to break-glass, is used in English even in teams that work in other languages, and I wanted the terms to stay recognizable. The book is engine-independent for a related reason. A team might run PostgreSQL, MySQL or something else, and might use a cloud identity service, a secrets manager or a hand-rolled script. The patterns are the same across all of them: separate the owner from the users, give each workload its own identity, make credentials expire, put an identity-aware entry in front of humans, check what runs before it runs, run writes through something that can refuse them, record what happens, review it. Products and their syntax change often. When a real product is named, it is named as an example, and its behavior is described in terms that should hold across versions.

A note on certainty

I have tried to say which of my recommendations are firm and which are preferences. Some are firm because the cost of being wrong is exposure of customer data or loss of the ledger, and I say so when that is the case. Most are judgment calls, and a thoughtful team could weigh them differently depending on its size, its regulators, its customers and how often it is paged. Where I do not know, I will say "I don't know" and name the variables that would settle the question.

If you disagree with a recommendation, I would like you to be able to find the premise you disagree with. That is why each chapter spells out its reasoning and its costs. A design you understand and have adapted to your own constraints is worth more than one you copied from a book.

Chapter 1 starts with the door itself.

PART I

Foundations

Chapter 1 — The Door Nobody Designed

Consider Ledgerly in its early years. A prospective customer sends over a security questionnaire, and one line looks routine: "List every person and system that can connect to your production database." Alex, who looks after the databases alongside other duties, picks it up on a quiet afternoon.

Alex starts with the password manager, which holds one admin entry that the whole backend team knows. Next come the five services, each storing that same password, and the laptops, where desktop database clients remember it. Then the analytics copy, a nightly restore of an unmasked dump, and a few older dumps in shared storage that nobody remembers creating. Every place Alex checks points to another one.

The questionnaire has a second line: "How do you know what they did?" Alex opens the database log and finds one role name on every line, whether the statement came from a service, from an engineer fixing a row, or from a report that should have been reading a replica. The honest answer to the first question is a pile of places, and the honest answer to the second is that Ledgerly does not know.

Alex does not need a better password. Alex needs to know what access is made of, and what a short, defensible answer would look like. That is the question for this chapter: what is access to a production database made of, and what would it take for Ledgerly to answer that questionnaire without a week of digging?

This chapter describes Ledgerly's starting state, as Chapter 2 does, and adds no target-state step yet. Its job is to give you the vocabulary and the map for the steps that follow.

The problem, and why it is hard

Access to a production database is rarely designed. It accumulates. The first service needs a way in, so someone creates an admin role and a password. The second service reuses it because the first one worked. An engineer needs to look at a row, so the password goes into a desktop client. A scheduled job needs data and gets the same connection string as everything else. None of these steps is a decision about access, and the sum of them is the door the chapter title names.

Look at what Ledgerly's door looks like by the time Alex goes counting. The shared admin password sits in a team password manager, and suppose, as an illustration, that about twenty people know it. The admin role it unlocks owns every table. All five services connect with that role, using the same password stored in each service's configuration, and the deployment pipeline runs migrations with it too. Engineers reach the primary from their laptops through the company VPN, with clients that remember the password and commit every statement as soon as it runs. Access is standing: nobody ever loses it, including people who moved to another team a year ago.

The replicas accept the same admin login, and reporting queries sometimes run on the primary. The analytics copy is a restore of an unmasked dump, and the dump files linger. An external accounting vendor never connects to any Ledgerly database, but it receives exports of invoice and ledger data, and the export step of `reporting-job` writes those files with the shared admin login. The database log shows one role. Nothing in this arrangement records which person ran which statement.

It is tempting to treat this as a discipline problem: tell people to be careful, change the password more often. I would not start there. A rule that works only when everyone remembers it fails the first time the person on call is tired. The system gives every principal the same capabilities and the same name, so a person who wants to be careful has nothing to be careful with: no narrower role to pick, no way to say "read-only" to the database, no session that carries a name.

The same applies to the remedy people reach for first, which is to handle the password better: store it in a vault, rotate it, hand it out only to people who ask. That is credential distribution, and it is worth doing, but it an-

swers one question: who holds the secret. Production database access is not a credential-distribution problem; it is an authorization, execution, and accountability problem. Authorization asks what each principal may do, and through which route. Execution asks how each statement reaches the data, whether anything inspects it first, and how far it can reach. Accountability asks whether the evidence ties each statement to a person and a request. A perfectly stored password settles who holds the secret and nothing else: the person holding it can still run a statement nobody inspected, and the log can still show a shared name.

That is why this is an engineering question rather than a policy one. A policy states what should happen. A design determines what can happen, and what leaves a record when it does. The rest of this chapter takes the shape of a design problem: what access consists of, what we want it to do, and where each part is covered in this book.

Access is five parts

Before judging any design, it helps to have a precise definition of what is being designed. I find it useful to say that access to a production database has five parts: an identity, a grant, a path, an execution and a record. The table says which question each part answers and which pillar of the book's thesis it belongs to.

Part	Question	Pillar
Identity	Who or what is acting: one person or one workload instance?	Accountability
Grant	What may that principal do at all: roles, privileges, ceiling?	Authorization
Path	How does the connection arrive: network reach, gateway, pooler?	Authorization (where)
Execution	How does a statement reach the data? Who or what submits it, whether it is inspected before it runs, how far it can reach (transaction, row guard, timeout), and how it is committed.	Execution
Record	What evidence remains, and does it tie the statement to the person, the request and the result?	Accountability

The three pillars map onto the parts as follows. Authorization is grant plus path, decided per request. Execution is what happens to each statement. Accountability is identity plus record. A password touches only the first of these, and only at the door.

The **identity** is the verified claim that a principal is who it says it is. A **principal** is anything that authenticates: a person, a service, a scheduled job, a vendor integration. Principal is the word to use when a statement holds for humans and workloads alike, and this book uses it that way throughout. A role is not an identity. A **role** is a named bundle of privileges inside the database, and it can be shared by many principals, which is exactly what the shared admin role does.

The **grant** is what the principal is allowed to do once it is in. Here a grant is the act, and the resulting record, of giving a privilege to a role, or a role to a **login**; Chapter 3 extends the definition. A **privilege** is one specific thing, such as reading a table, and the login is the database-level account a principal connects with. The **path** is the route a connection takes to reach the database: a network, a proxy, a gateway, a laptop.

Execution is how a statement reaches the data: who or what submits it, whether anything inspects it before it runs, what bounds its effect, and how it is committed. A grant says a role may update `invoices`. It does not say that this update should touch one row, that someone looked at it first, or that it runs inside a transaction that can be rolled back. In the starting state nothing answers those questions: the desktop client submits whatever is typed and commits it on the spot. Chapter 9 is about this part.

The **record** is the evidence afterward, the answer to "who did what, and when?", and, for a write, "to which rows, under which request?"

Each part can fail on its own, yet each depends on the others to be useful. Without a distinct identity, a precise grant has nobody to attach to. Without a chosen path, an identity can be bypassed by a route that never asks for one. Without a controlled execution, a narrow grant still allows a statement that is wider than anyone meant. Without a record, you cannot tell whether the other four worked. This dependency is why fixing only one part, usually the password, leaves the rest of the problem standing.

A test: can the log answer the question?

One way to find out which parts are missing is to ask the log a question and watch where the answer stops. The following pseudocode traces a single statement through the five parts, using only what the starting state records.

```
function explainStatement(logLine):
    login = logLine.login
    // identity: does this login map to exactly one principal?
    principals = principalsUsing(login)
    if count(principals) != 1:
        return { answer: "unknown", stoppedAt: "identity" }
    // path: did the connection arrive by a route we chose?
    if not isKnownPath(logLine.source):
        return { answer: "unknown", stoppedAt: "path" }
    // grant: was this statement within what the principal was meant
    to do?
    if not withinIntendedGrant(principals[0], logLine.statement):
        return { answer: principals[0], stoppedAt: "grant" }
    // execution: was a write reviewed and bounded before it
    committed?
    if isWrite(logLine.statement):
        if not logLine.reviewedStatement or not
logLine.countShownBeforeCommit:
        return { answer: principals[0], stoppedAt: "execution" }
        if logLine.rowsAffected is unknown or logLine.rowsAffected >
logLine.expectedRows:
        return { answer: principals[0], stoppedAt: "execution" }
    // record: does the evidence tie the statement to the person, the
    request and the result?
    if logLine.correlationId is empty or logLine.result is unknown:
        return { answer: principals[0], stoppedAt: "record" }
    return { answer: principals[0], stoppedAt: null }
```

In Ledgerly's starting state, `principalsUsing("admin")` returns about twenty people, five services and the deployment pipeline, so the function stops at the first check for every statement. That is not a bug in the log. The log is truthful about what it saw, which is one login. The gap is in the design that put many principals behind one login. Fix the identity, and the same function would next stop at the execution check. The founding incident is the case it exists for: a one-row fix whose `WHERE` clause matches a whole batch has `rowsAffected` far above `expectedRows`, was never reviewed, and never showed its count before `COMMIT`, so it stops at "execution" even though the person is known and the grant allowed it. A statement that passes that check can still stop at "record" if nothing ties it to a request and a result.

Applications and people, one question

When people hear "database access", they often picture engineers with query tools. That picture is incomplete in a way that matters. Much of the traffic to Ledgerly's primary comes from software: `billing-api` serving the web app, `invoice-worker` sending reminders, `payments-service` talking to payment providers, `reporting-job` running at night, and `admin-console` acting for support staff. Software connects more often, for longer, and with less supervision than any person does.

Services and people fail in different ways, so Part II covers applications and Part III covers people. The contrast is concrete:

	Services	People
Frequency	Constant, around the clock	Occasional, in bursts
Supervision	None while running	A person is watching the result
Credential lifetime	Short-lived and issued by a broker (Chapter 4)	One session or one approved window (Chapters 7 and 8)
Typical failure	A leaked secret, an exhausted connection pool, a deploy that points at the wrong database	A statement that is wider than meant, run by hand under pressure

Both rest on the foundations in Part I.

There is one more kind of principal that is easy to miss: a service acting on behalf of a person. `admin-console` is used by support staff, but in the starting state it connects with the shared admin login like everything else, so the database sees the console, never the person who clicked. Sam in support looks at one customer at a time, and the design lets the console reach every customer's data under a name that is nobody's. Chapter 3 shows how the console passes the person's identity along with each action, and Chapter 14 how the audit records are joined, so that the person survives the hop.

Security and reliability are the same question

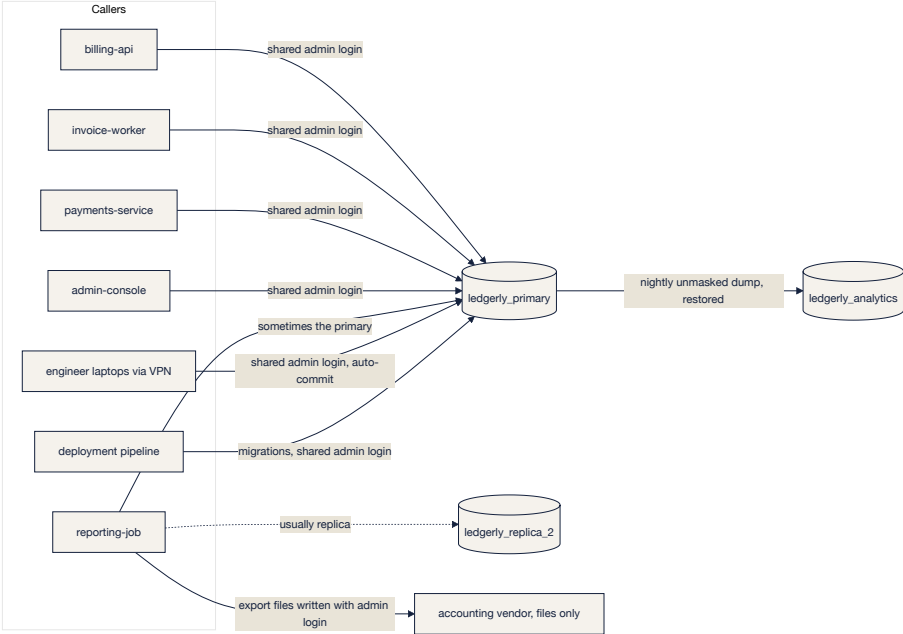
Teams often keep two lists. One is about security: who could steal data, who could abuse access. The other is about operations: what could take the database down or corrupt it. They are usually owned by different peo-

ple and reviewed at different times. For access, I would merge them, because the questions are the same.

Take three events that sound unrelated. A statement with a wrong or missing `WHERE` clause updates every invoice instead of one. A laptop client leaves a connection open and holds a slot all evening. A dump of the production database is copied to shared storage and read by someone who had no business reading it. The first is a data integrity problem, the second an availability problem, the third a confidentiality problem. Yet each one asks the same thing of the design: why was a single principal able to do something this wide, without a narrower route, a limit or a record?

The three events share a design cause: no narrowing of grant, lifetime or path, and nothing to show afterward. The wrong `WHERE` clause is mostly a grant and execution gap, the open connection a lifetime and limit gap, and the copied dump a path gap. Narrowing the grant, limiting the lifetime, choosing the path, bounding the statement and keeping the record help with all three events at once. That is why a single design serves both lists. A role that cannot read `payment_tokens` protects customers from a leak, and a role that cannot write `ledger_entries` protects the ledger from a bad batch update. Both come from the same habit of granting narrowly.

The diagram below shows Ledgerly's starting state. Look at how many arrows end at the same node, and at the one arrow that leaves it for the analytics copy.



Two things stand out. First, every path to the primary carries the same name, so the database cannot tell a worker from a person. Second, the data leaves through a side route as well: the nightly restore copies everything, including columns that no analyst needs, into a store with its own audience.

The range of ways in

Between the shared password and a fully controlled design there is a range of patterns, and the later chapters place each one on it. The diagram runs from the least to the most control over each statement. Look at what each step adds, and at where the first statement-level check appears.



Each step to the right adds a property and costs something. A bastion narrows the path, but the session on the far side is still unsupervised. A session proxy ties a connection to a person and records it, but it hands over a session in which anything permitted by the grant can be typed. A gateway that mediates execution looks at each statement before it runs and can refuse it. A product path removes the session altogether: the person asks

for a defined operation, such as "resend this invoice", and never writes SQL. Nothing here says the rightmost pattern is always the best one; it is also the most to build and the slowest to change. Chapter 7 compares the first four on concrete paths, Chapter 9 covers the statement-level checks, and the closing chapter scores the patterns by pillar.

What good access looks like

I would write the goal down before choosing any tool. I use five properties, and each one has a price.

Least privilege is the practice of granting only what a task needs, for as long as it needs it. The price is the effort of finding out what each task needs, and the friction when a task turns out to need more. A team that adopts it without a fast way to ask for more will watch people route around it.

Short life means access ends on its own. A grant that lasts an hour does not need anyone to remember it. Contrast this with **standing access**, which exists continuously whether or not it is used. A standing grant is a promise that nobody will need to ask again, and it is convenient for the first month and expensive for every month after, because nobody remembers what the promise covered. The cost of short life is more asking, and an expiry that can arrive halfway through a repair.

Observability means every use of access leaves a record that names the principal, and the record sits somewhere the principal cannot change. The cost is storage and, more importantly, attention: a log nobody reads is security theater, since it costs storage and gives no protection. Chapter 14 returns to this.

Reversibility has two meanings, and both matter. Access should be easy to take away, and a mistake made with access should be possible to undo or at least to contain. Revocation has to be routine, and undoing a mistake needs reviewed fixes and backups that restore. One cheap form of reversibility is to run a write inside an explicit transaction and show the row count before `COMMIT`: an update that was meant to touch one row and reports, say, four hundred is stopped while it can still be rolled back. Chapter 9 covers how a gateway makes that the default. The cost is time spent on rehearsals.

Low friction is the one teams forget to write down. If the legitimate path is slow, people will find the illegitimate one, and they will be right to, because the incident at 2 a.m. does not wait. The cost of low friction is that it constrains the other four: every control has to be cheap enough to use during the worst hour of the week.

These five pull against each other. Short life adds friction. Observability adds a hop: a session that is recorded has to pass through something that records it, and that something can be slow or down. Least privilege multiplies roles. I do not think there is a setting that maximizes all five. The useful question is not "who should have access?" but "which statements may run, through what, and with what record?" Each later chapter answers a slice of it, and says what the answer costs.

A first inventory

Before you adopt any control, I would write down what exists. The following sketch shows the shape of an inventory for Ledgerly's starting state. The point is to have one row per door, with a named owner and a reason, so that a row with no owner is visible as a gap.

```
door:      desktop client on engineer laptops
principals: about twenty people (illustrative)
login:     shared admin
reaches:   ledgerly_primary, ledgerly_replica_1, ledgerly_replica_2
network source: company VPN, any laptop
grants:    owns every table
can write?: yes, auto-commit, no row limit
record:    database log shows the shared role only
owner:     nobody
reason:    "needed to fix rows during incidents"
ends:     never
```

Fill in one block per door: each service, the scheduled job, the deployment pipeline, the laptops, the analytics restore, the dumps in shared storage, and the export files. Most teams find that the `owner` and `ends` lines are the ones left blank. An inventory like this is not a project plan. It is the thing that turns "who can reach the primary?" from a day of digging into a document you can read.

A map of this book

The chapters that follow take each part of access, and each failure above, in turn. The table links every question this chapter raised to the place where the book answers it.

The question	Pillar	Where it is answered
Who and what are we protecting data from, and from what kinds of events?	All three	Chapter 2
How do we give each principal a distinct identity and the narrowest role that works?	Authorization	Chapter 3
How do services get credentials without a shared password in a file?	Accountability (identity)	Chapter 4
How do connection limits and timeouts become a policy that protects the database?	Authorization and execution (bounds)	Chapter 5
How do reads reach replicas, and what happens to the other copies of data?	Authorization	Chapter 6
By which paths do people reach the database, and what sits on each path?	Authorization (path), accountability	Chapter 7
How does someone get write access for a task, and how does it end?	Authorization	Chapter 8
How is a statement checked before it runs, and how is it committed?	Execution	Chapter 9
How do ad-hoc queries and data fixes stay reviewable?	Execution	Chapter 10
Who changes the schema, and with what access?	Execution	Chapter 11
Which data needs extra protection, and how is it masked or withheld?	Authorization	Chapter 12
What about backups, exports and the other routes around the controls?	Authorization (path)	Chapter 13
How do we keep a record that holds up, and review it?	Accountability	Chapter 14
What changes during an incident?	All three, under pressure	Chapter 15
What do we do first, and how do we keep it up?	All three	Chapter 16

The book has five parts: foundations, application access, human access, protecting the data and its copies, and proving and operating all of it. Each chapter names the pillar it serves in its opening. You do not have to read in order. If your immediate problem is the shared password in five configuration files, Chapter 4 is the place to start, but Chapter 3 will save you from creating the next shared role by accident.

Trade-offs and costs

The first cost is **time and attention from a small group**. Someone has to build the inventory, define roles, set up credentials and run reviews. In a small team, this is often one person's part-time job, and it competes with product work. If the work never gets scheduled, the design stays on paper. I would rather see a small, finished first step than a large plan.

The second cost is **friction for whoever is on call**. Every control puts a step between a tired person and a repair. Narrow roles mean the first attempt sometimes fails with a privilege error at the worst moment. Short-lived access means asking again. Decide in advance what the fast path is, and rehearse it. A control that is bypassed during every incident is not a control; it is a delay before the bypass.

The third cost is **new systems that must be available when needed**. A gateway, a secrets store and a log store are each something else that can be down. Before you adopt one, ask what happens to the person who needs the database during that outage, and keep an answer ready.

There are also cases where the full design is the wrong place to start. A single-service prototype with no customer data, run by two people, does not need a gateway or a review calendar. It does need its own login separate from any person's, because that is cheap, and a note saying what would make the team revisit the question: the first customer, the third engineer or the first external integration. I do not know the right moment for your company. It depends, and here is on what: how sensitive the data is, how many principals touch it, and how quickly a bad statement can do damage you cannot reverse.

Finally, the good-access properties have a limit. They reduce the blast radius and make behavior visible. They do not make a database safe, and no chapter in this book claims otherwise. **Blast radius** is how much can be af-

fectured if one principal, credential or action goes wrong, and the work of this book is to make that radius smaller and more visible, not to make it zero.

Common mistakes

These are design traps, not personal failings. Each one is tempting for a reason, and each one has a remedy that does not depend on anyone being more careful.

- **Treating a password change as the remedy.** A new password feels like progress, and it removes a former colleague's knowledge. But if, say, twenty people and five services still share it, the design is unchanged. The fix is a design in which nobody needs to share a password, which is what Chapters 3 and 4 build.
- **Fixing the grant and forgetting the path.** Narrow roles help only if connections have to use them. If laptops can still reach the primary with an old admin login, the narrow roles are optional. Inventory every path, not only every role.
- **Leaving out the applications.** Reviews of "who has database access" often list only people, because people are on an org chart. The services hold the more capable credentials and use them more. Include every workload, every scheduled job and every script.
- **Making the safe path slower than the unsafe one.** If reading production takes a ticket and a day while the shared password takes, say, ten seconds, the shared password wins every time. Aim for a safe path that is no slower.

Warning: An unmasked dump in shared storage is the starting state's most direct route to customer data, and it is open to anyone who can read the folder. Treat finding and removing these files as part of the first inventory, before any redesign of roles.

Key takeaways

- Production database access is not a credential-distribution problem; it is an authorization, execution, and accountability problem, and a pass-

word answers only the first question, at the door. Access is a design of five parts, namely Identity, Grant, Path, Execution and Record, and a weakness in any one of them undermines the others.

- A shared admin account is not a decision anyone made; it accumulates, and the remedy is a design in which nobody needs to share a password.
- Applications connect more often and with less supervision than people do, so any treatment of access has to cover services, jobs and tools as principals alongside humans.
- Security and operational reliability are one question for access, because a wrong **WHERE** clause, a leaked connection and a copied dump share a design cause: no narrowing of grant, lifetime or path, and no record.
- Good access has five properties, which are least privilege, short life, observability, reversibility and low friction, and they pull against each other, so every control should be presented with what it costs.
- The first step is an inventory with one row per door, including the services, the laptops and the data copies, where a missing owner or a missing end date shows up as a visible gap.
- Controls reduce blast radius and make behavior visible, but they do not make a database safe, and the honest goal is a smaller radius and a clearer view.

The next chapter turns from the shape of access to the reasons for protecting it. Chapter 2 asks who and what the controls are for, sorts the events that can go wrong into three kinds, and writes a threat model for Ledgerly that the rest of the book can point back to.

Chapter 2 — A Threat Model for Production Data

Imagine that Ledgerly's security contact sends the platform team a short question before a quarterly review: "Who are we protecting the production database from?" Alex, who runs the databases, answers within a minute: outsiders, mostly. Then Alex pauses, because the answer does not match the team's own tickets from the past year. A support request that touched the wrong customer. A reporting query that slowed invoice writes for an afternoon. A dump restored on a laptop for one day of debugging, which nobody was reminded to delete. A vendor export that went out with one column more than intended.

None of those involved an outsider. None involved bad intent either. All of them involved a door that was open to someone who had a reasonable task, and a role that allowed more than the task needed. Alex's first answer was the one that sounds like security. The second list is the one that describes the system.

This chapter describes Ledgerly in its starting state, as Chapter 1 did, and adds no control of its own. What it adds is a written threat model, the reference that every later control in the book is chosen against. It frames the threats against all three pillars: what a principal is allowed to do, how each statement reaches the data, and what evidence remains. The question it answers is the one Alex could not answer in a minute: who and what can hurt the data, through which doors, and how far does the damage travel?

The problem, and why it is hard

Access design without a threat model tends to follow the loudest worry. If that is an outside attacker, the team builds a perimeter: a VPN, a firewall rule, a network allowlist. Those matter. But a perimeter answers "who may get near the database?" and says nothing about what a person or service

can do once inside. In Ledgerly's starting state, anyone on the VPN with the shared password can do everything the admin role can do, which is everything.

The obvious alternative is a list of controls from a checklist, applied in order. A checklist does not know which of your doors carry the most risk. It will ask for rotation of a password that nobody could rotate without an outage, and say nothing about the nightly restore that copies every Restricted column to a store with a much wider audience. Controls chosen without a model are chosen by availability and fashion. Some reduce exposure. Some are security theater, friction without a matching reduction in risk, and the team cannot tell which is which because it never wrote down what the control was for.

A written model is needed for three reasons. First, the doors are technical objects: a login, a network path, a file in shared storage, a pipeline variable. A policy can say "access must be appropriate", but only a design decides what a login can reach. Second, the biggest risks come from the interaction of ordinary choices, as the Preface describes: a shared password, an auto-committing client and a role that owns every table. No single choice looks wrong, so a review of single choices will not find the problem. Third, the model has to be cheap enough to maintain, or it will be written once and left to drift.

A threat model, in the sense used here, is small. It lists who and what can act on the data, through which path, what could go wrong, how far the effect travels, and which control addresses it. It is not a risk score or a prediction. It is a way of asking the same question about every door, so that the answers can be compared.

Note: The book describes threats only as far as they justify a control. This chapter names what can go wrong, such as a credential leaking or a query landing on the wrong database, and does not describe how anyone would make it happen. If more detail would not change the design, it is left out.

Who and what reaches the database

Chapter 1 said that access to a production database has five parts: an identity, a grant, a path, an execution and a record. A threat model starts by listing the principals and asks the five questions of each. Ledgerly has nine kinds of actor worth listing. Two of them, the vendor and the outside attacker, are not principals in the strict sense, which is why they belong on the list: they are easy to leave off it.

Services

The services (`billing-api`, `invoice-worker`, `payments-service` and `admin-console`, with `reporting-job` treated next as a job) are the heaviest users of the database. They run unattended, hold their credentials in configuration, and behave only as correctly as their code. A service can misbehave without anyone doing anything wrong: a retry loop that opens connections until the limit is reached, a query that scans a table because an index was dropped, a deployment that points a staging build at the primary.

In the starting state every service connects with the same admin login, so each service's blast radius is the whole database, even though `billing-api` and `invoice-worker` each need only a handful of tables. A defect in the smallest service has the reach of the largest.

Batch jobs

`reporting-job` deserves its own line because a job runs on a schedule, often at night, with nobody watching. It tends to read a lot of data, hold long transactions and produce files. In the starting state it sometimes reads the primary, and its export step writes the vendor files with the shared admin login. Its normal behavior already looks like bulk access, which makes its abnormal behavior harder to spot than a person's.

On-call engineers

The on-call rotation is under the most pressure to have broad access, and with good reason: a page at night is not the moment to wait for an approval that nobody is awake to give. In the starting state they use the shared admin password through desktop clients that remember it and

commit on every statement. The combination of urgency to fix and no friction to change anything makes this the group where accidents are most likely, and the group the design can protect best.

Support

Support staff use `admin-console` to look at one customer at a time. The threat here is scope, not malice. A tool that can display every customer, backed by a role that can read every column, answers a one-invoice question with a system that could show every invoice. The design question is how to make the broad capability unavailable when only the narrow one is needed.

The data team

The data team works on `ledgerly_analytics`, which in the starting state is a nightly restore of an unmasked dump: every Restricted and Confidential column, reached by a group whose job is to look at data in bulk. Their access is legitimate. The concern is that the copy has a wider audience, looser handling and fewer logs than the primary, so a control on the primary does not protect what sits in the copy.

Vendors and integrations

The external accounting vendor never connects to a Ledgerly database. It receives files. That changes the shape of the threat: the exposure is not a login but an export, and its blast radius is whatever columns the export contains and for how long the copy exists on the vendor's side. The relevant questions are what the file holds, who owns it afterward, and whether anyone would notice if the file grew a column.

CI/CD

The deployment pipeline is a principal too, even though it is infrastructure rather than a person or a service. In the starting state it runs migrations with the same admin login, because the admin role owns every table and nothing else can change them, and the credential sits in the pipeline's configuration. A pipeline that can change any table and holds a standing credential is a compromise risk and an error risk at once, since a bad migration runs with the owner's full privileges.

Platform and DBA staff

The small platform group runs the databases, and it is the one actor that can change the controls themselves: roles, grants, the entry that humans pass through, the log. In the starting state it uses the shared admin login like everyone else, so nothing separates its work from anyone's. In the target design it holds a role for managing access, `ledgerly_access_admin`, which three principals use through their own logins. That does not remove the threat; it moves it. A mistaken grant or a disabled log setting by someone with this role changes what every other row of the model means.

For that reason the model lists the **control plane** as an asset in its own right: the access gateway, the credential broker, `ledgerly_access_admin` and the audit store. Data is protected by the controls, and the controls are protected by a much smaller set of people and systems. The exposure is small in volume and large in effect, and the questions are the same as for any door: who can reach it, what could it change, and who would notice.

The outside attacker

An outside attacker is not a principal until they obtain a credential, so the useful way to model them is by what they would need to obtain, not by what they might do. The defender's questions are: which doors accept a credential that could leak, what would it reach, and how would we find out? The answers come from the other eight actors. A leaked credential inherits the blast radius of whoever it belonged to.

Three classes of threat

The same actors can cause harm in three different ways, and the controls differ for each. Mixing the classes leads to mismatched controls: an approval of intent, designed for misuse, does little against an accident, because the approver agrees the task is legitimate and the statement still goes wrong.

Errors

An error is an action that was meant to be correct and was not: a statement run against the wrong environment, a condition that matches more rows than intended, a query that is too heavy for the primary, a script

pointed at production by a stale configuration. The actor is a principal doing their job.

I would expect this class to be the most frequent in most teams, for a structural reason rather than a statistical one. Misuse needs a person with a motive, and compromise needs an attacker with an opportunity, but an error needs only a person with a task. I have no number for how much more frequent errors are, and I would not trust one if offered. The conclusion for design holds either way: controls aimed at errors protect against the class every team has, whether or not it ever faces the other two.

Errors are best addressed by narrowing what a single statement can reach, since people cannot be asked to be careful forever. Roles that cannot touch tables they do not need (Chapter 3), timeouts that stop a runaway query (Chapter 5), analysis of each statement before it runs (Chapter 9), a distinct route for writes with four-eyes review (Chapters 8 and 10), and a reviewed way to repair data afterward (Chapter 10) all belong to this class.

The four-eyes step helps here only on one condition: the reviewer sees the statement and its expected effect. Approving the intent ("fix invoice 1042") is a control for the misuse class, and it leaves the error class untouched. Approving the statement and the number of rows it should change addresses errors.

Misuse and curiosity

Misuse here means a principal with legitimate access using it for something outside the task: reading a customer's record out of interest, exporting more than the work needs, looking at colleagues' data. It includes plain curiosity, which is probably the most common form and the hardest to call wrongdoing. It also includes the rare deliberate case.

The design response is not suspicion of the team. A team does not need to assume bad faith to notice that a standing grant makes browsing possible and leaves no trail. The controls that matter keep access tied to a task and make it visible: read-only roles that withhold Restricted columns, masked views of Confidential data (Chapter 12), time-limited grants (Chapter 8), and an audit log that a reviewer actually reads (Chapter 14). Without evidence, "was this access appropriate?" cannot be answered for anyone, including the people who did nothing wrong.

Compromise

Compromise means a credential or a machine is in the hands of someone who is not the intended principal: a password that left through a configuration file, a laptop that is lost, a pipeline variable read by something it should not trust. The defender does not need to know how it happened to design for the fact that it can. The design controls what the credential could do, for how long, and whether anyone would know.

Compromise is where the properties of a credential matter most. A credential that is shared, static and unrestricted turns the loss of one laptop into the loss of every table. A credential that is scoped to one role, short-lived and issued to one session turns the same event into a bounded and visible one. Chapters 4 and 7 cover how credentials are issued; Chapter 3 covers how little they should carry; Chapter 14 covers how to notice.

Note: Classifying an event is often a judgment call. A one-row fix that reaches a whole batch is an error. The same statement run on purpose by someone with no task is misuse, and run with a credential taken from a laptop it is compromise. The database sees the same statement each time, which is the argument for logs that name a login and for controls that do not depend on guessing intent.

What is being protected

Three properties of the data are worth protecting, and the model should say which one each threat touches. They are the familiar confidentiality, integrity and availability, and each has a Ledgerly-specific meaning.

Confidentiality is about who can read what. For Ledgerly the weight is uneven. `iban`, `account_holder_name` and `token_value` are Restricted, so their exposure would be hard to undo. `contact_name`, `contact_email`, `contact_phone` and `billing_address` are Confidential customer PII (personally identifiable information). `invoices` and `payments` amounts are Internal. A threat that touches Internal data deserves less design effort than one that touches Restricted data, and the model should show that difference instead of treating all columns as equal. Data-protection

regimes of this kind tend to ask what personal data you hold and who can see it, which is the same difference; your legal and compliance colleagues own the interpretation.

Integrity is about whether the data stays correct. For most tables an error can be corrected by a later write, but `ledger_entries` and `audit_events` are append-only, and for the ledger, integrity matters more than secrecy: a wrong entry is fixed by a new correcting entry through `payments-service`, never by an edit. An integrity threat to those two tables is therefore different in kind. It is also the one where evidence matters most, since a ledger that can be changed quietly is a ledger nobody can rely on.

Availability is about whether the database serves the services that depend on it. It is the property most often forgotten in security discussions, and the one where accidents dominate. A long lock, a query that scans a large table on the primary, or a flood of connections from a retry loop can stop invoicing and payment collection as thoroughly as an attack could. Availability gets first-class rows in the model, tied to connection policy (Chapter 5) and to where reads are allowed to go (Chapter 6).

A fourth property sits beside these three: **accountability**, meaning that the record can say which principal did what. It is the record part of access. In the starting state it is missing, since the log shows only the shared role, and that absence raises the cost of every other threat.

Blast radius

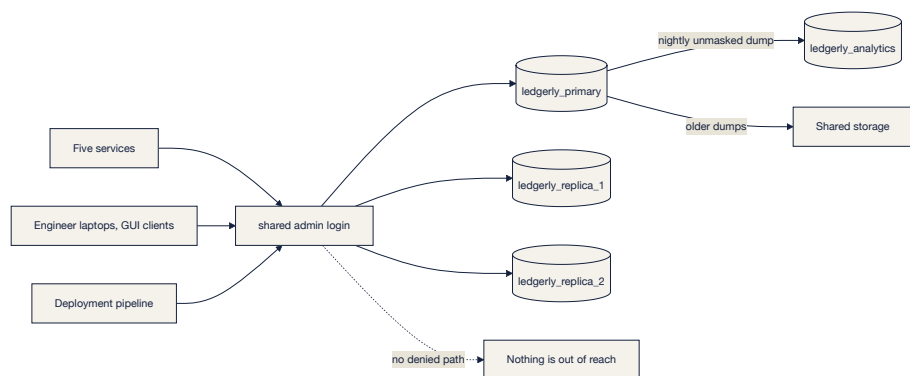
The model needs one more tool to compare threats: a way to say how far a single failure travels. **Blast radius** is how much can be affected if one principal, credential or action goes wrong. The term is only useful when it is used about a specific thing, so the question is always "what is the blast radius of this login?" and never "what is our blast radius?"

A blast radius has seven dimensions, and it helps to look at each separately:

- **Data reach:** which tables, columns and rows the principal can touch.
- **Action reach:** whether it can read, write, delete or change the schema.

- **Environment reach:** which databases it can connect to: the primary, the replicas, the analytics copy.
- **Time reach:** how long the access stays valid, from one session to forever.
- **Copy reach:** where else the data it can read has been duplicated, and who can reach those copies.
- **Statement reach:** how much a single statement can change before anything stops it: the rows it touches, whether it runs in a transaction that can be rolled back, and whether a timeout bounds it.
- **Network origin:** which networks and machines can present the credential at all.

The shared admin login in Ledgerly's starting state maxes out every dimension. It reaches all tables and all columns, it can read, write, delete and alter, it connects to the primary and both replicas, it has no expiry, and its data already has copies outside the primary. Any statement is unbounded and commits at once, and any laptop on the VPN can present the credential. The diagram shows what to look at: everything on the left holds the same credential, so everything on the right is inside one radius.



Compare it with the target-state role `app_reporting_job`: read-only, on a replica, with no access to Restricted columns. A defect that could expose Restricted data in the starting state is, for that role, a heavy query on a replica. The defect is identical; the blast radius changed.

To keep the comparison honest, write the radius down per login, as a list of questions and not a score, and put two logins side by side. The sketch below is illustrative, a review aid and not a tool the book builds on. An un-

known answer is recorded as unknown, because a model that rounds unknowns down to "fine" hides the cases it exists for.

dimension	shared admin login	app_reporting_job
data reach	every table	unknown until Chapter 3
lists the tables		
restricted cols	all	none
action reach	read, write, delete, DDL	read
environment	primary, both replicas	one replica
time reach	no expiry	unknown until Chapter 4
decides		
copy reach	unmasked restore, dumps	the export files it
writes		
statement reach	unbounded, commits at once	read-only transaction,
statement timeout		
network origin	company VPN, any laptop	unknown until Chapter 7
decides		

The comparison is useful because it is uneven. Where the second column says "unknown", the model has found a question the design has not answered yet, and that is a finding and not a gap in the table.

Two limits apply. A radius describes what a login could do, not what it will do, so it is a planning limit and not a forecast. And it says nothing about how likely each threat is. A narrow radius on a door that is open all day may still deserve more attention than a wide radius on a door that opens once a quarter, which is why the model has to combine radius with exposure.

The Ledgerly threat model

With the actors, classes, properties and blast radius in place, the model fits in one table. The table below covers Ledgerly's starting state. Each row is one actor and one path, with the class of threat that matters most for that pair, the impact on the properties above, the chapters that supply the control, and the **residual risk**: what is still true after those controls are in place. "Exposure" is a qualitative judgment (high, medium or low) of how often the door is used and how easy it is to misuse, and it is written down so that someone can disagree with it.

Actor	Path	Main class	Impact	Exposure	Controlled in	Residual risk
Services (all five)	Shared admin login from each service's configuration	Error, compromise	Any table, all columns; availability if connections run out	High	Chapters 3, 4, 5	A defect still has the reach of its own role
reporting-job	Reads the primary on some runs	Error	Heavy reads compete with writes; availability of the primary	Medium	Chapters 5, 6	A heavy read on a replica still slows other replica readers
On-call engineers	Laptop, GUI client, shared admin password, auto-commit	Error, compromise	Unscoped writes; integrity of any table	High	Chapters 7, 8, 10	A hand-typed statement inside a granted role can still be wrong
Support	admin-console acting under the shared login	Misuse, error	Browsing of Confidential customer data; wrong-customer edits	Medium	Chapters 3, 10, 12	Support still sees the columns one customer's case needs
Data team	Unmasked analytics restore and older dumps	Misuse, compromise	Confidentiality of every Restricted and Confidential column	High	Chapters 12, 13	Masked data can still reveal values by inference
Accounting vendor	Exported files	Error, misuse	Confidentiality of exported columns and their lifetime	Medium	Chapters 12, 13	The vendor keeps the file after delivery
Deployment pipeline	Admin login stored as a pipeline variable	Error, compromise	Schema and data integrity; all tables	Medium	Chapters 4, 11	A reviewed migration still runs with owner rights
Platform and DBA staff	Shared admin login now; ledgerly_access_admin in the target design	Error, misuse, compromise	Grants, controls and evidence can be changed	Medium	Chapters 3, 7, 14	The role exists, so its misuse is possible; only detection remains
Control plane (asset)	Gateway, credential broker, audit store	Error, compromise	Every other row changes meaning	Medium	Chapters 4, 7, 14	Compromise of one component weakens the evidence it holds

Actor	Path	Main class	Impact	Exposure	Controlled in	Residual risk
Any human session	A statement typed by hand within a granted role	Error	Unbounded statement, committed at once	High	Chapters 9, 10	A statement can be valid, approved and still wrong in effect
Any network origin	Company VPN reaches the primary directly	Compromise	A leaked credential works from any laptop on the VPN	Medium	Chapter 7	A credential stolen from an allowed origin still works there
Outside attacker	Any leaked credential from the rows above	Compromise	Inherits the radius of the credential's owner	Low to medium	Chapters 4, 7, 14	A credential stolen inside its lifetime works until it expires
Every actor	No named login in the database log	All classes	Accountability for every other row	High	Chapters 7, 14	The record is only as trustworthy as the store that holds it

Four observations follow from the table, and they are the reason to build it.

First, the shared admin login appears in most rows. The table shows one problem with many faces, and the first control, a separate role per principal (Chapter 3), addresses a large part of it at once. A model that reveals the order in which to work is doing its job.

Second, among the highest exposures are doors that were never meant to be doors: the analytics restore and the older dumps. None of them is an access path in anyone's mental diagram of the database. That is why Chapters 6 and 13 treat every data copy as a door with an owner and an expiry.

Third, the residual-risk column is the honest part of the table. Every control leaves something: a granted role can still be misused, a valid statement can still be wrong, a record can only be trusted as far as its store. A row with no residual risk written down is a row where nobody asked.

Fourth, the control-plane, human-session and network-origin rows name an asset or a path rather than an actor, and the last row names no single actor either. Missing accountability is a property of the whole system, and

it raises the cost of every other row, because without a named login no incident can be scoped. Chapter 14 closes that row. Earlier chapters build the identities that make closing it possible.

Warning: A completed threat model is a map of where Ledgerly is weakest. Keep it in a place with the same access restrictions as the systems it describes, and do not paste it into a public ticket or a wide-audience wiki. A document that lists unmasked copies and shared credentials is useful to the people who maintain it and to nobody else.

Keeping the model alive

A threat model written once is a snapshot, and the table above is out of date the day Ledgerly adds an integration or the data team gets a new copy. I would treat it as a living document with a named owner, like a run-book, kept in a repository with the same read restrictions as the systems it describes, since the table lists weaknesses. This has a cost. The access configuration usually lives in a more widely readable repository, so a change there does not automatically show up in the model's review. I would accept that cost and close the gap with a rule: a pull request that changes the access configuration links to the matching model change, or says why none is needed.

Some events should trigger an update without anyone having to remember:

- A new service, job, vendor or tool that reaches production data.
- A new data copy: a new replica, an export, a staging refresh, a dump for a one-off analysis.
- A change in who can reach what: a reorganization, a new on-call rotation, a vendor contract that ends.
- An incident or near miss, which is evidence about which row was wrong.
- A finding in an access review (Chapter 14), which is evidence about which assumption drifted.

Each row should record enough that a stranger can challenge it: the actor, path, class, impact, exposure judgment, control and its owner, the date of the last review, and the assumption the row depends on. The last field is the one teams skip and the most useful: "the vendor export contains only these columns" is a claim that can be checked, and a row that depends on an unchecked claim is a guess.

A small check can find rows that have gone stale. It does not decide anything. It points a human at the rows most likely to be wrong, and it treats a missing review date as stale. It also takes the events listed above as input: when a pipeline creates a login, a role or a data copy, it records an event, and an event no row covers opens a review item, so the update does not depend on anyone remembering.

```
function findStaleThreatRows(rows, maxAgeDays, accessEvents):
  stale = []
  // an event from the pipeline: a new login, role or data copy
  for each event in accessEvents:
    if not anyRowCovers(rows, event):
      stale.append(newReviewItem(event))
  for each row in rows:
    // a row with no review date has never been reviewed
    if row.last_reviewed is null:
      stale.append(row)
    else if now() - row.last_reviewed > maxAgeDays:
      stale.append(row)
    // a control with no owner is a control nobody maintains
    else if row.control_owner is null:
      stale.append(row)
  if count(stale) > 0:
    notify("platform", "threat model rows need review")
  return stale
```

The cadence is a trade-off. A quarterly pass is something a small team can keep, and a yearly one is rarely enough, but I would pick the interval by how fast Ledgerly's doors change. The security and compliance contacts set the questions the review asks, and the platform team answers them with facts about the system.

Trade-offs and costs

A threat model costs time to write and more time to keep. For a small team the first version plausibly takes a few days of careful discussion, and the maintenance is a recurring item that competes with feature work. If

nobody owns it, the cost is paid once and the benefit decays. In a very small system with one service and one database, the full table is more than the problem needs; a half page listing the doors and the worst-case radius of each is enough.

The model can also create false confidence: a tidy table suggests that every threat is on it. It lists the threats someone thought of, and an unlisted door is invisible to it. The best defense is the update triggers above, and the habit of asking after every incident, "which row would have predicted this, and if none, what is the missing row?"

The exposure judgments are subjective. Two engineers will rate the same door differently, and the disagreement is informative, but a rating implies more precision than the evidence supports. I would keep it to three levels and resist computing scores.

There is a human cost in listing people's groups as actors. Calling on-call engineers or support a threat actor can read as an accusation. The answer is in the framing: the model describes paths and roles, not individuals, and its entries say what the design allows. The teams in the table appear because they have the most legitimate need to reach data.

When not to use the full model: if nobody can currently answer "who has access?", start with an inventory of logins and data copies before writing threats, because a model built on an inaccurate inventory describes a system that does not exist. And do not use it as a substitute for a decision. It ranks and explains; the team still chooses which controls to build first and which risks to accept in writing.

Common mistakes

- **Modeling only the outside attacker.** It matches the common picture of security, and the controls are familiar. The cost is that errors and curiosity go unaddressed. Give each of the three classes its own rows in the table.
- **Listing controls instead of threats.** "We have a VPN" records an answer without the question. State what can go wrong first and the control second, so a control can be retired when its threat no longer applies.

- **Leaving out actors that are not people.** Pipelines, jobs and exports are easy to overlook because nobody logs in as them, and they tend to hold the most standing power. List every principal and every export.
- **Treating data as uniformly sensitive.** If every column is "critical", the model gives no guidance on where to start. Use the four data classes (Restricted, Confidential, Internal, Public) so the weight of a threat follows the weight of the data.
- **Skipping availability.** A runaway query does not look like a security event, so it misses security reviews. For a payments business it can matter as much. Give availability its own rows.
- **Writing it once.** The model is accurate for a few weeks and then becomes folklore. Assign an owner and add update triggers.

Key takeaways

- A threat model is a short structured list of actors, paths, threat classes, impacts and controls, and its job is to let a team compare doors against each other.
- Errors, misuse and compromise need different controls, and a design that only considers outsiders leaves the first two classes unaddressed; approving intent addresses misuse, while approving the statement and its expected effect addresses errors.
- Every actor belongs on the list, including pipelines, batch jobs, vendor exports and the platform staff who run the controls, because the principals that nobody logs in as often hold the most standing power, and the control plane is an asset too.
- Confidentiality, integrity and availability each need their own rows, and accountability, the ability to say which principal did what, raises the cost of every other threat when it is missing.
- Blast radius is always about one specific login, credential or action, and it is described along seven dimensions: data, action, environment, time, copies, statement and network origin.
- In Ledgerly's starting state one shared admin login appears in nearly every row, so a separate role for each principal is the control that shrinks the most of the table at once.

- The model is only as good as its upkeep: give it an owner, tie updates to events such as new services and new data copies, and keep it where only the people who maintain the system can read it.

The model tells you which doors matter most, and in Ledgerly's case nearly all of them lead back to one login. Chapter 3 begins taking that login apart: who the principals are, what a role should hold, and how little each of them needs.

Chapter 3 — Identities, Roles and Least Privilege

Imagine a Thursday evening at Ledgerly. A new table, `invoice_disputes`, went live on Tuesday, and the security contact asks the platform team a short, reasonable question: who can read it? Alex starts with the grants and finds none, because every caller connects as `admin`, which owns the table and can read anything. Alex then looks for who actually did read it. The database log has thousands of lines for the table, all written by the same login: the worker, the nightly report, two engineers on their laptops and the support tool.

Nobody can place the new table. The honest answer to "who can read it?" is "whoever can reach the database", and the honest answer to "who did?" is "someone". Nobody did anything wrong. The design never asked the database to tell these callers apart, or to limit any of them.

Chapter 1 split access into five parts: **Identity**, **Grant**, **Path**, **Execution** and **Record**. The book's thesis is that access is an authorization, execution and accountability problem, not a credential-distribution one. This chapter serves the authorization pillar. It covers the first two parts, who is asking (Identity) and what that principal may do (Grant), and it marks where grants stop, because the Execution part picks up there (Chapter 9).

It adds the first target-state steps from the book's running example. Every table is owned by `ledgerly_owner`, and nobody connects as the owner. Each service gets its own application role with the narrowest grants that work. The chapter also introduces the full role schema, the way principals reach roles, column-level grants and the ceiling for just-in-time access. Later chapters build on these definitions and do not repeat them.

The problem, and why it is hard

A database answers two questions on every statement: who is this, and is that principal allowed to do this? A shared account makes the first answer meaningless, and a broad account makes the second answer always yes. Both are easy to arrive at. One login, one password and full rights is the cheapest thing that works on the day a team creates its first database.

The cost arrives later, as missing evidence. When the people and services that share the admin login (say, twenty people, five services, the deployment pipeline and the export step) all use it, the database log cannot say which of them ran a statement. The application log describes what the application meant to do, not what reached the database. That is a design problem, not a discipline problem, because no amount of care by individuals brings back information the design discarded.

The obvious remedy is to write a policy: grant only what is needed. A policy does not work here, because the database makes the broad choice the easy one. Granting everything takes one statement and never fails.

Granting the narrowest set that works takes knowledge of every query a workload runs, including the monthly job nobody remembers and the retry path that runs twice a year. A grant that is too narrow breaks production at the worst moment. A grant that is too wide breaks nothing, so nobody notices it. Under that asymmetry, privileges only ever grow.

Ownership adds a second trap. In most engines the owner of a table can change its structure, drop it and grant access to others. If a service connects as the owner because it creates tables at startup, the service holds the power to rewrite the access model of the database. If a person is the owner, the model depends on that person's employment. Ownership is a structural privilege, and it should not live in anything that handles traffic or anything that handles a career.

The remedy is a structure, not an instruction: a distinct login per principal, a small set of roles that carry privileges, and a way to keep both in step with what the team intends.

Principals, logins and roles

This chapter works on Identity and Grant, and touches Path where a role says where it may connect. A **principal** is anything that authenticates. In Ledgerly's world there are four kinds worth separating.

- **Humans.** On-call backend engineers, support staff, the data team and the platform team. They come and go, change teams and work from different places.
- **Services.** `billing-api`, `invoice-worker` and `payments-service`. They are long-running, they run as many instances, and they carry the heaviest traffic.
- **Jobs.** `reporting-job` and its export step. They run on a schedule and finish. A job usually needs less than a service, and for a shorter time.
- **Tools.** `admin-console` is the clearest case. It is a service in how it connects, but it acts for a person, so two principals are involved in every action: the tool and the human behind it.

The external accounting vendor never connects to any Ledgerly database; it receives files, and Chapter 13 treats that path.

Identity, login and role are three different things

The book keeps three words apart, because mixing them is the source of most confusion in access design.

An **identity** is the verified claim that a principal is who it says it is. A **login** is the database-level account a principal connects with. A **role** is a named bundle of privileges. A principal has an identity, connects through a login, and the login is a member of one or more roles. The login is what appears in the database log. The role is what decides what the login may do.

This gives the model one sentence worth remembering: nobody logs in as a role. A role exists to be granted to logins and to hold privileges, and it is never a person. If the log shows a role name where a person or workload instance should be, the model has been bypassed somewhere.

For humans, the access gateway (Chapter 7) creates a login per session that carries the person's identity, and the login receives `human_readonly` by default. For services, each instance connects with a login that is a member of its `app_*` role. In the target state that login is dynamic and short-

lived (Chapter 4). In the starting state it is the shared admin login. The consequence for the central question is plain: the log names a login, and a login maps to one person or one workload instance, so "who ran that statement?" has an answer.

Why a shared account destroys evidence

Chapter 1 made the case in full; the short form is that a shared account cannot attribute a statement, cannot be revoked for one person, and cannot be scoped below what its broadest caller needs.

One path needs care: `app_admin_console` is a workload, but its users are people. The console passes the human principal's identity with each action, and the sketch below shows how.

```
function consoleAction(person, action):
    correlationId = newCorrelationId()
    outcome = "failed"
    // set on the session the console borrowed, cleared when it is
    returned
    setSessionContext({ actor: person.id, correlation: correlationId
})
    try:
        run(action)
        outcome = "done"
    finally:
        clearSessionContext()
        // a failed or half-finished action is recorded too
        auditLogAppend({ actor: person.id, correlation: correlationId,
                        workload: "admin-console", action:
action.name,
                        result: outcome })
```

The database login names the workload, and the session context and `audit_events` carry the person and the correlation id, so the application log, the audit trail and the statement's own record share one key. Chapter 14 covers how the records are joined. The context is set by the console, so it is a claim made by the workload and not a fact the database verified; the pooler caveat in the row-rule section applies to it as well.

A role hierarchy with four layers

I would build the role schema in four layers, from the one nobody uses outward to the ones people use. Each layer has one job.

Layer one: the owner

`ledgerly_owner` owns every schema object: schemas, tables, views and sequences. Nobody logs in with it. It exists so that ownership is attached to neither a person nor a service.

```
CREATE ROLE ledgerly_owner;  
// no login is ever created for this role; objects are created  
// under it only by a migration that assumes it  
  
CREATE ROLE ledgerly_migrator;  
GRANT ledgerly_owner TO ledgerly_migrator;  
// membership grants the owner's privileges, not ownership: objects  
// belong to the active role, so the migration always assumes  
// ledgerly_owner explicitly, on every engine
```

Layer two: the migrator

`ledgerly_migrator` runs schema changes. The deployment pipeline uses its login, humans do not, and it holds no data privileges of its own. It is a member of `ledgerly_owner`. On some engines that membership passes on the owner's privileges without any explicit step, but not ownership: a new object belongs to whichever role is active when it is created, so the migration always assumes `ledgerly_owner` explicitly.

The migrator holds standing owner privileges, exercised through a short-lived credential. The privileges never expire; only each credential does. Three things bound it: the credential lives only in the pipeline, it is short-lived (Chapters 4 and 11), and pipeline procedure is the only sanctioned way to use it. Schema changes are possible without any person or service holding ownership, and if those bounds hold, every use of the owner corresponds to a pipeline run. Chapter 11 covers how migrations are reviewed.

Backfills that belong to a schema change run inside the migration as `ledgerly_migrator`. Any other bulk change to data is a data fix and follows Chapter 10. I draw that line because the migrator's power is large, and it should not become the route for every awkward change.

Layer three: application roles

Each workload gets its own role. One role per service costs five definitions to keep in step with five code bases. In exchange, the blast radius of each credential is the size of that service's job: if the one for `invoice-worker` leaks, it exposes what the worker could do, and the worker never needed `payment_tokens`.

The roles are `app_billing_api`, `app_invoice_worker`, `app_payments_service`, `app_reporting_job` and `app_admin_console`. The prefix says who uses it: `app_` for workloads, `human_` for people. Roles carry no person names and no environment name.

Layer four: human roles

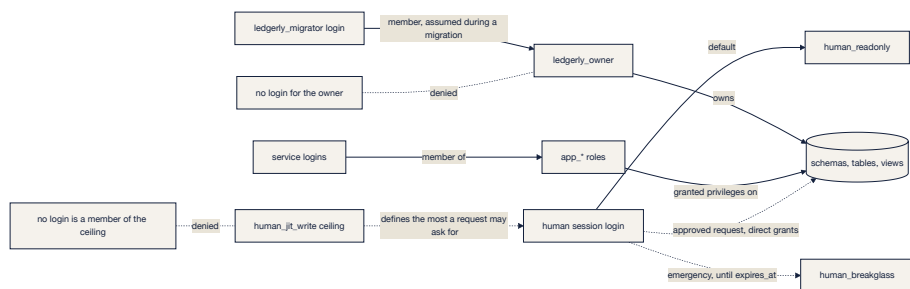
People get three roles, and they differ in how they are held.

- `human_readonly` is the default. It reads for investigation, withholds Restricted columns and masks Confidential columns where Chapter 12 says so.
- `human_jit_write` is a ceiling. It is never granted to a login. Each approved request grants a subset of it, on named tables and named columns, directly to the session login, and then takes it away.
- `human_breakglass` is broad emergency access with heavy logging (Chapter 8 defines it) and mandatory review afterward.

The ceiling is the least intuitive of the three, and I return to it after the grant mechanics.

The diagram

The diagram below shows who is a member of what, and who owns what. Look at the two things that are not connected: no login exists for `ledgerly_owner`, and no login is a member of `human_jit_write`.



Grants: schema, table and column

A **grant** is the act, and the resulting record, of giving a privilege to a role, giving a role to a login or to another role, or, for JIT access only, giving a privilege directly to a session login. A **privilege** is the specific thing a grant allows, such as `SELECT` on one table. The three meanings keep the vocabulary precise, and in this chapter I will say which one I mean.

Start from nothing

The starting point for any role is no privileges at all. Add grants until the workload works, and stop. Engines differ in what a new login can do by default: on some, any login may see object names or create objects in a default schema. List what every login receives automatically, and remove what a role does not need.

Privileges apply at three levels, and they add up: a role holds the union of everything granted to it at any level, and a narrow grant never takes away a broad one.

- **Schema level.** Whether a role may see the objects in a schema at all. On engines that require it, table-level grants do nothing without it, which makes schema access a useful coarse switch.
- **Table level.** `SELECT`, `INSERT`, `UPDATE` and `DELETE` on a whole table.
- **Column level.** The same privileges on named columns only.

Column-level grants matter because sensitive data is unevenly spread: `customers` holds Confidential contact details, `bank_accounts` holds Restricted `iban` and `account_holder_name`, and `payment_tokens` holds Restricted `token_value`. A table-level grant says yes to every column of

the table. Because privileges add up, restricting columns means granting only at column level and holding no table-level grant on that table for the role, directly or through another role it belongs to.

Application grants

The first two application roles, at table level:

```
GRANT SELECT, INSERT, UPDATE ON customers, invoices, invoice_lines TO
app_billing_api;
GRANT SELECT ON payments, ledger_entries TO app_billing_api;
GRANT INSERT ON audit_events TO app_billing_api;

GRANT SELECT, INSERT, UPDATE ON invoices, invoice_lines TO
app_invoice_worker;
GRANT SELECT ON customers TO app_invoice_worker;
GRANT INSERT ON audit_events TO app_invoice_worker;

// not granted to either role: bank_accounts, payment_tokens,
// UPDATE or DELETE on ledger_entries and audit_events
```

`app_billing_api` reads `ledger_entries` and `payments` but cannot write them, because only `payments-service` writes `payments`, `bank_accounts` and `payment_tokens` and inserts `ledger_entries`. Neither role holds `DELETE` on anything; if a service needs it, the grant is added for that table with a reason, and the review notices.

Column-level grants for the other roles

Roles with a dagger in the matrix are column-restricted. The column lists below are illustrative: which Confidential columns appear, and in what masked form, is Chapter 12's decision.

```
// illustrative column lists
GRANT SELECT (id, created_at, status) ON customers TO
app_reporting_job;
GRANT SELECT (id, customer_id, amount_minor, currency, status)
    ON payments TO app_reporting_job;
GRANT SELECT ON invoices, ledger_entries TO app_reporting_job;

GRANT SELECT (id, created_at, status) ON customers TO
app_admin_console;
GRANT SELECT, UPDATE (status) ON invoices TO app_admin_console;
// UPDATE (status) is a column-level grant: the tool can change an
invoice's
// status and no other column of the row
```

The last grant is the one I like most. `app_admin_console` may change an invoice's `status` and nothing else in the row, so the grant states the intent and the database enforces it even if the tool's code changes later. `app_reporting_job` holds no write privilege on any main table: it writes its results to `ledgerly_analytics`.

Note: On engines that replicate physically, roles are defined once for the whole cluster, so where each role may connect is a connection-layer rule (Chapters 5 and 7) and not something the grants express. Chapter 6 gives the detail.

The matrix as the reference

The table below is the target-state reference for the six roles that read or write the main tables. S is `SELECT`, I is `INSERT`, U is `UPDATE`, D is `DELETE`, a dash means no access, and a dagger (†) means column-restricted: Restricted columns are withheld, and Confidential columns are masked where Chapter 12 says so.

Table	app_ billing_ api	app_ invoice_ worker	app_ payments_ service	app_ reporting_ job	app_admin_ console	human_ readonly
customers	S, I, U	S	S	S†	S†	S†
bank_ accounts	—	—	S, I, U	—	—	—
payment_ tokens	—	—	S, I, U, D	—	—	S†
payments	S	—	S, I, U	S†	S†	S†
invoices	S, I, U	S, I, U	S	S	S, U (status only)	S
invoice_ lines	S, I, U	S, I, U	—	S†	S†	S†
ledger_ entries	S	—	S, I	S	—	S
audit_ events	I	I	I	—	I	S

I read the matrix by column. Each column is a workload's job description written as privileges, and a reviewer can ask whether it is still true. Across it, `payments-service` alone reaches `bank_accounts`, only `payments-`

service touches `payment_tokens`, apart from a column-restricted read by `human_readonly`, and no human role writes by default.

Append-only is structural

No application role, no human role and no JIT grant holds `UPDATE` or `DELETE` on `ledger_entries` or `audit_events`, and that includes `human_breakglass`. A correction is a new entry, never an edit. Ledger corrections are issued only through the correction path of `payments-service`, never by a human login.

There is one structural exception: `ledgerly_owner`, and so a running migration, can alter these tables, because the owner can alter or drop the table and grant itself any privilege on it. That is why Chapter 11 reviews every migration that touches them. The control sits in the review, not in the privilege.

Defaults, ownership and the superuser

Default privileges

Many engines can make a grant apply to tables created later, whether as a default privilege attached to an owner or as a schema-wide grant. That is convenient at first and a trap later, because a new table silently inherits whatever the grant says.

My preference is an empty default. A new table is readable by nobody but the owner, and the migration that creates it states each grant in the same file. The cost is a few more lines per migration; the benefit is that one review shows the whole access effect of a change.

Warning: A default privilege that grants `SELECT` on future tables to a broad role such as `human_readonly` exposes the next sensitive table the moment it is created. Review such defaults first, and remove any that cover future tables.

The notation is illustrative, because default privileges have no neutral syntax.

```
// illustrative notation: defaults for objects created by
ledgerly_owner
DEFAULT PRIVILEGES FOR ledgerly_owner ON TABLES: none

// each migration that creates a table states its own grants
CREATE TABLE <new_table> (...);
GRANT SELECT ON <new_table> TO app_billing_api;
```

The danger of ownership

In common engines the owner can drop an object, change its definition, grant privileges to anyone and, in many cases, ignore row-level restrictions that apply to others. Whoever holds ownership can change the access model itself.

So the rules around `ledgerly_owner` are strict, and the strictness is cheap. No login exists for it. Application roles never own tables, even ones they create for their own use, because "the service creates its table at start-up" is how a workload ends up owning part of the schema. The owner is the least used role in the system, which is the correct state.

The superuser

Most engines have a built-in administrative role that bypasses privilege checks, though managed services may withhold it. I call it the superuser here; it is outside the Ledgerly role schema because it comes with the engine.

I would treat it as a sealed instrument. No application, job or human session uses it in daily work. Its credential is held where the platform team can retrieve it and the retrieval leaves a record, and it is for engine-level operations no role can do, such as repairing replication. The honest limit is that it can undo every control in this chapter, and I do not know of a design that removes that power without making recovery impossible. Make its use rare, visible and reviewed; Chapter 15 returns to it. Routine administration, such as applying role definitions or issuing logins, uses `ledgerly_access_admin` instead. If the superuser is used for routine tasks, some role is missing a privilege it should have.

`ledgerly_access_admin` is one role used by three principals, each through its own login: the pipeline step that applies the access definition, the credential broker (Chapter 4) and the access gateway (Chapter 7). It

creates logins and manages memberships, owns no tables and holds no data privileges, and that is still wide power: whoever holds it can grant anything inside the roles it administers. Separate logins keep the three apart in the log, though not in what they may do.

What bounds the control plane's own power is therefore not the role but the surroundings. Each of the three logins is reachable only from its own component. Every action goes to the audit log (Chapter 14), and the drift check below compares the result with the code. Changes to the access definition go through their own small set of reviewers. These limits make misuse visible; they do not make it impossible, and I would say so in the access review.

Narrowing further: views, row rules and tenants

Views

A **view** is a stored query that behaves like a table. A role granted access to the view reads what the query exposes without holding a privilege on the underlying table.

```
// illustrative: a narrowed projection owned by ledgerly_owner
CREATE VIEW customer_support_summary AS
  SELECT id, created_at, status FROM customers;
GRANT SELECT ON customer_support_summary TO app_admin_console;
```

Note: In many engines a view runs with the privileges of its owner, not of the caller. That is what makes the narrowing work, and it is also why a view that selects too much leaks it to every role granted on the view. Some engines also let you choose the other behavior. Check what yours does by default, and test the view with a role that holds no privileges on the base table.

I prefer column-level grants where the engine supports them, because the grant lives next to the table. A view is better when the shape is the point: a join, a computed value, or a stable interface over a changing table. The book does not use a per-request view for JIT access, for two reasons: a view must be owned, so the gateway that creates it would need owner power, and a view made for each request breaks the ownership model.

Row rules and tenants: a decision rule

A row rule narrows rows inside a cell of the matrix and does not change what the cell says. Support looks at one customer at a time, so a rule such as "app_admin_console sees only invoices rows where customer_id equals the case's customer" fits where a column filter does not. Engines disagree on syntax, so the book shows no notation.

Ledgerly's customers are businesses, and each expects not to see another's invoices. That isolation is mostly the application's job: `billing-api` filters by customer on every query, and no role separates one tenant from another, because the application role must read all of them to serve any.

Use a row rule, for support or as a second tenant layer, only when two conditions hold: a wrong row is severe, and the application sets the context value reliably for every session. The cost is that every statement carries the predicate and something must set the context per session. Behind a pooler, a value set on one session can reach the next borrower of the same connection (Chapter 5 returns to that). When the context is missing, the rule must return no rows, and the first test is that case. A rule that fails open is worse than none. Separate databases are a stronger boundary at a much higher operational price; which fits depends on how many tenants you have and what you promised them.

The ceiling for just-in-time access

`human_jit_write` defines the ceiling: the most a just-in-time (JIT) request may ask for, expressed as privileges on named tables and columns. JIT access is access granted for a specific task and a limited time; Chapter 8 covers approvals and lifetimes, and this chapter only defines the ceiling.

The role is never granted to a login. A request is a list of (privilege, table, columns) triples that fits inside the ceiling, stored with the request. On approval, exactly those triples are granted directly to the requester's session login, and on expiry exactly those triples are revoked. A shared role cannot be scoped per request, so the per-request scope lives in the grants and the role draws the outer edge.

```

// the ceiling: what a JIT request may ask for, never granted to a
login
GRANT SELECT ON customers, payments, invoices, invoice_lines
    TO human_jit_write;
// id and created_at are assigned by the database, never changed or
supplied
// by a data fix, so the write privileges below name every other
column
GRANT UPDATE (<every column except id, created_at>)
    ON customers, payments, invoices, invoice_lines TO
human_jit_write;
GRANT INSERT (<every column except id, created_at>)
    ON customers, payments, invoices, invoice_lines TO
human_jit_write;
// DELETE only where a request names it, and never on a table with
// Restricted columns
GRANT DELETE ON customers, payments, invoices, invoice_lines
    TO human_jit_write;

// illustrative column lists; Restricted columns are not named
anywhere
GRANT SELECT (customer_id), UPDATE (customer_id)
    ON bank_accounts TO human_jit_write;
GRANT SELECT (customer_id, provider, last_four, expires_on),
    UPDATE (customer_id, provider, last_four, expires_on)
    ON payment_tokens TO human_jit_write;
// no table-level SELECT on either table: it would include the
Restricted columns
// no INSERT or DELETE on either: a new or removed row sets or
destroys
// Restricted values

```

Read the block for what it leaves out. `bank_accounts` and `payment_tokens` are open only for the few named non-Restricted columns, and never for row identifiers or timestamps: `iban`, `account_holder_name` and `token_value` can never appear in a JIT request. INSERT is limited to the first four tables, and it cannot name `id` or `created_at`, so a restored or new row always receives database-assigned values. The two append-only tables and the access request records, which belong to the gateway, are outside the ceiling entirely. Every triple except a `DELETE` names its columns.

Two costs are stated once, here. Columns that link a row to its owner, such as `customer_id`, are not Restricted, so a JIT `UPDATE` can change them. Such an owner-link change stays inside the ceiling, but it is a dangerous-operation class: the execution policy gates it, and it is routed to the product path (Chapter 10) and not run as raw SQL. The gate and the product

path are the control, not four-eyes approval alone. And a JIT `SELECT` shows any Confidential column it names unmasked, which is part of what the approver accepts.

What keeps the ceiling honest is a check, not trust. The gateway refuses any request outside it (Chapter 8 shows the check), and the drift test below asserts that no login is a member of `human_jit_write`, the failure that would turn a ceiling into a standing grant.

Where the engine has no column-level grants

Not every engine supports column-level grants. The book uses exactly one fallback: where the engine cannot grant on columns, the Restricted columns move into separate tables, `bank_account_secrets` and `payment_token_values`.

```
table bank_account_secrets:      // owned by ledgerly_owner
  bank_account_id    text primary key
  iban               text not null
  account_holder_name text not null

table payment_token_values:      // owned by ledgerly_owner
  payment_token_id    text primary key
  token_value         text not null

// reachable by app_payments_service and human_breakglass only;
// outside the ceiling, never part of a JIT request
```

With the fallback, the main tables hold no Restricted column, so a table-level JIT grant on them cannot reach Restricted data. The cost: the request detail drops to table level, and the approver has approved every non-Restricted column of the table, Confidential ones included, not only the ones the requester named.

Break-glass and the operations that sit above the schema

`human_breakglass` is the only human role that reads Restricted columns. It reads every table and writes `INSERT` and `UPDATE` on every table except the two append-only ones; on `bank_accounts` and `payment_tokens` it

writes only `UPDATE`, and only on non-Restricted columns. It is for emergencies, not routine fixes, and it is absent from the matrix because only a session holds it, for a short window; no standing workload does.

Restricted columns are changed only through an application path. A human never writes them with raw SQL, in a JIT session or otherwise: `admin-console` asks `payments-service`, the only writer, after a second person approves, and the change is recorded in `audit_events`. Chapter 10 describes that path. These facts mark the edge of what grants will ever do. A grant says which tables and columns a session may touch; it cannot say which statement, how many rows, or whether this particular change is the one that was approved. That is the job of the Execution part, and Chapter 9 covers how the gateway checks each statement. Knowing where grants stop helps you decide where a new capability belongs.

Roles as code, and drift

A role schema that exists only in the database is one nobody can review. The grants above are short enough to read, and they should live in version control, change through review and be applied by the pipeline, like the schema they protect.

Where the definitions live

I would keep two kinds of file. Migrations create tables and state each table's grants. A separate, reviewed access definition lists the roles, the memberships between roles and the ceiling. The pipeline applies both as separate steps. The access definition is applied through the pipeline step's own login in `ledgerly_access_admin`, the same role the broker and the gateway use through their own logins; it holds no data privileges. Creating roles is a stronger power than changing tables, so its step should have its own small set of reviewers. The reviewer's question is not whether the SQL is valid but whether this grant is the narrowest that makes the change work. A reviewer with the matrix open can answer that in a minute.

Narrowing an existing system

Ledgerly starts with one shared login that owns everything. I would not move from there to the matrix in one change, because a grant that is too narrow breaks production and the team cannot know what it missed until

it fails. A path with four steps:

1. **Give each workload its own login first, with the same privileges as before.** Nothing breaks, and the log immediately starts naming the workload.
2. **Observe.** Use the statements recorded under each login to list what the workload actually touches, over a period long enough to include the slow monthly job. A statement log can hold literal values, so treat it as sensitive: it may contain customer data or Restricted values that were pasted into a query. Restrict who reads it, set its retention, and prefer logging statement shapes with parameters where the engine offers it.
3. **Define the narrow role and move the login into it,** first in a staging environment that runs the same jobs, then in production with the old privileges kept at hand for a short while.
4. **Remove the old privileges** and watch for denied statements.

The cost is calendar time, not effort. Teams tend to skip the second step, and the predictable result is a rarely used path that breaks months later.

Drift

Drift is the gap between the access the code declares and the access the database holds. It appears when a problem is fixed at night with a hand-written `GRANT`: the grant works, the incident ends and nobody writes it down. The check compares declared with actual on a schedule, and it must fail loudly when it cannot read one side.

```

function checkRoleDrift():
    declared = loadDeclaredAccess()
    try:
        actual = readActualAccess()
        requests = loadAccessRequests()
        known = loadGatewayLogins()
    on ReadFailed:
        // an unreadable state is not a clean state
        notify("platform", "role drift check could not read a source")
        raise DriftCheckFailed

    for each extra in difference(actual.roleGrants,
declared.roleGrants):
        notify("platform", "grant not in code: " + extra)
    for each login in actual.loginsOwningObjects():
        notify("platform", "login owns objects: " + login)
    for each login in actual.membersOf("human_jit_write"):
        notify("platform", "login is a member of the ceiling: " +
login)
    for each login in actual.allLogins():
        if login is human and login not in known:
            notify("platform", "login unknown to the gateway: " +
login)
            // memberships must match what the login was issued for
            for each role in difference(actual.rolesOf(login),
expectedRoles(login)):
                notify("platform", "unexpected membership: " + role + "
for " + login)
            // a direct grant is legitimate only as part of a live request
            live = unexpiredTriples(requests, login)    // empty for
service logins
            for each g in difference(actual.directGrantsOf(login), live):
                notify("platform", "direct grant without a live request: "
+ g + " to " + login)
            for each login in actual.membersOf("human_breakglass"):
                if not hasOpenBreakGlass(requests, login):
                    notify("platform", "break-glass member without a request:
" + login)

```

`expectedRoles(login)` is `human_readonly` for a gateway session login, `human_breakglass` as well while a break-glass request is open, and the one `app_*` role the broker issued the login for. The check therefore catches a hand-written `GRANT app_payments_service TO <session login>` or `GRANT ledgerly_owner TO <login>` as well as a direct privilege grant.

Here `roleGrants` covers only role-to-role memberships and privilege-to-role grants, which the code declares. Logins are not in code: they come and go with sessions. Human logins are checked against the gateway's records. The direct grants of every login, service logins included, are com-

pared with the `access_request_privilege` triples that have not expired. A service login has no requests, so any direct grant on one is flagged; that is how a hand-written `GRANT` of a privilege to a service login at night gets noticed. A grant that outlived its request, or a break-glass member with no open request, is flagged even when the revoke job reports success. JIT access works by direct grants and expiry by a revoke job; if the job stops, the grants remain, and this check notices independently of the job (Chapter 8 covers the pairing). The check reads grants and memberships only. It does not see what a session does with them, which is the Execution part again.

The check repairs nothing, because an automatic revoke of unknown grants in production is a new way to break production at night. Nor does it say whether the declared state is right: a wrong grant, correctly declared, passes. That judgment belongs to the access review of Chapter 14.

Trade-offs and costs

Every control in this chapter has a price, and the price is paid by specific people.

Per-workload roles cost maintenance. Five roles mean five definitions to keep in step with five code bases. When a service needs a new table, someone updates the role in the same change; if the grant lands after the code, the deploy fails. The mitigation is to make the grant part of the migration. With one service and two engineers this overhead is not nothing, and a smaller schema of an owner, one application role and one read-only role is a reasonable first step.

Narrow grants will cause denied statements. Some denials are mistakes in the grants and some are code that reached further than it should have. Whoever is on call pays for both when a denial appears during an incident. Running staging with the same roles as production reduces the surprise; ignoring the cost leads teams to widen a grant in a hurry, and the hurry is how drift starts.

Column-level grants add review load. Each column list must be maintained when the table changes. A new sensitive column is exposed by default under a table-level grant, so for sensitive tables I grant on named columns and accept the extra lines. Where the engine lacks the feature, the fallback costs joins, extra tables to migrate and a coarser JIT approval.

Views and row rules add moving parts. They are worth their weight when the shape or the isolation is the point, and a poor trade when they are added to feel careful. A row rule whose context can go missing is security theater: it adds friction for every query and fails open in the case it was written for.

Ownership discipline slows some people down. Engineers used to creating a table in a console must now go through a migration. The friction is intended, and it is worth less on a throwaway development database than on production. Apply the model to production first, and relax it elsewhere only when you can say what you give up.

When not to use this much structure. A prototype with one developer and no customer data does not need the full role schema. The owner, application and read-only split is enough until the second service or the first sensitive column. Grow the schema when "who ran that?" cannot be answered from the log, or when one workload's credential could do more damage than its job justifies.

Common mistakes

- **Treating the application role as the owner.** It is tempting because migrations then run at startup with one fewer step. The service then holds the power to drop and re-grant. Use a separate migrator and an owner with no login.
- **Granting to match last year's queries.** A role copied from the broadest existing login keeps every privilege that was ever convenient. Derive the role from what the workload does now, observed under its own login.
- **Adding a role per person.** Roles are bundles of privileges, and logins are per person. A role named for an individual turns the schema into a staff list. Keep roles about jobs.
- **Using one role for several services because they are "similar".** The services drift apart and the role keeps the union of what each needs. A role per service costs a few lines.
- **Leaving defaults to future tables.** A default privilege saves a line in each migration and exposes the next sensitive table to a broad role. Keep the default empty and grant explicitly.

- **Granting the ceiling role to a login for one busy week.** The ceiling then becomes a standing grant with a friendly name. Keep the ceiling unattached: only the request check reads it, and the drift test flags any membership.
- **Hiding a sensitive column behind a view and leaving the table readable.** The view narrows nothing for a role that can reach the base table. Revoke on the table and test with a role that holds no table privilege.
- **Trusting a declared schema nobody compares.** Roles in version control do not stop the 2 a.m. `GRANT`. Compare declared and actual on a schedule, and make the comparison fail loudly when it cannot read.
- **Using the superuser for routine work.** It works every time, which is why it spreads. When a routine task needs it, add the missing privilege to the right role.

Key takeaways

- A shared login destroys evidence and scope together: the log cannot say who acted, and the login must be allowed to do whatever its broadest caller does.
- Keep identity, login and role apart: a principal has an identity, connects through a login, and the login is a member of roles; nobody logs in as a role.
- `ledgerly_owner` owns everything and has no login, `ledgerly_migrator` assumes it only during a migration, and each workload gets its own `app_*` role sized to its job.
- Start every role from no privileges, use column-level grants where sensitive columns sit beside ordinary ones, keep default privileges empty, and fall back to separate secrets tables only where the engine cannot grant on columns.
- `human_jit_write` is a ceiling that is never granted to a login: each approved request grants a subset directly to the session login, and no request can include a Restricted column.
- Roles and grants belong in version control and reviewed pipeline steps, and a scheduled drift check that fails loudly when it cannot read the database keeps the declared and actual states together.

- Every layer costs someone something: maintenance for roles, denied statements for narrow grants, review load for column lists, and a coarser approval where the engine lacks column-level grants.

Roles say what a principal may do once it is connected, but they say nothing yet about how a service proves who it is. Chapter 4 takes the next step: how credentials reach workloads, why short-lived and dynamic credentials change the shape of the problem, and what replaces the password in the configuration file.

PART II

Application Access

Chapter 4 — Credentials and Secrets

Imagine a Friday afternoon at Ledgerly. Someone on the backend team has moved to another group, and the security contact asks, politely, whether the shared admin password can be changed. Alex, from the platform team, agrees it should be. Then Alex starts listing where the password lives: the configuration of all five services, the pipeline's variables, the export step of `reporting-job`, a team password manager, and the saved connections in several desktop clients.

Changing it in the database takes a second. Changing it everywhere, at the same moment, means a coordinated restart of every service during business hours, and Robin, who is on call this weekend, asks the fair question: what happens to the invoices in flight? The change is postponed to "a quiet window" that never arrives. The password outlives the person, and then the next person.

This chapter answers the question that Friday raised: how does a service get a credential, and how do you change one without an outage? It adds the next target-state step from the running example. Credentials for services become short-lived or dynamic, and they reach workloads by identity instead of through a shared file.

Credentials belong to the Identity part of the five-part model, and they are necessary but not the answer. The book's thesis is that "Production database access is not a credential-distribution problem; it is an authorization, execution, and accountability problem." Better credential handling shrinks who can obtain a login. It says nothing about what a login may do once it is in, or what it did, and those are the jobs of the chapters on grants, execution and records.

The problem, and why it is hard

A **credential** is what a principal presents to authenticate; a **secret** is any value that grants access if disclosed. A password is both. Chapter 3 gave every service its own role, so that a login says what it may do. This chapter is about the other half: how the login proves who it is, and how long that proof remains good.

A password has a life cycle even when nobody designs one. It is created, copied to wherever a service can read it, used for months, and eventually changed or forgotten. Each copy is a place where a person or a process can read it, and the question "who can see this password?" has as many answers as there are copies. In the starting state the answer is "roughly everyone on the backend team, and anything that can read a service's configuration."

The obvious remedy is a better password: longer, random, stored somewhere tidier. That helps with guessing, and guessing was never the main risk. The risk is the number of places the value sits, and the fact that none of them can be emptied without touching the others. A strong password in five configuration files is still five copies.

The second difficulty is **rotation**, which means replacing a credential with a new one on a schedule or on suspicion. Rotation is hard because a credential is shared state between two systems: the database knows the value, and so does every consumer. Changing one side first breaks the other. A team that cannot change a password without an outage will, sensibly, stop changing it, and a rotation policy that nobody can execute adds a calendar entry and removes no exposure.

What credentials cannot do

Take a login from the reporting job, valid for one hour, leaked into a chat channel. It is a good credential by every measure in this chapter: it is unique, short-lived and fetched by identity. For its hour it can still read the three columns of `customers` the role is granted (`id`, `created_at` and `status`), on every row, because that is what `app_reporting_job` is allowed to read. Short lifetimes shrink the window; they do not shrink the role. Three things bound a leak, and none of them is the credential itself: the **role** it carries (Chapter 3), the **network reach** that decides where it

can be used from (Chapter 7), and the **log** that records what it did (Chapter 14). A login that can be presented only from the pooler's network, with a role that cannot see Restricted columns, is a smaller incident than any password policy can make it.

The same holds for the strongest design below. A broker, a lease and workload identity all answer who may obtain a login. They do not decide what the login may do, and they do not stop a permitted login from running a harmful statement.

This is an engineering question, not a policy one. A rule that says "rotate every 90 days" has nothing to say about how the five services learn the new value at the same instant. The answer lies in how credentials are issued, delivered and replaced, which is a design the team has to build.

The life of a credential

It helps to follow a credential from birth to retirement and ask, at each stop, who can read it and what would reveal it. The usual places are these.

Place	Who can read it	How it tends to leak	Better home
Configuration file in the repository	Everyone with repository access, forever, through history	Copied into a fork, a chat, a ticket	A secret manager, referenced by name
Environment variable	The process, its children, and anyone who can inspect the process or its crash dumps	Printed by a debug endpoint or an error report	A file or memory handed over at start, with narrow file permissions
Container image or build artifact	Everyone who can pull the image	Baked in at build time, so every copy carries it	Nothing in the image; fetched at start
CI/CD variable and job log	Pipeline administrators; sometimes every job	Echoed by a script, printed by a failing command	Short-lived value fetched by the job's own identity
Team password manager	Everyone in the shared vault	Shared by habit; no record of who read it	Individual access with a record, for the few that remain

Place	Who can read it	How it tends to leak	Better home
Desktop client's saved connection	Whoever uses that laptop	Lost or backed-up laptop	No saved credential; session credentials from a gateway (Chapter 7)

Two things stand out in the table. First, the leaks are almost never clever. They are copies made for convenience: a value pasted into a ticket so a colleague can reproduce a problem, a variable echoed while debugging a deployment. Second, the "better home" column is mostly the same answer: the value lives in one managed place, and consumers fetch it when they need it.

Warning: A secret that has appeared in an image layer, a repository history or a job log has to be treated as disclosed, even if nobody has read it. Deleting the line does not delete the copies. The remedy is to replace the credential, which is why a team needs a rotation path that works before the day it is needed.

A secret has one more property worth naming: how long it stays valid. Say a password is valid for three years: it carries three years of exposure from every copy ever made. An old copy of a credential that lasts an hour is worth nothing. The durations are illustrative. That is why lifetime matters more than strength.

Secret managers and how a service reads a secret

A **secret manager** is a service that stores secrets, controls who can read each one, and records every read. HashiCorp Vault and the secret stores of the large cloud providers are examples. The pattern matters more than the product: one place holds the value, access is decided per principal, and a read leaves a record.

Moving the password into a secret manager does not by itself reduce the number of readers. It changes who they are and gives each read a name: a specific service identity reads a specific path, and the secret manager says when. That is a real gain for investigation and revocation, and a modest gain for exposure.

The first secret

A service needs something to prove itself to the secret manager before it can read anything. If that something is another stored password, the problem has moved one level up and nothing is solved. This is the bootstrap question, sometimes called the first-secret problem.

The answer is **workload identity**: an identity the platform gives to a running workload, such as a service instance or a job, so that it can authenticate without a stored password. The platform that starts the workload vouches for it, and the secret manager trusts that statement. The service never holds a long-lived value of its own for this step. I treat workload identity as the thing that makes the rest of the design possible, because it ends the regress.

Reading at start, and reading again

A service has two reasonable moments to read its database credential: once at start, or whenever it opens a connection. Reading once is simple, and it works with static credentials that rarely change. It fails when the credential does change, because the running process keeps the old value until restart. Reading at connection time, through a client that asks the secret manager again after an authentication failure, copes with rotation without a restart.

```
function connectWithCredential(service):
    attempts = 0
    while attempts < maxAttempts:
        try:
            slot = readSecret(path: "db/" + service + "/active")
            credential = readSecret(path: "db/" + service + "/" +
slot)
                return openConnection(host: "db-pooler.internal.example",
                                       login: credential.login,
                                       password: credential.password)
        on AuthenticationFailed:
            // the value may have rotated since the last read; read
again
            attempts = attempts + 1
        on SecretStoreUnavailable:
            // fail the start; never fall back to a value baked into
the image
            raise StartRefused
    raise StartRefused
```


The block shows the two failures that matter. An authentication failure triggers a fresh read, so a rotation does not become an outage. An unreachable secret manager refuses the start, because falling back to a spare copy recreates the file this design removed. That refusal has a price, and the Trade-offs section returns to it.

Note: The secret manager itself is a door. Who may read the path `db/billing-api`, who may change it, and whether reads are logged are the same questions as for the database. Granting every service read access to every path would rebuild the shared password with an extra step.

Rotation without an outage

The cure for "change one side first and the other breaks" is to never have a moment where only one value works. This is the **two-credential pattern**, often called A/B rotation. Each service has two logins, A and B, both members of the same `app_*` role. At any time one is active and the other is idle.

To rotate, you change the idle login, publish the new value, tell consumers to switch, wait until the old login is no longer in use, and then invalidate it. At no point does a consumer hold a value that the database has stopped accepting. In pseudocode, with the failure points marked:

```

function rotateServiceCredential(service):
    active = readSecret(path: "db/" + service + "/active")    // "a" or
    "b"
    idle = otherSlot(active)
    if openSessions(loginFor(service, idle)) > 0:
        // an earlier rotation did not finish; changing this login
cuts live sessions
        raise RotationIncomplete
    newPassword = generatePassword()
    // the idle login has no users, so changing it breaks nothing
    setLoginPassword(login: loginFor(service, idle), password:
newPassword)
    try:
        writeSecret(path: "db/" + service + "/" + idle,
                    value: { login: loginFor(service, idle), password:
newPassword })
        writeSecret(path: "db/" + service + "/active", value: idle)
    on SecretStoreUnavailable:
        // the active login is untouched; the idle one holds a value
nobody can read
        notify("platform", "rotation stopped before publish: " +
service)
        raise RotationAborted
    // consumers re-read on their next connection; wait for the old
login to drain
    waitUntil(openSessions(loginFor(service, active)) == 0, deadline:
drainDeadline)
    if openSessions(loginFor(service, active)) > 0:
        // the old value still works; the rotation is not finished
        notify("platform", "old login still in use after rotation: " +
service)
        raise RotationIncomplete
    setLoginPassword(login: loginFor(service, active), password:
generatePassword())
    auditLogAppend({ action: "rotate", subject: service, actor:
"rotation-job" })

```

Three details carry the design. First, the database and the secret manager are separate systems with no shared transaction, so the order of the steps is what limits a failure: the value that is live is never changed before its replacement is readable. A failure at any step leaves the previous credential working. Second, the drain check. Rotation is finished only when nothing uses the old login, and a service that never re-reads its secret is found at that moment, not in an outage three weeks later. Third, the last step overwrites the old login with a value nobody keeps, but it runs only after the drain. While any session still holds the old login, the old value works.

The cost is that every service now has two logins and a job that knows both, and the consumer must re-read the secret after a failure. A service that reads its credential once at start needs a rolling restart to rotate, and that restart is where an A/B rotation turns into a calendar event again. Rotating the shared admin password in the starting state is hard for this reason: there is no second login, and no consumer can re-read anything.

I would rotate on a schedule the team can sustain rather than the shortest one it can write down. A suspected leak is a different path, described in "When a credential leaks".

Dynamic and short-lived credentials

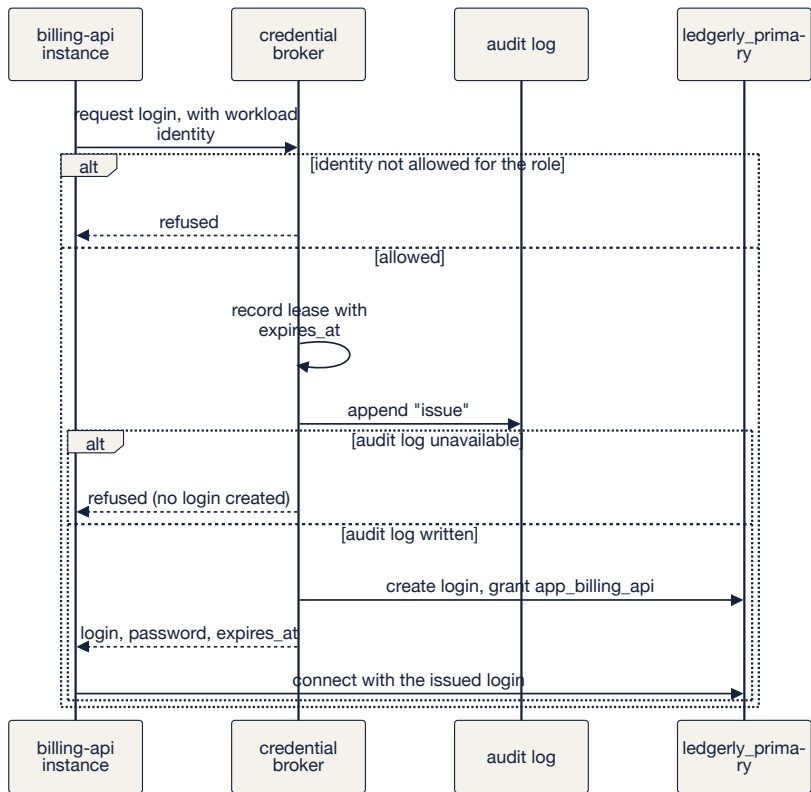
A rotated static credential still has a long lifetime between rotations. The next step is to make the lifetime short enough that rotation stops being an event. A **short-lived credential** expires on its own within a short, defined time. A **dynamic credential** is a short-lived credential generated on request for one principal, so two requests never get the same value. All dynamic credentials are short-lived; not every short-lived credential is dynamic.

The usual mechanism is a **credential broker**: a service that, on request, creates a login in the database, grants it membership in the right `app_*` role, and hands the login's credential to the requesting workload together with a **lease**. A lease is a record that says who holds the credential, for which role and until when. When the lease expires, or someone revokes it, the broker removes the login.

In Ledgerly the broker connects with its own login, a member of `ledgerly_access_admin`, the role that creates and drops logins and manages role memberships. It owns no tables and is not the owner, and nobody uses it for data work. That is a narrower power than ownership, and it is still a wide one: whoever holds the role can grant anything inside the roles it administers, application and human roles included. The broker needs only a small part of that power. Handing it less would mean splitting the role, which has costs of its own; here the broker uses the role as it is. The role is one role used by three principals, each through its own login: the pipeline step, this broker and the gateway. Chapter 3 says what

bounds that power. For the broker, the guard is the one statement made here: its credential is held like the owner's, only its own workload identity can fetch it, and every action it takes goes to the audit log (Chapter 14).

The sequence below shows an instance of `billing-api` asking for a login. Look for the two places where the request is refused: before anything exists, and when the audit log cannot be reached.



In pseudocode, the order of operations is what keeps the sequence correct. Function names such as `createLogin` are illustrative, as elsewhere in the book.

```

function issueServiceLogin(request):
    identity = verifyWorkloadIdentity(request.proof)
    if request.role not in rolesAllowedFor(identity):
        raise IssueRefused
    password = generatePassword()
    login = identity.workload + "-" + randomSuffix()
    expiresAt = now() + ttlFor(request.role)
    // the lease comes first: the sweeper reads it, so no login can
    exist
    // without a record of when it must end
    leaseId = recordLease({ login: login, identity: identity.workload,
                           instance: request.instance, role:
request.role, expires_at: expiresAt })
    try:
        auditLogAppend({ action: "issue", subject: login,
                        actor: "credential-broker", lease: leaseId })
    on AuditLogUnavailable:
        markLeaseVoid(leaseId)
        raise IssueRefused
    // a failure here leaves a lease with no login; dropping a missing
    login is harmless
    createLogin(login, password: password, valid_until: expiresAt)
    grant(request.role, to: login)
    return { login: login, password: password, expires_at: expiresAt }

```

The lease is written before the login exists, and the audit entry is written before the login is created. If the database fails after the lease is recorded, the sweeper finds a lease with no login and treats it as already revoked. If the audit log is down, no credential is issued at all. This is a deliberate choice, and it means an unreachable audit log stops new instances from starting, a cost that Chapter 14 discusses from the other side.

Expiry and revocation

An expiry only stops new connections from authenticating. A session that is already open keeps running after the password has expired, so expiry has two parts: the login is made invalid, and the open sessions on it are closed. That is also the difference between rotation and **revocation**, which makes a credential or grant stop working now. Rotation replaces; revocation ends.

```

function revokeExpiredLeases():
  for each lease in expiredLeases(before: now()):
    try:
      closeSessions(login: lease.login)
      dropLogin(lease.login)
    on LoginNotFound:
      // the login is already gone, which is the goal
    on RevokeFailed:
      // an expired credential that still works is an incident
      notify("platform", "expired login still active: " +
lease.login)
      continue
    markLeaseRevoked(lease.id)
    try:
      auditLogAppend({ action: "revoke", actor: "lease-sweeper",
                        subject: lease.login, lease: lease.id })
    on AuditLogUnavailable:
      // the revoke is done; the log catches up from the retry
queue
      queueForRetry(lease.id)

function checkOverdueLeases():
  // scheduled separately, so one failing job cannot hide the other
  if count(expiredLeases(before: now() - graceMinutes)) > 0:
    notify("platform", "leases past expiry and not revoked")
  if now() - lastSuccessfulRun("revokeExpiredLeases") > maxRunAge:
    notify("platform", "lease sweeper has not completed recently")

```

Renewal follows the same discipline. A running instance asks the broker to extend its lease before it ends, and the broker agrees only while the lease is live and the total lifetime is below a ceiling. Without a ceiling, renewal turns a short-lived credential into a long-lived one that happens to be reissued. The lifetime itself is a judgment call. I do not know the right number for your system, and it depends on three things: how long an instance runs between restarts, how many connections each instance holds, and how long a renewal takes when the broker is under load. As an illustration only, think of a lease of one hour renewed every thirty minutes.

The gain over A/B rotation is that a leaked credential expires on its own. The login names the instance, since it was created for that instance, so the database log answers "who ran that statement?" down to the workload. This delivers what Chapter 3 promised: a login that maps to one workload instance.

Workload identity and IAM-based database authentication

There is a way to remove the stored password altogether. Some platforms let the database trust the platform's own identity system: the workload asks the platform for a short-lived token, and the database accepts that token in place of a password. Cloud providers offer this under names such as IAM database authentication, and some databases can verify tokens from an external identity provider.

The attraction is that there is no secret to store, fetch or rotate. The token is minted for the workload when needed, is short-lived (typically minutes), and is bound to the identity the platform already knows. Revocation is the platform's job: removing the identity's right to request tokens stops new tokens, and tokens already issued may be accepted until they expire. The login in the database still exists, is still a member of its `app_*` role, and still appears in the log.

The costs are specific:

- **Platform coupling.** The design works where the platform supports it and nowhere else, so a team with workloads on two platforms builds the pattern twice.
- **Mapping.** Someone has to say which platform identity corresponds to which login and role. That mapping is a grant, and it deserves the same review as any other (Chapter 14).
- **Open connections.** On common engines the token is checked when a connection is opened. As with any expiry, an open connection outlives its token (see "Expiry and revocation"), so a limit on connection lifetime is still needed (Chapter 5).

Which of the three approaches to use depends less on fashion than on what the platform offers and what the team can operate.

Approach	Stored secret?	Lifetime	Needs	Main cost
Static credential in a secret manager, A/B rotation	Yes, one pair per service	Until the next rotation	Secret manager, rotation job	Consumers must re-read; the rotation job must keep running

Approach	Stored secret?	Lifetime	Needs	Main cost
Dynamic credentials from a broker	Only the broker's own	Minutes to hours, per instance	Broker, <code>ledgerly_access_admin</code> , lease sweeper	Broker is a high-value, always-available component
Workload identity tokens	None for the database	Short-lived, typically minutes	Platform and database support	Platform coupling; tokens checked only at connect

None of these removes the credential problem; each moves it. The broker's own login is a static credential, held in the secret manager, fetched by the broker's workload identity and rotated A/B. The token design relies on the platform's identity system being correct. What changes is how many secrets there are, how long each one is useful, and who can read them.

TLS and certificate verification

Everything above concerns where a credential is kept and how long it lasts. Before it reaches the database, it crosses a network, and a connection that is not encrypted carries the password, and the data, in a form that anything on the path can read. Encryption in transit closes that, and it is the baseline. It has to hold on every leg. A request often passes through a pooler or a gateway before it reaches the database, and each of those is a separate connection with its own encryption and its own certificate check. Encrypting the leg from the client to the pooler says nothing about the leg from the pooler to the database.

There is a second job that is easier to skip: knowing that the server is the right one. A connection can be encrypted to a destination that is not the database. **Certificate verification** is the client checking that the server's certificate was issued by an authority it trusts and that the certificate names the host it meant to reach. Without it, encryption protects against someone reading the traffic and does nothing about someone answering the connection.

The setting that matters is the client's verification mode. Most database clients offer a mode that encrypts but does not verify, and it is popular because it works with any certificate, including expired and self-signed ones.

I regard that mode as half the job: it hides the traffic from a passive observer and gives a false sense that the server is the right one. The remedy is to distribute the certificate authority to each service and verify the name.

That brings its own costs. Certificates expire, and an expired server certificate is an outage with an exact date on it, so expiry needs monitoring and a renewal procedure that is rehearsed. Distributing the authority certificate to every consumer is work, and it has to be updated when the authority changes. Client certificates, where the certificate is the credential, work with short lifetimes only if something issues them automatically.

When a credential leaks

Even with every control above, a credential can leak, and the design should say in advance what happens next. The first step is to separate the two questions that panic merges: how do we stop the credential from working, and what did it do while it worked?

For a dynamic credential, the first answer is a revoke: end the lease, close the sessions, and the login is gone within the time it takes the broker to act. For a static credential in an A/B setup, a suspected leak is not the planned rotation. The planned job waits for the old login to drain, and a leaked value stays valid for as long as anyone holds a session on it. In a leak, cut the leaked login first: remove its roles, close its sessions and change its password without waiting for the drain, then publish a new value to the idle login and switch the active pointer. Instances that still hold the old login lose their connections and fail until they re-read the secret. That is an availability cost, and I would pay it: a leaked credential that keeps working is the larger problem. A service that cannot re-read its secret needs a restart, and the team should know which services those are before the day it matters. For a shared admin password in the starting state, the first answer is the outage this chapter began with, which is the strongest argument for doing the work before an incident.

The second question is answered by the log. A login that maps to one workload instance lets you see the statements it ran, the tables it touched and the hours it was active. That narrows the review to the blast radius of one role rather than the whole database. The step-by-step procedure, the

order in which to act, and how to rehearse it belong to Chapter 15, which covers access during incidents. Here the point is that a team that has built revocation as a routine step can use it on a bad day without inventing it.

Data-protection and payment-card regimes tend to ask how access is granted, how long it lasts and how it is revoked. The leases, the revocation steps and the audit entries described here are the records that answer those questions.

Ledgerly's migration plan

The move from the shared admin password to short-lived credentials is a sequence of small, reversible steps. I would order them by how much risk each step removes and how little it can break.

1. **Inventory every copy of the admin password.** Services, pipeline variables, the export step, the password manager, saved connections in client tools. The list is the work plan. Chapter 1's inventory is the starting point.
2. **Create a login per service, in its `app_*` role.** Use the roles from Chapter 3. At this stage the credentials are static, but each is separate, stored in the secret manager, and read by identity.
3. **Move services off the admin login one at a time.** Start with `reporting-job`, whose blast radius is smallest, then `invoice-worker`, `admin-console` and `billing-api`. Do `payments-service` last, because it is the only service that reaches `payment_tokens` and `bank_accounts`, and an error there costs the most. Each move is a deploy that can be rolled back.
4. **Give migrations and the export step their own logins.** The pipeline uses a short-lived login in `ledgerly_migrator` (Chapter 11); the role itself holds standing owner privileges, and only the credential is short-lived. The export step uses `app_vendor_export` (Chapter 13). Neither uses the admin login after this step.
5. **Switch each service to A/B rotation.** Make each service re-read its secret after an authentication failure, run the first rotation in a quiet period, and watch the drain check. A service that does not drain is a finding, and the finding is the point of this step.

6. **Introduce the broker for the services that renew easily.** Start with a service that restarts often and holds few connections. Add the others as the team learns how the leases behave. Keep static A/B as the fallback.
7. **Retire the shared admin login.** Remove it from every configuration and from the password manager. Once ownership has moved to `ledgerly_owner` (Chapter 3), strip its privileges and drop it, or disable it. If the admin turns out to be the engine's superuser, as a managed service's master user often is, it becomes what Chapter 3 calls the sealed instrument. That is a credential nobody uses in daily work. It is held where the platform team can retrieve it, and each retrieval is recorded. Change its credential once, after the whole inventory is cleared.

A team could stop after step five and still be much better off: each service has its own credential, reads it through an identity, and rotates it without an outage. The broker is an addition for teams that can operate it.

Trade-offs and costs

Every move in this chapter shifts cost from an incident to a routine. Here is where the routine gets expensive.

Dependence on the secret manager. Once credentials come from a secret manager, an unreachable secret manager stops new instances from starting, and with short leases it eventually stops running instances from renewing. That makes it part of the availability of the database path. Run it with the care you give the database: more than one instance, a tested recovery, and an on-call rotation that knows it exists. Caching a credential for the length of a lease softens a brief outage, and lengthens the time a revoked credential can still be used.

Connection storms. If every instance of a service renews at the same minute, or restarts together after a deploy, the database sees a burst of new connections and authentications. Authentication is not free, and a burst of its own making can slow the database for every other role, a connection storm without any outside cause. Stagger renewals with jitter, keep the pool of connections stable between renewals, and let Chapter 5 shape how many connections a role may hold.

Harder debugging. A login that exists for one hour is gone when someone reads the log the next morning, and a name like `billing-api-4f2a` means nothing without the lease record. Keep leases and their identity for as long as the audit log is kept, and give whoever is on call a lookup from login to instance. If that lookup is missing, the short-lived credential makes investigation slower, not faster.

A high-value broker. The broker's login can create logins in every application role, so it is worth an attacker's effort. Keep its code small and reviewed, and give it the attention you give the thing it protects; a broker that gets less is a risk in itself.

When not to use this. A team with a single service, a handful of engineers and a managed secret store may be well served by a separate static credential, held in the secret manager and rotated A/B by hand twice a year. A broker is one more service to run and one more system for the team to learn; below a certain size, the added moving parts will cause more outages than they prevent. If you cannot name who would be paged when the broker is down, I would wait. Workload identity tokens are the cheapest step when the platform offers them, and still need the mapping reviewed.

Common mistakes

- **Treating the secret manager as the whole answer.** It is tempting because the move is visible and quick. The shared password is still shared if every service reads the same path. Give each service its own path and its own login, and review who can read each one.
- **A rotation nobody consumes.** A job rotates the value, but services read it at start and never again, so the new value sits unread until the old one is retired and something fails. Test that consumers pick up a rotation, and make the drain check part of the job.
- **Falling back to a baked-in value.** A "last known good" copy in the image makes start-up reliable and brings back the file the design removed. Refuse the start, alert, and repair the dependency.
- **Encrypting without verifying.** Turning on TLS satisfies a checklist and the encrypt-only mode makes the connection work. Verify the certificate and the host name, and monitor the expiry date.

- **Lifetimes without a ceiling.** Renewal with no limit rebuilds a long-lived credential. Cap the total lifetime of a lease and make the instance re-authenticate with its identity.
- **Treating the broker's role as small.** `ledgerly_access_admin` is wider than the broker's job, so a broker that is taken over can reach as far as the roles it administers. Restrict which principals may call it and which `app_*` roles each may request, and review its audit entries. Treat the wide role as a cost you have accepted, not as a limit.

Key takeaways

- Every copy of a credential is a reader, so the most useful change is to reduce the number of copies and give each remaining read a name.
- Rotation is hard because a credential is shared state; two logins per service make it possible to change one while the other keeps working.
- A rotation that the consumers do not pick up, or that nobody runs, is a calendar entry and not a control, so the drain check and an alert on a rotation job that has stopped running belong to the design.
- Dynamic credentials give each workload instance its own login with a built-in end, at the price of a broker, a lease record and a dependency on the secret manager.
- Workload identity removes the stored password where the platform supports it, and moves the trust to the platform and the mapping between identity and login.
- Verifying the server's certificate and name is what makes TLS protect the credential; encryption without verification hides the traffic and leaves the server's identity unchecked.
- When a credential leaks, the speed of revocation and the clarity of the log decide how contained the event is, and that depends on work done before the leak.

Credentials decide who gets in; they say nothing about how many connections they open or how long a statement may hold the primary. Chapter 5 turns the connection itself into a policy: limits per role, pooling, timeouts and the labels that make a runaway query easy to find.

Chapter 5 — Connections as Policy

Consider Ledgerly on a quiet Tuesday night. A teammate raised the parallelism of the nightly aggregation in `reporting-job` so that it would finish before the morning, a change that looked harmless in review. At 01:40 the job starts, opens a connection for each of its worker threads and begins to read `payments` and `invoices` from the primary, using the shared `admin` login. Within a few minutes `billing-api` starts to return errors to customers in a time zone where it is already morning. Robin, on call, is paged for a customer-facing outage.

Robin opens the database and finds that nothing is broken. The primary is up, disk activity is normal, and no statement has done anything wrong. The server has a fixed number of connection slots, the report has taken most of them, and the API is waiting at the door. Because every caller uses the same login, the database cannot say that the report is the cause. It can only say that the slots are full.

This chapter answers a question that sits between identity and data: how do we make the connection itself a place where policy is applied? In the five-part model of Chapter 1 it works on two parts. The connection is the Path, because the pooler decides how services arrive and how many of them may. It is also the first bound on Execution, because limits and timeouts say how much one caller, and one statement, may take from everyone else. The step from the running example is that connections are shaped by a pooler and by limits and timeouts per role.

The problem, and why it is hard

A connection is a resource before it is anything else. On most engines each one costs the server memory and a share of its scheduling, and the server allows only a fixed number of them. The limit is a property of the server,

not of the caller, so the first caller to ask for many gets many. Nothing in the default setup says that the nightly report should have fewer slots than the customer-facing API.

A password does not touch any of this. It decides who gets in; it says nothing about how many slots they hold or for how long. That is why a book that is about more than distributing credentials needs a chapter on connections.

Three habits make the problem worse. First, the number of connections is a product, not a sum: the instances of a service multiplied by the size of each instance's pool, multiplied again whenever an autoscaler adds an instance. Say a team sets a pool of twenty on a service that scales to ten instances. It has asked for two hundred connections, and has probably not decided that on purpose. Second, a connection is held for as long as its statement runs, so a slow query occupies a slot the whole time, and slots are what the next caller needs. Third, a failing caller tends to try harder. A client that times out and retries immediately turns a slow database into a crowd at the door.

The obvious response is to raise the server limit. It helps for a week and then makes the next incident larger. More connections mean more concurrent work on the same primary, and the database does not get faster at running a hundred statements at once because a hundred are allowed.

The controls live in several places that nobody owns together: the role definition, the application's pool, a possible pooler, and the settings that decide how long a statement may run. Availability is one of the properties at stake in the threat model of Chapter 2, and accidents dominate it. A shared login turns every accident into everyone's accident.

In the starting state, then, all five services and every engineer on a VPN draw from one pool of server slots under one name. There are no timeouts beyond whatever each client library defaults to, and there is no record of which caller holds which connection. The sections that follow replace that with a budget per role, a pooler that enforces it, timeouts that set an upper limit on how long anything may hold a slot, and a label on each connection that says who owns it.

A connection budget per role

The first step is the cheapest. Chapter 3 gave each service its own application role, which means a limit can finally attach to a name. A limit per role says: this principal may hold at most this many connections, no matter how many instances it runs or how hard it asks. When it reaches the limit, new connections from that role are refused, and every other role keeps working.

That refusal is the policy. It moves the failure from "the database is full for everyone" to "this caller is out of slots", and the second is a much better page for Robin to receive. The blast radius of a greedy caller becomes the caller itself, and the refusal names the caller, which is the accountability half of the same step.

The numbers need reasoning, not a formula. I would work from the server's limit downward. First subtract a reserve that nothing else may use, because the people and tools who repair the system also need to connect. The reserve covers platform sessions and the logins of the three principals that use `ledgerly_access_admin` (the pipeline step that applies role definitions, the credential broker and the access gateway). Then divide the rest among the application roles by what each one needs at its busiest, plus some room. The following table is an illustration for the primary. The numbers are hypothetical, not a recommendation.

Principal	Role	Connection limit (illustrative)	Why this size
billing-api	app_billing_api	80	Busiest service; serves the web app and the public API
invoice-worker	app_invoice_worker	30	Queue-driven; concurrency is set by the worker count
payments-service	app_payments_service	30	Steady; talks to external providers between statements
admin-console	app_admin_console	15	Narrow, audited actions by a few humans
Migrations	ledgerly_migrator	3	One migration at a time, from the pipeline

Principal	Role	Connection limit (illustrative)	Why this size
Human sessions on the primary	Session logins of kind JIT and break-glass, created by the gateway	8	Only sessions with an approved request run here (Chapter 6); routine reads go to <code>ledgerly_replica_2</code> . The roles <code>human_jit_write</code> and <code>human_breakglass</code> carry no limit, because no login is a member of the first and the second is granted per session
<code>reporting-job</code>	<code>app_reporting_job</code>	10	Transition only: it reads the primary in the starting state and during the move, and the row goes to zero in the target state
Reserve	access administration and platform	10	Only for repair and for issuing logins
Unallocated	none	14	Headroom, so a rounding error is not an outage

Suppose the server allows 200 connections. The allocated rows add to 186, and the remaining 14 are deliberately left unassigned. The routine human sessions are not in this table because they do not run on the primary. Their budget is set on `ledgerly_replica_2`, where routine reads go; say, 20 slots there. The `reporting-job` row exists only while the report still reads the primary. In the target state it reads from `ledgerly_replica_2`, with its own budget on that host, set at the connection layer in the pooler (Chapter 6 covers the move). The role definitions might carry the limits in a notation like this one, which is illustrative and belongs to no engine:

```
// illustrative notation: the limit belongs to the role, not to a
person
ROLE LIMIT app_billing_api      CONNECTIONS 80;
ROLE LIMIT app_invoice_worker   CONNECTIONS 30;
ROLE LIMIT app_payments_service CONNECTIONS 30;
ROLE LIMIT app_admin_console    CONNECTIONS 15;
ROLE LIMIT ledgerly_migrator     CONNECTIONS 3;
ROLE LIMIT app_reporting_job     CONNECTIONS 10;
```

Note: Engines differ in what a limit or a setting attaches to. Some count connections per role, some per login, and some only per database or per server. On some engines a limit or a default written on a role applies only to connections that log in as the role itself, and is not inherited by the logins that are members of it. Chapter 4 gives each service instance its own dynamic login, a member of the `app_*` role, so on those engines the role-level limit and the role-level timeouts below would silently not apply.

Two places can carry them instead: the broker, which can set a per-login limit and the same settings when it creates each login, and the pooler, which can carry the settings but may not be able to sum the logins of a role into one total. The pooler section below sets out three ways to get a role total when it cannot.

Two properties make the table useful. The limits are checked where the connection starts, so a caller learns of the refusal at once and not after the first slow statement. And the reserve sits outside every application role, so that on the worst night the platform team can still log in to fix things. A budget that fills the whole server has the same flaw as a shared admin password: it works until the moment someone needs it most.

The sums do not have to be conservative. If `billing-api` and `payments-service` rarely peak together, you can let the application limits add up to more than the server allows, and accept that two simultaneous peaks might collide. I would do that only with the reserve held back, and only after the team has watched the actual usage for a while. The default is a budget that adds up, because a budget that cannot be violated is easier to reason about at three in the morning.

Pooling: where the connections are shared

A pool keeps a set of open connections and lends them to callers, so that each request does not pay for a new connection and the database sees a bounded number of them. There are two places to put one.

Nearly every service already has a pool in its database library, and it should be sized from the work one instance can do at once, not from the traffic it hopes to serve. A small pool that makes callers wait briefly is

kinder to the database than a large one, and it should give up after a short wait with an error the service can turn into a clear response. Start small, watch how long callers wait, and raise the size on evidence.

The pooler in front of the database

A **pooler** is a database proxy whose main job is to accept many client connections and multiplex them over a smaller number of server connections. PgBouncer and ProxySQL are common examples. The service connects to the pooler as though it were the database, and the pooler decides when a real connection is lent out.

The question the book keeps asking applies here too: whose name does the database see? Chapter 4 gave every service instance its own dynamic login, created by the credential broker and a member of the service's `app_*` role. The pooler has to fit that model, and I would build it in this shape. The instance presents its own login and credential to the pooler. The pooler verifies them against the database and keeps server connections under that same login. It does not hold a password store of its own, and it does not hold a login of its own that every client shares.

A rotated or expired credential reaches the pooler through the database, not through the pooler. When a lease ends or is revoked, the broker drops the login (Chapter 4), so the pooler's next attempt to authenticate or to open a server connection under that login is refused, and it closes the idle server connections it holds for it.

How a per-role limit still holds has no universal answer, and I would not assume your pooler can give one. Many poolers keep their limits per login and database pair, and some add a limit per login or per database. I know of no general capability that sums the connections of many different logins into one role-level cap, so read your pooler's documentation instead of taking this chapter's word for it. If yours cannot do it, the role total has to come from somewhere else. There are three places, and each has a price.

- **The pooler connects upstream with one stable login per role.** The instance's dynamic credential then authenticates the client to the pooler only, and the pooler's per-login limit is the role's limit, exactly. The price is the identity model of Chapter 3:

- The database log shows one name per role, so attribution to an instance moves to the pooler's log and to the application label described below.
 - The pooler must verify dynamic credentials itself, which makes it a consumer of the broker's lease state.
 - It also holds a long-lived credential for each role, to be guarded and rotated A/B as Chapter 4 describes.
 - Revoking an instance's login at the broker no longer closes its server connections; the pooler has to end that client's sessions.
- **The database enforces a per-role limit, where the engine supports it for member logins.** It is exact, it sits where nothing can bypass it, and the instance logins and their attribution stay intact. The price is that support varies, as the note above says. Where it works, the refusal comes from the database after the connection has already cost an authentication, and nothing multiplexes.
 - **The broker caps how many logins it issues per role.** With a per-login limit set when each login is created, the role's total cannot exceed the number of logins times that limit. The price is that this is an upper bound and not an exact count, and that the cap has to match the autoscaler's instance ceiling. A scale-out past the cap becomes a deploy that cannot get credentials, which is an availability problem, and renewals that briefly overlap an old and a new login need room under the cap.

I would try the database-side limit first, where the engine honors it for member logins. Where it does not, the broker cap keeps the identity model and costs a ceiling to maintain. I would take the single upstream login per role only when connection counts force multiplexing across instances, and I would write down what the log no longer says. All three are better than the design that maps every client of every role to one shared login. It recreates the shared admin account one layer down: the log shows a single name again, and a limit applies to everyone at once.

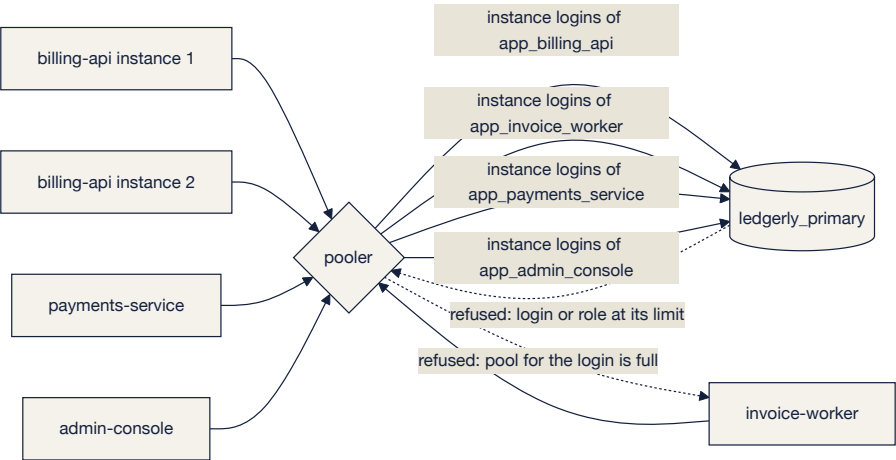
In the recommended pass-through shape the pooler buys little multiplexing. In transaction mode the clients of one instance share that instance's server connections, but two instances do not share with each other, so a

role's total is at least the number of live instances times a small minimum. What the pooler buys is three other things.

- **Central refusal.** One place says no, per login and per pool, in the same way for every client library.
- **The same limits for every service.** The pool sizes and waits in each service stop being the only line of defense.
- **A network chokepoint.** Services enter the database only through the pooler. The network policy, which says which hosts may open a TCP connection and which Chapter 6 defines and Chapter 7 writes, admits the pooler's hosts and no one else, so a credential that leaks from a service is useless from anywhere the pooler is not reachable from. Reach bounds a leak, as Chapter 4 says, and the pooler is where that bound is easy to state.

The cost is a new system in the path, covered in the trade-offs below. For a single small service with one instance, the application's pool and a role limit are enough, and the pooler is not worth adding until the instance count or the need for one chokepoint says otherwise.

The following diagram shows the pass-through shape, which leaves the instance logins visible to the database. Look for the two places where a caller is refused: at the pooler, when the pool for one login is full, and at the database, when a login or, where the engine supports it, the role is at its limit.



Session mode and transaction mode

A pooler lends a server connection for a period, and the length of that period is its **mode**. In session mode the client keeps the same server connection from connect to disconnect; the pooler saves nothing except the cost of connecting. In transaction mode the client holds a server connection only for the length of one transaction, and the next transaction may run on a different one. Transaction mode multiplexes far more clients over few connections, and that is why it is attractive.

It comes with a cost that is easy to miss, because it concerns authorization and not performance. Anything a client sets on its session lives on one server connection, and in transaction mode the client does not own one. Settings made outside a transaction may vanish on the next statement, or worse, may stay on a connection that the pooler lends to someone else. The things that depend on session state include temporary tables, prepared statements on many setups, advisory locks, session-level timeouts, and any context value your own design stores in the session.

That last item touches the book directly. In Chapter 3, a row rule may depend on a value such as `current_case_customer()`, and `app_admin_console` carries a human principal's identity into the audit log and into `audit_events`. If either value is set once per session and read later, then in transaction mode it can be missing or, in the bad case, belong to the previous client.

Warning: In transaction mode, a context value set at session level can be left on a server connection and read by the next client's transaction. A row rule or an audit record that reads it may then act for the wrong customer or the wrong person. Set such values inside the transaction, scoped to it, and make the code that reads them fail when the value is absent.

The pattern that holds up is to set the context inside the transaction and to let it end with the transaction. The next block shows the shape, and it is written to stay correct when the work fails part-way.

```

function runAsPrincipal(principal, customerId, work):
  transaction:
    // transaction-scoped: it disappears at commit or rollback, so
    // the next client cannot inherit it from this connection
    setLocalContext("acting_principal", principal.id)
    setLocalContext("case_customer", customerId)
    try:
      work()
    on ContextMissing:
      // a row rule or audit write found no value: stop, do not
guess
      raise ActionRefused
    // the commit happens here, once, after all of the work
succeeded
    // on any error the transaction rolls back, and the context goes
with it

```

The database log still shows the instance's login, but in transaction mode an instance's clients share its server connections, so the log cannot say which of the instance's threads or requests held a slot. To find that out you need the pooler's own view of its clients, and it needs to be planned before the first incident. That is why the next sections on timeouts and naming matter.

Humans do not use this pooler. They reach the database through the access gateway, which creates a login for each session (Chapter 7). Their limits are the subject of the last part of the next section, because a person can do something a service cannot: lift the limit.

Timeouts as policy

A slot is lent for as long as a statement or a transaction runs, and nothing on most engines ends either of them by default. Timeouts are how the platform team says how long a role may hold a resource, and they are the part of this chapter that bounds Execution: a statement that runs too long is cancelled where it runs, and the slot goes back. They belong to the role definition and the pool configuration, and not to the goodwill of each client library, which is why this chapter treats them as policy.

Three timeouts belong to the database and two to the pool.

Timeout	Belongs to	What it stops	Who feels it
Statement	Database	A query that runs far longer than the role should ever need	The caller of that statement
Lock wait	Database	A statement queued behind a lock, and everyone queued behind it	The caller; also any caller waiting on the same row
Idle in transaction	Database	A transaction opened and then left, holding locks and slots	The application that opened it
Connection lifetime	Pool	A connection that stays open for days and keeps its old credential or state	The pool, which must reconnect
Pool acquire wait	Pool	A caller waiting forever for a free connection	The service handling the request

Different roles need different values, because they do different work. A request in `billing-api` is a person waiting for a page; a statement that needs thirty seconds there is a bug. `invoice-worker` may legitimately run a longer statement while it builds a batch. `reporting-job` runs long reads by design, on a replica where a long read does not block the writers. A shared default would be wrong for all of them, so the values attach to the role. The block below uses illustrative notation, and the values are hypothetical.

```
// illustrative notation: defaults for the role; see the note above on
// members, because the broker or the pooler may have to apply them
per login
ROLE SETTING app_billing_api      statement_timeout 5s, lock_wait 2s,
idle_in_transaction 10s;
ROLE SETTING app_invoice_worker  statement_timeout 30s, lock_wait 5s,
idle_in_transaction 30s;
ROLE SETTING app_reporting_job   statement_timeout 15m, lock_wait 5s,
idle_in_transaction 60s;
ROLE SETTING human_readonly      statement_timeout 60s, lock_wait 5s,
idle_in_transaction 5m;
ROLE SETTING ledgerly_migrator   statement_timeout none, lock_wait
3s, idle_in_transaction 60s;
```

The last line reflects a migration lesson that Chapter 11 covers: a schema change may run for a long time, but it should wait only briefly for a lock.

Two rules about order matter more than the individual values. The database's statement timeout should be shorter than the application's request deadline, because if the application gives up first it abandons a statement that is still running and the slot stays occupied; and the pool's acquire wait should be the shortest of all, so that callers learn about congestion early. When a deadline does fire first, the connection may still carry a running statement, so the service closes it and does not lend it to the next caller.

The idle transaction is the surprising one. A service that opens a transaction, calls an external payment provider and waits with the transaction still open lets the provider's slowness hold locks and a slot on the database. The long-term fix is to keep external calls outside transactions, which is a code change; the timeout keeps the cost bounded while that change is on the backlog.

A connection that lives for weeks keeps whatever it was born with, so a maximum lifetime makes the pool replace its connections regularly. It should be shorter than the credential's lifetime, with some variation between connections so that they do not all reconnect in the same minute. Chapter 4 explains why an expired credential does not close a session that is already open; the lifetime is how the pool closes its own.

Defaults a session can lift

A timeout set as a role default is a default. On many engines a session may change its own settings, including its statement timeout and its read-only mode, and may switch role. For a service that is acceptable: the limits exist for accidents, and code does not lift its own timeout by accident. For a person with a SQL client it is not, because the session that needs the limit is the session that can lift it.

Two controls sit where the session cannot reach them. The gateway opens each human session with its limits applied and refuses a request to change the settings on a guarded list; Chapter 9 covers how the gateway does this. The cost is that clients which send such settings when they connect stop working until someone has listed them. Human sessions get their limits from the gateway: it sets a per-login connection limit by session kind when it creates the session login, and it counts how many sessions a person may hold. Neither is a role limit, since `human_jit_write` is a ceiling

that no login is a member of. Neither is left to the client. For any limit that has neither, the table of controls should say that it is a default and not a guarantee.

Naming the caller

When Robin looks at the connections, the useful question is "whose is this?", and the login answers the first half. A role says that a connection belongs to `billing-api`. It does not say which instance, which release, or which run of a scheduled job. Most engines and poolers allow a client to attach a short label to its connection, usually called an application name, and it shows up in the database log and in the view of current activity.

A convention is worth more than the feature. I would put three things in the label: the service, the instance or run, and the release. For example `reporting-job/run-0412/v31` says which job, which night's run, and which build. The label turns "someone is holding forty connections" into "the Tuesday run of the report holds forty connections, and it is the new build".

Two cautions come with it. First, the label is supplied by the client and can be set to anything, so it is for diagnosis and attribution, not for authorization. The login is the identity; the label is a note on the envelope. Second, a pooler in transaction mode shares server connections between clients, so the label on the server side may not be the client's. Some poolers forward it for each transaction, some do not, and the pooler's own log of its clients is then the record that carries the name. Find out which your setup does before you depend on it.

The label also feeds Chapter 14: an audit log is easier to read when each line has the service and the run, not only a login. The cost is a line in each service's startup, and one more convention to keep when a new service appears.

Connections are not the only resource a role can consume. Memory for sorting and joining, temporary disk space, CPU and the number of parallel workers are also shared, and engines differ widely in whether they can limit them per role. Where the engine cannot, the isolation has to come

from placement: a heavy reader goes to a replica, which is the argument of Chapter 6. Check which limits your engine can set per role before promising any of them to anyone.

Ledgerly: the night the report ate the API

Return to the Tuesday night. The starting state explains it in one sentence: `reporting-job` ran as `admin` against the primary, so it competed for the same slots as `billing-api` under the same name, with no limit on either and no timeout on anything.

Alex's plan does not begin with the pooler. It begins with what is cheapest and gives the most protection, and moves down the list as the team learns.

1. **Limits per role.** With the application roles from Chapter 3 in place, each gets a connection limit from a table like the one above. The report loses the ability to take the API's slots at once, even though it still reads the primary on the first night.
2. **Timeouts per role.** `app_billing_api` gets a short statement timeout and a short idle-in-transaction limit, and the report gets a long statement timeout that does not apply to the API.
3. **Labels.** Each service and each scheduled run sets its label. The next page names the run, not only the role.
4. **A pooler.** The five services connect through it and it becomes their only way in, with the role total taken from one of the three places above.
5. **A move to the replica.** `reporting-job` reads from `ledgerly_replica_2` with its own budget, and the primary no longer carries its work. Chapter 6 describes the move itself.
6. **An alert before refusal.** The platform team watches the saturation of each pool, not only the refusals. A pool that stays at, say, ninety percent of its limit for hours says "raise the limit or fix the caller" while there is still time.

The job also needs a change of its own, because a limit that refuses the job is only useful if the job reacts well to being refused. A job that retries at once after a refusal is the crowd at the door in a smaller form.

```
function acquireWithBackoff(role):
  for attempt in 1..maxAttempts:
    try: return acquireConnection(role, waitUpTo: acquireWait)
    on PoolEmpty: sleep(backoffWithJitter(attempt)) // slow down,
do not retry harder
  raise ServiceBusy
```

With these in place, the same change that raised the report's parallelism would have a different result. The job reaches its limit and is refused, it slows itself down and takes longer, and the API does not run out of slots because of it. A job that gives up on a batch should also leave a record of what it skipped, so that nothing reads half-built aggregates as whole; that is the job's own design and goes beyond this chapter. The review that approved the change would still have been fine, because the design no longer relies on a reviewer noticing the interaction between two services.

Trade-offs and costs

Per-role limits cost tuning. A limit that is too low refuses legitimate load, and the refusal looks like an outage to the service that receives it. Limits need an owner and a review whenever a service scales out or a new workload appears, because the sum of instances times pool sizes can pass the limit without any single change looking wrong. The platform team becomes the place where those questions are asked, which is useful and also a new queue.

A pooler is a new system in the path. Every query now passes through it, and when it is down, every service is down. It must be available during the incident in which you most need it, so it needs its own redundancy, monitoring and runbook. In transaction mode it also removes session features that some code depends on, and it makes the database log harder to read. If you have a single small service with one instance and a pool of ten, the application's own pool plus a role limit is enough.

Timeouts cancel legitimate work. A statement timeout set too tight kills a good statement, and a naive retry repeats the work and the cancellation. Services need to handle cancellation, and for writes this means thinking about whether the operation can be run twice without harm. The values for `ledgerly_migrator` and for data fixes (Chapter 10) differ from those of applications, which means more configuration to keep correct.

A limit nobody watches protects nothing. A statement timeout so long that it never fires, or a role limit that no alert covers, adds a line to the configuration and no bound. The test is whether anyone would notice if the control stopped working.

Labels are cheap and spoofable. They help people and cost discipline, but they prove nothing, and a log that relies on them for attribution is relying on a convention.

When not to use these controls: do not add per-role limits before roles exist, since a limit on the shared `admin` role only limits everyone at once. Do not set aggressive timeouts on a service before you know how long its normal statements take, because the first cancelled invoice run will teach the team to turn the timeouts off. Measure the normal first, then set the limit above it.

Common mistakes

- **Raising the server limit to fix an incident.** It is tempting because it works at once and costs nothing today. It moves the failure to a larger crowd and removes the early signal. Give each role a budget and leave the server limit as the sum.
- **Sizing the pool for the traffic you hope to have.** More connections feel like more capacity, but the database has a fixed amount of useful concurrency. Size the pool for the work one instance can do at once.
- **Mapping every client to one login in the pooler.** It is the easy configuration, and it brings back the shared admin account. Keep each instance's own login, or choose one stable login per role explicitly and accept what the log no longer says.
- **Using transaction mode with session state.** The code worked in session mode, so nobody tests it in transaction mode. Context for row rules, identity for audit and temporary tables must live inside the transaction, or the mode must stay at session level for that role.
- **One set of timeouts for every role.** A single default is too tight for the batch and too loose for the API, and the usual outcome is that it gets raised for everyone. Attach the values to the role.
- **Treating the application name as identity.** The label is set by the client. Use the login to authenticate and the label to explain.

- **Filling the server with application budgets.** When the limits add up to the whole server, the platform team cannot connect during the incident. Keep a reserve outside every application role.

Key takeaways

- A connection is a limited resource and a point of policy: a limit that belongs to the server and not to a role lets the first greedy caller take everyone's share.
- A per-role connection budget turns "the database is full for everyone" into "this caller is out of slots", which shrinks the blast radius of a run-away job to the job itself.
- Keep a reserve of connections outside every application role, so that the people and tools that repair the system can always connect.
- The pooler is where services enter, so it is the place for central refusal and the network chokepoint. Many poolers cannot sum a role's dynamic logins, so the role total comes from the database, the broker or one upstream login per role, each with a named price, as the pooler section sets out.
- Transaction-mode pooling removes session state, so any context that a row rule or an audit record depends on has to be set inside the transaction and has to fail loudly when it is missing.
- Statement, lock wait and idle-in-transaction timeouts belong to the role, and the database's statement timeout should fire before the application's deadline. A session can lift a default, so for people the limit has to be enforced where they cannot change it.
- Application names make connections explainable but prove nothing, so the login remains the identity and the label remains a note.

A connection limit and a pooler decide how many callers reach the primary, but they do not change how much work lands on it. Chapter 6 takes the next step: moving the reads to replicas and other copies, and the access questions that arrive with every copy.

Chapter 6 — Reads, Replicas and Data Copies

Imagine a Tuesday morning at Ledgerly, a few days before month end. Jordan, from the data team, has been asked for a number: how many invoices were issued last month, split by customer segment and by whether a payment attempt had already failed. The analytics copy would be the natural place to ask, but it is a nightly restore, and the restore is a day old. The question needs today's data. Jordan opens a desktop client, connects with the shared admin login, and runs a query that joins `invoices`, `invoice_lines` and `payments` across the whole month.

Nothing in the query is wrong. It is a reasonable question, written by someone who is good at their job. But the connection points at the primary, because that is the host the client remembered, and the query reads several large tables while `billing-api` is serving customers who are about to pay their invoices. The web app slows down, Robin gets paged, and Robin and Jordan spend twenty minutes working out that the report and the slowdown are the same problem. Nobody did anything unusual. The design offered one place to read from, and every read went there.

This chapter adds one step of the target state: reads go to replicas, and reporting never touches the primary. In the five-part model of Chapter 1 it is about the Grant and the Path working together, which is what authorization means in this book. The grant decides what a reader sees, and the path decides which database the reader can reach at all. It also deals with what follows from the step, which is that every replica, and every other copy of the data, is a new place where access has to be designed. The question is this: when data is read in more places than it is written, which reads go where, who may read each copy, and how do you stop the copies from multiplying into doors nobody owns?

The problem, and why it is hard

A database that takes writes and also serves every read is doing two jobs with one set of resources. Writes need the primary: there is one of it, it is the source of truth, and it must stay available for them. Reads can be repeated, they can be spread out, and many of them tolerate data that is a few seconds old. Moving reads away from the primary protects the writes, and protecting the writes is protecting the ledger.

The obvious approach is to add a replica and point the read-heavy workloads at it. That works for about a week. After that, three problems appear, and none of them is solved by the replica itself.

The first is that a replica is not the primary. In most deployments replication is asynchronous: the primary commits a change, and the replica applies it shortly afterward. The gap is called **replica lag**. A read that lands on a lagging replica returns a true answer about a slightly older world. For a monthly report that is invisible. For a customer who has just paid an invoice and is looking at a page that still says "due", it is a support ticket.

The second problem is that the privileges travel with the data. A physical replica is a full copy, including the Restricted columns, and anything that can read the replica can read what a reader of the primary could read, unless the grants say otherwise. The next section explains why a role cannot be made safer on one replica alone.

The third problem is that copies multiply. The first replica is for the application. The second is for reporting, then for human investigation, then for the export step. A warehouse follows, fed by a schedule or by change data capture. Someone restores a snapshot into a staging database. Each copy was a sensible answer to a real load or convenience problem on its day. Each also holds production data, carries its own credentials and its own network path, and is rarely on anyone's list of ways into the data. A **side door** is a data copy or export path that bypasses the access controls of the primary. This chapter is about the copies you build on purpose, and Chapter 13 returns to the ones that appear by accident.

A policy that says "do not run reports on the primary" does not help Jordan. The client remembers one host, the login works on all of them, and the primary is the one that always has fresh data. The control has to be in how reads are routed and in what each login is able to connect to.

Splitting the read path

I would start by treating the read path as its own access surface, with its own inventory, its own roles and its own rules, and by asking one question of every read: does this read need the primary? A read needs it when it decides something, such as a balance before a payment is collected, or when it must see a write made a moment ago in the same session. Everything else may go to a replica. How a service classifies and routes its own reads is application design, and this book leaves it aside.

The access rule is the other half, and it is two lines. Reporting reads and human reads never fall back to the primary: if the replica is unavailable, the job waits or fails and the platform team is told. And the logins used for them cannot connect to the primary at all, so a mistake cannot send them there. A nightly aggregate that runs two hours late is an inconvenience. One that quietly switches to the primary because the replica was down is how the starting state came about in the first place.

A read-only role is not a harmless one

Reads are usually treated as load. They are also the main confidentiality exposure: a result set that leaves the system changes nothing, leaves nothing to roll back, and can be copied anywhere. A role that cannot write can still read every column it is granted, and one heavy query from it can hold the primary down. "Read-only" also means three different mechanisms, which differ in whether the session can undo them.

Mechanism	What it stops	Can the session undo it?
A grant without write privileges	Writes by that role	Not by itself; but a session that can switch to a role with write privileges is no longer bound by it
A read-only transaction default	Writes in that session, by default	On many engines, yes: the session can set it back or start a read-write transaction
A physical replica	Every write on that host	No: the host refuses writes, and the session has no setting to change that

For people, the replica is the strongest of the three, and that is one reason routine human reads go there. What none of them limits is how much a session may read. A row cap on a session and tiering for exports belong to

Execution, and Chapter 9 covers them.

Grants travel to the replica

The replica question that security reviewers ask first is "what can someone do on it?" On engines that replicate physically, roles and grants live in the cluster's catalog, and that catalog is copied along with the data. A role is defined once, on the primary, and it appears on every replica. A replica is read-only, so a role cannot be created, altered or granted on it alone. It follows that a role cannot be made safer on one replica: narrowing one reader's surface on a replica needs either the connection layer or a separate copy. The privileges you designed in Chapter 3 are in force on every replica, and which database a login may reach is a rule of the connection layer.

That has a useful side. `human_readonly` withholds Restricted columns and masks Confidential ones where Chapter 12 says so, and it does so on the replica for the same reason it does on the primary. `app_reporting_job` reads on a replica with no access to Restricted columns. If the grants were built with least privilege in mind, a replica gives a reader no more than the role allows.

It has a risky side as well. In the starting state the replicas accept the shared admin login, which owns every table. A reader on a replica with that login sees every column, which is the same exposure as the primary minus the ability to write. People tend to treat replicas as safe for exactly this reason, and the result is that the replicas are where the broadest access lives.

Pointing each login at its replica

The target state gives each replica a purpose, and the purpose is enforced where connections are accepted. Two terms are used in fixed senses from here on. A **network policy** says which hosts may open a TCP connection to a database host; Chapter 7 writes one. A **connection policy** says which login may authenticate to which database, and from where.

The connection policy needs a mechanism, because the grants cannot carry it and the network policy cannot see logins. On engines that have one, the mechanism is the host-based access rule that each server reads from its

own configuration. That rule is keyed on the source of the connection and on a login pattern or role, and it is not part of the replicated data, so the primary and each replica can hold different rules. The block below is one such policy, written as the rules the three servers would hold.

```
connection policy: one rule set per server, not replicated
  ledgerly_replica_1:
    allow from pooler hosts      login app_billing_api,
app_admin_console // browsing, support lookups
    deny everything else

  ledgerly_replica_2:
    allow from pooler hosts      login app_reporting_job,
app_vendor_export
    allow from gateway hosts    login pattern: kind readonly //
human_readonly, Chapter 7
    deny everything else

  ledgerly_primary:
    deny from anywhere          login app_reporting_job,
app_vendor_export
    allow from gateway hosts    login pattern: kind jit, kind
breakglass
    deny from gateway hosts    login pattern: kind readonly //
routine reads use replica_2
```

The cost is that three rule sets now have to be kept in step with the role schema, and a change that lands on one server and not another is drift. The rule sets belong in the same pipeline that applies role definitions (Chapter 3), and a check that compares each server's rules with the policy belongs in the same review. Where an engine has no such rule, the same decisions fall to the pooler and the gateway, and to the network policy that admits them.

The pairing of `human_readonly` with `ledgerly_replica_2` means a person investigating a problem reads from the same replica that reporting uses. That is a deliberate trade: one replica to protect, monitor and size, in exchange for human investigation competing with the nightly job for capacity. If the two start to interfere, a third replica for humans is a sensible split, and the connection policy changes by one block.

The denials on the primary matter as much as the allowances on the replicas. `app_reporting_job` and `app_vendor_export` have no business on the primary. If their logins cannot connect there, the "sometimes reads the

primary" behavior of the starting state is no longer something a configuration mistake can bring back. A job that is pointed at the wrong host fails to connect, which is a visible and cheap failure.

The exception in the other direction is a JIT or break-glass session. Every human session login is a member of `human_readonly` by default, so the primary rule cannot be written against that role: it would either cut off break-glass or mean nothing. It is written against the kind of session the access gateway opened. A routine session is sent to `ledgerly_replica_2`. A session that needs to write, or to read in an emergency what a replica cannot show quickly enough, is opened on the primary, and the request for access is made from that session (Chapters 7, 8 and 10 describe the model).

That leaves a window on the primary: until a request is approved, such a session can do what `human_readonly` does. Three rules keep the window small, and Chapter 7 states them as part of the session kinds. Only people whom policy allows to request that kind may open it. The reason is logged when the session opens, and the session has a short opening window in which a request has to follow. An alert fires if none does. A person who only wants to look at data does not open such a session; that is what the replica is for, and it is why routine reads are not allowed on the primary.

The replica has a host, too

A physical replica is a copy of the data files, so its hosts, volumes, snapshots and replication credentials reach the same data without passing through a database role. I would apply the same host access rules, storage encryption and retention to replicas as to the primary, and write that down when the replica is created, because nobody will think of it later. Chapter 13 returns to snapshots and backups.

Reporting, analytics and the choice of copy

Replicas solve a load problem for reads that are cheap and frequent. Reporting is the opposite: reads that are rare, large and slow. The question is whether reporting belongs on a replica or on something else, and the answer depends on what the reports need.

Three places a report can read

Option	Freshness	Load on the primary	Access surface	Fits
A reporting replica (<code>ledgerly_replica_2</code>)	Seconds to minutes	Replication only	Same data and roles as the primary	Operational reports, exports, investigation
A separate copy loaded on a schedule (<code>ledgerly_analytics</code>)	As old as the last load	One load per schedule	Its own roles and its own data	Models, dashboards, bulk analysis by the data team
A copy fed by change data capture	Seconds to minutes	A reader of the change stream	A new pipeline and a new store	Near-real-time analytics, many consumers

I would send operational reporting to a replica and bulk analysis to a separate copy, and I would resist putting the data team's work on the replica for a reason that is not about load. A replica carries the shape and the sensitivity of the production schema. The data team does not need Restricted columns, and in the target state does not see raw Confidential columns either, because the analytics copy is masked as it is loaded, except the display hints that exist to be shown (Chapter 12 describes how and names the exception). A replica cannot be masked per consumer without changing what the replica is. A separate copy can be built for its audience.

That is also why the analytics copy is not called a replica anywhere in this book. It is loaded on a schedule from the primary, it has different roles, and its content may differ from the source. Calling it a replica invites the assumption that it carries the same controls, and in the starting state it carries fewer.

Change data capture is the option I would reach for last. It adds a pipeline that has to be kept running, a store that has to be secured, a lag of its own and a schema contract between producer and consumer. For a company the size of Ledgerly I would take the scheduled load first and write down what would make me change my mind: for instance, reports that need data newer than the schedule allows, or a second team with the same need.

The cost of each copy

Every option in the table has the same hidden item: a new thing to protect. A copy is a place where the data lives, a set of logins that can read it, a network path, a retention rule and an owner. A copy without an owner is a side door, and a copy with an owner and no expiry is a side door that takes longer to notice.

It helps to keep a short register of copies, with the same fields for every one. The block below is illustrative notation, and the point of it is that the `owner` and `ends` lines cannot be blank.

```
table data_copy:
  name          text primary key    // "ledgerly_replica_2"
  kind          text not null       // replica, analytics, export,
dump, staging
  source        text not null       // what it was made from
  contains      text not null       // data classes present:
Restricted, Confidential, ...
  readers       text not null       // roles or logins that may
read it
  owner         text not null       // a team, never a person
  ends         timestamp null      // null only for replicas and
the analytics copy
  review_due    timestamp not null
```

For Ledgerly the register starts with five entries: the two replicas, the analytics copy, the export files that go to the accounting vendor, and the old dumps in shared storage. Four are on purpose and the dumps are not, which is why the register is useful: the dumps get an owner and a near end date. Chapter 13 uses the same register for the copies that live outside the databases.

Note: A register that nobody opens is not a control, so it is tied to the step that creates copies. The provisioning step refuses to create a replica, a snapshot restore, a staging database or an export job unless a row exists with an owner and, where the kind requires one, an end date. The access review (Chapter 14) then asks which copies exist and who can read each.

The refusal has a cost: someone fills in a row before a team with a deadline can get its copy, and the first weeks produce complaints. It also covers only copies made through the provisioning step, so a copy made by hand

is still a side door, and Chapter 13 deals with it.

Reach, not only grants

A role that is correct on paper is still wrong if the wrong login can connect. For every copy, the grant withholds what the audience does not need (Chapters 3 and 12), and reach decides which network paths and principals can open a connection at all. On a flat network a leaked login on any service reaches every copy, and reach bounds what a leak is worth, as Chapter 4 says. The table states the reach rule for each place; the network policy that enforces the host side is Chapter 7's, and the connection policy above enforces the login side.

Place	Reachable from	Must not be reachable from
ledgerly_primary	Pooler hosts, the gateway for JIT and break-glass sessions, the pipeline step that applies migrations	Reporting and export jobs, routine human sessions, the web tier directly
ledgerly_replica_1	Pooler hosts, for the services that browse	The gateway for routine reads, the analytics load, the web tier directly
ledgerly_replica_2	Pooler hosts for reporting and export, the gateway for routine human reads	Services that run the application, the web tier directly
ledgerly_analytics	The load job, reporting-job for its results, the gateway for analytics sessions	The services that run the application

Migrating Ledgerly's read path

Consider how Alex would move Ledgerly from the starting state to this design without breaking the nightly job on the first try. The order matters more than the speed, and I would take it in five steps.

1. **Inventory the reads.** For each service, list the queries it runs and sort them into reads that need the primary and reads that do not. Most of `billing-api`'s listing queries are the second kind. Its writes and its post-write redirects are not. The query logs are the raw material for the routing rules.

2. **Give each replica a purpose.** Write the connection policy for `ledgerly_replica_1` and `ledgerly_replica_2` and apply it to the new logins first, leaving the shared login in place for the moment.
3. **Move reporting-job and the export step.** Point both at `ledgerly_replica_2` with their own logins, and then deny those logins on the primary. Run them side by side for a few cycles and compare the outputs before turning the old path off.
4. **Move browsing reads.** Route `billing-api` and `admin-console` browsing reads to `ledgerly_replica_1` behind a flag, and raise the share gradually.
5. **Close the shared login on the replicas.** Once nothing legitimate uses it there, remove its ability to connect to either replica. This is the step that makes the rest permanent, and it is the one teams delay.

The data team's path changes too. Jordan's month-end question now has two answers. If a day-old number is fine, it goes to `ledgerly_analytics` through `human_analytics_reader`. If it needs today's data, it goes to `ledgerly_replica_2` through the access gateway with `human_readonly`, a session that is logged, that cannot see Restricted columns and that cannot touch the primary. What Jordan loses is the ability to point a desktop client at any host and have it work. What Jordan gains is that a heavy query is a problem for the replica and a mild one for everyone else.

Protecting a replica from its readers

A replica that serves humans and a nightly job will see heavy queries, and on some engines a long query can delay replication or be cancelled so that replication can continue, depending on configuration. The controls are the ones from Chapter 5, applied to the replica's roles: a shorter statement timeout for `human_readonly` than for `app_reporting_job`, a per-login connection limit that the gateway sets on each human session login, and an export step that runs outside the nightly aggregate window. Add a lag alert, and send the page to whoever owns the replica in the register. An alert nobody receives is not a control.

Trade-offs and costs

A replica is a system to run. It has to be monitored, patched, sized and recovered, and for a small team one replica and a nightly copy may be all that is needed. The access argument still holds without it: you decide which logins reach the primary and which do not.

Sharing ledgerly_replica_2 has a price. People and the nightly job compete for one replica's capacity, in exchange for one place to protect. The split into a third replica is cheap in policy terms, and expensive in operations.

Connection policy is more configuration. Per-server rules have to match the role schema and survive a failover, a rebuild and a new replica. Drift between servers is the failure, and the comparison check is the price of keeping it visible.

Some screens will be behind. Moving reads off the primary means lag, and someone will notice. Agree with support and product in advance which screens may be behind, and check the lag first when something does not add up in an investigation on a replica (Chapter 15).

The register adds friction. Requiring a row before a copy exists slows the team that wanted the copy, and that slowness is the intended effect. A register that nobody has to fill in at creation is a list of what someone remembered.

There are cases where this chapter's advice does not apply. A small system whose primary has headroom and where the engineers who read from it already connect through a controlled path may not need a replica for load, and separating the read path has a smaller payoff. A system with very strict freshness needs for every read may have to run most reads on the primary and put its effort into limiting what those readers can do. A replica is also not a backup; Chapter 13 covers backups.

Common mistakes

- **Treating a replica as less sensitive than the primary.** It cannot be written to, so it collects broad access: the shared login, a standing read role for everyone, a wider firewall rule. It holds the same data, so it gets the same grants and an equally narrow connection policy.

- **Letting reporting or human reads fall back to the primary.** Falling back is tempting because it keeps the report running when the replica is down. Fail the read instead, and deny those logins on the primary so that a mistake cannot send them there.
- **Sending reads that decide something to a replica.** The read is cheap and the replica is idle, so it looks like an easy win. A decision made on stale data and followed by a write is how balances and statuses drift. Sort reads by what follows them, not by how heavy they are.
- **Building a warehouse by pointing a human at a replica.** The data team then carries production's schema, production's sensitivity and production's capacity limits. Give analytics its own copy and its own roles (Chapters 12 and 13).
- **Adding copies without a register.** Each one was a favor to a team with a deadline. Six months later nobody can say how many exist. Make the owner and end date part of creating a copy.
- **Forgetting the host and the snapshots.** The roles are right, and the volume snapshot of the replica is readable by anyone with infrastructure access. Include the host, the storage and the snapshots in the controls for every copy.
- **Assuming a role can be tightened per replica.** On engines that replicate physically, roles are cluster-wide. Narrow one reader's surface at the connection layer or with a separate copy.

Key takeaways

- Writes need the primary and most reads do not, so the first question for the read path is whether a read decides something or follows a write; reporting and human reads never fall back to the primary.
- "Read-only" is three mechanisms, and only the replica cannot be undone by a session, so a read-only role limits writes and not what a session may read.
- A replica holds the same data as the primary and is governed by the same cluster-wide roles, so the grants from Chapter 3 and the connection policy together decide what a reader can see.

- The connection policy is a per-server rule keyed on source and login, kept in step with the role schema; each replica has one purpose, and the shared admin login is accepted on none of them.
- A JIT or break-glass session on the primary needs a person who may open it, a logged reason, a short window and an alert when no request follows.
- Analytics belongs on a separate copy built for its audience, and calling it a replica hides the fact that it carries different controls.
- Every data copy needs an owner, a reader list and, except for the replicas and the analytics copy, an end date, and the step that creates copies refuses to run without them.

The read path now has its own roles, its own destinations and a short list of copies, but we have not yet said how a person gets to any of them. The next chapter takes up the paths in for humans, and compares a bastion, a database proxy and an access gateway.

PART III

Human Access

Chapter 7 — Paths In: Bastions, Proxies and Access Gateways

Imagine that Alex, from the platform team, is asked a plain question during a planning meeting: "If we turned off one path into the production database tomorrow, which one would hurt?" Nobody answers quickly. Alex goes to find out, and starts with the network. The VPN gives every engineer a route to the primary. There is also a bastion host that someone set up for a contractor two years ago, and it still has a listening port. The desktop clients on a dozen laptops hold saved connections, and one of them was set up to tunnel through the bastion, while another goes straight over the VPN.

Then Alex asks Robin, who is on call this week, how Robin would reach the database at night. Robin names two paths without hesitating and a third after thinking about it. Each path works, none was designed, and each one ends in the same shared admin login. The network team can say who is on the VPN. The database can say that the admin login connected. Nobody can say which person was behind a given connection, and nobody can turn off one person's access without turning off everyone's.

So the question for this chapter is: what should stand between a person and the database, so that reaching the network and reaching the data are two separate decisions, each with a name attached? This chapter adds the access gateway, the first step of the human side of Ledgerly's target state. In the book's five parts it carries the Path, and the identity of whoever enters by it: the network decides who may knock, and the gateway puts a name on every session. The roles a gateway session receives, and how a person asks for more than reading, belong to Chapter 8.

The problem, and why it is hard

A network path and a database path look like the same thing, because the first one is what you need before you can try the second. A person who is on the VPN can open a connection to the database port. Whether that person may then do anything is decided by a username and password, and in Ledgerly's starting state the password is shared. The network said "you may knock", and the database could not tell who knocked.

That is the heart of the problem. Network controls answer the question "which machines may talk to which machines", and they do it well. They do not answer "which person is this, what are they allowed to do right now, and what did they do". When a network rule is asked to stand in for those questions, it fails in a quiet way: the rule stays correct while the access it grants goes stale. An engineer who changed teams is still on the VPN. A contractor whose engagement ended is still in a group that nobody reviews.

Three more things make the human path harder than the service path.

First, people are the most varied clients. A service connects from a known place with a known driver. A person might use a command-line client, a desktop GUI, a notebook, a browser tool, or a script they wrote last Tuesday. Each has its own way of remembering credentials, its own default for auto-commit, and its own habit of downloading results to the local disk. Chapter 1 listed the clients that remember the password and auto-commit; that is not an unusual setup, it is the default behavior of many tools built for convenience.

Second, people connect from the least controlled places. A laptop travels, sleeps, gets backed up and is sometimes shared at home. A saved connection with a password in it moves with the laptop, and so does any result set that was exported to a file. The database cannot see any of that.

Third, the path is used under pressure. The engineer who needs the database at night is not in the mood to learn a new tool, and a path that is slow or fragile will be bypassed by whoever has the means. A design that is correct on paper and loses to the old shortcut at 3 a.m. has not changed anything, which is why the shortcut has to be removed and not only discouraged.

A policy that says "do not share the admin login" cannot be checked, because the database log shows the admin login and nothing else. A design in which every session has its own login can be checked, and so can a network rule that lets only one thing reach the database. The rest of this chapter compares the ways of building such a design.

Four paths in, compared

Before the comparison, two terms, because the words are used loosely in the industry and the book needs them fixed.

A **bastion** is a hardened host that is the single network path to a protected system. It forwards connections. It does not understand the database protocol, so it cannot tell a `SELECT` from a `DROP`, and it does not know who is behind the connection unless the person logged in to the bastion itself.

An **access gateway** is an identity-aware intermediary that authenticates a human, issues a short-lived credential, and records the session. A **database proxy** is something different: a protocol-aware intermediary that pools, routes or filters database connections. A gateway may be built on top of a database proxy, but the two answer different questions. A proxy knows the protocol; a gateway knows the person. This chapter uses each term only in this sense.

Here are the paths a team usually considers, with what each one knows about the person and what it leaves behind. The first four put a person in a session; the last two keep the person out of one.

Path	What it knows about the person	What the database log shows	Main cost
Public endpoint	Nothing beyond the database login	A login, from any address	Exposes the database port to the whole internet
VPN to the database	That the person is on the network	The shared or per-person login	Network membership is treated as identity
Bastion	Who logged in to the bastion host, if accounts are per person	The login the person typed, from the bastion's address	One more host to patch, and it sees no statements
Access gateway	Identity from the company's single sign-on	A session login that maps to one person and one session	A new system that must be available when someone needs it

Path	What it knows about the person	What the database log shows	Main cost
Product path, no SQL	The person's role in an internal tool	The service's own login, with the person in the tool's audit trail	Each task needs a feature built; free-form questions have no path
Submit-and-run executor	The person who submitted, and the reviewer who approved	The executor's login, running a statement set that was reviewed	Every task waits for review; nothing is interactive

The last two rows are not ways of reaching a database at all. The person never holds a session, and the statement runs from a component that checked it first. They belong in the comparison because for many tasks they are the better answer: a person who never opens a session cannot leave one open. Chapter 1 introduced the range of such patterns, Chapter 9 covers how the gateway checks a statement, and Chapter 10 covers the data-fix patterns; this chapter covers the sessions that remain after the tasks that fit those patterns have moved out.

The public endpoint

I would not expose the database to the public internet for human use, and the argument is short. A reachable port invites automated traffic that has nothing to do with Ledgerly's people, and every credential for that database then becomes something worth guessing. The only defense left is the strength of the login, and the book's premise is that logins leak.

There are narrow cases where a managed database is only reachable publicly, and the provider offers an allowlist of source addresses. If you are in that position, treat the allowlist as the minimum and move to a private path as soon as the provider allows it. The test is simple: could someone who has never worked for you open a connection to the port? If so, every other control in this book starts from a worse place.

The VPN

A VPN is a reasonable network layer. It narrows who can reach the database to people who have been given a route, and most companies have one already. Its limit is what it proves. It proves that a person, or a person's laptop, passed the VPN's checks. It does not say which database login they used afterward, and it does not record statements.

In Ledgerly's starting state, the VPN is the whole control. Everyone on it can reach the primary, and the admin login is the same for all of them. I would keep the VPN, but I would give it a smaller job: it carries people to the gateway and to the other internal tools, and it no longer carries them to the database port.

The bastion

A bastion adds one useful property: a single place where the network door can be closed. The database's firewall allows only the bastion's address, so turning off a bastion account turns off a person's network path. If the bastion has per-person accounts and records who logged in to it, you gain a rough record of who was connected and when.

What a bastion does not give you is anything below the network layer. Once a person has an open tunnel through it, the connection to the database is whatever the person's client makes it. The login is still typed by the person or saved in their client. The statements are invisible to the bastion. So a bastion tidies the network picture without fixing the identity picture, which is why a team can have one and still be unable to answer "who ran that statement?".

A bastion also has a quiet maintenance problem. It is a host with a shell, so it needs patching, key management, and an owner who remembers it exists. The contractor's host in the opening is the usual shape of the failure: it was correct on the day it was built and nobody was assigned to it afterward.

The access gateway

The access gateway is where I would send people, for the reasons below.

It authenticates the person against the company's single sign-on, so the identity is the same one the company already controls. Leaving the company, changing teams and failing a multi-factor check all take effect at the gateway without anyone editing the database. It issues a credential that belongs to one session and dies with it, so there is no password to save. And it records the session, so "who ran that statement?" has an answer in two places: the gateway's session record and the database log, which names a login that maps to exactly one person.

Gateways come in two broad shapes. Some sit in the data path: the person's client connects to the gateway, and the gateway opens the connection to the database on the person's behalf. Others only broker the credential and step aside, so the client connects to the database directly with the short-lived login. The first shape can record statements and enforce rules on the connection. The second is simpler and puts fewer things in the path, but then the network has to allow people to reach the database port, which brings back part of the problem this chapter started with.

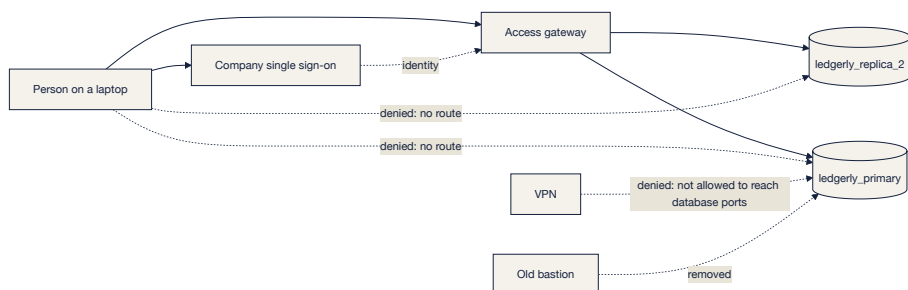
For Ledgerly I would choose the data-path shape, because it lets the network rule be absolute: only the gateway's address may reach the database hosts. The price is that the gateway handles every human statement, so its capacity and availability matter. I come back to that below.

Separating network reach from database access

The most useful change in this chapter is not a product. It is separating two questions that the starting state answers together.

The first question is **network reach**: which machines may open a connection to which database host and port. The second is **database access**: which login may do what, once connected. In the starting state, the VPN answers the first and a shared password answers the second. After the change, a network rule answers the first for everyone at once, and the gateway answers the second for each person separately.

The diagram shows the target arrangement for humans. Look at the denied arrows: they are the paths that exist today and will not exist afterward.



Two enforcement points sit behind that diagram. The network rule on the database hosts allows connections only from the gateway, from the services' own networks, from the pipeline and from the credential broker. And the gateway only forwards a session to the database that the session's kind allows: a routine read session goes to `ledgerly_replica_2`, and a session opened to make a JIT or break-glass request goes to `ledgerly_primary`, as Chapter 6 set out.

I would write the network rule down as a short list that anyone can read, because the list is what an access review checks. Chapter 6 defined the difference: this list is the network policy, which decides which hosts may connect at all, while the connection policy decides which login may authenticate to which database. If it contains an entry nobody can explain, that entry is a path.

```
network policy (illustrative):
  ledgerly_primary:
    allows access gateway
    allows service networks of billing-api, invoice-worker,
      payments-service, admin-console
    allows deployment pipeline
    allows credential broker, for access administration
    denies everything else, including the VPN

  ledgerly_replica_1:
    allows service networks of the read-only services
    allows replication traffic from the primary
    denies everything else, including the VPN

  ledgerly_replica_2:
    allows access gateway
    allows reporting-job network
    allows replication traffic from the primary
    denies everything else, including the VPN

  ledgerly_analytics:
    allows analytics-job network
    allows access gateway, masked views only
    denies everything else, including the VPN
```

The same list continues with what leaves the database hosts and what can reach the gateway.

```
network policy (illustrative, continued):
    egress from the database hosts:
        allows replication, the audit log, backups
        denies everything else

    the gateway's own exposure:
        reachable from the VPN and the office network, nothing else
        reaches the database hosts, the audit log, single
sign-on
```

Notice how little the VPN has to do in this model. It no longer guards the database. It carries people to the gateway, and the gateway decides the rest. The VPN matters less; each person's access is now tied to a named session.

Two entries in the list are easy to forget. Egress comes first. A database host that may open outbound connections anywhere can be used to send data out, and a short allowlist (replication, the audit log, backups) turns that into a visible event. The gateway's own exposure comes second: it is now the one reachable door, so it is itself on a short list, and it must not be reachable from anywhere the VPN does not already reach.

The list also assumes that the network can tell the callers apart. On a shared cluster network, where several services come from the same address range, it cannot: a rule that allows "the services' networks" allows all of them. In that case the network rule keeps out everything that is not a service or the gateway, and it is the pooler and the login that separate one service from another (Chapters 5 and 6). I would write that down as the rule for the cluster, so that nobody reads the network list as more precise than it is. Which login may authenticate to which database is the connection policy of Chapter 6, and this chapter only points to it.

The access gateway in Ledgerly

Here is what the gateway does, in the order it does it, and what happens when a step fails. This is the part I would test carefully, because a gateway that fails open becomes the new shared password with better branding.

What a session is

A session starts when a person authenticates. The gateway checks the identity, decides which kind of session the person is entitled to ask for (a routine read session for most people; JIT and break-glass kinds only through the requests of Chapter 8), and then creates a **session login**, the account the person's session uses. It is created on `ledgerly_primary` and replicates (Chapter 6). The session then connects to the database its kind belongs to. That login carries the person's identity in its name, so it appears in the database log, and it is a member of `human_readonly` by default. The credential for it is generated for this session, handed to the gateway and never shown to the person, so there is nothing to save.

A login name has a fixed shape, so that a log line can be read without a lookup: a prefix that says it is a session login, the person's short identifier, and the session's own number, as in `sess-robin-0412`. The name identifies; the gateway's record is what maps it to the person and the request, and nothing should parse the name to decide what the session may do.

To create and remove logins, and to grant and revoke role memberships, the gateway needs privileges in the database. It gets them through its own login for the role `ledgerly_access_admin`. The pipeline step for role definitions and the credential broker also use that role, each through its own login (Chapters 3 and 4). That makes the gateway's login a sensitive credential: whoever holds it can grant anything inside the roles it administers. I would keep it in the same place and under the same rules as the owner's credentials, give it no data privileges, and audit every use of it (Chapter 14). Copying it into scripts or chat for convenience would recreate the shared password one level up.

Who may open which kind of session

`mayRequestKind` is policy, and I would keep it as data that a reviewer can read, not as a condition buried in the gateway's code. One table answers it: the single sign-on group, the kinds that group may open, the longest each may last, the database it lands on, its defaults (read-only, statement timeout, row cap) and whether it may export.

```

session policy (illustrative):
  group backend-engineers:
    routine      max 8 hours, ledgerly_replica_2, read-only, row
cap, no export
    primary-bound max 2 hours, ledgerly_primary, only to file a
request
    group analysts:
      routine      max 8 hours, ledgerly_analytics, masked views,
export logged
    group contractors:
      routine      max 4 hours, ledgerly_replica_2, masked views,
no export

```

The primary-bound kinds are the sensitive ones, since a session on the primary is one request away from write access. Four rules keep them rare: only named groups may open one; it must be opened with a logged reason; the gateway waits only a short time, say fifteen minutes, for a request to be filed from it; and a session that ends without a request raises an alert, because "opened the primary and did nothing" is a pattern worth noticing. The two-hour cap holds only until a grant: when Chapter 8's `grantJitAccess` or `declareBreakGlass` approves a request, the gateway resets that session's `ends_at` to the request's `expires_at`, and `endSession` below closes the session at that time. Chapter 8 starts from the other side, with the request.

The order of operations

The gateway, the audit log, the database and the single sign-on are separate systems. No transaction spans them, so the order of steps and the handling of a half-finished session have to be designed.

```

function openSession(person, wantedKind):
    identity = verifySingleSignOn(person)
    if not mayRequestKind(identity, wantedKind):
        raise SessionRefused
    login = newLoginName(identity)
    endsAt = now() + maxSessionLength(wantedKind)

    // the record comes first: a login with no record is what the
    sweep hunts
    recordSession({ login: login, identity: identity, kind:
wantedKind,
                    ends_at: endsAt, status: "opening" })

    try:
        auditLogAppend({ actor: identity, action: "session-open",
                        subject: login, ends_at: endsAt })
    on AuditLogUnavailable:
        markSessionRefused(login)
        raise SessionRefused

    try:
        // always on the primary: logins are cluster-wide and
    replicate
        createLogin(login, on: "ledgerly_primary", password:
freshSecret())
        grant("human_readonly", to: login)
        // replication lag: wait until the target database knows the
    login
        waitUntilLoginVisible(login, on: databaseFor(wantedKind))
    on DatabaseError:
        // a login with no role is harmless, but it must not linger
        dropLoginIfExists(login)
        markSessionRefused(login)
        raise SessionRefused
    markSessionOpen(login)
    return connectionFor(login, databaseFor(wantedKind))

```

Read the block for its failure handling. The record is written before the login exists, so a crash in the middle leaves a record that says "opening" and no login, or a login that the record explains. The audit entry comes before the login: if the log store is unreachable, no session opens. That choice has a cost, which the trade-offs section names. For a routine read session the gateway waits until `ledgerly_replica_2` has received the login, because replication lag would otherwise make the first connection fail now and then. If creating the login or granting the role fails, the gateway tries to clean up and refuses the session. If even the cleanup fails, a login with no privileges exists, and the sweep below removes it.

Ending a session

A session ends in one of three ways: the person disconnects, the idle limit passes, or `ends_at` passes. The last two matter because people leave tabs open. An open connection outlives an expired credential (Chapter 4), so the gateway closes the connections itself and then removes the login. The same function serves a person who leaves the company: a deprovisioning event from single sign-on calls `endSession`, and the maximum session length is the bound for the case where that event never arrives.

```
function endSession(login, reason):
  // the single owner of a session's end; Chapter 8's expiry calls
  it too
  try:
    closeConnections(login)
  on CloseFailed:
    // a drop that fails on a live connection is retried by the
    sweep
    notify("platform", "connections not closed: " + login)
    // requestsOf returns only requests not yet marked revoked
    for each id in requestsOf(login):
      try:
        revokeRequestAccess(id, from: login)
      on LoginNotFound:
        // nothing is left to revoke
      on RevokeFailed:
        notify("platform", "session grants not revoked: " + login)
        return
    markRevoked(id)
    auditOrQueue({ action: "revoke", actor: "gateway",
                  subject: login, request: id })
  try:
    revokeIfExists("human_readonly", from: login)
    dropLoginIfExists(login)
  on DatabaseError:
    notify("platform", "session login not removed: " + login)
    return
  markSessionEnded(login, reason)
  auditOrQueue({ action: "session-end", subject: login, reason:
    reason })
```

Here `auditOrQueue` appends the record to the audit log and, if the log is unreachable, queues it for retry and notifies the platform team. The revoke and the drop already happened by then, so a log outage delays the entry and never the access removal. A login that is already gone is treated as success: a session that Chapter 8's expiry path ended earlier is closed cleanly here, its record is marked ended and no alert fires.

The sweep calls `endSession` for every session that should already be over.

```
function sweepSessions():
    // endSession reports a stuck login and returns, so one failure
    // does not stop the loop
    for each s in sessionsPastEnd(before: now()):
        endSession(s.login, "expired")
    for each s in sessionsStuckOpening(before: now() - openingGrace):
        endSession(s.login, "abandoned")
    for each login in loginsWithoutRecord():
        endSession(login, "orphaned")
```

The order is the point. A request's grants are revoked from its stored record and marked revoked only after the revoke worked, and only then is the login removed, because some engines refuse to drop a login that still holds privileges. A failed revoke or drop notifies the platform team and leaves the record for the next sweep; a failed audit write is queued and does not stop the loop. The sweep gives a session that is still "opening" a grace period, so it does not remove a login that is only seconds old. Chapter 3's drift check, which compares the logins that exist against the gateway's records and the grants against the stored triples, runs on its own schedule, so one stopped job does not hide the other.

Which database, which role

Every session lands where its kind belongs. A routine session is a `human_readonly` login that connects to `ledgerly_replica_2`. A session opened to make a JIT or break-glass request is opened on `ledgerly_primary`, and Chapter 8 shows how the request is made from that session and how the grants end with it. The gateway enforces the destination rule that Chapter 6 wrote as a connection policy: it is the gateway that knows the kind, and the database host that refuses a login meant for another one.

A person can have two sessions open at once, one routine and one with a request. I would keep that possible and keep the sessions visibly separate, so that nobody uses the privileged one for a casual lookup. The human cost is a second connection in the person's client. The benefit is that the privileged session is easy to find, short, and easy to end.

What the gateway records

Because Ledgerly's gateway sits in the data path, it can record each statement a session sends, along with the start, the end and the rows affected. I would store that record where the people it describes cannot edit it; Chapter 14 covers where. One record looks like this:

```
statement record (illustrative):
  login:      sess-robin-0412
  person:     robin
  kind:       jit
  request:    req-0412
  fingerprint: UPDATE invoices SET status = ? WHERE id IN (?)  //
  literals redacted
  param_hash: 9c1e...
  rows:       12
  started_at: 03:21:07
  ended_at:   03:21:07
```

Payment-data and financial-reporting regimes tend to ask who touched production data and by what path; the session record is that answer, and the reader's legal and compliance people own the interpretation. Two limits are worth knowing. A statement log tells you what was asked, not what was returned, and I would not store full result sets, because they would turn the log into a copy of the data. The fingerprint carries the shape of the statement and a hash of its parameters instead of their values, for the same reason. And a log that nobody reads records without protecting anything, so decide who looks at it, and when: a weekly sample of sessions that held a JIT request is a start, and the access review is the other occasion.

Capacity and the connection budget

Every human session is also a database connection, and Chapter 5's budget gave humans a fixed slice, say twenty connections, as an illustration. Routine engineer sessions all land on `ledgerly_replica_2`, so that replica is the host of the human slot, and its connection limit is where the number is finally enforced. The gateway should enforce that slice itself: a limit per person, a limit per kind of session, and a clear refusal when the budget is spent. A refusal at the gateway is a message to a person. A re-

fusal at the database, when the budget has run out, can crowd out a service. Putting the limit at the gateway keeps the human budget from becoming the service's problem.

The laptop and the local copy

The gateway controls the path to the database. It does not control what happens to the data once it arrives on a laptop, and that is where a quiet part of the risk lives.

Two things stay on the laptop. One is the connection setup: the saved host, the saved login and, in the starting state, the saved password. The other is whatever the person does with a result: an exported file, a pasted table, a local copy of a few thousand rows to "play with". Each is a data copy in the sense of Chapter 6, a place production data lives that has no owner and no expiry.

I would handle these in three layers, from cheapest to most expensive.

Remove what the client can remember. Because session credentials are generated for one session and never shown, a desktop client has no password to save. The saved connection now holds a gateway address and the person's single sign-on, and the old saved entries for the primary can be deleted during the migration, with a short list of what to delete and where. This layer costs almost nothing and removes the most common leak.

Choose defaults that fit the task. The gateway can open sessions with read-only defaults and a statement timeout from Chapter 5. It can also cap the rows a statement returns: it counts rows as it streams them, closes the cursor at the cap, tells the person, and writes the event to the statement record; a session that hits the cap repeatedly is what the weekly sample looks at first.

Defaults that the person can lift do not stay defaults. A client can send a `SET` that raises the timeout or turns read-only off, so the gateway keeps a list of guarded settings and refuses any statement that changes them. The cost is that some clients send such statements on connect and will break; that is a cost to name in the migration, and Chapter 9 covers how the gateway decides which statements to refuse. Clients that auto-commit are

less dangerous when the session has no write privileges to commit with. A browser-based console, run by the gateway itself, goes one step further, because nothing is downloaded unless an export button exists and is logged. That console is slower and less pleasant than a native client, and I would offer it as an option for people who only look and never need their own tools, not as a requirement.

Decide what leaves, explicitly. Whether a result may be exported from a session is a rule of its own, and it belongs with the masking and minimization work of Chapter 12 and the export paths of Chapter 13. The gateway can apply the rule, but it cannot make it unnecessary. A person can read a screen, take a photograph of it or retype a value, and I do not know a cost-effective control for that. The gateway reduces the amount of data that leaves by accident and makes deliberate exports visible; it does not make leaving impossible.

Note: If your engineers need large extracts for analysis, the answer is usually not a bigger export limit on the gateway. It is a masked analytics copy designed for that purpose, which Chapters 6 and 12 describe. The gateway is the wrong tool for bulk data movement.

One door, one point of failure

A single entry point is easy to explain, easy to log and easy to close. It is also a single point of failure, and it is worth saying that plainly before recommending it. If the gateway is down, nobody can reach the database through the front door, including the on-call engineer who needs the database most when something else has already broken.

There is no answer that removes this cost. There are answers that move it somewhere you can handle.

Keep the gateway small. It authenticates, creates logins, forwards and records. Every extra feature is another reason for it to be down. I would resist adding result transformation and an approvals interface to the same service, and run those as separate components that the gateway calls but does not depend on to start a read session. Where a statement is checked before it runs, that check is a component of its own too (Chapter 9).

Run more than one instance, and put the state somewhere durable.

Two instances behind the same address, sharing the session records, mean a restart or a bad deploy does not end every session. The session store and the audit log are dependencies too, and their availability is part of the gateway's availability. That is the reason the opening function above refuses a session when the log is unreachable: it is a deliberate choice that ties human access to the log, and it should be a choice that the team made on purpose and wrote down.

Decide what the degraded modes are. There are two failures to plan for. One is the gateway being down while the databases are fine. The other is the audit log being down while the gateway works. For each, someone accountable has to have decided in advance whether human access stops or continues by a narrower route, and what is recorded in the meantime. The book's decision for a gateway outage is that there is no second door: human access stops, break-glass included, since a break-glass session is a gateway session with a heavier role. The way to keep that rare is the redundancy above, not an emergency login outside the gateway. Chapter 8 gives the reasoning and what a team does if it decides otherwise, and Chapter 15 describes rehearsing the outage. For an audit log outage, reads on a replica may buffer entries and writes may not (see the trade-offs below).

Treat the gateway as production. It has an owner, an on-call path, a health check that a person can read, and a deploy process with a rollback. The platform team owns it. If the gateway is a side project of one engineer, the single point of failure is that engineer.

The gateway also concentrates a different risk: a compromise of the gateway is a compromise of access to everything it can reach. That is why its `ledgerly_access_admin` login is guarded as described earlier, and it is no reason to avoid the gateway: the starting state has a shared password, copied to everyone on the team, in the same position.

Ledgerly's move from many paths to one

A team in Ledgerly's position would not switch everything on one day. I would take the steps in an order that keeps a working path available at each point.

1. **Inventory the paths.** List every way a person can reach any Ledgerly database today: the VPN routes, the bastion, the saved client entries, any scripts that connect with the admin login. Alex's walk in the opening was this step, and it should end in a written list that the platform team keeps.
2. **Stand up the gateway beside the old paths.** At first it creates session logins on the primary, where logins are defined, grants them `human_readonly`, and connects them to `ledgerly_replica_2`, and nothing else. The old paths still work. A few people use the gateway for their daily reading and report what is awkward.
3. **Move reading first.** Reading is most of what people do, and it carries the least risk to try. Once the awkward parts are fixed, close the replica to the VPN and delete the saved entries that pointed at it.
4. **Move the primary last.** Direct access to the primary is what people lean on during an incident, so I would close it only when the JIT and break-glass routes of Chapter 8 are working and rehearsed. Closing it earlier would take the emergency path away before the replacement is ready.
5. **Remove the old bastion and the admin login's reach.** The contractor's bastion is turned off. The shared admin login stops being accepted from human networks. Services keep their own logins until Chapter 4's work replaces them.
6. **Test the denials.** For each former path, someone tries it and confirms that it fails. A denied path that nobody tested is an assumption.

The cost of the migration is mostly human. People have to learn a new connection flow, and some of their scripts will break because they assumed a standing login. Plan a window during which both work, and write down the date when the old path closes. Without a date, the old path stays open, because it always seems cheaper to leave it.

Trade-offs and costs

A smaller design that may be enough. A gateway is a new system in the critical path, and if you cannot staff its on-call path and its deploys, a smaller design is the better start. If four engineers reach one database over a private network, using personal, short-lived logins issued by something

your team already runs, you may already have most of the value. I would write down what would make you change your mind: a contractor joining, a second database, a request for session recordings, or the first incident where the log could not say who did it. The gateway is a response to those conditions, not a milestone.

Failing closed has a price. Refusing a session when the audit log is unreachable keeps the record complete and makes the log store a dependency of every human session. Some teams prefer to open read sessions with a local buffer and send the entries later. That is a defensible choice for read-only sessions on a replica, and a weaker one for sessions that write. If you choose it, say which sessions it applies to, and monitor the buffer.

Statement recording is a data copy. The fingerprint record redacts literals, but it still names tables and columns, and any field left unredacted, such as a free-text note or a client-supplied comment, can carry a value. Treat the log with the classification of the most sensitive thing a record might contain, restrict who reads it, and set a retention period that someone owns.

Latency and fidelity. A gateway in the data path adds a hop and may not support every client feature or every protocol extension. Some tools behave oddly through an intermediary: very large transfers, long-lived cursors, and unusual authentication flows are the usual places. Test the tools your people actually use before you promise that the old path will close.

When not to use a gateway. A single-service prototype with no customer data does not need one. A system whose only human access is through a managed console that already provides identity, short-lived credentials and logging may not need a second layer in front of it. A bastion is a reasonable answer when the real need is a single place to close the network door and when the protected system has its own strong, per-person authentication. In both cases the same questions apply: who is the person, how long does the access last, and where is the record.

Common mistakes

- **Treating the VPN as the identity layer.** It is tempting because the VPN already knows who is on it and everyone has it. It proves network membership, not which database login was used or what it did. Keep

the VPN as the way to reach the gateway and add a layer that attaches a person to each session.

- **Putting a bastion in front of the database and calling the work done.** It tidies the network and leaves the login shared and the statements invisible. Choose a bastion for what it does, a single closable network door, and look for identity and recording elsewhere.
- **Leaving the old path open during the migration.** It seems prudent to keep both paths for a while, and it is, but without a closing date the old one outlives the new one. Write the date, announce it, and test that the old path fails afterward.
- **Opening a session whose end nobody enforces.** A session limit that exists only in documentation is a standing grant with a longer name. End sessions at `ends_at`, close the connections as well as dropping the login, and run a sweep that finds sessions and logins that should not exist.
- **Skipping the degraded-mode decision.** A team that builds a gateway without deciding what happens when it is down will decide under pressure, usually by reopening the old shared path "for a minute". Decide the emergency route in advance, write it down and rehearse it.
- **Recording everything and reading nothing.** Capturing every statement feels like diligence. Without an owner who samples it and an access review that looks at it, the log is a data copy with no purpose. Decide who reads it and what they are looking for.
- **Reading the network list as more precise than it is.** On a shared cluster network the rule cannot tell one service from another, and a reviewer who assumes it can will approve a path that is wider than it looks. Write down which control separates the callers instead.

Key takeaways

- Network reach and database access are separate questions, and a design that answers both with the same control cannot say which person did what.
- A bastion makes the network path singular but does not understand the database protocol, so on its own it leaves the login shared and the statements invisible.

- An access gateway authenticates a person through single sign-on, creates a session login that carries that identity, and removes it when the session ends, which is what makes the database log name one person.
- The network rule that allows only the gateway to reach the database hosts is the control that makes the gateway the only door, and it should be a short list that an access review can read.
- A single entry point is a single point of failure, so the gateway should be small, run in more than one instance, owned by the platform team, and paired with a degraded-mode decision made before the incident.
- The gateway cannot control what a person does with data that has reached a laptop; it can remove saved credentials, set read-only defaults and make exports visible, and the rest belongs to masking and minimization.
- Migrate reading first, close the primary last, set a closing date for each old path, and test that the denied paths are denied.

The gateway gives every person a session with a name on it and a default of reading, but reading is rarely the whole job. The next chapter looks at what happens when someone needs more than `human_readonly` for a limited time: just-in-time access, break-glass access, and what the approval costs the people who wait for it.

Chapter 8 — Just-in-Time and Break-Glass Access

Imagine a Tuesday night at Ledgerly, a little after three in the morning. An alert says that a batch of invoices has been stuck in the status `sending` since the `invoice-worker` restarted, and the reminder emails for the morning run will not go out until the batch is cleared. Nothing is exposed and nothing is lost; the cost of the problem is a delay. Robin, the on-call backend engineer, opens a routine session through the access gateway. The session is read-only and runs on a replica, which is what Robin wants for the first ten minutes: look, count, understand.

The look confirms the problem, and the fix needs an `UPDATE` on one column of `invoices`. Robin writes the one-statement fix down and files a request for it. The approver is the secondary on-call engineer, who is asleep, and the page has not been acknowledged. Robin has a laptop, a runbook and a decision to make: wait for an answer, or find a way around the question. In the old Ledgerly the decision would not have existed, because the shared admin password was one paste away. In the new Ledgerly the way around has to be designed, and so does the way back.

This chapter adds the next target-state steps from the running example: humans have no standing write access, writes arrive as just-in-time grants inside the ceiling from Chapter 3, and an emergency role exists that is rarely used and always reviewed. In the book's five parts it is mostly the Grant: who approves what, for which statement, for how long. It answers Robin's question: how does access appear when needed, how does it end with the task, and what happens when the normal path is too slow?

The problem, and why it is hard

Standing access is a promise that nobody will need to ask again. It is convenient for a month, and after that nobody remembers what the promise covered. People change teams and keep their grants; a role widened for

one incident stays wide. When an access review arrives (Chapter 14), the reviewer faces grants with no task attached, and the answer to "is this still needed?" is "probably". The cost of a standing grant is every day it is not used, because it still carries the blast radius of whatever it covers.

The obvious replacement is **just-in-time (JIT) access**: access granted for a specific task and a limited time. It is easy to state and easy to get subtly wrong in five places.

Scope. The easy implementation grants a role to a login for an hour. If that role is `human_jit_write`, the login receives the whole ceiling, update rights on six tables for a task that needed one column of one, and the grant is a standing one with a timer attached. The scope has to live in the grants themselves, one request at a time.

Expiry. Grants do not expire on their own on common engines. Something has to take them back, usually a scheduled job, and a job can stop, lose its credential, fail on one login and carry on, or run against the wrong list.

Approval. A second person reading every request has a human cost: someone is woken, someone waits. If approvals are too slow, people find the faster path, which is the shared password the book is trying to retire. If the approver always answers within seconds, approval no longer reduces exposure.

The emergency. An access scheme with no sanctioned way to move quickly grows an unsanctioned one. The question is whether the fast path was designed, logged and reviewed, or improvised.

The operation. A grant names a table and a column and says nothing about which rows or which statement. An approver who reads "`UPDATE on invoices, status`" has approved a capability, and whatever runs inside it is outside the approval. For approval to mean "a second person saw what will happen", the request has to name the operation and the grant has to follow from it.

A policy that says "access must be temporary" does not tell the system what to do when the revoke job is down. The rest of the chapter wires grants, records and timers so that it does.

Just-in-time access: requests that name an operation

A JIT request is built from five elements, each there because of one of the failures above. Each is tagged with the part of the book's model it serves.

- The ceiling (Grant), `human_jit_write`, defines the most a request may ask for. It is never granted to a login (Chapter 3).
- A write request (Grant) names the reviewed fix it is for, in `fix_id`. Its (privilege, table, columns) triples are derived from that fix, inside the ceiling, and stored in `access_request_privilege`.
- Approval (Grant) is of the statement, and the grant enforces its bounds: the approver reads the statement, the gateway grants exactly the derived triples to the requester's session login, and the approved statement set is recorded as `approved_hash`. Expiry revokes exactly those triples, read from the same record.
- Writes in a JIT session (Execution) are forwarded only from the runner, which refuses anything that does not match `approved_hash`.
- An independent reconciliation (Record) compares the grants the database holds with the requests that justify them, so a stopped job is noticed by something other than the job.

I will take them in the order a request meets them.

What a request contains

A request is made from the session that will use it, and that session runs on the primary. A session carries at most one live request; a task that outlasts its window continues in a new session with a new request. Routine investigation stays on the replica with `human_readonly` (Chapter 6), and Chapter 7 says who may open a primary-bound session. The gateway knows which login belongs to which person, so nobody types a principal or a login.

Here is the shape of Robin's request, before anyone has answered it:

```

access_request:
  id:          "req-0412"
  kind:        "jit"
  requester:   "robin"
  requester_login: "sess-robin-0412"
  approver:    null
  fix_id:      "fix-0093"
  approved_hash: null           // set at approval, never by the
runner
  reason:      "alert-7731: invoices stuck in sending"
  expires_at:  now() + 1 hour   // illustrative window
  status:      "pending"

access_request_privilege:           // derived from fix-0093, not
typed
  ("UPDATE", "invoices", "status")

```

Three things are worth noticing. The reason points at something checkable, an alert or a ticket. The grant is narrow, and Robin needs no `SELECT` triple because the session already holds `human_readonly`. And Robin did not type the triple: it was derived from `fix-0093`, so the grant and the statement the approver reads cannot disagree. A request that holds only `SELECT` triples, for instance to read a Confidential column unmasked, has no fix and no hash.

A grant restricts privileges and columns, not rows. The `UPDATE` above can change `status` on every invoice the session can see. A JIT grant bounds what a mistake can reach by column and by time; what bounds it by row is the statement the approver reads and the runner executes, with the count-first discipline of Chapter 10.

Binding the request to the statement

The fix is a reviewed statement set with its filter and its expected row count, in the form Chapter 10 describes, and the request names it by `fix_id` and carries its `approved_hash`. The hash comes from the fix's merge record or the approved request record and never from the runner's input, and in a JIT write session the gateway forwards writes only from the runner. A request that holds only `SELECT` triples has no fix, hash or runner, and the approver reads the column list. Chapter 9 states the rule in full and covers how the gateway checks each statement.

Checking a request

The gateway checks the shape of the request before anyone is asked to approve it, and checks it again, with the approver, at grant time.

```
function checkRequestShape(request):
    // runs before anyone is asked to approve
    // live = approved and not yet revoked; this request is not yet
    counted
    if request.status != "pending" or
    hasLiveRequest(request.requester_login):
        raise GrantRefused
    if request.expires_at is null or request.expires_at <= now():
        raise GrantRefused
    // a window longer than the gateway allows is refused, not clamped
    if request.expires_at > now() + maxJitWindow:
        raise GrantRefused
    for each p in request.privileges:
        // an empty column list is never read as "all columns"; table-
        level
        // triples exist only on engines using the secrets-tables
        fallback
        if p.privilege != "DELETE" and isEmpty(p.columns):
            if not tableLevelAllowed(p.table):
                raise GrantRefused
    // refuses a table outside the ceiling and any Restricted column
    if not withinCeiling("human_jit_write", request.privileges):
        raise GrantRefused
    checkOperationBinding(request)
```

```

function checkOperationBinding(request):
    if hasWriteTriple(request.privileges):
        fix = fixRecord(request.fix_id)    // null when fix_id is null
        if fix is null:
            raise GrantRefused
        // the triples come from the fix; a typed list that differs is
        refused
        if request.privileges != triplesFromFix(fix):
            raise GrantRefused
        else if request.fix_id is not null or request.approved_hash is not
        null:
            // a SELECT-only request has no fix and no hash
            raise GrantRefused

function checkJitRequest(request):
    // runs again at grant time, once an approver is named
    checkRequestShape(request)
    if request.approver is null or request.approver ==
    request.requester:
        raise SelfApprovalNotAllowed
    if request.fix_id is not null:
        if request.approver == fixRecord(request.fix_id).author:
            raise SelfApprovalNotAllowed

```

The self-approval lines are where four-eyes is enforced in code: whoever asks, or wrote the fix, cannot be the one who answers. A pending request has no approver, which is why the shape check leaves those lines out; the approver is filled in at the moment of approval, and the full check runs at grant time.

I would refuse an over-long window instead of silently shortening it: a shortened one leaves the requester believing they have an hour when they have twenty minutes, and the surprise arrives mid-repair.

`withinCeiling` is where Chapter 3's ceiling becomes enforceable. It rejects a triple whose table is outside the ceiling, whose privilege is not offered on that table, or whose column list includes `iban`, `account_holder_name` or `token_value`. It rejects `INSERT` and `DELETE` on `bank_accounts` and `payment_tokens`, because a new or removed row sets or destroys Restricted values. It rejects any triple other than a `DELETE` that has no column list, except a table-level triple on an engine that uses the `secrets-tables` fallback. And it rejects anything touching `ledger_entries`, `audit_events` or the access request records, which no human role writes.

One thing stays inside the ceiling on purpose. `UPDATE (customer_id)` on `bank_accounts` or `payment_tokens` passes `withinCeiling`, because `customer_id` only links a row to its owner and is not `Restricted`. Moving the row to another customer still changes whose account or token it is, so the execution policy routes it to the product path (Chapter 9). The approver's checklist catches it by hand.

Who approves, for how long, for what scope

A JIT request has three variables: the approver, the window and the scope.

The approver is a person who understands the tables in the request and is not the requester or the author of the fix. For Ledgerly, the natural approver for `invoices` is another backend engineer, usually the secondary on-call engineer. The natural approver for anything on `payments` or `payment_tokens` is someone from the team that owns `payments-service`, because that team knows what a changed `status` there will trigger downstream. The platform team runs the access tooling, and I would not make it the default approver for data it does not own: it becomes the bottleneck, and an approver who cannot judge the change.

```
function chooseApproverPool(request):
    owners = emptySet()
    for each p in request.privileges:
        owners.add(owningTeam(p.table))
    pool = eligibleApprovers(owners)
    // neither the requester nor the author of the fix may answer
    pool.remove(request.requester)
    if request.fix_id is not null:
        pool.remove(fixRecord(request.fix_id).author)
    if pool is empty:
        raise NoApproverAvailable
    return pool
```

An empty pool fails visibly and never falls back to a wider one. Otherwise the pool's order is the escalation order: the page goes to the first person, then the next, inside the same pending request, which is refused only when it reaches `expires_at` unanswered.

The window should be as long as the typical task, not as long as the worst one. Say a data fix usually takes twenty minutes, the form offers an hour, and the gateway's maximum is four hours. Extending a grant is a new request with a fresh approval, not an edit of the old one, because an edit

would let a single approval stretch indefinitely. When a repair outlasts its window, the session ends and any open transaction rolls back with it, so a repair written as a single transaction (Chapter 10) loses its work cleanly instead of leaving half of it applied.

The scope is the triples and the statement they come from. The approver reads both, and the approval screen shows the triples as the grants they will become: "UPDATE on `invoices (status)`, until 04:12". A request that names a Confidential column in a `SELECT` shows that column unmasked, and the screen says so in words, not only in the column list.

Approval happens on the gateway's own screen, through single sign-on with a fresh multi-factor check at the click, not as a reaction under a chat message: the approval is the control, so whatever records it must know who gave it.

What the approver checks

The approver's decision is narrower than "is this a good idea?": approve the statement, and let the grant enforce its bounds. This checklist mirrors the reviewer's checklist for the fix in Chapter 10, from the access side.

1. Is the reason a checkable alert or ticket, and is it still open?
2. Does the statement on the screen do what the reason says? Read the statement, not the requester's summary of it.
3. Does the filter bound the rows, and does the expected count match what the requester saw?
4. Do the derived triples cover the statement and nothing more? Under the secrets-tables fallback, "nothing more" means "all non-Restricted columns of the table", and the screen says so.
5. Is the window about as long as this task, and under the maximum?
6. Does the statement touch `customer_id` on `bank_accounts` or `payment_tokens`, or any Restricted column? Then it is not a JIT data fix. Send it to the product path and refuse the request.
7. Am I neither the requester nor the author, and would I be comfortable explaining this approval to the reviewer next week?

An approver who cannot answer one of these asks the requester while the request stays pending. Silence is not approval, and neither is a column list.

Granting

Approval is the moment where four records have to agree: the gateway's request record, the audit log entry, the grants in the database and the session. No transaction spans them, so the order matters, and so does what each failure leaves behind.

```
function grantJitAccess(request):
    checkJitRequest(request)

    // the hash comes from the records, as "Binding the request"
    explains
    hash = approvedHashFor(request)
    // the approved record comes first: the revoke job reads it, so a
    grant
    // can never exist without its triples and its end
    markApproved(request.id, approved_hash: hash)

    // audit next: if the log is unreachable, no grant is made
    try:
        auditLogAppend({ actor: request.approver, action: "grant",
                        subject: request.requester_login,
                        request: request.id, fix: request.fix_id,
                        approved_hash: hash, triples:
request.privileges,
                        expires_at: request.expires_at })
    on AuditLogUnavailable:
        markRefused(request.id)
        raise GrantRefused

    // the session's end follows the approved window, as "Ending the
    grant" says
    setSessionEnd(request.requester_login, request.expires_at)

    // separate systems, no shared transaction: a failure part-way
    leaves an
    // approved record with no grant or only some triples
    for each p in request.privileges:
        grant(p.privilege, on: p.table, columns: p.columns,
            to: request.requester_login)

    // the window may end between the check and the grants: end it now
    if request.expires_at <= now():
        revokeRequestAccess(request.id, from: request.requester_login)
```

Read it by its failure modes. If the audit log is down, the request is refused and nothing is granted: a grant that cannot be recorded cannot be reviewed, and I would rather stop one repair than create one invisible one. If the gateway dies after the record and the log entry but before the grants, the request is approved and has no effect, which exposes nothing.

If it dies after two of three triples, the login holds two, the record holds three, and the later revoke takes back all three, since revoking a triple that was never granted is harmless.

The last lines handle a race. The window does not restart on approval, so the shape check refuses an approval that arrives after `expires_at`; what remains is an approval that passes a moment before the window closes, and those lines take its grants back at once.

The audit entry carries the fix and the hash along with the triples, so the approver, the statement and the window sit in one record. Payment-data and financial-reporting regimes tend to ask who approved a change to production data, on what basis and for how long, and this record answers all three; legal and compliance own the interpretation.

Ending the grant

A grant ends in three layers, and none of them is trusted to be the only one.

The first layer is the session. The gateway ends it at `expires_at` and removes the login, with no scheduled job involved: the clock that bounds the access belongs to the component that created it. A primary-bound session opens with the two-hour cap of Chapter 7, so on a grant, from `grantJitAccess` or `declareBreakGlass`, the gateway resets the session's `ends_at` to the request's `expires_at`, and `endSession` of Chapter 7 closes the session at that time; a window longer than two hours is therefore not cut short.

```
function endSessionAtExpiry(session):  
    // one live request per session, so its end is the session's end  
    waitUntil(session.request.expires_at)  
    // never raises; a failure leaves the record for the next sweep  
    endSession(session.login, "expired")
```

The ending itself is `endSession`, the single place that closes connections, revokes what the session's requests granted and removes the login, and `sweepSessions` calls the same function (Chapter 7).

The second layer is the revoke job, which reads the same record and removes what the request granted. `revokeRequestAccess` derives its work from `access_request_privilege`: for a JIT request, it revokes each

stored triple; for a break-glass request, it revokes the role `human_breakglass`. It does not read a separate list that could disagree, and it treats a missing login as already revoked.

```
function revokeExpiredGrants():
  // expiredGrants: kind jit or breakglass, status approved,
  expires_at
  // passed, not yet revoked
  for each item in expiredGrants(before: now()):
    try:
      // revokes what the request granted: its stored triples
    for "jit",
      // the role human_breakglass for "breakglass"
      revokeRequestAccess(item.request_id, from: item.login)
    on LoginNotFound:
      // the login went with the session; nothing is left to
    revoke
      on RevokeFailed:
        // an expired grant still live is an incident; retried
    next run
      notify("platform", "expired grant still active: " +
    item.id)
      continue
      markRevoked(item.request_id)
      try:
        auditLogAppend({ action: "revoke", actor: "revoke-job",
          subject: item.login, request:
    item.request_id })
      on AuditLogUnavailable:
        // the revoke is done; the log catches up from the retry
    queue
      queueForRetry(item.request_id)
      notify("platform", "revoke not yet logged: " + item.id)
```

The revoke continues when the audit log is down. The asymmetry is deliberate: a grant without a log entry is an exposure, while a revoke without one is a gap that can be filled later.

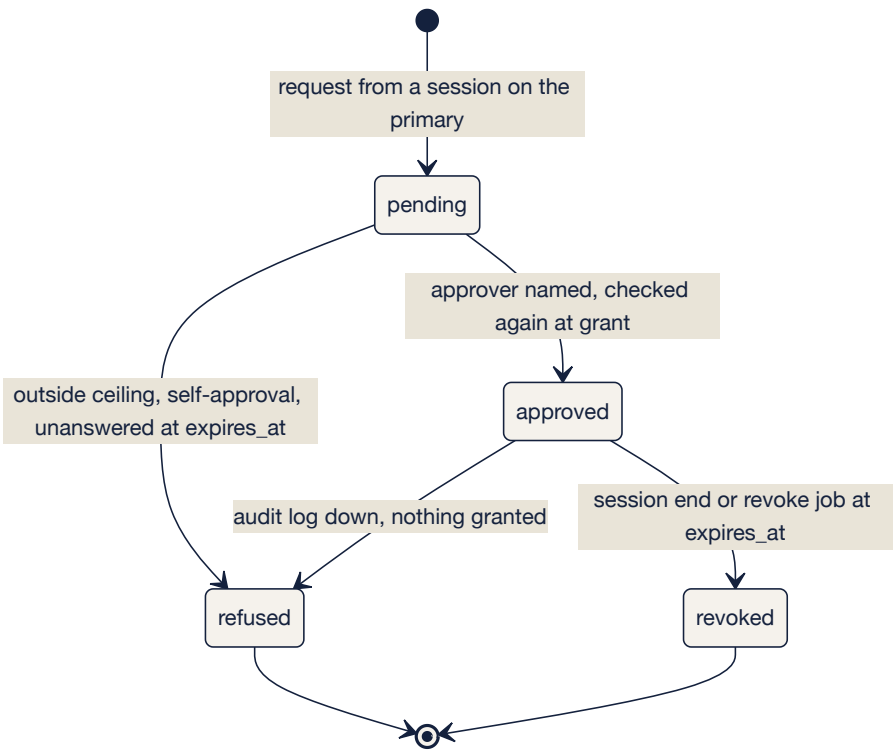
The third layer is the reconciliation, and it is a different kind of thing. It does not take access away. It asks, from the outside, whether the first two layers did their job:

```
function checkOverdueGrants():
  // scheduled separately from revokeExpiredGrants, so one failure
  // cannot hide the other
  if count(expiredGrants(before: now() - graceMinutes)) > 0:
    notify("platform", "grants past expiry and not revoked")
  if now() - lastSuccessfulRun("revokeExpiredGrants") > maxRunAge:
    notify("platform", "revoke job has not completed recently")
```

Alongside it, the drift check from Chapter 3 reads the grants the database actually holds and compares them with the unexpired triples in the requests. A direct grant with no live request behind it, a login that is a member of the ceiling, a member of `human_breakglass` with no open breakglass request: each is flagged, even when the revoke job reports success. The overdue check watches the records the job should have acted on; the drift check watches the database the job should have changed. A job that reports "done" while changing nothing is caught only by the second.

The reconciliation repairs nothing on its own, and I would keep it that way: an automatic revoke of unknown grants is a way to break production at night. It notifies the platform team, and a person decides.

The state a request moves through is small, and drawing it shows where each layer acts. Look at the exits from `approved`: a revoke, or a refusal when the audit log is down and nothing was granted; nothing comes back from `revoked`.



Self-approval, and no automatic approval for on-call

Self-approval is refused, always. `checkJitRequest` raises `SelfApprovalNotAllowed` and nothing overrides it, including seniority and urgency. The urgent case has a different path, which is the next half of this chapter.

Automatic approval is the other tempting shortcut, because the friction is highest at night. For reads I would not build it: every session login already holds `human_readonly`, and the one thing a JIT `SELECT` adds is a Confidential column shown unmasked, which is exactly what an approver should look at. A write changes what the services act on next, and a machine cannot tell whether `status` is the right column. Keep writes with a human approver and spend the effort on making that approver easy to reach.

If you add an automatic approval later, for a read that really does carry something new, the rule must require an explicit column list, and an empty or missing list must refuse the request visibly. Under the secrets-tables fallback a table-level read has no column list at all, so a rule that treats "no named Confidential column" as acceptable would approve every Confidential column in the table.

Requests from automation, and the policy as code

A job, a runbook script or an AI agent that wants to write is a principal like any other: it opens its session under its own workload identity, names a fix, and waits for a human approver, and the record names both the automation and the person who triggered it (Chapter 9 covers what it may do inside a session). The approval logic itself is code. `withinCeiling`, `chooseApproverPool`, `maxJitWindow` and the shape checks live in the repository, go through review, and sit in the drift check's baseline; a ceiling edited by hand in a console is the original problem one level up.

Abuse cases

Three cases are bounded by the controls above and removed by none. Collusion between requester and approver is narrowed to the columns, statement and window in the record, and the pair is visible in the audit entry; the access review (Chapter 14) is where a pair that approves each other far more often than chance shows up. A socially engineered ap-

prover ("approve req-0412 now, the batch is blocking payroll") is met by the checklist: read the statement on the gateway's screen, check the alert in the reason, and answer nothing sent through another channel. A hijacked single sign-on token cannot write in a JIT session, because the gateway forwards writes only from the runner and the runner refuses a hash that was never approved. The residual risk sits in reads inside the granted triples and in break-glass, which has no approver; the step-up check at declaration and the alert are what bound it.

Break-glass access

Break-glass is emergency access with a defined logging set and mandatory review. The name comes from the glass-fronted fire switch: reachable by anyone who needs it, and breaking it is visible to everyone.

In the book's model it is the role `human_breakglass`, and it is the only human role that reads Restricted columns. It reads every table and writes `INSERT` and `UPDATE` on every table except the two append-only ones; on `bank_accounts` and `payment_tokens` it writes `UPDATE` only, and only on non-Restricted columns. It never writes `ledger_entries` or `audit_events`, because no human role does. It is not a route for routine fixes, and it is not a data fix path: a data fix is planned, reviewed and bounded (Chapter 10), and break-glass is none of those things by design.

A break-glass request is self-declared. It has no approver. The person in the session says an emergency exists, gives a reason, and the gateway marks the request approved and grants the role to the same session login until `expires_at`. The same revoke job and the same session timer take it back.

The approver is the whole control for JIT, so an emergency role without one may seem odd. Break-glass exists for the cases where the approver pool has not answered, and an approval step would reproduce that failure. The control moves from before the access to around it: an alert at declaration, the logging set below, a short fixed window, and a review that follows without exception.

```

function declareBreakGlass(session, reason):
    // a hijacked token alone must not be enough to declare
    requireFreshMultiFactor(session.principal)
    // one live request per session: a break-glass declaration
    supersedes it
    supersedeLiveRequest(session.login)
    // the window is the gateway's fixed maximum; there is no argument
    for it
    window = breakGlassMaxWindow
    request = newRequest(kind: "breakglass", requester:
session.principal,
                                requester_login: session.login, approver:
null,
                                reason: reason, expires_at: now() + window)
    markApproved(request.id)
    try:
        auditLogAppend({ actor: session.principal, action:
"breakglass",
                                subject: session.login, request: request.id,
                                expires_at: request.expires_at })
    on AuditLogUnavailable:
        markRefused(request.id)
        raise GrantRefused
    setSessionEnd(session.login, request.expires_at)
    grant(human_breakglass, to: session.login)
    // paged after the grant: a paging outage must not block an
    emergency;
    // the review queue below is the net for a page that never arrived
    queueReview(request.id, reviewer: "not-the-requester")
    notify("security", "break-glass declared: " + request.id)
    notify("platform", "break-glass declared: " + request.id)

```

Three decisions in that block need a comment.

The window is fixed by the gateway, at two hours in Ledgerly's illustration. No longer window is available on request, because "I need it a little longer" turns an exception into a habit. If the emergency outlasts the window, the person declares again and the second declaration alerts again; that is the intended friction.

The audit-log check is the same as for JIT. If the log is down, the declaration is refused. I accept that cost, because a break-glass session with no record is the one thing a review cannot reconstruct, and the review is what makes the role tolerable. What on-call does when the normal path and the log are both down belongs to Chapter 15.

The alert is a notification to the security contacts and the platform team that someone has opened the broadest access Ledgerly has, not a request for permission. In a healthy system it is rare enough to be read; if it fires weekly it will stop being read, and break-glass is being used for something that deserves a designed path.

The break-glass logging set

"Heavy logging" is not a standard until it is defined, so for a break-glass session the logging set is these four things:

- Every statement, as a fingerprint with its literals redacted and a hash of the parameters, with the start, the end and the rows affected. Redaction matters most here: this session reads Restricted columns, and a literal in the log would copy them somewhere without column-level protection.
- The entries as they happen. Each statement is appended to the audit log as it runs, not collected at the end, so a session that is cut off still leaves its record.
- A one-line note on every write. The gateway refuses a write without one, which costs seconds and gives the reviewer the intent beside the statement.
- The same record as any other session: the login, the person, the request and the reason, so the three records of Chapter 14 can be joined.

One risk remains, and it differs by state. Today, with no gateway or with one that only brokers the credential, a break-glass session is the one place where an unbounded UPDATE on a table it may write is still possible. The window, the row counts, the entries and the review bound the damage and expose it; they do not prevent it. In the target state the gateway of Chapter 9 runs write sessions with auto-commit off and a row cap, its matrix denies an unbounded write in this lane, and a write without a note is refused. What stays possible is a write that fits inside the cap and still hits the wrong rows. The cap is hardest to size here, because an emergency is by definition not the planned task; until the policy sets one, the unbounded write is an accepted risk, and the reason the review exists.

Review after the fact

Every break-glass use is reviewed, by someone other than the person who used it. The review is not a hearing; it asks the questions a design review asks: was this an emergency that the normal path could not serve; what did the session read or write; did it read Restricted columns, and was that necessary; and could a JIT request have served the same need, and if so, what would have made that request fast enough?

The last question is the one that pays. Each review either confirms that break-glass was the right tool or finds a gap in the normal path that made it the only tool. Fixing the gap (a missing approver, a short window, a table the ceiling does not offer) reduces the next use. The review also needs its own clock:

```
function checkBreakGlassReviews():
  for each r in breakGlassRequests(endedBefore: now()):
    review = reviewFor(r.id)
    if review is null and now() - r.expires_at > reviewDeadline:
      notify("security", "break-glass review overdue: " + r.id)
    if review is not null and review.reviewer == r.requester:
      notify("security", "break-glass reviewed by its own user:
" + r.id)
```

A review nobody does is a control that has quietly stopped existing, so its deadline raises an alert.

When the gateway is not there

Humans reach the database only through the access gateway (Chapter 7), and both JIT and break-glass grants are made by the gateway to a session login it created. A gateway outage is therefore also a break-glass outage. This is a trade-off in the design, and Chapter 7 states the same decision from the gateway's side.

The alternative is a second door: an emergency login outside the gateway. It is attractive on the night it is needed and dangerous every other night, because a credential that works without the gateway works without its logging, identity and expiry; it is the shared admin password with a better name.

I would rather make the gateway hard to lose than make its absence survivable by weakening everything else: more than one instance, few dependencies (the audit log, the request records), a rehearsed failure, and a health signal that on-call sees first. If all instances are down at once, no human data path remains, and I would not invent one here. The sealed instrument of Chapter 3 exists for engine-level work, it needs a network route that Chapter 7 closes to people, and it bypasses the append-only and Restricted protections, so it is not a data access path. Chapter 15 returns to what to do in that case.

The choice is not free. A team with a fragile gateway may reasonably want a second path while it matures. Then write down what it is, who can retrieve its credential, how its use alerts, and what retires it: a path with an owner and an end date is a decision, and one without is a leftover.

When the engine has no column-level grants

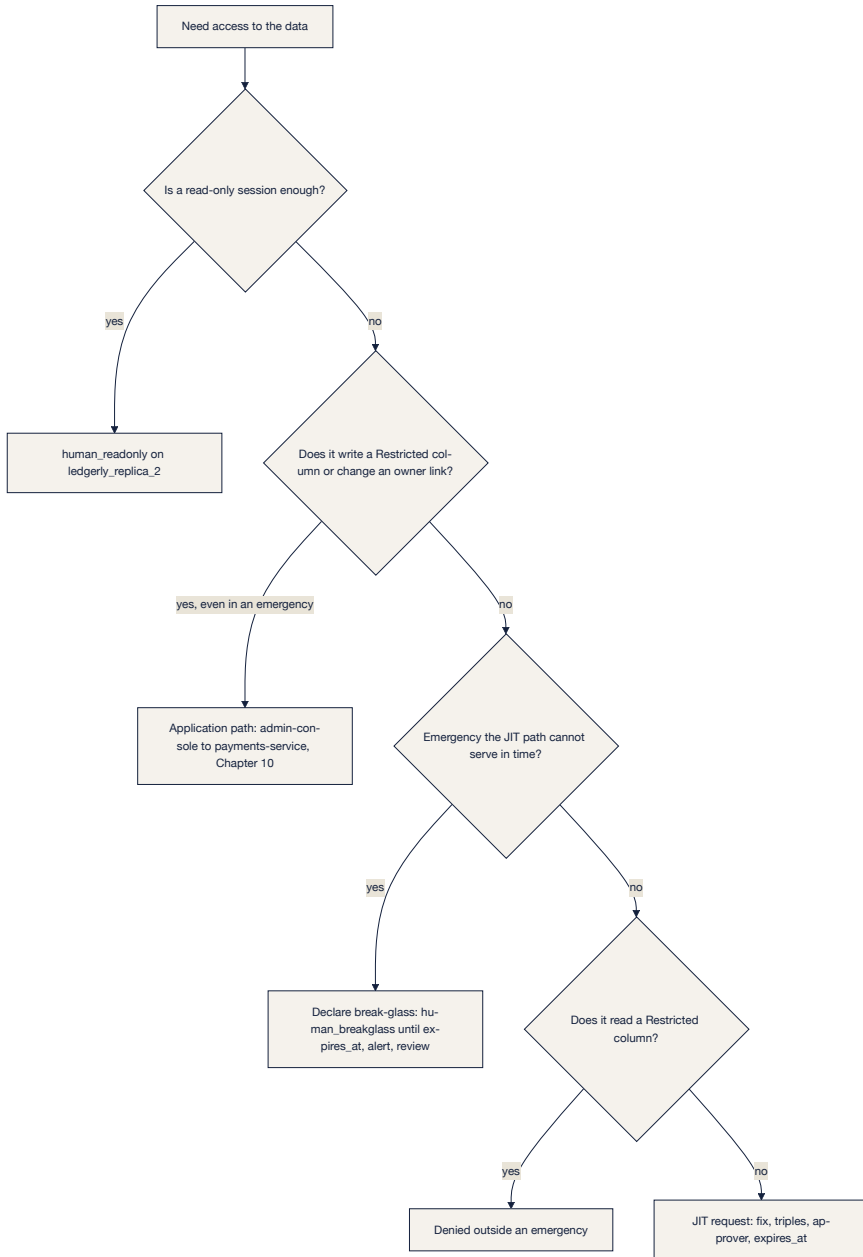
Everything above assumes that the engine can grant a privilege on named columns. Chapter 3 introduced the one fallback the book uses for engines that cannot: the Restricted columns live in separate tables, `bank_account_secrets` and `payment_token_values`, outside the ceiling and reachable only by `app_payments_service` and `human_breakglass`. The main tables then hold no Restricted column, so a table-level JIT grant on `bank_accounts` or `payment_tokens` cannot reach one. Such a request has no column list: `columns` is null, and the triple reads "UPDATE on `payment_tokens`".

The price falls on the approver, who has approved every non-Restricted column of the table, Confidential ones included. If the requester wanted to change `expires_on`, the approver is also approving `provider` and `last_four`. The approval screen must say "all non-Restricted columns of this table" in words, and the statement the approver reads carries more of the weight. A column group that a team expects to open often deserves its own small table.

The fallback does not change the rules for Restricted data. A person still never writes it with raw SQL; the change goes through `admin-console` and `payments-service` after a second person approves (Chapter 10).

Choosing the path

A person at three in the morning needs to know which door to use, and the answer belongs in the runbook as a short flow. The diagram below is the shape of it. Look at the two ends that no human data route serves: a write to a Restricted column, which goes to the product path even in an emergency, and a read of one outside an emergency, which is denied. The design says so instead of leaving the decision to whoever is under pressure.



Ledgerly at 3 a.m.

Back to Robin. The stuck rows were counted in the routine session, and the number matches the runbook. The write is the request bound to `fix-0093`, for an hour, citing the open alert. `chooseApproverPool` selects the

secondary on-call engineer and the team's lead, in that order, and the page goes to the first. The old Ledgerly would have opened the shared password here. The new one has a wait, and the wait is the design working.

If no one answers, the runbook asks the question that decides everything: is this an emergency the approver pool cannot serve in time? Stuck reminders can wait until morning, so Robin leaves a note on the alert and stops, an outcome the old password never allowed.

Suppose instead that `payments-service` were failing, payment collection had stopped for every customer, and diagnosis needed a Restricted column that no JIT request may contain. That is a case for break-glass. Robin declares with the alert as the reason; the gateway grants `human_breakglass` for its fixed window; the security contacts and the platform team are told. Next morning someone who is not Robin reads the log and asks the review questions, one of which is why diagnosis needed a Restricted column. The answer might lead to a safer diagnostic view next quarter.

Trade-offs and costs

Friction is the price, and it must be spent where it buys something. A JIT request costs the requester a minute to write and the approver a minute to read, at best, and a woken engineer at worst. For a task that happens twice a quarter, this is cheap. For a task that happens twice a day, it is a product feature wearing an operational disguise, and Chapter 10 describes how to move it into `admin-console`.

Binding to a fix costs preparation. The fix has to be written down before the request can be filed, which is slower than a grant alone. It is the price of approving what will run instead of what might.

Approval fatigue is a failure of the design, not of the approver. An approval that arrives within seconds, every time, from the same person is theater: either the requests are trivial and `human_readonly` should cover them, or the approver has stopped reading. The remedy is smaller requests, a screen that shows what the grants will be, approvers who know the tables, and a pool large enough that nobody is the default. Two signals belong in the access review: how often requests are refused or sent back, and how often the same approver answers. A refusal rate of zero is as likely to mean that nobody is looking as that the requests are good.

The on-call engineer carries the cost first. The approver is woken, the requester waits, and the team learns a runbook it rarely uses. Make the approver pool a rotation, publish it, and make sure the people on it have the context to read a request about `invoices` or `payments`.

The gateway is a dependency of every write. The revoke job, the audit log and the request records are all dependencies of a grant; when one is down, grants stop, which limits exposure and slows incident response. Chapter 7 describes how to reduce the odds.

The reconciliation produces noise. The drift check and the overdue check will flag legitimate cases, such as a revoke that runs a minute late. Tune the grace period so that a flag means something; a muted check is worse than none, because the team believes it is covered.

Break-glass has a social cost. A person who declares one is telling the security contacts that the normal path failed. If the review feels like a trial, people will hesitate during real emergencies; if it is a design review, they will not. Blameless review is what keeps the role usable.

When not to use this. A team of four engineers with one low-risk database and no Restricted data does not need approval pools and a reconciliation job on day one. A manual process with a reason, a time limit, a person who confirms and a record may be enough for a while, if the grants are removed when time runs out and someone checks. Build the pattern in the order of its failures: first the expiry, then the record, then the approver, then the alert.

Common mistakes

- **Granting the ceiling role to a login "for this one task".** Every JIT session then holds the whole ceiling, and the request's scope is decorative. Grant the triples to the session login and keep `human_jit_write` as a ceiling no login belongs to; the drift check flags a member.
- **Approving the triples and not the statement.** A column list is short and easy to read, and approving it approves a capability. Bind the write request to a reviewed fix, derive the triples from it, and take `approved_hash` from the fix's record.

- **Revoking from a separate list.** A hard-coded table list will eventually miss a table the request granted. Derive the revoke from `access_request_privilege`, the record that drove the grant.
- **Trusting the revoke job alone.** A job that has run every night for a year is easy to stop watching. Give the session its own end time and run an independent check that compares grants with requests.
- **Writing the audit entry after the grant.** If the log is down, a grant exists that nobody can see. Write the record, then the log entry, then the grants, and let a failed log entry stop the grant.
- **Automatic approval of on-call requests.** Reads need no request and a write needs a second person. Shorten the wait with a good approver pool.
- **Making break-glass long enough to be comfortable.** A window that never expires inside a working day becomes a working role. Fix the maximum in the gateway and let a second declaration alert again.
- **Leaving the logging set undefined.** A reviewer cannot judge a session whose record was never specified. Name the set before the first use.
- **Skipping the review because nothing went wrong.** Reviews are where the normal path gets repaired. Give the review a deadline and raise an alert when it passes.
- **Building a second door for the gateway's bad day.** An emergency login outside the gateway bypasses its identity, logging and expiry. Invest in availability first; if a second path is unavoidable, give it an owner, an alert and an end date.

Key takeaways

- Standing access costs most on the days it is not used, so the useful question is who needs access now, for which task, and how the system will know the task is over.
- `human_jit_write` is a ceiling and is never granted to a login; a write request names its reviewed fix (`fix_id`), the triples are derived from the fix, and the approver approves the statement, whose hash (`approved_hash`) the runner must match.

- Revocation is derived from the same `access_request_privilege` record that drove the grant, enforced by the session's own end time, backed by a revoke job, and checked by an independent reconciliation that trusts neither.
- Writes need four-eyes: an approver who is neither the requester nor the fix's author, answering on the gateway's screen after a fresh multi-factor check. There is no automatic approval for on-call.
- Break-glass is a self-declared `human_breakglass` session with a fixed maximum window, a defined logging set, an alert at declaration and a mandatory review by someone other than the user. It is neither a routine path nor a data fix path, and its one accepted risk is, today, an unbounded `UPDATE`, and in the target state, once the gateway of Chapter 9 enforces a cap, wrong rows within the cap.
- A design with no sanctioned fast path grows an unsanctioned one, so the emergency route is designed, logged and rehearsed, and its dependency on the gateway is a cost you name.
- Where the engine cannot grant on columns, the secrets-tables fallback keeps Restricted columns out of every request, and the approver approves whole tables of non-Restricted columns.

A grant, even a bound one, says what a session may touch and for how long, not how carefully the statement behind it is checked. Chapter 9 turns to that: what the gateway does with a statement before it runs. Chapter 10 then turns to the work itself: how to run a data fix so that the right rows change, the wrong rows do not, and a second person can see why.

Chapter 9 — Mediated Execution: Query Analysis and Dangerous Operations

Ledgerly's support team has an assistant in its chat tool. Sam asks it to void a batch of duplicate invoices from a billing retry, and the assistant, running under its own identity, drafts a fix: an `UPDATE` on `invoices` with a `WHERE` clause on the retry's batch label. Jordan from the data team reviews the statement and approves it. The statement is valid SQL, the grant covers it, the approver read it, and the label was correct when Jordan checked it.

The statement still matches more than the twelve rows the fix named. A second job reused the batch label two days earlier, and nothing in the statement's text says so. Nobody was careless: Sam described the task accurately, the assistant wrote what it was asked to write, and Jordan reviewed what was shown. What was missing was a step between "approved" and "committed", a place where something looks at what the statement will do to the data and not only at what it says.

Chapters 3 to 8 decided who may hold a grant, over which path, and for how long. This chapter adds the Execution part of the five-part model: the step where the gateway handles each statement, instead of handing over a session and trusting whoever holds it. The question it answers is this: what should stand between an approved statement and the data, when approval alone cannot say what the statement will do?

The problem, and why it is hard

A session handout gives a person a login and lets their client talk to the database directly. Everything after that is the client's business: what is typed, how many rows it touches, whether it commits on the spot. The grant bounds which tables and columns are reachable, and nothing

bounds what any single statement does inside that reach. In the starting state a desktop client submits whatever is typed and commits it immediately.

Mediated execution reverses the arrangement. The statement goes to a component that is not the person's client, and that component decides whether it runs, with which limits, and when it commits. The gateway of Chapter 7 is already in the data path, so it is the natural place for this. The person still writes the statement, or an automation does; what changes is that no statement reaches the data without passing through the loop in the next section.

This is a spectrum, not a switch. The cheapest form is a session handout with a few settings the gateway enforces, such as a statement timeout and a row cap. The next form checks each statement before forwarding it. The most mediated form approves each statement individually and runs it from a worker, so the person never holds a database session for the write at all. Some tools approve each query rather than a time window. At the time of writing, QueryProxy, for example, has a second person approve the statement and a worker run it, so the developer never holds the database credential. That trades the speed of an interactive session for an approval bound to the exact statement, and other query-approval tools work in a similar way.

Mediation is hard for three reasons. The first is that SQL is expressive: the same effect can be written many ways, and whether a statement is "dangerous" often depends on data the checker has not read. The second is that every check has a cost in latency, in false refusals and in the people who learn to work around it. The third is that a checker that is trusted too much becomes the thing nobody reviews. The position this chapter takes is that statement analysis reduces mistakes and is not the authorization boundary. The grants are the boundary. The gateway's job is to make the common wrong statement hard to run by accident, and to make every write carry a bound that something other than the writer enforces.

The gateway's loop for each statement

Every statement a session sends, human or not, goes through the same loop. The order matters more than any single step, so here it is as pseudocode before the pieces are explained.

```
function handleStatement(session, statement):
    if statement.changesSetting(guardedSettings):
        return refuse(session, "guarded setting")
    verdict = evaluateStatement(session, statement)
    // the record comes first: a statement that cannot be recorded
    // does not run
    try:
        recordIntent(session, statement, verdict)
    on RecordUnavailable:
        return refuse(session, "record unavailable")
    if verdict.action == "deny":
        return refuse(session, verdict.reason)
    if verdict.action == "gate":
        return routeToGate(session, statement, verdict) // approval,
    // or the product path
    if statement.isRead:
        return streamRows(session, statement, cap: session.rowCap)
    // cursor closes at the cap
    // auto-commit is off: the runner owns the transaction of a fix,
    // the gateway owns the rest
    tx = transactionFor(session)
    result = execute(tx, statement)
    tx.rowsSoFar = tx.rowsSoFar + result.rows // counted across
    // every statement of the fix
    if tx.rowsSoFar > verdict.bound:
        failTransaction(tx) // its owner can
    // only roll back
    return refuse(session, "more rows than approved")
    showRowCount(session, tx.rowsSoFar) // before COMMIT,
    // while undo is free
    if tx.ownedByGateway:
        if not countAccepted(session, tx.rowsSoFar, verdict.bound):
            rollback(tx)
            return refuse(session, "count not accepted")
        commit(tx) // the commit point
    // for a person or a migration
    recordOutcome(session, statement, result) // failure is
    // queued and alerts; a commit stands
    return result
```

Guarded settings. The gateway opens each session with its limits applied: the statement timeout, a read-only default, the role and the connection limit by session kind (Chapter 5). A client can send a `SET` that lifts any of them, so the gateway keeps a list of guarded settings and refuses a state-

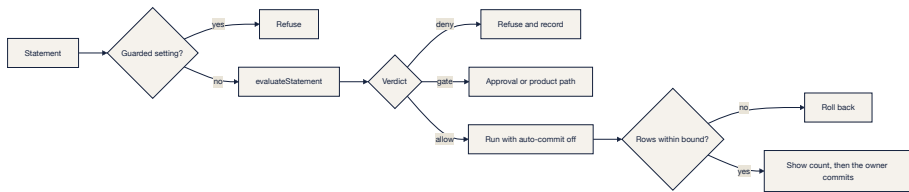
ment that changes one. The word "non-overridable" is earned only where a mechanism enforces it. A timeout set as a role default can be lifted by the session unless the gateway refuses the `SET` or the platform pins it, so a limit that has neither is a default and not a guarantee. The cost is that some clients send such settings when they connect and stop working until someone lists them.

The row cap. For reads, the gateway counts rows as it streams them and closes the cursor at the cap, tells the person, and records the event. The cap does not say a person may only see so many rows over a day, and a determined person can page through a table in slices. It says that a single mistaken query does not pull a whole table into a laptop. Large legitimate extracts belong on the export path of Chapter 13 and the masked analytics copy of Chapter 12, where the tier decides who approves.

Auto-commit off for writes. A write session runs each write inside an explicit transaction. For a person or a migration the gateway opens it; for a data fix the runner of Chapter 10 opens it, and the gateway forwards every statement of the fix into that one transaction. Either way the gateway counts the affected rows across all statements and compares them with the bound, while the decision to `COMMIT` stays with the transaction's owner: a gateway that committed each statement of a fix on its own would break the runner's single transaction, and would make its dry run a real change. When the count passes the bound, the gateway refuses the statement and marks the transaction failed, so the owner can only roll back. This is the cheap form of reversibility from Chapter 1: an update that was meant to touch one row and reports four hundred is stopped while undo is still free. For an interactive session, "shown" means a person sees the count and confirms. For the runner and for automation, "shown" means the gateway compares the count with the bound the approval recorded and fails the transaction on any excess. The cost is the extra step, and the transaction's locks, which stay held while the person decides; a confirmation prompt that nobody answers holds a row lock, so the prompt has a timeout that rolls back.

Record first. The gateway writes the intent before the statement runs and the outcome after. There is no transaction that spans the gateway's log and the database, so the order is what carries the guarantee. If the intent cannot be written, the statement does not run. If the outcome cannot be writ-

ten after a commit, the commit stands, the entry is queued, and the platform team is alerted; a log that only works when nothing is wrong is not worth the name. Chapter 14 covers where the record lives.



Classifying a statement

The gateway decides what to do with a statement in `evaluateStatement`. It classifies the statement by what it does and where, looks up the matrix in the section after next, and for a write that the matrix allows with a bound, finds the bound and compares the planner's estimate with it. Every refusal and every gate comes from the matrix; the function has no early returns of its own besides the parser's.

```

function evaluateStatement(session, statement):
    parsed = parse(statement)
    if parsed.failed or parsed.isMultiStatement:
        return deny("cannot be analyzed")           // fail closed
    // nine classes, one per matrix row; the most specific class wins
    class = classify(parsed)
    cell = matrix[class][session.kind]              // deny, gate,
allow or allow with a bound
    if cell.action != "allow":
        return cell                                  // a deny, or a
gate with its route
    if class == "read":
        return allow(cap: session.rowCap)
    if cell.bound == "fix":
        binding = checkBinding(session, statement)  // the next
statement of the approved fix
        if binding.action == "deny":
            return binding
        bound = binding.bound
    else:
        bound = session.writeBound()                // the break-glass row cap, or
the backfill rule's limit
        if bound is none:
            return deny("no approved count")
        estimate = plannerEstimate(parsed)           // asks the
planner; runs nothing
        if estimate > bound * tolerance:
            return deny("estimate exceeds the approved count")
        return allow(bound: bound)

```

Three choices are worth defending. First, a statement the parser cannot read is refused. If the gateway ran what it could not analyze, the analysis would apply only to the statements that were never a problem. A client that sends batches of statements in one string breaks under this rule, and that is a cost to name in the migration; the answer is to send them one at a time.

Second, classification looks at the effect and not at the name of the person or the tool. A `DELETE` in a migration and a `DELETE` in a JIT session are the same class, and the matrix decides what each lane may do with it. Having one classifier with one table, and not a separate rule set per path, is what keeps the policy small enough to review.

Third, the planner's estimate is compared with the expected count, but only as an early refusal. The planner estimates from statistics, and statistics go stale; an estimate can be too low as easily as too high. The estimate lets the gateway refuse a statement that is plainly wrong before it

takes locks. The control is the real count, compared inside the transaction before `COMMIT`. The tolerance is a parameter the platform team sets and records, and I would start tight and loosen it only when false refusals are measured, not guessed.

What static analysis cannot see

A statement's text says little about its effect on rows. A parser can see that a `WHERE` is missing, that a predicate is always true, that a statement touches a table it should not, or that it calls a function. It cannot see how many rows a predicate matches today, which is the problem in the opening: a valid filter, on a label that now matches more rows than anyone expected.

Several other cases sit outside what analysis can settle. A filter can be valid and still wide, matching most of a table because of the data and not because of the syntax. A join can multiply rows. A view or a function can hide a write or a wider read behind a name that looks harmless, and the gateway cannot see inside a function body that the role is allowed to call. A triggered side effect happens after the statement and is invisible to its text. And a statement can be exactly right in shape and wrong in meaning: the right rows by the filter, the wrong rows by the intent, which no tool can judge.

Open-source tools such as QueryProxy check statements at submission, rejecting an `UPDATE` or `DELETE` whose `WHERE` is missing or always true and capping unbounded `SELECT`s; at the time of writing its own documentation describes this parser as a defense layer rather than the authorization boundary, which is the point of this section, and other gateways and approval tools take similar approaches.

What follows from this is a division of work. Analysis catches the common mistakes cheaply and early. The grants (Chapter 3) set what is reachable at all, so a missed statement is still bounded by the role. The count check in the transaction catches the case where the data disagreed with the author. The approval, with a second person who reads the statement and the expected count, catches the case where the shape was right and the intent was wrong, some of the time. None of these is complete, and the design relies on their overlap and does not claim any one of them as the boundary.

The same limit shapes the export tier of Chapter 13. The gateway computes an export's tier from the source columns the query reads, not from the names of the result columns, and a source it cannot resolve counts as the highest exportable tier; where even that cannot be decided, the export is refused. That is the same fail-closed rule as the parser's: what analysis cannot read, it does not wave through.

The deny, gate and allow matrix

Every statement class gets one of three answers in each lane. **Deny** means the gateway refuses and records the refusal. **Gate** means the statement does not run as sent: it needs an approval it does not yet have, or it belongs on a different route. **Allow** means it runs under the limits of the loop above; where a cell says "bound", the write is allowed only inside a bound that comes from the source the cell names, and the real count is compared with it before `COMMIT`. The columns are the lanes the book has described, and the rows are the statement classes the classifier produces.

Statement class	Human read session	JIT write	Break-glass	Data-fix runner	Migration
Read, within the row cap	Allow	Allow	Allow	Allow	Allow
Bounded write, within the approved count	Deny	Allow, bound to the fix (runner only)	Allow, bound to a row cap (note required)	Allow, bound to the approved hash and count	Allow, bound to the back-fill rule
Unbounded write	Deny	Deny	Deny	Deny	Gate: named approver
Owner-link change	Deny	Gate: product path	Gate: product path	Gate: product path	Gate: named approver
Write to a Restricted column	Deny	Gate: product path	Gate: product path	Gate: product path	Gate: named approver
Export	Gate: by tier	Deny	Deny	Deny	Gate: named approver

Statement class	Human read session	JIT write	Break-glass	Data-fix runner	Migration
Schema change (DDL)	Deny	Deny	Deny	Deny	Allow: reviewed lane
Privilege change	Deny	Deny	Deny	Deny	Gate: platform review
Write to ledger_entries or audit_events	Deny	Deny	Deny	Deny	Gate: platform review

Reading it row by row shows the design. The human read session has no write path, so every write class is denied there, and the only gate is the export tier. A JIT write session forwards writes only from the runner, so its bounded-write cell is an allow with a binding and not an open door. Break-glass is the one lane where a person writes directly, which is why its bounded write is allowed only under a row cap and with a note, and why the size of that cap is the hardest sizing decision in the book: an emergency is not the planned task, so no cap can be picked from the task. The cap should be set from what the break-glass role would plausibly need to repair and reviewed after each use. An unbounded write is denied there. What is left is a write that stays inside the cap and hits the wrong rows, an accepted risk that the review exists to expose, as Chapter 8 states; Chapter 8 also says what holds today, before a gateway enforces the cap.

The owner-link change deserves its own row. `UPDATE` of `customer_id` on `bank_accounts` or `payment_tokens` passes the JIT ceiling, because the column links a row to its owner and is not Restricted. Moving the row to another customer still changes whose payout account or payment token it is. The classifier therefore names it as its own class, and every lane except a migration answers with a gate that routes it to the product path: `admin-console` asks `payments-service`, a second person approves, and the change is recorded in `audit_events`. A migration in the normal lane gates it for a named approver, as Chapter 11 does for any flagged statement, and the expedited lane refuses it; a schema change has little business reassigning ownership, and if a bulk reassignment is ever needed, it is a product feature to build and not a statement to run. Writes to Restricted columns (`iban`, `account_holder_name` and `token_value`) take the same route, which is why no lane lets a human write them with raw SQL. The

one migration case is moving Restricted columns into the separate secrets tables of Chapter 11, which goes to a named approver and not to a refusal.

The migration column matches what Chapter 11 describes. Its classifier flags statements for a named approver or for platform review, and refuses only in the expedited lane. In this table the flags show up as gates where a person has to say yes, and the allows are the reviewed schema change and the backfill that stays inside its rule. Append-only tables are the same: a migration that touches them goes to the platform team, and the other lanes deny it. A second rule set for forbidden statements would be a second policy to keep in step with this one, so the migration pipeline reuses the classifier and the matrix and adds only its lane rules.

The matrix is deliberately small. Nine classes and five lanes fit on a page, and a policy that fits on a page gets read. Each new class or lane added to it is a decision someone has to defend, and the usual failure is a matrix that grows an exception for every incident until nobody can say what it forbids. I would keep it in version control, reviewed like code, with the platform team as owner and the data team as a required reviewer for the write rows.

Binding approval to the statement

An approval that covers "a write on `invoices`" lets any write on `invoices` through. Statement-bound approval means the approval names this statement, and the gateway refuses any other. Chapter 8 describes the request and Chapter 10 the runner; this section is the rule that connects them, stated once.

A request for a write carries a `fix_id` and an `approved_hash`. The hash is the hash of the statements that were reviewed, taken over the whole set when a fix has more than one, and its source matters more than the check. It comes from the fix's merge record, or from the approved request record, and it never comes from the runner's input. If the runner supplied its own hash, the runner would be checking its own work, and a change to the statement would arrive together with a matching hash. A hash that originates in the record that a second person approved is a hash that someone other than the writer vouched for.

```

function checkBinding(session, statement):
    request = loadApprovedRequest(session.requestId)    // from the
store, not from the caller
    if request is none or request.status != "approved":
        return deny("no approved request")
    if request.fixId is none:
        return deny("no fix named")                    // a SELECT-
only request has no fix and no hash
    fix = loadMergeRecord(request.fixId)                 // the
statements, read from the record
    if hashOf(fix.statements) != request.approvedHash:
        return deny("record differs from the approved hash")
    next = fix.statements[session.transaction.statementCount]
    if normalize(statement) != normalize(next):
        return deny("statement differs from the approved one")
    return allow(bound: fix.expectedRows)               // the count
comes from the hashed record

```

The gateway checks one statement at a time but the hash covers the set: it loads the statements from the merge record, confirms that their hash is the approved one, and accepts only the next statement in that order. The bound is the expected row count in the same record, and not a number the request carries, so the hash protects the count as well. The commit stays with the runner's transaction, so a fix with several statements either lands whole or not at all.

Two rules complete the binding. In a JIT write session, the gateway forwards writes only from the runner's connection. The person's own client can still read inside the granted triples, but a write sent from it is refused, because otherwise the hash would be a check that a client could route around. A guard is a mechanism only if the session cannot write any other way. The second rule is the reverse case: a JIT request that holds only `SELECT` triples has no fix and no hash, so there is nothing for a write to be bound to, and the write is denied. A reading request cannot be turned into a writing one by sending a different statement.

Where the gateway only brokers a credential and the client connects directly, none of this is enforceable, and the book says so plainly: the hash then records an intention and prevents nothing. The binding is a property of the data path and it exists only where the gateway is in it.

The runbook case is the one with a template. A runbook's merge record stores the hash of the statement template, and the request carries only the key list. The runner substitutes the keys into the stored template. For this

case the gateway checks the hash against the template and not against the substituted text: it confirms that the submitted statement is the approved template with only the key list filled in, and it uses the number of keys as the bound, together with the request's triples. The gateway never takes a statement or a hash from the caller that wants the work done.

Callers that are not people

Jobs, runbook scripts, ChatOps bots and AI agents all submit statements, and none of them is a person. The rule in this chapter is that they take the same loop, as principals of their own. There is no lighter path for automation, because automation is where a mistake repeats at machine speed.

An automated caller has a workload identity from the platform and not a person's session, and a read-only default. It opens its session under its own principal, so its reads are subject to the same row cap, timeout and guarded settings as anyone's. If it wants to write, it follows the same route as a person: it names a fix, an approver who is a person other than the one who triggered the automation approves it, and its writes go through the runner with an expected count. The two-person rule of Chapter 8 holds here too: whoever asked the automation to draft the fix cannot be the one who approves it, and there is no automatic approval by a caller that is not a person. An agent that proposes a statement is proposing it, and the approval is a human reading it.

The record names both parties: the automation that submitted the statement and the person who triggered it. This is the accountability half. A record that said only "the assistant did it" would end the trail at a machine, and a record that said only "Sam asked" would hide which tool turned the request into SQL. Both names are stored, and the access review asks about both.

An AI agent adds one concern that the other callers do not: its statements are generated from text, some of which it did not write. A ticket, a log line or a customer message can end up in the context that produces the statement. The defense is not to read the text more carefully. It is that the agent's privileges do not change with what it reads. Its grants are the same, its writes still need a person's approval of the exact statement, and a

statement that arrives by a route the agent should not have, such as a write outside the runner, is refused by the same matrix. Whatever the agent was persuaded to propose, it cannot approve it, and it cannot run it without the bound.

For a read-only agent, the useful limits are the ordinary ones: a tight row cap, a short timeout, a view or a masked copy in place of Restricted columns, and a per-principal budget, so that a loop that goes wrong exhausts its own allowance and not the shared one. For a writing agent, the useful limit is that its writes are few and small, because each one needs a person. If that makes the agent too slow to be useful for writes, the right conclusion is that the task should become a product feature that the agent calls, not that the approval should be relaxed.

Trade-offs and costs

Mediation costs time. Every statement passes through one more component, and a gated write waits for a person. For interactive reads on a replica, the added latency is small and mostly invisible. For writes it is the point: the delay is where the second look happens. I would not remove it for urgent cases; I would make the break-glass lane's gate quick and loud, and review it afterward.

It costs compatibility. A gateway that parses statements and refuses what it cannot parse will break some clients, some drivers and some habits. Refusing SET statements, multi-statement strings and unusual syntax produces a list of things that stop working. The list is shorter if it is collected before the rollout, from the statement records of the existing sessions, than if it is discovered through support tickets. Budget a period where the gateway records its verdicts but only refuses the clearest cases, and move to enforcement per class as the false refusals fall.

It costs a component that must be available. The statement check sits in the path of every write, and Chapter 7 describes why that component should stay small. Where the check is down, the default that bounds the damage is that writes stop and reads on the replica continue. A fail-open check teaches the team that the check is optional, and the first outage ends the policy.

It costs a policy to maintain. The matrix, the list of guarded settings, the tolerance and the row caps are all parameters someone owns. A policy nobody owns drifts toward whatever the last incident needed. The matrix is the one artifact to keep small, reviewed and versioned.

There is also a cost that cannot be paid with engineering: analysis gives a false sense of safety. A team that has a parser that rejects `UPDATE` without `WHERE` can start treating a statement that passed as a statement that cannot do harm. The matrix says what the gateway refuses, and the opening of this chapter shows what a passing statement can still do. The count check and the second person exist for that case, and they should not be thinned out because the parser is good.

Common mistakes

Treating the analyzer as the boundary. The analyzer is a layer that catches the common wrong statement. The grants decide what is reachable, and a design that lets the analyzer carry that weight fails the first time a statement is valid and wrong.

Letting the runner or the caller supply the hash. A hash that arrives with the statement proves only that the statement matches itself. The hash comes from the merge or approved request record, and the check runs in the gateway, where the caller cannot edit it.

Auto-commit left on for write sessions. With auto-commit on, the row count is a report after the fact. Turning it off, and showing the count before `COMMIT`, is what makes a wrong count cheap.

Failing open when the parser cannot read a statement. Running what could not be analyzed makes the analysis cover only the harmless cases. Refuse it, and keep a short list of the clients that need changing.

A lighter path for automation. Giving jobs and agents a shortcut around the loop, because they are trusted or because they are in a hurry, removes the controls where the volume is highest. Automation takes the same loop under its own principal.

A matrix that grows an exception per incident. Each exception is a decision with no owner. Keep the table to the classes and lanes that exist, and when a case does not fit, change the design or build the feature.

Reading a row cap as a limit on exposure. A cap bounds one query, not what a person can read in a day. If exposure is the concern, it belongs to the masking and tiering work of Chapter 12 and Chapter 13.

Key takeaways

- Execution is how a statement reaches the data: who submits it, whether anything inspects it first, what bounds its effect and how it is committed. Grants do not answer these questions.
- Mediated execution puts a component between the statement and the data. The cheapest form is a few enforced settings; the strongest approves each statement and runs it from a worker.
- Every statement, from a person or an automation, goes through one loop: guarded settings, evaluation, record first, a row cap for reads, and auto-commit off for writes with the row count shown before COMMIT.
- Analysis reduces mistakes and is not the authorization boundary. The grants are the boundary, and the real row count checked inside the transaction catches what the parser and the planner cannot.
- One small matrix of statement class by lane gives each answer as deny, gate or allow. The owner-link change and writes to Restricted columns are gated and routed to the product path.
- Approval is bound to the statement: the `approved_hash` comes from the merge or approved request record, never from the runner, and in a JIT write session the gateway forwards writes only from the runner.
- Jobs, runbooks, ChatOps and AI agents are principals with their own identity, a read-only default and the same loop. Their records name both the automation and the person who triggered it.

The loop and the matrix say what the gateway does with a statement. They do not say how to write a fix that is worth approving: how to prove the count, keep a snapshot, plan the rollback and have a second person review it. Chapter 10 turns to that work.

Chapter 10 — Ad-hoc Queries and Data Fixes

Imagine a Thursday afternoon at Ledgerly. A bug in `invoice-worker` has been fixed and deployed, but while it was live it left a batch of invoices in the status `sending`, even though the emails had gone out. Support is asking about them, the customers' reminders are paused, and Robin, the on-call backend engineer, can see the shape of the repair in one line: set the status back to `sent` for the affected invoices and nothing else.

Robin opens a session through the access gateway. The default session is read-only, so the line cannot run. Robin needs to ask for write access, and that request has to name tables and columns. Somebody else has to look at the statement before it touches the primary. Meanwhile the clock is running, and the repair is small enough that the process around it feels larger than the work.

This chapter adds one step of the target state: writes happen through a reviewed data fix procedure and just-in-time grants within the `human_jit_write` ceiling. Chapter 7 gave humans a single door, and Chapter 8 built the grants that open it for a limited time. Here we ask what a person should do once the door is open: how to read, how to write, and how to make the second kind of work safe enough that it can be done calmly on a Thursday afternoon.

In the five-part model of Chapter 1, this chapter is about **Execution**. Robin already holds every credential the repair needs, so the risk is not in who has a credential. It is in what the grant authorizes, what one statement will do when it runs, who read it first, and what evidence remains. The question for the chapter is this: what is the smallest, most reviewable way for a person to change production data, and when should the change stop being a person's job altogether?

The problem, and why it is hard

Every system that holds real data eventually reaches a state its own code cannot repair. A worker crashed between two steps. A migration left a column half populated. A customer was merged into another by mistake. The application has no endpoint for "put this right", because nobody expected it to be needed, and writing the endpoint would take longer than the repair. So a person writes a statement and runs it against production. That is a **data fix**: a change to production data made by a person outside the normal application path, to repair a state the application cannot repair.

The first difficulty is that the dangerous tool and the useful tool are the same tool. A SQL session that can answer "how many invoices are stuck?" can, with one more keyword, change all of them. In the starting state of Chapter 1, the GUI client holds the shared admin login, auto-commits every statement, and executes whatever text is highlighted. The distance between a question and an irreversible change is one word.

The second difficulty is that the person doing the work is usually under pressure and usually right. A process that treats each fix as a threat teaches people to route around it, and the route-around is worse than the original risk: a standing grant, a shared password, a copy of production on a laptop. A process that treats each one as routine teaches nobody to look twice, and the one statement in a hundred that is wrong reaches every row. The safe path has to be the easy one.

The third difficulty is that the unit of risk is not the person but the statement, and a statement carries almost no information about its own size. `UPDATE invoices SET status = 'sent' WHERE id = 'inv-0042'` and the same line with the `WHERE` clause cut off differ by a few characters. Nothing in the text says "this will touch nine thousand rows"; only the data says that, and the data is read after the fact unless the process asks the question first.

The fourth is that the answer cannot be a policy alone. "Count the rows before you update" is read once and followed until the first busy day, so it has to be built into the path: a runner that refuses to commit when the count is wrong, and a session that cannot write any other way.

Ad-hoc queries have a quieter problem. An **ad-hoc query** is a query a person writes and runs once, outside any application. In a read-only session it cannot change anything, but it can read a great deal, it can be expensive for the database to run, and its results leave the system in a screenshot, a spreadsheet or a chat message.

The default: a read-only session

A human session starts as a login that is a member of `human_readonly`, connected through the access gateway to `ledgerly_replica_2`. It can read the tables the role covers, with Restricted columns withheld and Confidential columns masked where Chapter 12 says so. It cannot write, and nothing the person types will change that.

Most investigation fits inside this session, and the person does not need to ask for anything or wait for anyone. I would defend this default on cost alone: a design in which the common task needs no approval is a design in which the approval for the uncommon task still means something.

Three habits keep the shared replica and its results safe. **Bound the query:** the role-level statement timeout (Chapter 5) and the gateway's row cap (Chapter 7) stop the runaway case, and selecting counts and samples first keeps accidental exports small. **Select the least that answers the question:** `id, status, due_on` is better than `SELECT *`, partly because it is faster and partly because it keeps Confidential columns out of the result unless the question needs them. **Treat results as data copies:** a result pasted into a ticket or a spreadsheet is a copy of production data under different access rules (Chapter 13 treats such copies as side doors), so paste identifiers and counts, not rows with personal content, and name the query that produced them.

One caveat matters for the next section. The replica is behind the primary by some amount, usually small and sometimes not. A count taken there is an estimate of what the primary holds now. For investigation that is fine. For deciding how many rows a write will touch, it is not, and the data fix procedure below takes the authoritative count on the primary, inside the transaction that does the write.

Writing is a different act

A read can be repeated, abandoned and ignored. A write changes what every other principal sees, including the services that are running at that moment and will act on the new state before anyone checks it. A wrongly updated `status` can start a reminder email; a wrongly updated `due_on` can change what a customer is told they owe. That is why the book gives writing its own path, with its own approval, its own time limit and its own record.

That path is the just-in-time access of Chapter 8, with one property that matters here: a write request is bound to a reviewed fix. The request carries the fix's `fix_id` and an `approved_hash`, and its (privilege, table, columns) triples are derived from the fix, not typed by the requester. The most any request may ask for is the ceiling defined by `human_jit_write`. Three consequences shape how the fix is written.

First, **the request is built from the fix**. A fix that updates `invoices.status` needs exactly one triple: `UPDATE` on `invoices` for the column `status`. The guard in the next section reads before it writes, and that read uses the plain `SELECT` on `invoices` that `human_readonly` already holds, so no `SELECT` triple is needed. The exception is a fix that snapshots a Confidential column, such as `billing_address` on `customers`: under `human_readonly` the column is masked, and a snapshot of masked values is useless for a rollback, so the request names a `SELECT` triple for it and the approver sees it.

Second, **a Restricted column never appears in the request**. `iban`, `account_holder_name` and `token_value` cannot be named, and `INSERT` and `DELETE` on `bank_accounts` and `payment_tokens` are unavailable, because inserting or deleting a row sets or destroys Restricted values. One more change sits inside the ceiling and is still not an ordinary data fix: `UPDATE` of `customer_id` on `bank_accounts` or `payment_tokens`. The column is not Restricted, but moving a row changes whose payout account or payment token it is. The execution policy gates this owner-link change and routes it to the product path described below (Chapter 9). If a fix needs any of these, it is not a data fix.

Third, **the approver sees the statement, not only the triples**. Chapter 9 states this rule of statement-bound approval and how the gateway enforces it. For a data fix the approval screen shows the statement, the filter and the expected count next to the triples, and a request for all columns is a request for a different kind of trust. Fixes written with a narrow footprint are approved faster, which is the incentive you want.

The request record for the stuck-invoice fix, in the engine-independent notation from Chapter 3:

```
// request record for the stuck-invoice fix; values are illustrative
access_request:
  id          "req-0187"
  kind        "jit"
  requester   "robin"
  approver    "alex"
  fix_id      "FIX-0031"
  approved_hash "a41c...07" // set at approval, from the fix's
merge record
  reason      "reset invoices left in 'sending' by worker bug"
  expires_at   now() + 1 hour

access_request_privilege: // derived from FIX-0031, not typed
  ("req-0187", "UPDATE", "invoices", "status")
```

On engines without column-level grants, the Restricted columns already live in separate tables outside the ceiling (Chapter 3). A JIT request on the main table is then a table-level grant, and the approver has approved every non-Restricted column of it, Confidential ones included. The grant no longer limits the statement's footprint, so the statement itself carries more of the review.

Anatomy of a data fix

A data fix is a short procedure with a fixed order. The order matters more than any single step, because each step exists to make the next one cheaper to get right.

1. **State the intent.** One sentence: what state is wrong, what state is right, and which rows are in scope. If it cannot be written in one sentence, the fix is probably two fixes.
2. **Count the affected rows with a `SELECT`.** "Forty invoices" and "nine thousand" are different conversations, and the count is the first place a

mistake shows itself.

3. **Write down the expected row count.** The fix declares the number it expects to change. This is the guard.
4. **Take a before snapshot.** The primary keys and the old values of exactly the columns the fix will change, saved somewhere that outlives the session.
5. **Run the change inside a transaction.** Compare the number of rows changed with the expected number. If they differ, roll back.
6. **Check a postcondition inside the same transaction.** A query that proves the state is now what the intent said, for example "no invoice in the list still has the status `sending`".
7. **Commit, then record.** The commit is the point after which other principals see the change. The after state, read inside the transaction, goes into the record next to the before snapshot.

The pseudocode puts the guard in the runner and not in the person's memory. It treats the database, the snapshot store and the audit log as separate systems, so no transaction covers more than the database. The runner is handed a request id and nothing else; the first function decides whether that request may run at all.

```
function loadApprovedFix(request_id, session):
    request = loadRequest(request_id)          // the gateway's record,
not the runner's input
    if request.status != "approved" or request.expires_at <= now():
        raise RequestNotLive
    if request.requester_login != session.login: // granted to this
session's login only
        raise RequestNotForThisSession
    fix = loadFixScript(request.fix_id)
    // approved_hash was set at approval from the fix's merge record,
so
    // the script is compared with the record, not with itself
    if hashOf(fix.statements) != request.approved_hash:
        raise ScriptNotReviewed
    if request.privileges != fix.privileges:
        raise GrantsDoNotMatchFix
    return fix
```

The second function runs it, and everything it does is bounded by the first:

```

function runDataFix(request_id, session):
    fix = loadApprovedFix(request_id, session)
    // the authoritative count, on the primary, before anything is
written
    before = readRows(fix.target_filter, fix.columns, using: session)
    if count(before) != fix.expected_rows:
        raise RowCountMismatch
    saveSnapshot(fix.id, before)
    transaction:
        // re-read under the lock: a changed value must fail, not pass
        locked = readRowsLocked(fix.target_filter, fix.columns, using:
session)
        if locked != before:
            rollback()
            raise RowsChangedSinceSnapshot
        // one source: the change runs on the same filter that was
counted
        changed = execute(fix.change, where: fix.target_filter, using:
session)
        if changed != fix.expected_rows or not
fix.postcondition(session):
            rollback()
            raise FixGuardFailed
        after = readRows(keysOf(before), fix.columns, using: session)
        // commit point: other sessions see the change only after this
line
        commit()

    try:
        saveAfterState(fix.id, after)
        auditLogAppend({ actor: session.login, action: "data-fix",
                        request: request_id, fix: fix.id, rows:
changed })
    on AuditLogUnavailable or SnapshotStoreUnavailable:
        queueForRetry(fix.id)
        notify("platform", "fix committed, record pending: " + fix.id)

```

Several details in these blocks are deliberate.

The runner never supplies its own hash. It loads the fix that the request names and compares the hash of its statements with the `approved_hash` in the request record, which Chapter 9 traces back to the merge record. Retyping the statement changes the text between review and execution, and the review attached itself to the earlier text.

The snapshot is saved before the transaction. A failure of the snapshot store then stops the fix with nothing changed. The cost is that the rows could change between the snapshot and the write: a worker may move an

invoice from `sending` to `failed` in that gap, and the key would still match the filter.

The transaction reads the rows again under a lock. It compares the whole result, with the columns the fix changes, against the snapshot. Any difference in the set of rows or in a value aborts the run with `RowsChangedSinceSnapshot`, and the author starts again from a fresh count. On engines that cannot lock rows by a filter, the re-read still runs inside the transaction; it narrows the gap and does not close it, and the expected count and the postcondition are what remain.

The record is written after the commit. The snapshot store and the audit log are other systems, so the after state is saved once the commit has happened. This is the opposite order from the JIT grant in Chapter 8, where a grant must not exist without its log entry.

The order is defensible here because Ledgerly's gateway sits in the data path (below): it forwards the runner's statements and records the session, so a committed change is on record even if the audit append fails. The entry is retried and the platform team is told. On a gateway that only brokers credentials there would be no session log of the statements, and the audit entry would have to come first.

A dry run shows what a count cannot. The count is a `SELECT`, so it misses what the statement sets in motion: a trigger that writes elsewhere, a cascade, a rule. A real dry run executes the change inside a transaction and rolls it back:


```

function dryRunDataFix(request_id, session):
    fix = loadApprovedFix(request_id, session)
    if count(readRows(fix.target_filter, fix.columns, using: session))
    != fix.expected_rows:
        raise RowCountMismatch      // a wrong filter never reaches
the locks
    transaction:
        countsBefore = countsIn(fix.watch_tables, using: session)
        changed = execute(fix.change, where: fix.target_filter, using:
session)
        report = { rows: changed,
                    sample: sampleAfterValues(fix, using: session),
                    related: diff(countsBefore,
countsIn(fix.watch_tables, using: session)) }
        rollback()      // nothing is kept; the row locks are held
until here
    return report

```

The report holds the number of rows, a sample of the after-values and the change in row counts in the fix header's `watch_tables`. It needs an approved request and the write grants, so it cannot exist while the pull request is under review; it runs inside the approved window, holds the same row locks while it runs, and cannot see effects outside the database, such as an email. Where the cost of a surprise is high, the author runs it in the first minutes of the window, attaches the report to the request, and the approver reads it before the commit step, at the price of a pause.

A guard is a mechanism only if the session cannot write any other way. Ledgerly's gateway sits in the data path (Chapter 7) and in a JIT write session forwards writes only from the runner's connection, so the hash check, the count and the snapshot cannot be skipped. The transaction stays the runner's: the gateway forwards each statement into it and counts the rows across the whole fix, but never commits on the runner's behalf, which is why a failed postcondition rolls back every statement. Chapter 9 covers how the gateway checks each statement against the merge record and where its inspection stops. The price is that every JIT write depends on the runner and the gateway being available. With a gateway that only brokers the credential, the runner is a convention, not protection against a statement without a `WHERE` clause.

Where the snapshot goes

The before snapshot is a data copy, and it needs an owner and an expiry like any other. A snapshot of `status` values on `invoices` is Internal data. A snapshot of a Confidential column, such as `billing_address` on `customers`, is Confidential, and the store needs access rules at least as strict as the source table.

I would keep snapshots with the fix's record, with a retention period the platform team sets: long enough for "was this fix correct?" to be answered, short enough that the snapshot does not become a second, ungoverned copy of customer data. That is days for invoice statuses and perhaps a full billing cycle for a billing address. Choose a number, write it down, and let it expire.

The rollback plan

A before snapshot is not a rollback plan until someone has written down how it would be used. The plan is one or two sentences in the fix's header, such as "restore `status` from the snapshot for the same keys", and it should be possible to turn the snapshot into a fix script mechanically.

The rollback is itself a data fix. It goes through the same request and review, because other principals may have acted on the new state in the meantime and restoring the old values may overwrite legitimate updates. The runner for a rollback therefore checks that the current values still match the after state recorded by the original fix, and refuses to touch rows that have since changed. A reverse fix that cannot say "these rows moved on, handle them separately" causes the next repair.

Some changes have no rollback, and the header should say so. An email, a webhook or a call to a payment provider cannot be taken back by restoring a column. For those, the conversation happens before the fix runs: which effects does this state change trigger, and should the services that react be paused first?

Two tables have no rollback in the ordinary sense, because no human role can edit them. `ledger_entries` and `audit_events` are append-only. A correction to the ledger is a new entry, issued only through the correction path of `payments-service`, never by a human login, so a fix that appears to need a change there is a request for that path, not for a grant. A trigger

that rejects every `UPDATE` and `DELETE` on both tables makes the rule hold even if a grant is widened by mistake. It has a cost: `ledgerly_owner` can still drop the trigger, so the real review point stays on migrations that touch these tables (Chapter 11).

A `DELETE` is the same procedure with a harder rollback

The anatomy above is written for `UPDATE`. Suppose a retry in `invoice-worker` inserted duplicate rows into `invoice_lines`, and the repair removes them. `DELETE` on `invoice_lines` is inside the `human_jit_write` ceiling, and it differs in three ways.

The grant bounds the table and not the rows, because a `DELETE` triple has no columns, so the bound comes from the filter, the expected count and the review, and the filter should be a list of keys. The snapshot is the whole row, since nothing else will hold the values afterward, and a table with Confidential columns makes the snapshot as sensitive as the table. The rollback is a re-insert, which is another data fix with its own request. An `INSERT` triple names its columns, so the re-insert request decides whether `id` and `created_at` are among them. If they are not, the restored row receives new values for both, and anything that referenced the old identifiers is not restored. If they are, the approver sees a key and a time-stamp being written, which no ordinary fix does, and what the engine does with a supplied key depends on how it generates keys. The header states which choice the rollback makes.

```

fix:
  id          "FIX-0044"
  intent      "Remove the duplicate invoice_lines rows created
by a retry.
              No other row is deleted."
  target_filter "id IN (<ids of the duplicates>)"
  change       "DELETE"
  expected_rows 12                      // illustrative
  snapshot     "whole rows"
  privileges    ("DELETE", "invoice_lines") // no columns for
DELETE
  rollback     "re-insert from the snapshot as a new fix; its
INSERT triple
              leaves out id and created_at, so the rows get new
values and
              references to the old ids are lost"
  effects      "invoice totals are recomputed by the worker on
next run"
  watch_tables ("invoices")

```

Where losing the identifiers is not acceptable, the reviewer asks for a cheaper protection than the rollback: a foreign key that rejects the delete while another row still points at the line, or a different repair that does not remove the row.

Review: four-eyes, in a repository

A data fix is code that runs once, so it gets the control engineers already trust: a pull request. The fix is a script in a repository, with a header, a reviewer and a history, and the history answers, six months later, who changed a status and why.

```
// data-fixes/2026-10-06-FIX-0031-reset-sending-invoices.fix
fix:
  id          "FIX-0031"
  intent      "Invoices stuck in 'sending' after the worker bug
              are set to 'sent'. No other column changes."
  target_filter "status = 'sending' AND updated_at < '<cutoff>'"
  change      "SET status = 'sent'" // applied to
target_filter only
  expected_rows 40
  postcondition "no invoice matches target_filter"
  privileges    ("UPDATE", "invoices", "status")
  rollback      "restore status from snapshot for the same ids,
              only where status is still 'sent'"
  effects       "reminders resume for these invoices"
  watch_tables  ("invoices") // read by the dry run
  author       "robin"
  reviewer     "alex"
```

The `privileges` field connects the repository to Chapter 8. The request's triples are copied from it, so the approver looks at the request and the fix side by side, and `loadApprovedFix` refuses a request whose triples differ from the fix. A statement that touches a column its privileges do not name fails at run time, which is the intended outcome; the grant is the second line of defense behind the review.

What the reviewer reads

The reviewer's job is narrower than "is this a good idea?".

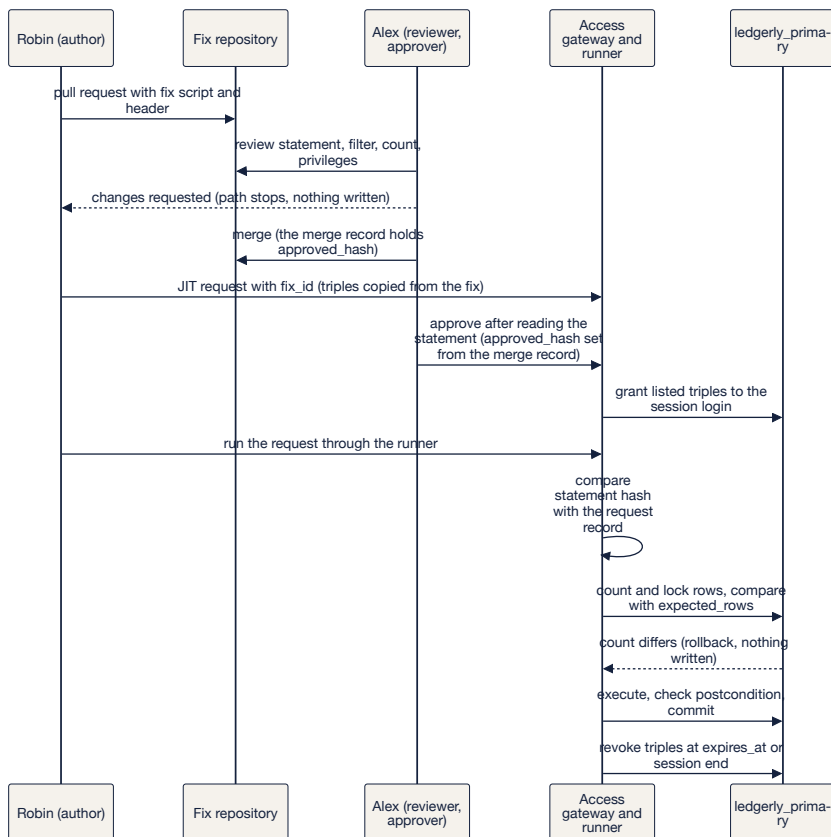
- **Does the statement match the intent?** They should agree in the tables, the columns and the filter.
- **Is the filter specific?** A filter that selects by key is the safest; a filter that selects by state and a time cutoff is acceptable if the count agrees; a filter with no restriction at all is rejected on sight.
- **Is the expected count plausible, and where did it come from?** A reviewer who sees `expected_rows: 40` can ask for the count query's output in the pull request. A number nobody can source is a number nobody checked.
- **Do the privileges name only what the statement needs?** Compare them line by line.
- **Does the fix touch a Restricted column, an append-only table, or a table outside the ceiling?** If so, it is not a data fix, and the review ends with a pointer to the right path.

- **What does this change trigger?** Reminders, emails, jobs that read the column. This question catches the quiet cases.
- **Does the preview agree with the intent?** The pull request holds the count query's output and a `SELECT` preview of the rows and the after-values; the dry-run report is attached to the request later, and a trigger or a cascade shows up there and nowhere else.
- **Is the rollback sentence usable?** For a `DELETE`, it says what the re-insert cannot restore.

Review is four-eyes in the book's sense: a second person reviews before a risky action runs. The merge fixes the text: a change to the script after review produces a different hash, and the runner refuses it. The approver of the JIT request and the reviewer of the fix are two acts. On a large team I would prefer two people, so the approver checks the access against the fix instead of re-reading it; on a small team they will often be the same person, which is acceptable if neither is the author. The author never approves their own fix or their own grant.

The sequence

The diagram shows the path from a request to a committed change. Look for the places where the path can stop with nothing written: the review, the hash comparison and the runner's guards.



When the change is to a Restricted column

The ceiling forbids Restricted columns in any JIT request (Chapter 3 gives the reason), so a whole class of repairs has no SQL path. Imagine that a customer's `account_holder_name` on `bank_accounts` was entered with a typo. A person cannot fix that in a JIT session, because the column cannot be named in a request. Nor can a break-glass session do it: `human_breakglass` holds no write privilege on Restricted columns, whatever the process says.

The path is the application. `admin-console` asks `payments-service`, the only writer of `bank_accounts`, to make the change; `admin-console` requires a second person to approve it before `payments-service` performs it, and the change is recorded in `audit_events`. It is a product path, not a data fix: it has a form, a validation, a reason field, an approver and a record, and it runs as the service's own role, not a human's.

This costs more than a SQL statement, and I think it should. Those values are the ones whose exposure damages customers in ways that are hard to undo, and the application is the only place where validation and the full set of side effects live. The cost is minutes for the person who needs the change; the saving is that a hand-typed repair can no longer introduce a malformed bank account number. A wrong token in `payment_tokens` is likewise replaced by `payments-service` through the provider's own flow, not patched.

Break-glass is not a data-fix route

`human_breakglass` writes where the JIT ceiling does not reach, and it is the only human role that reads Restricted columns (its exact privileges are in Chapter 3). On paper it looks like a convenient way to make a repair the JIT path refuses. It is not, and the reason is what the role lacks.

Break-glass is self-declared. It has no approver, no `fix_id` and no runner, and its safeguards are the logging set and the mandatory review that Chapter 8 defines. A limiting action that stops ongoing harm, reviewed afterward, is what the role is for (Chapter 15 covers the incident side). The lasting repair of the data is a different act and is always a data fix: run in a break-glass session, it skips the one control that reads the statement before it runs, in a role where an `UPDATE` without a row guard is possible. Chapter 8 states that as an accepted risk, and routing repairs through the role would make it routine.

The second reason is incentives. If break-glass can do a repair, it becomes the fast lane, and its alert stops meaning anything. A data fix that "needs" break-glass signals something else: the night approval cycle is too slow, the ceiling lacks a column the fix legitimately needs, or the fix belongs on the application path. If a repair cannot wait, stop the harm with the smallest available action and run the data fix through the normal path once the pressure is off.

Turning a repeated fix into a product feature

A data fix is a symptom: a state is reachable that the application cannot undo. The first occurrence is a fix; the third is a feature request that nobody has filed. I would keep a directory of fix scripts, tagged by the kind

of repair, and when the same tag appears again, someone asks whether it belongs in `admin-console`. There is nothing precise about "three times": a repair of ten rows twice a year may never justify a screen, while one that runs every week, needs a review each time and could be done by support in a minute is a feature waiting to be written.

Take the stuck invoices from the opening, and suppose the state recurs. Chapter 3's grant table already shows what `app_admin_console` may do to `invoices`: `SELECT` and `UPDATE` on `status` only. The product version is an action on the invoice page, available to a narrow group, that resets the status for one invoice, asks for a reason and writes the action to `audit_events` with the human principal's identity. It carries its own checks: only invoices in the stuck state, only one at a time, and a limit on how many a person can reset in an hour.

The conversion moves the risk from the statement, which pays for a request, a review and a runner on each repair, to the code, which is reviewed, tested and versioned once. Support can use the action without a database session, so every repair that becomes a feature is a recurring write that no longer needs human access.

```
function resetStuckInvoice(actor, invoice_id, reason):
    if not actor.canResetInvoices:
        raise NotAuthorized
    if reason is empty:
        raise ReasonRequired
    if countResetsBy(actor, within: "1 hour") >= resetLimit:
        raise RateLimited

    transaction:
        invoice = readInvoiceLocked(invoice_id)
        if invoice.status != "sending" or invoice.updated_at >
cutoffForStuck():
            rollback()
            raise NotStuck
        setInvoiceStatus(invoice_id, "sent")
        // commit point: the status change and the business record
agree
        appendAuditEvent({ actor_principal: actor.id, action: "reset-
stuck-invoice",
                                subject_id: invoice_id, reason: reason })
        commit()
```

The audit event and the status change are both in `ledgerly_primary`, so a single transaction covers them; `audit_events` is the application's business record, not the audit log of Chapter 14. In the runner above, the audit log is a separate system and the order of operations carries the safety.

Not every repair should be a feature. A button that anyone can press has a different risk profile from a script that two people reviewed, so repairs that change money, tokens or bank details stay on a four-eyes path even when they are frequent. A repair becomes a self-service action only if its worst case is bounded by the action's design; otherwise it becomes a request with an approval inside the product. The test is whether the repair will recur and whether a person is currently the only one who can do it. If both answers are yes, the sprint is an investment and not a cost.

Runbooks for the repairs you can predict

Some fixes recur in kind even when the details change: reassigning an invoice to the correct customer, replaying a failed export, repairing a duplicated row. For these, a **runbook** is the right artifact: a written, followed procedure for a repeatable task. It has the same skeleton as a fix script, with the parameters left blank. It names the symptom that triggers it, the diagnostic queries that confirm it, the template of the fix and the privileges it needs, the source of the expected count, the effects to check, the rollback, and who may author and review. It also says when to stop and escalate: "if the diagnostic count exceeds the number in this paragraph, this is not the known problem".

A runbook can also carry a pre-approval, so that a predictable repair does not wait for an approver at night. Five conditions decide it, and the runner checks them, not the person:

- the filter selects by key, a list of ids, and not by a state and a cutoff;
- the expected row count is at most N, a number the runbook sets for its own repair (say, 20 for stuck invoices; illustrative);
- the triples are exactly the runbook's fixed set;
- the repair triggers no outbound effect, as the runbook's `effects` field records;

- the statement is the runbook template with only the key list substituted, and the template's hash was approved when the runbook was reviewed.

The merge record of the runbook stores that template hash. The key list is the only part the request carries, and the runner substitutes it into the stored template and checks the result against the request's triples; it never takes a hash or a statement from its own caller (Chapter 9 covers how the gateway checks the substituted statement). If any condition fails, the request falls back to full review with an approver.

The cost is that a pre-approved run executes with nobody reading the key list that night, so the runbook's row limit bounds a wrong list, and someone other than the runner reads the record on the next business day. A runbook also ages: one for a table that has since changed shape gives a stale statement the appearance of review. Tie each runbook to a date and an owner, and treat a failed run as a prompt to revise it, not an inconvenience to bypass.

A WHERE-less UPDATE, analyzed without blame

Consider Ledgerly in its starting state. An engineer connects with a desktop client that remembers the shared admin password and auto-commits. The task is the stuck invoices. The statement is written on two lines in the editor: the `UPDATE` on the first, the `WHERE` on the second. The client executes the highlighted selection, and the selection covers the first line only. The statement runs, and every row in `invoices`, say nine thousand of them, has the status `sent`.

One version of this story ends with somebody being told to be more careful. It teaches nothing, because the next engineer in the same chair faces the same distance between a question and an irreversible change. The useful question is what in the design let a single statement reach every row.

- **The login could write everywhere.** The shared admin role owns every table, so a mistake was not confined to the table or the column the task needed.
- **The client committed on its own.** Nobody saw "nine thousand rows affected" before the statement became final.

- **The count was never asked for.** Nothing required the person to know how many rows to expect, so the number printed after the fact carried no meaning.
- **No one else had read the statement.** The only reader was the author, who read the intent back into the text.
- **There was no before snapshot.** The old statuses were gone, and the repair of the repair had no starting point.
- **The log said "the shared role".** The question "who ran this?" had an answer only through chat history.

Recovery starts from a backup, reconciles whatever happened since, and informs everyone downstream (Chapter 13 and Chapter 15 deal with those mechanics). The design set its cost before the statement was typed.

Now run the same afternoon through the target state. Robin writes the fix with its header and an expected count of 40, and Alex reviews it. The JIT request names `UPDATE` on `invoices`, column `status`, and the grant lasts an hour. If the filter had been dropped from the reviewed script, the runner would count nine thousand against an expected 40 and refuse before any statement ran. A statement typed into the person's own client would not be forwarded as a write (the gateway rule of Chapter 9), and in a plain `human_readonly` session the `UPDATE` would not run at all.

The guard does not make a fix correct. It makes one class of error loud: the statement touched a different number of rows than the author said. A statement that touches exactly 40 rows, but the wrong 40, passes, which is why the review, the postcondition and the snapshot exist as separate layers, and why Chapter 9 says what analysis cannot see. The guard is also only as good as its number. An expected count copied from the run's own output teaches the runner to agree with itself, so the number has to come from a query the reviewer can run independently, ideally before the statement was written. Where the engine has column-level grants (Chapter 3), a narrow grant limits what the cases the guard misses can reach.

Trade-offs and costs

The procedure is slower than typing a statement. A request, a review, an approval and a runner add minutes on a good day and much more when the approver is asleep. On a Thursday afternoon that is acceptable.

At night, under an alert, it is the cost that tempts people toward break-glass, and it deserves design attention: a reviewer on the rotation, the run-book pre-approval above for the repairs that qualify, and a stated limit for the longest the team will wait for a night approval, written down before the first night it is needed.

The review can become a reflex. An approval given in seconds adds friction without reducing exposure; Chapter 8 makes the full argument. The signal specific to this chapter is the pull request: if reviewers never ask for the count query's output and never request a change to a statement, they are reading nothing or reviewing only perfect fixes. The defenses are the checklist items that need an artifact: the count output and preview in the pull request, the dry-run report on the request, and an explicit comparison of the statement with the privileges.

Transactions hold locks. The runner starts the transaction after the review and ends it in seconds. A procedure that opens it first, "to be safe", and lets the human decide afterward makes other writers wait, fills the pooler's connections, and lets Chapter 5's timeouts fire on unrelated work. Keep the human decision outside the transaction.

Large fixes need batches. A fix that touches millions of rows cannot run as one transaction without long-held locks and a large transaction log. The usual response is to split it into batches, say a few thousand rows each (an illustrative size, not a recommendation), each with its own count and guard, with a runner batch mode that repeats the same checks. A failure partway leaves the data in a mixed state, so the fix has to be written to be re-run: the filter selects only rows still in the old state. A large repair is still a data fix; only the first population of a new structure belongs to a migration (Chapter 11).

Snapshots and the fix repository need an owner. A snapshot without an expiry has traded one risk for another, and for Confidential columns the second can be the larger. A directory of fix scripts is a record of how production drifted, and without an index by tag nobody will look for the previous occurrence of a problem.

When not to use all of this. In a small team with a database that holds nothing sensitive, a pull request for every status change is more ceremony than the risk justifies; counting first and using a transaction still pay for

themselves. On a development or staging database with no production data, none of it applies, and for a copy rebuilt nightly, such as `ledgerly_analytics`, a mistaken write costs a rebuild. The procedure scales with the cost of being wrong, and the primary is where that cost is highest.

Common mistakes

- **Treating the `WHERE` clause as the guard.** It is tempting because the clause is where the author's intent lives. But a clause is text, and text can be cut, mistyped or run in part. The remedy is a guard that compares the changed count with a declared expectation and rolls back on mismatch.
- **Counting on the replica and updating on the primary.** The replica lags, so the two numbers can differ for a good reason. The remedy is to take the authoritative count on the primary, inside the transaction, and to use the replica count only to explore.
- **Retyping the reviewed statement.** The reviewer approved a file, and a person copied it into an editor and changed one thing "while there". The remedy is a runner that compares the script's hash with the `approved_hash` in the request record, together with a gateway that forwards the session's writes only from that runner. On a gateway that only brokers credentials the guard is a convention, and the review has to carry it.
- **Asking for a broad grant "to be safe".** It makes the approval a judgment about trust instead of a check against a statement, and it widens what a mistake can reach. The remedy is to request the triples the statement needs, and to make a second request if the situation changes.
- **Using break-glass because the approver is slow.** The path is shorter and the fix is urgent. The remedy is to solve the slow night approval and keep break-glass for the emergencies it was defined for.
- **Writing a Restricted column through a "special" session.** Updating an `iban` directly feels like a kindness to the customer, and it removes validation, approval and the application's record in one step. The remedy is the application path, even when it is slower.

- **Saving a snapshot nobody governs.** The snapshot protects against a bad fix and becomes an ungoverned copy of Confidential data in a shared folder. The remedy is an owner, an access rule matching the source table and an expiry date.
- **Skipping the rollback sentence because the fix is small.** Small fixes are the ones that get reversed in a hurry. The remedy is a one-line plan, written before the run, that says what the snapshot restores and where it must not.
- **Leaving a repeated fix as a script forever.** The result is a recurring write that needs a human on the primary every time. The remedy is a periodic look at the tags in the fix directory, and a decision, even if the decision is "not yet".
- **Reviewing for correctness of intent instead of correctness of statement.** A reviewer who debates whether the repair is worth doing may never read the statement. The remedy is a checklist that asks the narrow questions in order.

Key takeaways

- A read-only session is the right default for investigation, and it works only if the common task needs no approval.
- Writing is a different act from reading, so it goes through its own request, review, time limit and record. The request carries a `fix_id` and an `approved_hash` taken from the merge record, its triples are derived from the fix, and the approver sees the statement.
- The safe order of a data fix is count, snapshot, transaction, row-count guard, postcondition, commit, and the runner, not the person's memory, should enforce it, with the gateway forwarding the session's writes only from the runner.
- A guard makes one class of error loud and does not make a fix correct, which is why review, narrow grants and a before snapshot exist as separate layers.
- A change to a Restricted column is not a data fix: it goes through `admin-console` to `payments-service` with a second approver and a record in `audit_events`, and break-glass is not a route around that.

- A repeated manual fix is a missing product feature, and each one that becomes an `admin-console` action is a recurring write that no longer needs a person on the primary.
- When a statement reaches every row, the useful question is what in the design allowed it, and the answer is a list of missing guards, not a person.

Data fixes change what is in the tables; the next kind of change alters the tables themselves. Chapter 11 turns to schema changes, to who runs them and with which identity, and to the habit of making them backward compatible so that the repair of one day is not the outage of the next.

Chapter 11 — Schema Changes and Migration Access

Imagine a Thursday afternoon at Ledgerly. The invoicing team wants a new column on `invoices` so that tax can be shown separately from the total, and the change is small: one column, a default, a backfill for old rows. Robin, who is on call this week, reviews the pull request and approves it. The pipeline runs the migration as `ledgerly_migrator`, the deploy finishes, and for about ninety seconds every request to `billing-api` that touches an invoice hangs and then fails.

Nothing in the migration was wrong, and nobody did anything unusual. A long `invoice-worker` batch transaction, left open while it waited on an external mail service, held a lock on the table, the migration queued behind it, and every statement that arrived after the migration queued behind the migration. Alex from the platform team finds the cause quickly. Then comes the harder question: the migration ran with the power of the owner of every table, it ran unattended, and the only thing between that power and the ledger was a review of a diff. How should schema changes reach the primary, with which identity, and how do we keep them from becoming either a hole in the access model or an outage?

This chapter adds one step of the target state: migrations run as `ledgerly_migrator` from the pipeline, and nothing else has the power to change the schema.

In the five-part model of Chapter 1, a migration is **Execution** with the widest bound there is. The pipeline holds the owner's privileges by design, so the risk is not in who holds a credential. It is in what the migration is authorized to do, how its statements execute, and what record survives. For migrations that means a short list of statement classes allowed on each lane (authorization), a path that decides when and how a statement runs, with a short lock wait and a retry (execution), and a record that the owner cannot rewrite (accountability).

The problem, and why it is hard

A schema change is the most privileged thing most teams do to a database on a routine day. Ownership lets the owner drop a table, change a column's type, rewrite grants and, on many engines, bypass the restrictions that apply to everyone else. Chapter 3 gave that power to `ledgerly_owner`, a role with no login, and let `ledgerly_migrator` assume it only while a migration runs. That arrangement answers who owns the schema. It does not answer who may start a migration, what a running migration is allowed to do, or what happens to everyone else while it runs.

Three separate risks sit inside the one activity.

The first is **authority**. Whoever can trigger the pipeline's migration step can, in effect, run owner-level statements on production. What is standing is the authority, in the sense Chapter 3 described: a standing owner authority, exercised through a short-lived credential. The bounds on it are procedural: the credential lives only in the pipeline, it expires with the job, and the pipeline is the only sanctioned way to use it. Those bounds are only as good as the pipeline's own access controls, which makes the pipeline part of the database's security boundary whether or not anyone drew it that way.

The second is **availability**. Many schema changes need locks, and the lock is where the damage happens. A statement that must wait for a lock does not only wait: on many engines, the statements that arrive behind it queue as well, so a migration that cannot get its lock in the first second can close the table to everyone for as long as the oldest transaction in front of it stays open. The size of the change hardly matters. A one-column change can be as disruptive as a rewrite if the wrong transaction is in the way.

The third is **drift**. A schema that is defined by migrations and also edited by hand is a schema that nobody can fully describe. A hotfix index added from a console on a bad night is useful, and a year later it is a difference between environments that nobody remembers, along with grants that match no file.

Each can be reduced by design. A migration role that is bounded by the pipeline, a lock wait that fails fast, a change shape that never needs a long lock, and a check that compares the live schema with the expected one are

all things you can build. A policy that says "be careful with migrations" is not on that list.

Who runs DDL, and as which role

DDL (data definition language) is the family of statements that create, alter and drop schema objects, as opposed to the statements that read and write rows. The rule of this chapter is short: DDL runs as `ledgerly_migrator`, from the pipeline, and no application role can run it.

The application roles hold no DDL

None of the `app_*` roles from Chapter 3 owns a table or may create, alter or drop one. This is easy to say and a little costly to keep, because many frameworks can create or change tables at startup, and some teams enable that because it removes a step from deployment. The cost of that convenience is that the service then needs ownership, and a compromised or confused service can drop and re-grant. I would turn automatic schema management off in production, and keep it for development databases where nobody has to explain a dropped table.

The visible side effect is that a deploy that includes a schema change is now two steps: the migration, then the service rollout. The ordering between them is a design question that the expand and contract pattern below answers.

The migrator and the owner

`ledgerly_migrator` has a login that the pipeline uses; it is a member of `ledgerly_owner`, and it holds no data privileges of its own. Membership gives the migrator the owner's privileges, but it does not make the migrator the owner. On many engines a new object belongs to whichever role is active when it is created, so a migration that runs under the migrator's own login would leave its tables owned by the migrator. The rule is therefore the same on every engine: a migration always switches to `ledgerly_owner` explicitly, because ownership follows the active role and not the inherited privileges. Every object it creates is then owned by `ledgerly_owner`, and the migration returns to the migrator's own role when it ends. In notation that belongs to no engine:

```

function runMigration(migration):
    login = fetchMigratorLogin(identity: pipeline.workloadIdentity)
    connection = connect(login: login, database: "ledgerly_primary")
    // lock_wait 3s, statement timeout none (Chapter 5)
    applyRoleSettings(connection, role: "ledgerly_migrator")
    if isApplied(connection, migration.id):
        close(connection)
        return // a retry never re-runs
    applied work
    // external record first: the owner can rewrite the database's own
    record
    auditLogAppend({ actor: login.name, action: "migration-start",
                    migration: migration.id, commit: migration.commit
    })

    try:
        assumeRole(connection, "ledgerly_owner")
        execute(connection, migration.statements)
        recordApplied(connection, migration.id)
    on LockWaitExceeded or DatabaseError as error:
        // owner rights end with the connection on every path
        resetRole(connection)
        close(connection)
        auditLogAppend({ actor: login.name, action: "migration-
failed",
                        migration: migration.id, error: error.kind })
        raise error
        resetRole(connection)
        close(connection)
        auditLogAppend({ actor: login.name, action: "migration-applied",
                        migration: migration.id })

```

Four details matter. The credential is issued to this job by the pipeline's workload identity at run time (the pattern of Chapter 4) and expires with the job, so a copied pipeline variable cannot hold a working owner credential for months and nothing in the repository holds one. The switch to the owner role is scoped to the statements of one migration. The record that a migration was applied is written in the same run, so the database knows its own version, which the drift check below depends on. And the external entry is written before the first owner statement, because `recordApplied` lives in a database the owner can rewrite. The cost is that an unavailable audit store stops the migration before it begins. That is the opposite order from the data fix in Chapter 10, where the gateway's session log already records the statements: here nothing but the pipeline sees what the owner's session does, so the record comes first. Every start entry

gets a matching end, `migration-failed` or `migration-applied`, and `isApplied` makes a retry safe even when the final audit append is the step that failed.

On some engines, recording the version and running the DDL can share one transaction, and on others a DDL statement ends the transaction around it. Know which kind you have. If a failure can leave the schema changed and the version unrecorded, the migration needs to be written so that running it a second time is harmless.

Humans do not hold the migrator

Humans never connect as `ledgerly_migrator`. If a person needs a schema change, the person writes a migration, and the pipeline runs it. That sounds like friction, and it is, but it buys one useful property: every use of the owner's power corresponds to a pipeline run, which has a commit, a review and a log. When the migrator credential also sits in a team password manager, that property disappears, and so does the reason for having the migrator at all.

What the pipeline can reach

Giving the pipeline the migrator credential moves a large power into a system that many people can influence. It is worth being explicit about that, because it is where this design can quietly go wrong.

What the pipeline can do

A job that runs `runMigration` can create, alter or drop any object in the schema, change any grant on objects the owner owns, and change the two append-only tables, `ledger_entries` and `audit_events`. No application role and no human role can edit rows in those tables, including `human_breakglass`; the one structural exception is that the owner, and so a running migration, can alter them. The control for that exception is not a privilege. It is the review.

So the question about the pipeline is: who can make it run something? That includes people who merge to the deployment branch, people who can edit the pipeline definition, people who can add a step to a job that

holds the credential, and anyone who can read the job's environment while it runs.

Bounding the reach

I would bound that reach in five ways, in rough order of how much each one removes.

1. **Only one job holds the credential.** The migration job for the production environment is the only place the migrator login can be issued, and it is issued by the job's identity, not by a variable that every job can read. Builds of unreviewed branches, test jobs and pull request pipelines never receive it.
2. **The pipeline definition is reviewed like the migrations.** A change to the migration job, to its permissions or to the branch rules that trigger it is a change to the owner's reach, and it needs the same reviewer who would check a migration.
3. **The trigger is a protected branch.** Merging to it requires a review by a person who is not the author, which is Ledgerly's four-eyes control for schema changes. For migrations that touch grants, ownership or the append-only tables, the reviewer must come from the platform team.
4. **The credential is short-lived.** A job that fails and leaves a log with a secret in it leaves a secret that has already expired.
5. **Every run is recorded.** The pipeline writes what it applied, as which login, from which commit, to a store the pipeline cannot rewrite, and it writes the start entry before the migration runs. Chapter 14 covers where that store lives.

The review itself can be partly mechanical. A small check can read the migration before a person does and flag the statements that need a closer look:

```

function classifyMigration(migration, lane):
    flags = []
    refusals = []
    for each statement in migration.statements:
        if statement.touches("ledger_entries") or
statement.touches("audit_events"):
            add(flags, "append-only table: platform review required")
            if statement.kind in ["GRANT", "REVOKE", "CHANGE OWNER",
                                "CHANGE DEFAULT PRIVILEGES", "CREATE
VIEW"]:
                add(flags, "changes access: platform review required")
            if statement.kind in ["CREATE TABLE", "ADD COLUMN"] and
(statement.table is new or statement.table in
sensitiveTables) and
not hasExplicitGrants(migration, statement.table):
                add(flags, "new structure without a grant decision")
            if statement.kind in ["INSERT", "UPDATE", "DELETE"]:
                add(flags, "data change in a migration: backfill rule
applies")
            if statement.kind in lockHeavyKinds:
                add(flags, "long lock: scheduled window and a named
watcher")
            if statement.kind in ["DROP TABLE", "DROP COLUMN"] and not
migration.isContractStep:
                add(flags, "destructive change outside a contract step")
            if lane == "expedited" and statement.kind not in
expeditedAllowlist:
                add(refusals, "not allowed in the expedited lane: use the
normal lane")
    return flags, refusals

```

In the normal lane the check does not decide anything. It makes the reviewer's attention land on the lines that change access, hold a lock or destroy data, and it makes the omission of a grant visible. A check that blocks a merge on every flag, with no way to acknowledge one, will be bypassed within a month, so a flag requires a named approver and does not forbid the change. The expedited lane is the one place where the check does refuse: a non-empty `refusals` list fails the job before it runs, and no approver can override it. The next section explains why.

The normal lane refuses nothing outright, and that is a choice: a hard list of forbidden statements would be a second policy beside the deny, gate and allow matrix of Chapter 9. Dropping `ledger_entries` is flagged twice, as an append-only change and as a destructive one, and the platform team has to say yes to it.

Why not a person with a ticket

Some teams prefer that a DBA applies each migration by hand after approval. For a small database with one or two schema changes a month that can be a reasonable choice, and the pipeline's reach question disappears. The cost shows up when the person is away, when the change must be applied identically to several environments, or when the manual step drifts from what was reviewed. The pipeline's weakness is its breadth of reach; the manual path's weakness is that nothing enforces that what was applied equals what was approved. I lean toward the pipeline with the bounds above, and I would revisit the choice if the pipeline's own access list proved impossible to keep short.

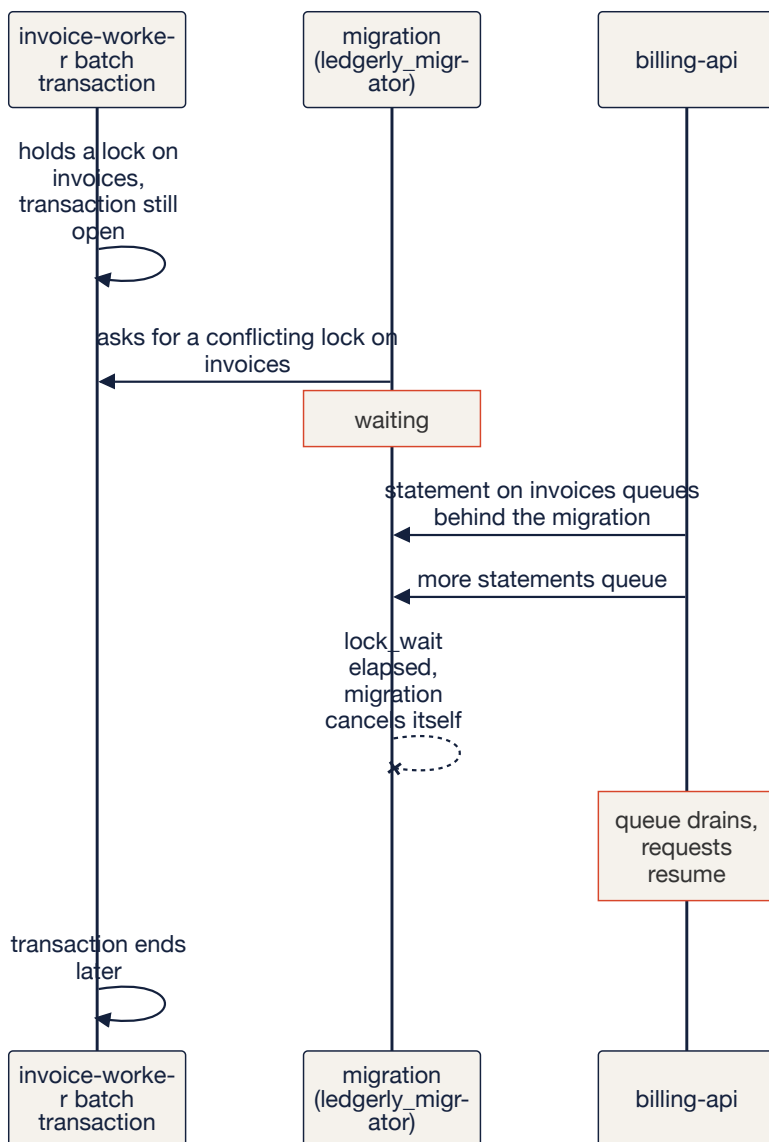
Locks and the availability risk

Now back to the ninety seconds at Ledgerly. The mechanism behind them is common across engines, and the defense is built from a handful of habits.

How a waiting migration closes a table

A migration that alters `invoices` needs a lock that conflicts with ordinary reads and writes. It can only take that lock when no transaction that holds a conflicting lock is still open. If a long transaction is open, the migration waits. While it waits, new statements on the table queue behind the migration's pending request, because letting them jump ahead would starve it. The table is then effectively closed until the old transaction ends or the migration gives up.

The sequence below shows the order of events. Look for the point at which the lock wait gives the migration an exit before the queue grows.



The point is the self-cancellation. Without a lock wait, the migration waits as long as that batch transaction stays open, and **billing-api** queues for that whole time. With a short lock wait, the migration fails after a few seconds, the queue drains and the pipeline can try again later.

A short lock wait and a patient retry

Chapter 5 set `lock_wait 3s` and no statement timeout for `ledgerly_migrator`. That pairing is deliberate. A schema change may need a long time to do its work, but it should wait a very short time to get started. The retry then belongs to the pipeline:

```
function applyWithRetry(migration, maxAttempts, pauseSeconds):
    attempt = 1
    while attempt <= maxAttempts:
        try:
            runMigration(migration)
            return
        on LockWaitExceeded:
            // a clean no-op only if nothing had committed: one
transactional
            // unit, or a locking step that comes first
            if not migration.retrySafe:
                notify("platform", "partial migration, not retried: "
+ migration.id)
                raise MigrationBlocked
            notify("platform", "migration waiting on a lock: " +
migration.id)
            sleep(pauseSeconds * attempt)
            attempt = attempt + 1
            // the table is busy for longer than the retry window; a person
decides
            notify("platform", "migration blocked after retries: " +
migration.id)
            raise MigrationBlocked
```

A few remarks on this. A failed lock wait is a clean no-op only when nothing has committed yet, which is why a migration carries a `retrySafe` mark: it is one transactional unit, or its locking step comes first. A multi-step migration on an engine where each DDL statement commits on its own is not retried blindly, because the second attempt would meet the first attempt's work. It stops and a person decides. The pause grows with the attempt count so that repeated tries do not themselves become a pattern of load. And the final state, `MigrationBlocked`, is a human question: find out which transaction is in the way, and decide whether to wait, to cancel it or to reschedule. That decision involves killing someone's session, which is an access question of its own and belongs to Chapter 15 when it happens during an incident.

Long statements and online changes

Some operations keep a lock for as long as they run, such as rewriting a large table to change a column's type. For those, the lock wait is not the problem; the duration is. Engines differ on which statements are quick, and the answer changes with the engine's version, so check each one.

Online schema change tools and non-blocking index builds avoid the long lock, but they create and drop objects, so they need the same owner reach as any migration and run as the migrator, inside the pipeline.

`classifyMigration` marks these statements as long-lock, and a long-lock migration gets a scheduled window and a named person watching, written in the migration's description. That is not a failure of automation; some changes are events.

Expand and contract

The safest schema change is one that old and new application code can both live with. The pattern for this is **expand and contract**: change the schema in additive steps, move the application across, and only then remove what is no longer used. It matters for this book because each step is a small migration with a small lock, and because it keeps deploys reversible.

Consider a hypothetical change at Ledgerly, used only to show the shape: renaming `total_minor` on `invoices` to `amount_minor`. The canonical schema keeps `total_minor`, and the example assumes a variant in which the column is renamed. The steps look like this:

1. **Expand.** A migration adds `amount_minor` as a nullable column, with a brief lock. Old code ignores it.
2. **Write both.** New releases of `billing-api` and `invoice-worker` write the new column as well as the old one, so old and new releases can run side by side.
3. **Backfill.** Old rows get values for the new column, as described below.
4. **Read the new column.** If the release misbehaves, the old column is still correct, so the rollback is a deploy and not a repair.
5. **Contract.** After a period in which nothing reads or writes `total_minor`, a final migration drops it. This is the only destructive

step, and it is gated separately: reviewed as destructive, and never in the release that introduced the new column.

Expand and contract costs time and attention. A change that could be one migration becomes four or five, spread over several deploys, and during the middle steps the application carries code for both shapes. For a small team with one service and a maintenance window, a single migration may be the better choice. For Ledgerly, with five services reading the same tables and no window the business accepts, the pattern is what keeps the lock short and the rollback cheap.

Backfills inside the migration, and data fixes outside it

Step 3 is a bulk change to data, and the same rule would apply to a split of the total into net and tax, with one difference: the old table holds no tax figure, so that backfill could not be computed and would have to leave the new columns empty until a data fix filled them. Chapter 10 defines a data fix as a change to production data made by a person outside the normal application path, to repair a state the application cannot repair, so the book needs a line here: a backfill that belongs to a schema change runs inside the migration as `ledgerly_migrator`; any other bulk change to data is a data fix and follows Chapter 10.

What makes a backfill belong to the schema change?

- Its only purpose is to give the new structure its first values, by a rule stated in the migration.
- It is described in the migration, reviewed with it and applied by the pipeline.
- Every value it writes could be recomputed from data already in the table.

If any of these is false, it is a data fix. The rename shows where the line falls: every value written to `amount_minor` is `total_minor` of the same row, a rule the migration states and a reviewer can check. Suppose the team notices, while backfilling, that some old invoices have a wrong `total_minor`. Correcting those totals is not part of the schema change. It follows Chapter 10, with a count of the affected rows, a guard on the ex-

pected row count and a second reviewer, because a copy that preserves a wrong value is still a correct backfill. The migrator's power is large, and it should not become the route for every awkward change.

On a large table, the backfill runs in batches, each a short transaction on a bounded set of rows. It pauses when replica lag grows (Chapter 6), gives up visibly if the lag does not clear, and can be stopped and restarted without losing work.

The same rule applies to the two append-only tables. A migration may add a column to `ledger_entries`, but it does not rewrite existing entries to fill that column with corrected values; a correction is a new entry, issued through the correction path of `payments-service`. A migration that does touch those tables is flagged by the classification step above and goes to the platform team for review.

Hotfix DDL and drift

Every team eventually faces the night when an index would fix a slow query, and the pipeline takes, say, twenty minutes. The question is what the access model says about the shortcut.

There is no console shortcut, so build a fast lane

In the target state nobody holds the owner's login. `human_breakglass` writes `INSERT` and `UPDATE` and carries no DDL, and the superuser is a sealed instrument that is not used for daily work (Chapter 3). So a person on the primary has no legitimate way to add an index by hand. That is deliberate, and it moves the question to the pipeline: how quickly can it run a reviewed change when the need is urgent?

I would give the pipeline an **expedited lane** rather than a bypass. The lane uses the same job, the same credential and the same record; it differs in timing, not in who must approve. Four-eyes stays: a reviewer who is not the author still approves, and the author does not merge alone. What relaxes is the scheduled waiting period and, for a short allowlist of statement classes, the platform-team requirement, with a follow-up review by the platform team the next working day. `classifyMigration` enforces the allowlist: building an index, dropping an index and adding a nullable column are eligible. A statement that changes grants, ownership or a view,

touches an append-only table or destroys data is not, because those are the changes the platform review exists for, and a migration that contains one is refused in the lane and goes through the normal one. Chapter 15 describes access during incidents; the point here is that urgency changes the speed of the process, not the identity that runs the DDL or the review a dangerous statement gets.

The cost is that the expedited lane is a standing option to skip some caution, and an option like that gets used. Track how often it is used, and treat a rising number as a sign that the normal lane is too slow, not that the team has become less careful.

Drift

Drift is any difference between the schema the migrations describe and the schema that exists in production. It enters through a hand-applied hot-fix, through a migration that was edited after it was applied, through an owner-level change by a tool that nobody remembered, and through grants altered outside the access definition. Drift in the schema surprises the next migration. Drift in grants is an access change that no review saw.

A scheduled check can find both kinds:

```
function detectDrift(environment):
  expected = buildExpectedSchema(appliedMigrations(environment))
  actual = readLiveSchema(environment)

  findings = []
  for each difference in compare(expected, actual):
    add(findings, difference)
  for each difference in compare(expectedGrants(),
    readLiveGrants(environment)):
    add(findings, difference)

  if length(findings) > 0:
    notify("platform", "schema or grant drift in " + environment)
    recordCheckRun(environment, findings)

function checkDriftCheckAge(environment):
  // scheduled separately, so a stopped drift job cannot hide itself
  if now() - lastSuccessfulRun("detectDrift", environment) >
    maxRunAge:
    notify("platform", "drift check has not completed recently")
```

Two things make this useful. The expected state is derived from the migrations and the access definition from Chapter 3, so there is exactly one source of truth to compare with. And the check covers grants, not only tables, because a grant that appears outside any file is the more important kind of drift for this book. When a difference is found, the resolution is one of two: write a migration that records the change, if it should stay, or write one that reverses it, if it should not. Resolving drift by editing the record of what was applied hides the change instead of reviewing it.

Note: Do not "fix" drift by applying a corrective statement directly as the superuser or the owner. The statement that removes the difference is itself an unreviewed change to production. Put it in a migration, and let it go through the lane.

New tables, new columns, and the grants that follow

Chapter 3 chose an empty default for new tables: a table is readable by nobody except the owner until a migration states a grant. This section is the other half of that decision, because it is in migrations that the decision either holds or is forgotten.

Grants live in the migration

Each migration that creates a table states its grants in the same file. A reviewer then sees the entire access effect of a change in one diff, which is the argument for the empty default. For a new table,

`invoice_adjustments` (an illustrative table, not part of the canonical schema), the file looks like this. Its free-text `reason` column is classified Confidential for the worst case, as `invoice_lines.description` is, so readers other than the owner of the table get column-level grants that leave it out:

```

CREATE TABLE invoice_adjustments (
    id          text primary key,
    invoice_id  text not null,
    reason      text not null,
    delta_minor integer not null,
    created_at  timestamp not null
);

// the migration states every grant it means to give
GRANT SELECT, INSERT ON invoice_adjustments TO app_billing_api;
// reason is Confidential free text: left out of the broad read grants
GRANT SELECT (id, invoice_id, delta_minor, created_at) ON
invoice_adjustments TO app_reporting_job;
GRANT SELECT (id, invoice_id, delta_minor, created_at) ON
invoice_adjustments TO human_readonly;

```

Forgetting a grant has two possible outcomes, and the difference is the reason for the empty default. With an empty default, the service that needs the table fails on its first query. That is loud, it is found in a test or at the first deploy, and it is fixed by a one-line migration. With a broad default privilege, a forgotten grant is silent in the other direction: the next table is readable by whoever the default names, and nobody notices because nothing fails. A noisy failure is cheaper than a quiet exposure.

The failure is only noisy if something runs the query. This is why I would include, in the pipeline, a post-migration smoke step that checks that each application role can use the tables it is supposed to use. The application roles have no login, so the step needs a mechanism. I would use a test login that is a member of each application role and switches to each in turn, where the engine supports it; the alternative is one test login per role. Either choice widens the pipeline's reach, because a login that can become any service role is worth more to an attacker than any one service credential. It is therefore issued by the job's workload identity, expires with the job and is used for this step only. In return, the step catches the forgotten grant before the service rollout does.

New columns on tables that already have grants

A new column behaves differently depending on how its table is granted.

Table grant style	A new column is...	Consequence
Table-level grant	readable by every role that holds the grant	A new sensitive column is exposed at once
Column-level grant	readable by nobody until named	A new column must be granted to be used

For tables with sensitive columns, such as `customers`, `bank_accounts` and `payment_tokens`, that argues for column-level grants where the engine supports them, which is the choice Chapter 3 made. The new column then starts closed, and the migration that adds it is where the team decides which roles may read it and whether it is Confidential or Restricted (Chapter 12 covers classification). A migration that adds a column to one of those tables without any grant statement is a prompt for a question, and the classification check can ask it.

Where the engine lacks column-level grants, the fallback from Chapter 3 applies: Restricted columns live in separate tables such as `bank_account_secrets` and `payment_token_values`. A new Restricted column belongs in one of those tables, never in the main table, because the main table's grants are table-level and the ceiling for JIT access is drawn around it. This is easy to violate by accident, since adding a column to the table everyone already knows about is the natural move, and it is the sort of mistake only a reviewer who knows the fallback will notice.

Views over sensitive tables

A view runs with the rights of its owner on many engines, which means a view over a table with Restricted columns can reveal what its readers' own roles could not. That is why `classifyMigration` flags `CREATE VIEW` as an access change, why the migration that creates one goes through the platform team, and why a view over a sensitive table is judged as an access change and not as a convenience.

Trade-offs and costs

The pipeline is a high-value target. A credential that can run owner-level DDL, held by a system that many people can edit, is a concentration of power. A reviewed pipeline does not stop someone with owner-level reach

from doing something unreviewed; it makes that something visible after the fact, which is a weaker control than it sounds. The bounds in this chapter reduce the risk without removing it. If your pipeline's access list is long and nobody owns it, fix that before relying on any of the rest.

Reviews add delay and are only as good as the reviewer. A four-eyes check on migrations slows small changes and can become a formality, especially when the reviewer sees, say, ten migrations a week and every one is a column. The classification check helps by pointing at the dangerous lines. It does not replace attention. A review that is always approved within seconds is the signal to watch, and Chapter 8 makes the full argument.

Expand and contract multiplies the number of changes. One logical change becomes several deploys. For a team with a maintenance window, one service and a small table, it may not be worth it. The pattern pays when many things read the table and nobody can stop them for the length of a change.

A short lock wait makes migrations fail. That is the intended behavior, and it is a cost. On a busy table a migration may need several attempts, and someone must decide what happens after the last one. If you set the wait too short for your workload, the pipeline will page the platform team for noise. Tune it from what your own long transactions look like, and say so in the runbook.

Drift checks produce findings that someone must own. A check that alerts a channel nobody reads finds nothing. Decide who owns each class of finding before you turn the check on.

When not to use this design. For a throwaway database, a prototype or a single-engineer project with no customer data, a migrator with an owner role and a reviewed pipeline is more than the situation asks for. Run migrations from a developer's machine, keep the production database off the critical path, and add the pipeline when someone besides you depends on the data. The design in this chapter is for the point at which the schema is shared by services, and a wrong change costs customers something.

Common mistakes

- **Letting the service create or alter its own tables at startup.** It removes a deploy step and gives the service ownership, which includes the power to drop and re-grant. Keep automatic schema management off in production, and run migrations as a separate step with the migrator.
- **Keeping the migrator credential in a team password manager "for emergencies".** It is tempting because the pipeline sometimes fails at the worst time. It turns every holder into an owner and removes the property that every use of ownership matches a pipeline run. Build the expedited lane instead.
- **Running a migration with no lock wait.** The migration waits as long as the oldest transaction in front of it, and every statement behind it waits too. Give the migrator a short lock wait and put the retry in the pipeline.
- **Hiding a data fix inside a backfill.** It feels efficient to correct bad values while the table is open anyway. The migration's review was designed for a column definition, not for a change to customers' amounts. If the change is not a first fill of the new structure by a stated rule, it is a data fix and follows Chapter 10.
- **Dropping the old column in the same release as the new one.** It saves a migration and removes the rollback. Keep the contract step separate, delayed and reviewed as destructive.
- **Relying on default privileges so new tables work without a grant line.** A broad default turns a forgotten grant into a silent exposure. Keep the default empty and let each migration state its grants.
- **Adding a Restricted column to a main table on an engine with no column-level grants.** The table's grants are table-level and the JIT ceiling is built around them. Put Restricted columns in the separate secrets tables.
- **Treating a drift finding as noise.** A grant that exists in production and nowhere in code is an access decision nobody reviewed. Resolve it with a migration that either records or reverses it.
- **Editing a migration after it was applied.** The environments that ran the old version now differ from those that run the new one. Add a new migration instead.

Key takeaways

- DDL runs as `ledgerly_migrator` from the pipeline, as the owner for the length of one migration, and no application role holds any DDL privilege.
- The migrator holds a standing owner authority, exercised through a short-lived credential: one job holds it, it is issued by workload identity, every run is recorded before it starts, and humans never use it.
- Because the pipeline can reach the owner's power, its own access list and definition are reviewed like a migration; migrations that touch grants or the append-only tables go to the platform team, and the expedited lane accepts only an allowlist of statement classes that `classifyMigration` enforces.
- A short lock wait with a retry in the pipeline keeps a blocked migration from closing a table, and online tools and additive steps handle changes that would otherwise hold a lock for a long time.
- Expand and contract keeps old and new code working together and makes rollback a deploy, at the cost of more migrations; a backfill that fills the new structure by a stated rule runs inside the migration, and any other bulk change to data is a data fix and follows Chapter 10.
- Drift in schema or grants is found by comparing the live state with what migrations and the access definition describe, and it is resolved by a migration, not by a quiet edit.
- Each migration states the grants for what it creates and keeps defaults empty, so a forgotten grant fails loudly instead of exposing a table quietly.

Schema changes are the one place where a pipeline, not a person, holds the most power over the data. The next part turns from who can reach the database to what is inside it: Chapter 12 asks which columns deserve protection, how to classify them, and how to mask or withhold them without breaking the work that needs them.

PART IV

Protecting the Data

Chapter 12 — Sensitive Data: Classification, Masking and Minimization

Imagine a Monday at Ledgerly. A customer writes to support to say that a payout has not arrived and asks whether the bank account on file is the right one. Sam, in support, opens the customer's page in `admin-console` and finds the whole account number on the screen. Reading it back in chat is the quickest way to answer, and a screenshot in the ticket is the quickest way to hand the case to a colleague. Nobody in the story does anything unreasonable, and by the end of the afternoon the number lives in a chat log and a ticket attachment.

Two desks away, Jordan from the data team is building a report on `ledgerly_analytics` and notices that the `customers` table there carries contact names, emails and phone numbers that no report needs. Robin, who is on call, is reading an application log to find out why a reminder failed, and finds the recipient's email address inside the error message. Each exposure is small, and the three share a shape: data arrived in a place where nobody had decided it should be.

This chapter adds one step of the target state: sensitive columns are classified, masked or withheld by default, and the analytics copy holds no raw Restricted data. The question it answers is what each person and each copy actually needs to see, and how to build the system so that it shows that and nothing more.

In the five-part model of this book (**Identity, Grant, Path, Execution, Record**), the chapter works on two parts. The Grant decides which columns a role may read at all, and Execution decides what form of a value comes back through the path. None of it depends on how a credential reached a person; a perfectly guarded login that reads a full account number has still exposed it.

The problem, and why it is hard

The controls from the earlier chapters decide who reaches which table and which columns. They do not know why a column matters. A grant on `customers` treats a status flag and a contact email alike, and the schema calls both of them `text`. Nothing in a column definition says that `invoice_lines.description` is free text that a person may have filled with a name, or that `iban` is the kind of value whose exposure cannot be undone by rotating something.

The second difficulty is that people need some of the data, and a rule that says nobody may see it produces workarounds. Support must answer questions about payout accounts. On-call must find out why a reminder failed. The data team must count customers by segment. When the sanctioned path shows nothing, the unsanctioned ones appear: screenshots, ad-hoc exports, a standing grant "until the project ends". The useful question is not whether a person may see a value, but which form of the value the task needs: the whole value, its last digits, a yes or no, or a count.

The third difficulty is that data does not stay in the table. It is copied into an analytics store, written into log lines, echoed in error messages, pasted into tickets and sent in exports. Every control on the primary stops at the database edge, and the value continues on its way. This is why the answer has to be built into the system and cannot stay a policy. A policy that says "handle personal data with care" cannot be executed by a migration pipeline, a scheduled load or a log formatter. The decisions have to be written into the things that run: the schema, the grants, the load and the logging code.

Data-protection and payment-card regimes, such as GDPR, KVKK or PCI DSS, tend to ask a family of questions: which personal data do you hold, why, where does it go, who can see it and for how long. A classification kept in the schema is the form in which you can answer them. Your legal and compliance people own the interpretation of any regime; this chapter only describes the engineering that makes the answers findable. The catalog is evidence of what the system does, not a claim that it satisfies any regime; Chapter 14 returns to what evidence an access review can collect.

Classification that lives in the schema

Data classification sorts data into named classes, and each class decides handling. Ledgerly uses four.

Class	What it means	Ledgerly examples	Default handling
Restricted	Exposure would damage customers or the company in ways hard to undo	iban, account_holder_name, token_value	Withheld from human_readonly; never in a just-in-time (JIT) request; changed only through an application path; absent from the analytics copy
Confidential	Personal or commercial data about customers and contacts	contact_email, billing_address, provider_reference, invoice_lines.description	Masked or withheld for human_readonly; unmasked only in a JIT SELECT with an approver; masked or dropped in the analytics copy, except the display hints that exist to be shown (provider, expires_on; see the decision table below)
Internal	Operational data without personal content	amount_minor, status, due_on, account_code	Readable by the roles that need it; copied to analytics
Public	Safe to publish	None among the eight tables	Integrity only

The Public row is empty on purpose. It exists so that "can this be published?" has an answer other than silence.

Four rules make the classes usable. They are the part I would write down before choosing any tool.

- **Classify columns, not tables.** customers holds a status flag and a phone number, and the two need different handling. For inventory purposes only, a table takes the highest class among its columns.
- **Classify free text for the worst case.** A column that people type into will eventually hold whatever people type.

`invoice_lines.description` is Confidential because it can contain personal data, and no filter can tell which rows do.

- **A derived value inherits its source's class until someone argues otherwise.** A keyed hash of an `iban` is Restricted until a person has argued that it is not. `payment_tokens.last_four` is Confidential and not Restricted because it exists to be displayed, and that decision was made once and written down.
- **Unknown is Confidential.** A column nobody has classified is treated as Confidential and marked provisional, so that the unfinished work is visible and the default is the closed one.

A catalog in the same repository as the migrations

A classification that lives in a wiki drifts, because nothing that changes the schema reads the wiki. I would keep it as a file in the repository that holds the migrations, and change it in the same pull request that adds the column. Some engines also allow labels or comments on columns; either form works as long as the pipeline can read it.

```
// illustrative catalog entry, one per column
table column_classification:
  table_name          text not null
  column_name         text not null
  class               text not null    // Restricted,
Confidential, Internal, Public
  reason              text not null    // one sentence, for the
reviewer
  owner                text not null    // a team, never a person
  human_readonly       text not null    // readable, masked,
withheld
  analytics_copy        text not null    // copy, transform, drop
  reviewed_on          date null        // null means provisional
```

The catalog makes two checks possible. The first runs against each migration, next to the flags described in Chapter 11. Unlike those flags, which ask a reviewer to look, this one blocks the merge, because classifying a column costs one line and a missing line is the thing the whole control exists to catch.

```

function checkColumnClassification(migration, catalog):
    problems = []
    for each column in addedColumns(migration):
        entry = catalog.find(column.table, column.name)
        if entry is null:
            add(problems, "unclassified: " + column.table + "." +
column.name)
            continue
        if entry.class == "Restricted" and not
inRestrictedHome(column):
            // a main table whose grants are table-level cannot
withhold it
            add(problems, "Restricted outside its home: " +
column.name)
            if entry.class == "Restricted" and entry.analytics_copy !=
"drop":
                add(problems, "Restricted column would reach analytics")
            if problems is not empty:
                raise MergeBlocked(problems)

```

Here `inRestrictedHome` means a table where grants can withhold the column: a main table with column-level grants, or the separate secrets tables of Chapter 3 on engines without them. When the person adding the column is unsure, the way out is not to skip the check but to classify the column as Confidential with `reviewed_on` empty, and let the next access review settle it.

The second check runs on a schedule and compares the live schema with the catalog, the same way Chapter 11 compares the live schema with the migrations. A column created by hand on a bad night shows up as a finding with an owner, and the finding is resolved by a migration and a catalog line, not by a quiet edit.

Minimization: the data you do not hold

Minimization is not collecting, copying or keeping what nobody needs. It is the cheapest control in this chapter, because a value that was never stored has no mask to maintain, no copy to find and no line to redact. Every other control here assumes the data exists; this one asks whether it should.

For each sensitive column, I would ask four questions in this order.

1. Does any feature or duty need this value? If the answer is "it might be useful", the answer is no.
2. Could we hold a surrogate instead? Ledgerly holds `payment_tokens`, provider-issued tokens that stand in for payment instruments, and not the instruments themselves. That one decision removes a whole class of value from the schema.
3. Could we hold less of it? A display hint such as `last_four` lets support and customers recognize an instrument without the instrument.
4. Could we hold it for less time? A closed customer's contact details have a purpose until the last invoice is settled and dispute windows have passed, and after that they are kept only for a short, defined period and then deleted.

Two decisions about where data lives matter more than any retention rule. The first is that **append-only tables should not carry personal data**. `ledger_entries` holds `amount_minor` and `account_code`, both Internal, and that is why a correction can be a new entry: nothing in the table ever needs to be erased to honor a request about a person. `audit_events` refers to its subject by an opaque `subject_id` for the same reason. The second is free text. A structured field with a fixed set of values is classifiable and maskable; a note box is neither, so where a feature can use a code, a status or a product reference instead of a sentence, I would choose that.

Retention is a job, not a sentence in a policy. Something has to run on a schedule, find rows past their period and delete or anonymize them, and that something needs its own narrow privileges, its own audit entries and an owner. I don't know the right period for your data, and neither does this book. It depends on what your contracts, your regulators and your own dispute process require, and your legal people have to answer that before an engineer writes the job. What an engineer can say is that backups, exports and the analytics copy hold old data too, and a deletion that reaches only the primary has not finished. Chapter 13 returns to that.

Masking: dynamic and static

Masking replaces or obscures a value so that the real one is not visible while the shape stays usable. **Redaction** is the blunt version: the value is removed from an output or a log. The two kinds of masking differ in where they act.

	Dynamic masking	Static masking
What changes	What a query returns	The copy itself
When	At read time, per reader	At load time, once
Raw value still stored	Yes, in the same store	Not in the copy
Typical use	human_readonly on a replica	The analytics copy, staging data
How it fails	Engine behavior, filters, direct grants	A stale or over-trimmed copy

Dynamic masking for humans

human_readonly is the default every session starts with, so its masking is the one most people meet. The notation below is illustrative, because engines express masking in very different ways: some offer a mask on the column, some an unmask privilege, some a fixed view per table.

```
// illustrative notation. The grant lists the columns the role may
read;
// each mask rule changes what a read through the role returns
GRANT SELECT (id, created_at, status, contact_name, contact_email,
              contact_phone, billing_address) ON customers TO
human_readonly;

MASK RULE contact_name_initials  ON customers (contact_name)
  FOR human_readonly USING initials();
MASK RULE contact_email_partial  ON customers (contact_email)
  FOR human_readonly USING keep_first(1);
MASK RULE contact_phone_tail     ON customers (contact_phone)
  FOR human_readonly USING keep_last(2);
MASK RULE billing_address_hidden ON customers (billing_address)
  FOR human_readonly USING redact();
MASK RULE provider_reference_tail ON payments (provider_reference)
  FOR human_readonly USING keep_last(4);

// free text has no shape to mask, so it is withheld: not in the
column list
// (illustrative column list)
GRANT SELECT (id, invoice_id, created_at) ON invoice_lines TO
human_readonly;
```

The choice of function follows from what the column is for. A technique is not better or worse in itself; each has a place and a way of failing.

Technique	What a reader gets	Suits	Where it stops helping
Partial	A first or last few characters	A screen, a pasted result	Short values show most of themselves
Redaction	A fixed marker or nothing	Addresses, free text	Cannot confirm anything
Keyed pseudonym	A stable surrogate	Joins across loads	Rotating the key breaks the joins
Coarsening	A broader value, such as a country or a month	Analytics	Combined columns can single out a person (Chapter 13)

A plain hash is not on the list. A hash of a value drawn from a set small enough that every candidate can be tried, such as a phone number, can be reversed by trying them, so a pseudonym needs a key held outside the database.

Not every Confidential column is masked. A column that exists to be shown stays readable: `payment_tokens.provider`, `last_four` and `expires_on` are the display hints that let a person recognize an instrument, and masking them would only push people toward the Restricted value they replace. Contact fields, the billing address and `provider_reference` identify a person or a transaction and are masked. `invoice_lines.description` is withheld.

That gives the rule Chapter 8 depends on. `human_readonly` already reads, so a JIT `SELECT` triple is needed only for a Confidential column that the role masks or withholds, and what it adds is that column unmasked. Restricted columns never appear in a JIT triple, so nothing here changes for them. The request and the approver's screen should therefore carry one explicit line, "Unmasked columns: `contact_email`", separate from the table list, so that the approver sees which masks the request removes and not only which tables it touches.

Warning: Check how your engine resolves a login that reads a column through `human_readonly` and also holds a direct column grant. If the mask wins, a JIT `SELECT` shows nothing new and the approval means nothing. If raw values win for every reader, the mask is decoration. Test both cases on a copy before relying on either.

Two limits belong next to the technique. First, on many engines a mask applies to what a query returns, not to what its conditions compare, so a reader who may filter on a masked column can learn something about its values through counts and matches. Treat dynamic masking as protection against casual exposure, such as a screen, a screenshot or a pasted result. Second, masking an output does nothing for the stored value, which remains in every backup, replica and dump of the primary.

The first limit gives a decision rule for inference. Ask how cheaply a reader could recover the value by filtering: a handful of equality probes on `contact_email` is cheap, and a count of customers by a masked phone prefix is cheap too. Where it is cheap and the reader must not learn the value, withhold the column and do not mask it, or deny predicates on the

masked column for that role. Where it is expensive, such as probing a long free value one character at a time through thousands of logged statements, masking plus the statement record is a reasonable control.

Masking in the result stream

Some access paths cannot rely on the engine to mask. A gateway that sits between the person and the database can apply rules to the result stream itself, before rows reach the screen or a result file. That helps where the engine has no masking feature, and the rules can be the same for every engine behind it.

A stream rule depends on knowing which column a value came from, and an alias such as `SELECT api_key AS k` defeats rules that match on the column name. Tools that mask results, QueryProxy among them, add rules that match on the value as well, which is a second layer, never a substitute for the grant. At the time of writing, it documents this limitation of name-based rules itself, and other tools take similar approaches. The consequence for Ledgerly is that a stream mask sits behind the column grants of Chapter 3, and a column that a role must never see is withheld by the grant, not left to a pattern.

Note: On engines without column-level grants, Ledgerly uses the fallback from Chapter 3: the Restricted columns sit in `bank_account_secrets` and `payment_token_values`. Masking `human_readonly` then usually means one fixed masked view per table, made once by a migration and owned by `ledgerly_owner`, with the role holding `SELECT` on the view and not on the base table. A table-level JIT grant on the base table shows every non-Restricted column raw, Confidential ones included, and the approver has accepted exactly that. This is a fixed view, not a view made per request, and it is a masking mechanism for Confidential columns. It is not a second fallback for Restricted ones: the only fallback for those is the secrets tables.

Static masking for the analytics copy

The analytics copy is where masking is cheapest and most useful, because the raw value can be left out of the copy altogether. Jordan's report needs customer segments, counts and amounts. It does not need names, emails or phone numbers, so for most Confidential columns the right treatment is drop, and a transform is for the few that analysts demonstrably use.

The load is a scheduled job, not a service, and it runs under its own login and the role `app_analytics_load`, like any other principal in the access review. The role holds `SELECT` only on the columns the catalog does not mark Restricted, so the grant is the first guard and the catalog the second. They are not independent, because the grant is derived from the catalog's Restricted marks: a column wrongly classified as Internal passes both. What the grant adds is that a column marked Restricted cannot be read even if a transform entry is wrong, and the independent check is the schema assertion in `verifyMasked`, which compares the copy with the plan. Within that grant, the load decides column by column from the catalog, and it fails closed.

```
function loadAnalyticsTable(table, catalog):
  plan = []
  for each column in sourceColumns(table):
    entry = catalog.find(table, column)
    if entry is null:
      // an unclassified column never travels; the load says so
      notify("platform", "not loaded, unclassified: " + table +
        "." + column)
      continue
    if entry.class == "Restricted" and entry.analytics_copy !=
      "drop":
      raise CatalogInvalid
    if entry.analytics_copy != "drop":
      add(plan, { column: column, treatment:
        entry.analytics_copy })
    // build beside the live table, check, then swap: a failure leaves
    the
    // previous load in place, so the copy never holds a half-masked
    table
    next = table + "_next"
    writeMasked(next, from: table, plan: plan)
    if not verifyMasked(next, plan):
      dropTable(next)
      raise MaskingCheckFailed
    swapTable(next, into: table)
```


`verifyMasked` makes two checks. It asserts the schema: the column set of the target must equal the plan, and no Restricted or dropped column may exist in it. Then it samples rows and fails if a transformed column still equals its source value. The schema assertion catches a column that arrived by a path the plan did not list; the sample catches a transform that was configured wrongly. Neither is a proof that the copy holds nothing sensitive, because the sample reads only some rows.

The price of failing closed is visible: when a developer adds a column and forgets the catalog, the migration check should have stopped it, and if it did not, the analytics table loads without that column and the data team notices a gap. I would accept that. A report with a missing column is a ticket; a copy with an unmasked column is an exposure that nobody notices.

A keyed pseudonym for joins deserves a decision of its own. It lets analysts follow one customer across loads without knowing who the customer is, and it works only while the key stays out of the data team's reach and unchanged. A rotation of the key changes every pseudonym, so history no longer joins. For many teams the opaque `customer_id` already does this job, and a second surrogate adds a key to guard for no gain.

Tokenization and application-level encryption

Tokenization replaces a value with a surrogate that can be mapped back only by a controlled service. It is not masking, because the mapping exists, and it is usually not encryption, because the surrogate is normally not computed from the value at all. Schemes that do compute it from the value under a key are encryption by another name, and they need the same key care. Ledgerly already benefits from it in one place: `token_value` is a provider's token, the provider holds the mapping, and `payments-service` is the only caller. The Ledgerly database therefore never holds the instrument.

Building a second tokenization service for bank account numbers is a larger step. It adds a system to the payment path that must be available when payments are, it needs a list of callers that may map a token back, and every mapping is an event someone should be able to review. I would not build it at Ledgerly's size. Restricting the columns, restricting the one ser-

vice that reads them and encrypting the values would give most of the protection with less to keep running. If a regulated partner or a payment provider requires it, the requirement decides.

Application-level encryption means the service encrypts a value before the database sees it, so roles, replicas, backups and dumps all hold ciphertext. This is its distinct strength: it protects the value in the data copies that a grant cannot reach (Chapter 13). Encryption done by the storage layer, on disk or in a snapshot, protects against a lost volume and does nothing against a query, and teams often mistake it for access control. Application-level encryption is closer to a real boundary, and it is a heavier commitment.

- **The key decides who can read.** The data is only as protected as its key. Give decrypt permission to the workload identity of `payments-service` and nothing else, so the decision is a question of who can decrypt and through what. Key hierarchies, rotation and key backup are a discipline of their own, and the key service's documentation covers them better than this book can.
- **The column loses its query features.** You cannot search, sort, range-filter or enforce uniqueness on ciphertext in the usual way, so choose columns that are only ever read back by identifier.
- **A service that can decrypt on request can be asked to.** Encryption moves the boundary to the service, so its callers, its rate and its audit entries become the control, and the audit entry is written before the decryption, as below.

```
function readPayoutAccount(accountId, onBehalfOf):
    row = selectBankAccount(accountId)           // the stored value is
    ciphertext
    // the audit entry comes first: if the log is unreachable, nothing
    is decrypted
    try:
        auditLogAppend({ actor: "payments-service", action: "decrypt",
                        subject: row.id, on_behalf_of: onBehalfOf })
    on AuditLogUnavailable:
        raise DecryptRefused
    // the key service checks the caller's workload identity, not the
    request
    return keyServiceDecrypt(row.iban, keyId: row.key_id)
```

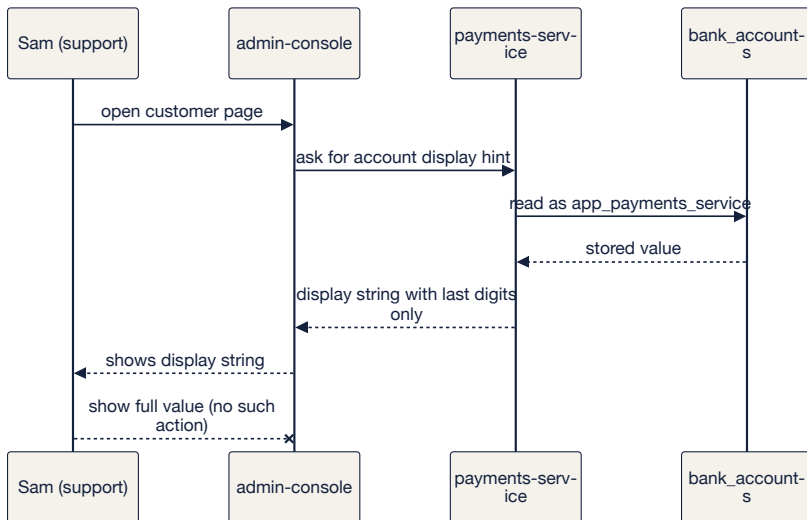
The matrix does not change if you encrypt. `human_breakglass` still reads the Restricted columns, but what it reads is ciphertext, and decrypting needs a key permission the role does not carry. I would decide in advance whether an incident that needs plaintext may ask for key access, with which approver, and write it into the break-glass runbook of Chapter 8. Leaving that choice for the incident is how a rule gets improvised.

Working without seeing: the support case

Back to Sam. The aim is that Sam can answer the customer fully and never sees the full value. Most support questions turn out to need less than the value.

What support needs to do	What it needs to see	How it is provided
Tell the customer which account is on file	The last digits	<code>payments-service</code> returns a display string; or the hint is stored when the account is written, as <code>last_four</code> is for tokens
Check that the person writing is the contact	A yes or no	A narrow check in the service that already holds the grant, which compares what the customer said with the stored value and answers "match" or "no match"
Look at the customer's invoices	Statuses, amounts, due dates, for this customer only	<code>app_admin_console</code> reads Internal columns of <code>invoices</code> , and the row rule on the case's customer, from Chapter 3, narrows the rows
Fix a wrong account number	Nothing, and not by hand	The application path: <code>admin-console</code> asks <code>payments-service</code> , a second person approves, and the change is recorded in <code>audit_events</code>

The diagram shows the first row. Look at the last arrow: an action that does not exist is a stronger control than one that exists and is refused.



I would not build a "reveal" button at the start. A reveal that needs a reason and logs the click sounds reasonable, and within a month clicking it is a habit and its log is something nobody reads. If a real need appears, make it a request with an approver, as Chapter 8 does for any unmasked Confidential read, and count how often it is used. A feature that is used daily is telling you that the display form is missing something.

Whoever is on call follows the same rule under a different pressure. Robin debugging a failed reminder needs to know that a recipient address was rejected, and rarely needs the address. When Robin does need the unmasked `contact_email`, that is a JIT `SELECT` that names the column, with an approver who reads the column list. On-call reads are never auto-approved: a rule that skips Confidential columns approves a request that grants nothing, and a rule that includes them removes the one check that carries meaning. The cost is a wait at a bad hour, and the way to pay it down is to remove the need for the value in the common cases, which the next section covers.

Sensitive data in logs and error messages

Logs are the data copy nobody designs. They are written by many components, read by many people, kept for a long time and searched with tools that were never part of the access model. Four places carry most of the leakage.

- **Application logs and error messages.** A message built from a value, such as "could not send to the address", puts the value in the log, in the error tracker and sometimes on a customer's screen. Drivers and ORMs can also log bound parameters when debug logging is on.
- **Database logs.** Statement logging can record literals, and a slow-query log keeps the text of a query along with the values in it.
- **Statement records.** The access gateway records each statement a person sends (Chapter 7), and a person who types a value into a condition has put that value in the record.
- **Tickets and chat.** A result pasted to explain a problem is a copy with the wrong audience and no end date.

Warning: A statement log holds literals, so the audit log of Chapter 14 and the gateway's statement records are themselves stores of sensitive data. Record statements as fingerprints with the literals redacted, classify the store at least Confidential, restrict who reads it and give it a retention period, or the control that watches the data becomes one more copy of it. Chapter 14 owns the mechanism.

My preferred rule for application logging is an allowlist, not a scrubber. A scrubber looks for patterns that resemble sensitive values and removes them; it catches the shapes its author thought of and misses the rest. An allowlist names the fields that may appear and drops everything else, so a new field is silent until someone chooses to log it.

```
function buildLogEvent(event, allowedFields):
  safe = {}
  dropped = 0
  for each field in event.fields:
    if field.name in allowedFields:
      safe[field.name] = field.value
    else:
      dropped = dropped + 1
  // the count is the signal: a field dropped every day is either a
  // missing allowlist entry or a value somebody tried to log
  safe["dropped_field_count"] = dropped
  return safe
```

Error messages follow the same discipline. An error carries a code and identifiers, such as an invoice id, a customer id and a delivery status, and not the value that failed. This is also how Robin's reminder case stops needing the raw address: if the failure record holds a status such as "recipient rejected" and the invoice id, the log answers the question without the address, and the gateway's record of who read the address, if anyone did, stays in one place.

Redaction in the log pipeline is a backstop, and it will miss things. The check I would not skip is a canary in tests: run the application with a placeholder of a recognizable shape in every sensitive field, and fail the build if it appears in any log output. It tests the allowlist where it matters, and it costs a few lines. Debugging gets slower when the log holds identifiers instead of values, and the `dropped_field_count` is where the resulting "why can't I see this" messages go, instead of a developer re-enabling parameter logging in production.

Ledgerly's column decisions

The table below pulls the chapter together. It is the decision record for the sensitive columns of the eight tables, in the form an access review can read. In that column, "Withheld" and "No access" both mean the role holds no grant on the column, so a query that names it is refused. The `human_readonly` column states the default for a session; the JIT column states whether a request may name the column.

Column	Class	human_readonly	JIT SELECT	Analytics copy
bank_accounts.iban, account_holder_name	Restricted	No access	Never	Dropped
payment_tokens. token_value	Restricted	Withheld	Never	Dropped
payment_tokens. provider, last_four, expires_on	Confidential	Readable	Not needed	provider and expires_on kept, last_four dropped

Column	Class	human_readonly	JIT SELECT	Analytics copy
customers. contact_name, contact_email, contact_phone	Confidential	Masked	Names the col- umn, un- masked	Dropped
customers. billing_address	Confidential	Redacted	Names the col- umn, un- masked	Dropped
payments. provider_reference	Confidential	Last four characters	Names the col- umn, un- masked	Dropped
invoice_lines. description	Confidential	Withheld	Names the col- umn, un- masked	Dropped
amount_minor, currency, status, due_on, account_code	Internal	Readable where the matrix grants	Not needed	Copied

Two entries deserve a comment. The Restricted columns are dropped from the analytics copy and never offered to a human role other than `human_breakglass`; the only route that changes them is the application path, with a second person approving (Chapter 9 covers how the execution policy routes gated operations there). The same path takes the owner link: `customer_id` on `bank_accounts` and `payment_tokens` stays inside the JIT ceiling because it is not Restricted, yet moving a row to another customer changes whose account it is, so an owner-link change is gated and routed to the product path, never made as an ordinary data fix. And where the engine has no column-level grants, the same table still holds, with one change: the Restricted columns live in the secrets tables, the JIT column reads "table level", and the approval screen says so in words.

Trade-offs and costs

Classification is upkeep. A catalog that is correct on the day it is written starts to age at once. Someone has to own each class decision, review the provisional ones and answer the question that an unclassified column raises. If nobody reviews the catalog, it becomes the kind of record that is complete and wrong, which is worse than an honest gap.

Masking costs debugging time. Robin cannot see the raw email, so a bug that depends on its exact characters needs a JIT request and an approver. I accept that cost for the access this chapter closes. It is not free, and a team that pays it every night will start looking for a way around, so measure how often the unmasked read is requested and fix the cause.

Dynamic masking depends on the engine. Behavior around filters, direct grants and views differs, and a mask that you have not tested on your engine is a hope. The view-based variant adds objects that migrations must keep in step with the base tables.

Failing closed produces gaps. The analytics load that refuses unclassified columns leaves the data team without a column until someone classifies it. Name who owns that wait, and keep the classification review fast enough that the gap is hours, not weeks.

Application-level encryption removes features and adds a dependency. The key service sits on the path of every read of the column, and its outage is a payments outage. Because the audit entry comes first, an unreachable audit log stops those reads too, which is a deliberate coupling and one more availability dependency. The loss of query features pushes search toward other columns, and the key backup is one more thing that must be tested.

Some controls here can become security theater. An unmask button that is always clicked, a catalog nobody reviews, a log scrubber that nobody tests and a masked column that any reader can probe with filters add friction or paperwork and reduce nothing.

When not to use all of this. A small system with a handful of tables and no customer personal data does not need a catalog file, a masked load or a key service. A one-page list of sensitive columns, column-level grants that leave them out and a log allowlist give most of the value. Add masking

when a second audience needs the data, add encryption when a copy you cannot grant around becomes a concern, and write down which event would make you move to the next step.

Common mistakes

- **Treating encryption at rest as access control.** It is tempting because it is one setting and it satisfies a checklist. It protects a lost disk and does nothing against a query. Keep the grants, and use application-level encryption where a copy needs protecting.
- **Masking in the client and not in the database.** A mask in the web interface is easy to build, and a direct connection, an export or an API response bypasses it. Mask where the data leaves the store.
- **Classifying tables and not columns.** One label per table over-protects the status flag and under-protects the phone number. Classify each column and derive the table's label from them.
- **Defaulting new columns to Internal.** Internal is the class of least friction, so it becomes the class of anything unreviewed. Make Confidential the default and clear it by review.
- **Masking free text with patterns.** A pattern finds the numbers and emails its author expected and misses a name typed into a note. Withhold free-text columns from broad readers.
- **Using an unkeyed hash as a pseudonym.** It looks irreversible. For a value with few possibilities it can be reversed by trying them. Use a keyed pseudonym whose key stays outside the database, or drop the column.
- **Copying production to staging "and masking it afterward".** The window between copy and mask is a copy of everything. Mask on the way, as the analytics load does, and treat staging as one of the doors in Chapter 13.
- **Adding a Restricted column to a JIT request to save a step.** The ceiling refuses it for a reason, and the temptation is strongest at night. Restricted values change only through the application path with four-eyes approval, and on engines without column-level grants they sit in the secrets tables.

- **Auto-approving on-call reads.** It removes the pause where a person checks which Confidential column is being unmasked. Keep the approver, and reduce the number of times the value is needed.
- **Logging request bodies at error level.** The body is the most useful thing for a developer and holds every field a customer typed. Log identifiers and statuses, and let an allowlist decide what else appears.

Key takeaways

- Classify every column, treat free text for the worst case, and make unclassified columns default to Confidential, so that new data starts closed and the work of opening it is visible.
- Keep the classification catalog in the repository that holds the migrations, and block a merge that adds a column without a class, because the check costs one line and its absence is the failure.
- Minimization comes first: data that was never stored needs no mask, and personal data kept out of append-only tables cannot become an erasure problem.
- Dynamic masking suits `human_readonly` and protects against casual exposure; static masking suits the analytics copy, where the raw value can be dropped entirely and the load fails closed on anything unclassified.
- Give people the form of a value their task needs, such as a display string or a yes or no, and keep Restricted columns out of every JIT request and every raw-SQL write.
- A mask in the result stream is a second layer behind the grant, never a substitute for it, and a load role that cannot read Restricted columns makes a catalog error fail closed.
- Logs, error messages and statement records are data copies: log identifiers through an allowlist, keep values out of messages, and classify the audit log itself.

The controls in this chapter protect data while it sits in the database and while a query returns it. The next chapter follows the data out, into backups, dumps, exports and staging copies, where none of these roles apply.

Chapter 13 — Backups, Exports and Other Side Doors

Picture a Friday afternoon at Ledgerly. Robin is chasing a bug in invoice reminders that appears only with realistic data: invoices with unusual due dates, customers with several contacts, line descriptions that hold free text nobody planned for. Staging has a dozen hand-made customers, and the bug does not show on any of them. Someone remembers that an older dump of production sits in shared storage, and Alex, who runs the databases, restores it into staging so the whole team can look.

The bug is found within the hour. The dump stays in the folder, and the restored database stays in staging. Staging has no access gateway, because it is not production. Its login is shared, every engineer can reach it, CI jobs run against it, and a contractor who joined last month has an account. `invoice-worker` in staging uses the same outbound email settings as production, and its reminder job is switched on.

Nobody decided to expose customer data. Each step was a sensible answer to an afternoon's problem, and the controls built in the earlier chapters protect a database that no longer holds the only copy of its contents.

This chapter adds the last step of the target state before Part V: backups, exports and staging copies are treated as doors with owners and expiry. The question is this: once data leaves the primary on purpose, for recovery, for a vendor, for a test or for a person who needs to look, how do you keep the controls with the data instead of with the database it came from?

In the five-part model (**Identity, Grant, Path, Execution, Record**), a copy is a place where the Grant and the Path no longer apply, because the data left through a route that carries neither. The chapter's work is to make the copy itself carry a Grant: a reader list, an owner and an end date, with a Record of who made it and who removed it.

The problem, and why it is hard

Chapter 6 defined a **side door** as a data copy or export path that bypasses the access controls of the primary. Replicas and the analytics copy are doors you build deliberately and can name. This chapter is about the rest: backups and snapshots, dumps, CSV files, staging databases, a few thousand rows on a laptop, a dashboard cache, a file in a vendor's inbox. Most of them were made for a good reason. Few have an owner.

The core difficulty is that access controls live in a system, not in data. The primary has an access gateway, a login per session, column-level grants, timeouts and an audit log. A dump of the primary has none of those, because a dump is a file. Whoever can read the file has everything in it, including columns the role schema withholds from every human role except `human_breakglass`. A backup that holds the Restricted columns is, for the people who can read it, a standing grant to those columns that no gateway issued and no review lists.

The obvious response is a policy: do not copy production data. It fails for a plain reason. Copying is how the work gets done. Recovery needs a copy by definition. A vendor needs files. A bug that depends on real data shapes needs something that looks real. A policy that forbids all of it is routed around, and the copies that result are the unmanaged ones. The better question is which copies are needed, who owns each, who may read it, and when it ends.

A second difficulty is that these doors are made by different teams for different reasons, so no single owner sees them all. The platform team owns backups, the data team owns the analytics copy, the backend team owns staging, finance owns the vendor relationship, and each engineer owns the contents of their own laptop. An inventory is the first control, and it is found in storage buckets, job schedules and configuration, not in a policy document.

A map of the side doors

Before the controls, a map. The table lists the doors this chapter covers, what each one holds in Ledgerly's case and who reaches it by default if nobody has designed otherwise.

Door	What it holds	Who reaches it by default	First control
Backups and volume snapshots	Everything, including Restricted columns and the secrets tables	Anyone with storage or infrastructure access	Separate powers to take, restore and delete; encryption with its own key
Dumps in shared storage	Everything, as a readable file	Anyone who can open the folder	Find, assign an owner, delete or move under the backup rules
Human exports (CSV, pasted tables)	Whatever the query returned	The person, then whoever they send it to	Tiered approval, expiry, no Restricted data
Staging and test databases	A copy of production, or a lookalike	Every engineer, CI, contractors	Masked or synthetic data, outbound messages off
Laptops	Result sets, local databases, saved files	The laptop's user and its backups	Remove the need; masked samples instead
BI tools	Cached extracts, scheduled reports, shared dashboards	Everyone the tool is shared with	Read the analytics copy only, per-person identity where possible
Vendor files	Invoice and ledger exports	The vendor's staff and systems	Narrow role, column list, retention, file expiry

Every row has the same shape of problem: a copy of data with the controls of a file system or a SaaS product, and a reader list that nobody wrote down. The register from Chapter 6 already exists for this reason, and I would extend it to every row above. A door that is not in the register is not a door someone will review.

```

table data_copy:
    name      text primary key    // extended from Chapter 6
    kind      text not null       // "staging_refresh_current"
    staging, laptop,
                                // backup, dump, export,
                                // bi_extract, vendor_file,
    replica, analytics
    source    text not null       // what it was made from
    contains  text not null       // highest data class present
    readers   text not null       // roles, logins or groups
    storage   text not null       // where the bytes live
    owner     text not null       // a team, never a person
    ends      timestamp null      // null only for replicas and
    analytics
    review_due timestamp not null // when the row is next reviewed
    deleted_by text null          // who removed it, once removed

```

The new fields are `storage` and `deleted_by`; `kind` gains the values `backup`, `laptop`, `bi_extract` and `vendor_file`, and `contains` records the highest data class present. The first new field says where to look when someone asks what can read the copy, because the answer is usually a storage permission and not a database role. The second closes the loop: an `ends` date nobody acts on is a wish, and the record of who deleted the copy is the evidence that the date meant something.

Backups and snapshots

A backup is the one side door nobody argues against, and that agreement is part of why it is so often left uncontrolled. It has to hold everything, because its job is to restore everything. That is the whole design, and it is also the whole exposure.

What a backup holds

Backups come in a few kinds. A **physical backup** or volume snapshot copies the database's files, often with a continuous archive of changes so the database can be restored to a point in time. A **logical dump** reads the data through the database and writes it as statements or rows. Both hold every table. The physical kind also holds the cluster's role definitions and whatever credential material the engine stores for them, and some logical tools do too, depending on how they are used.

Two consequences follow, and neither is visible in a diagram of the primary. The first is that the Restricted columns are in the backup, and so are the secrets tables on engines that use them (`bank_account_secrets` and `payment_token_values`, from Chapter 3). The role schema's rule that only the application roles and `human_breakglass` reach those values describes the database. It says nothing about the file.

The second is that the backup holds the state of the day it was taken. A backup from before the shared admin login was retired still contains that login. It holds the role memberships of people who have since changed teams. Rotation and revocation happen in the live system, and they do not reach back into files that already exist. When someone narrows a role on the primary, nothing happens to the restore of last year: the primary improves, and the copies keep the state of the day they were made.

Restore is read

I would put one sentence at the top of the backup design: the power to restore is the power to read everything. Whoever can restore a backup into a place they control can then log in there as an owner and read every column, and no gateway, column grant or masking rule in production applies, because production is not where the reading happens.

This changes who the control belongs to. Backup access is usually granted by infrastructure roles: someone can read the bucket, mount the snapshot or run the restore tool. Those roles are rarely reviewed with the same care as `human_readonly`, though they sit above it. The question for an access review is therefore not only "who can query the primary" but "who can restore, and into what".

I would separate three powers, and give them to different principals:

Power	What it means	Who holds it
Take	Create backups and write them to the store	The backup job's own identity, write-only to the store
Restore	Read a backup and bring it up somewhere	The platform team, per request, with a second person approving (four-eyes)
Delete	Remove or expire backups	The retention process; humans only through a reviewed request

The point of the split is that the job that writes backups cannot read them back, and nobody can both restore a backup and delete the recovery points, so a restore cannot be followed by removing the thing it was restored from. Where the storage supports write-once retention for a period, I would use it for the same reason, at a cost: you cannot delete a backup you made by mistake, such as one taken from a database that briefly held something it should not have, until the period ends. Decide in advance how that case is handled.

Encryption, and whose key

Backups should be encrypted at rest, and the interesting question is who holds the key. If the same administrator can read the bucket and use the key, encryption protects against a lost disk and nothing else. If the key is

held by a different principal than the storage administrators, and restoring requires both, then the storage team alone cannot read the data and the key holder alone has nothing to read.

That separation costs something specific. Restoring now needs two parties, and a backup whose key is lost is no longer a backup. A key-recovery path has to exist and has to be tested, which is one more procedure for the platform team to keep. Chapter 12 covers application-level encryption and the limits of key management; the narrower point here is that backup encryption is a gate on restore, and a gate is only worth having if it is a different gate from the one on the storage.

Encryption does not help against the person who is allowed to restore. It protects the file in transit and at rest from people and systems that are not part of the restore path. I would say that plainly in the design document, because "the backups are encrypted" is often heard as "the backups are safe", and those are different claims.

A restore request

Restore is the moment the backup becomes a live database, so the control belongs there. The block below shows a restore request with the order of operations written out. The backup store, the register and the audit log are separate systems, so no transaction covers them.


```

function restoreBackup(request):
    if request.approver is null or request.approver ==
request.requester:
        raise SelfApprovalNotAllowed
    if request.target not in isolatedTargets:
        raise TargetNotIsolated
    if request.ends is null or request.ends > now() + maxRestoreLife:
        raise RestoreNeedsEnd

    // the register comes first: a copy that exists with no record is
a side door
    registerCopy({ name: request.target, kind: "backup", source:
request.backup_id,
                    contains: "Restricted", readers: "none until access
applied",
                    owner: request.owner_team, ends: request.ends })

    try:
        auditLogAppend({ actor: request.approver, action: "restore",
                        subject: request.backup_id, target:
request.target })
        on AuditLogUnavailable:
            raise RestoreRefused

    // set before the copy exists, so a failure part-way still ends;
if this
    // call fails, the sweeper on `ends` removes the copy all the same
    scheduleDeletion(request.target, at: request.ends)
    // reachable only by the restore tooling until the last step
    startRestore(request.backup_id, into: request.target, reachable:
false)
    // every login returns with its backup-day credential, the
superuser and
    // ledgerly_access_admin included: nothing may connect before this
is done
    disableAllLogins(on: request.target)
    rotateSuperuserCredential(on: request.target)
    // the backup already holds a login with this name: drop it and
create it
    // again with a new credential, so the backup-day one never works
    replaceLogin("ledgerly_access_admin", on: request.target,
credential: "new")
    applyAccessDefinition(on: request.target)
    openTarget(request.target, readers: request.readers)

```

If the process stops after `registerCopy` and before the restore starts, the register holds an entry for a copy that does not exist, which a weekly check finds and removes. If it stops after the restore starts, the copy exists and is unreachable, and its scheduled deletion removes it; if even the scheduling failed, the sweeper that acts on `ends` in the register finds it. The reverse order would leave a copy that exists and is not listed, and that

is the failure that matters. The cost of the design is the delay: an approval and an isolated target before anyone sees a row. During an incident that delay is real, which is why the restore path should be rehearsed with approval arranged in advance, as Chapter 15 discusses for incidents. When the aim is to recover a few rows and not a whole database, the restore goes to the same isolated target, the rows are taken from it, and they go back into production as a data fix under the rules of Chapter 10.

A restored copy starts closed

A restored database brings back the roles, memberships and credential material of the day of the backup. I would treat it as hostile until the access definition has been applied again. In practice that means a restore target with no network route from services and no human logins, and a first step that disables every login the backup brought back, the superuser and `ledgerly_access_admin` included, because each returns with the credential it had on the day of the backup. Then the superuser credential is rotated, and only then is the access definition from Chapter 3 applied: roles, memberships and grants as code describes them now. The copy becomes reachable after that, and not before.

That step needs an access administrator, and it has to be one that belongs to the target. The backup already holds a login named `ledgerly_access_admin`, which came back with the credential of the day the backup was taken. The restore replaces it: the returned login is dropped and created again under the same name with a new credential, made for the restore and removed with it. Production's current credential for that role is never copied into the target, and the one in the backup is gone before anything can connect, because whoever holds either can grant anything inside the roles it administers, and a restore drill is exactly the place where credentials get pasted into a test environment. If the restored copy's own administrator is compromised, the blast radius is that copy.

Only after that does the restored copy get readers, and the readers are the people on the restore request. A restore for a drill needs no human readers at all: the drill checks that the data comes up and the counts match, and then the copy is deleted.

Retention and deletion

Retention is a trade between two costs. Keep backups longer and you can recover from problems that were noticed late, at the price of a larger pile of data that holds everything, including customer data the primary deleted long ago. Keep them shorter and you hold less, and some corruption will be older than your oldest backup. I do not know the right period for your company. It depends on how long a silent problem usually takes to be noticed, how much a lost month of data would cost, and what your legal and compliance people say about deleted data that lives on in backups. Privacy regimes such as GDPR or KVKK tend to ask what happens to a person's data once it is deleted from the live system, and the interpretation is theirs, not this book's. What engineering can supply is evidence, not compliance: a register and a deletion record that show what the system does.

One design that is worth discussing with them: backups age out on schedule, and a restore procedure re-applies deletions recorded since the backup was taken, so a restored copy does not bring back data that was removed on purpose. That adds a step to every restore and requires that deletions are recorded somewhere a restore can consult, which is another reason to keep deletion requests in a durable record. The step runs after the access definition is applied and before the copy opens.

```
function reapplyDeletions(target, backupTakenAt):
    // by execution time: a deletion recorded before the backup but
    run after
    // it is not in the backup's state, so it must be applied again
    for each request in deletionRecord.executedSince(backupTakenAt):
        auditLogAppend({ actor: "restore-tooling", action:
"reapply_deletion",
                                subject: request.id, target: target }) //
    record first
        deleteRows(on: target, subject: request.subject_id)
    // a failure here leaves the copy closed: openTarget has not run
    yet
```

Drills create copies

A backup that has never been restored has not been shown to work. Restore drills are how you learn that the backups work, and every drill creates a database containing Restricted data. I would run drills through the same request path as a real restore, with the same isolated target, the

same register entry and an end date within the day. The drill is also the rehearsal for the incident: if it needs a shortcut to finish, the shortcut will be taken at three in the morning.

Warning: A backup or dump that holds Restricted columns and sits in storage readable by a wide group is the same exposure as a standing grant to those columns, with no gateway and no log. If you find one, restrict the storage permission first and talk about retention afterward.

Dumps, exports and CSV files

Backups are the door with a purpose and a team. Dumps and exports are the doors usually opened by one person under time pressure, and they come in two kinds that should be handled differently: the exports that happen on a schedule and the ones a person makes.

Dumps made by hand

Ledgerly's shared folder holds a few older dumps. Each was made because someone needed a copy for an hour, and the copy stayed. A person's dump is a backup with no retention, no encryption policy and no owner. It is also, on engines where a dump reads through the database, evidence that someone held a login that could read every table.

I would do three things. First, take the dump out of the set of normal tools: the access gateway does not offer a whole-database dump, the human roles cannot read the tables a dump would need, and `human_breakglass` is the only human role that could, with the break-glass logging set of Chapter 8. Second, find the existing dumps, and before deleting any of them, check whether one is someone's only backup. That happens: a dump becomes the backup of a table nobody else copied. Build the real backups first, then delete the dumps. Third, give the person who needed the dump an alternative that does the job, which is usually the masked analytics copy or a test data set (see the section on test data below).

Human exports

Exports by people are the most common and the hardest to ban. Support needs a list of one customer's invoices. Finance needs a file of last month's payments. The data team needs a sample. I would tier them by the highest data class in the file, because a single rule that fits all of them is either too loose for the dangerous ones or too heavy for the harmless ones.

Tier	Contents	Control
Internal only	Aggregates, counts, status breakdowns, no personal content	Self-service, logged, no approval
Contains Confidential	Rows with customer contact or free text	Ticket with reason, one approver, an end date for the file; the rows come from a <code>JIT SELECT</code> session that names the columns (Chapter 8), so the approver reads the column list the file will hold
Contains Restricted	Bank account or token values	Not exported; the answer is a display form or an application path (Chapter 12)

The tier is computed at the gateway, not declared by the requester, and it fails closed: it is the highest catalog class (Chapter 12) among the source columns the query reads, not among the names of the result columns, because an alias or an expression such as `lower(contact_email) AS x` hides the source. A column whose source cannot be resolved, through a view, a CTE or a function, counts as the highest exportable tier, and where even that cannot be decided the export is refused. Chapter 9 covers the limits of static analysis. These tiers are an illustration of a shape. What matters is that approval is attached to risk. An approval that is granted within seconds, every time, for everything, is the theater Chapter 8 describes. The tier that needs a human approver should be small enough that the approver reads the request, and I would measure that by whether anyone ever says no.

The end date is the part teams skip. An export without an end date is a copy that outlives its question. The ticket that approves the file carries a date, the file goes to a place with an owner, and a sweeper removes it. Chapter 10's rule for ad-hoc work covers the lightest case: paste identifiers

and counts, not rows with personal content. Chapter 7 covers what the gateway can do about a bulk result set, which is limit it and make it visible, not prevent it.

The vendor export

The external accounting vendor never connects to a Ledgerly database. It receives files, which makes the export step a door you can design completely, because you control both ends of the path up to the vendor's inbox. In the starting state the export step of `reporting-job` writes those files with the shared admin login, so the file can contain anything the admin role can read. In the target state it uses `app_vendor_export`, and that changes what the file can hold, because a file cannot contain more than the login that wrote it could read.

```
// illustrative column lists: the vendor's real needs decide them
GRANT SELECT (id, customer_id, total_minor, currency, status, due_on,
created_at)
    ON invoices TO app_vendor_export;
GRANT SELECT (id, amount_minor, currency, account_code, created_at)
    ON ledger_entries TO app_vendor_export;
```

The grant is the control, not the query. A mistake in the export query can leave out rows or repeat them, and it cannot add `contact_email`, because the role cannot read it. `customer_id` is an opaque identifier and the vendor has no table to join it to. If the vendor's work needs customer names, that is a decision someone makes and approves, and does not arrive because a query was widened.

The export step connects to `ledgerly_replica_2`, as the connection policy in Chapter 6 says, and writes a file. The file then moves to the vendor by the channel agreed with them. The steps involve three separate systems, so the block below shows the order and what each failure leaves.

```

function runVendorExport(period):
    file = writeExportFile(period, as: "app_vendor_export") // local,
not yet shared
    manifest = { file: file.name, rows: file.rowCount, checksum:
file.checksum,
                columns: file.columnNames, period: period }

    // the record exists before the file leaves: a sent file with no
    // record is the failure to avoid
    registerCopy({ name: file.name, kind: "vendor_file", source:
"ledgerly_replica_2",
                contains: "Internal", readers: "accounting vendor",
                owner: "platform", ends: now() + fileLife })

    try:
        sendToVendor(file, manifest)
    on SendFailed:
        // nothing has left; the file stays local and is removed with
its end date
        notify("platform", "vendor export not delivered: " +
file.name)
        raise ExportNotDelivered

    // the file has already left: a log outage delays the entry, never
the export
    auditOrQueue(manifest)

```

Here `auditOrQueue` appends the record to the audit log and, if the log is unreachable, queues it for retry and notifies the platform team, as in Chapter 7. A failure of the log becomes a visible, retried gap and not a silent one, and no transaction spans the log and the vendor channel.

The manifest is the part people leave out. It lets both sides check that the file is the one that was meant, and it gives an access review something to read: this many rows, these columns, this period. If a column appears in a manifest that was not in last month's, that is a change somebody should have approved.

Two limits remain. Once the file has reached the vendor, its retention is a contract and a trust question: you can ask the vendor to confirm deletion, and you cannot check it. And the checksum shows that the file was not altered in transit, not that the vendor's staff are the only people who read it. For those two I would put a retention term in the agreement and a line in the access review that asks when the vendor last confirmed deletion.

Test data: staging, masked copies and synthetic data

Staging is where the Ledgerly scenario began, and it is the door most often opened out of kindness. Engineers want realistic data, and production is the only source of data that has the shapes real customers create. The cost is that staging has the access model of a test system, with the data of a production one.

Three kinds of non-production data

Kind	Realism	Exposure	Cost
Production copy	Highest	Same as production, with weaker controls	Cheap to make, expensive to own
Masked copy	High, with some shapes lost	Lower, not zero	A masking pipeline to build and maintain (Chapter 12)
Synthetic data	Lower, limited to what was modeled	Lowest	A generator to maintain, and bugs that need unmodeled shapes

I would not copy production data into staging unmasked, and the reason is not abstract. Staging usually has more readers than production, a shared login, CI that stores logs, and integrations configured to behave like production. It combines the largest audience with the weakest identity controls, and it holds the most sensitive data. If the data must be real, the controls have to be as strong as production's, and at that point staging is production under another name.

What a masked copy still is

Masking, covered in Chapter 12, replaces or obscures values while keeping the shape usable. A masked copy is far better than a raw one, and it is still a data copy, for reasons worth stating so that nobody treats "masked" as a synonym for "safe to hand out".

First, masking covers the columns you listed. Free text such as `invoice_lines.description` can hold personal data the classification never anticipated, which is why the book classifies it for the worst case, and a mask that skips it leaves it in the copy. Second, shape-preserving masks keep the rows that remain distinguishable: an unusual invoice amount, an unusual date and a small customer can together point at one

business. The values were masked, and the combination still carries information. Third, consistent masking, where the same input gets the same output so joins keep working, also lets the same person be recognized across copies and tables.

None of this means masking is not worth doing. It means a masked copy is handled as Confidential until someone has reviewed what is left, its audience is smaller than "everyone in engineering", and it has an end date like any other copy; the continuous analytics copy is the exception, and it is reviewed on its `review_due` date instead. The word for this is **re-identification**: the ability to work out who a record is about from fields that were left in. I would describe the risk to the team in those terms and not in the language of attacks. The design response is a smaller audience and an expiry, not a longer list of masked fields.

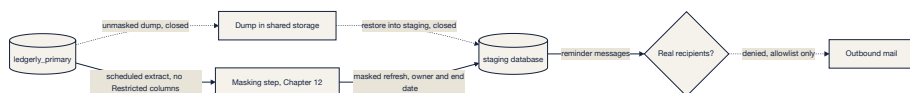
Synthetic data

Synthetic data holds no real person, so it carries the least exposure. Its limit is that it holds only the shapes someone thought to model, and the bugs that need real data are the ones nobody modeled. A generator that makes tidy customers does not make the customer with six contacts and a description in two languages.

I would grow the synthetic set from the bugs. Each time a problem needs production-shaped data to reproduce, the shape that caused it, and not the values, becomes a fixture: a customer with several contacts, an invoice line with an empty description, a due date at the end of a month. Over a year the set stops being tidy and starts being the useful kind of odd. It costs an engineer an hour after each such bug, and it removes the need for the next production copy.

Ledgerly's staging, redone

Return to Robin's Friday. The path that opened the door was a dump from shared storage into staging. The path I would build replaces it with a refresh that goes through the masking step and lands in a database with a short life and a named owner. In the diagram, look at the dashed paths: they are the ones the design closes.



The right half of the design is easy to forget. A staging environment holds data and also acts on it. `invoice-worker` in staging reads customers and sends reminders, and if its outbound settings match production, it sends them to the contact addresses in the data. A masked copy has no real addresses, which fixes that by accident. A raw copy turns a test run into messages to real customers. I would make the outbound path in every non-production environment deny by default and allow only an allowlist of internal test addresses, regardless of what the data holds.

Payment tokens follow a related rule. A token stands in for a payment instrument, and whether it works outside the system that issued it depends on the provider, which I cannot judge for yours. The default is that `payment_tokens.token_value`, `bank_accounts.iban` and `bank_accounts.account_holder_name` are never present in a non-production copy, masked or not (backups are the exception, because they must hold everything), and staging uses the provider's test tokens.

Note: Treat the staging refresh as a pipeline with a failure path, not a one-off. If the masking step fails, the refresh stops and staging keeps its previous contents. A refresh that falls back to an unmasked source for one run recreates the dump in the shared folder with extra steps.

BI tools, laptops and vendors

Three more doors deserve short treatment, because each takes data out of the database by a route that looks like a feature.

BI tools. A dashboard tool connects with one login, and everyone with access to the dashboard sees what that login can read. That reverses the identity model of Chapter 3: the database sees one principal, and, say, a hundred people see its data. Two controls help. Point the tool at `ledgerly_analytics` and not at a replica, so it reads masked data with a role of its own for the tool, an `app_` role like any other workload (`human_analytics_reader` is the data team's role for people), instead of

production roles. And, where the tool supports it, have it pass each person's identity to the database, so row and column rules apply per person. The tool also keeps extracts, caches and scheduled email reports of its own, which are data copies that go in the register. A dashboard is a convenient door, and a cached extract is the same door with the controls gone.

Laptops. The gateway removes saved credentials and sets read-only defaults, as Chapter 7 describes, and it cannot control what a person does with rows that reached the screen. The practical control is to reduce the reasons to keep a local copy: a masked sample, a staging database that behaves like production, and fixtures for the shapes that cause bugs. A team cannot prove that no laptop holds production data, and I would not claim to. A baseline such as full-disk encryption and a way to wipe a lost device reduces what a lost laptop costs, and it is a baseline, not an answer.

Other vendors. Accounting is the one the book follows, and others receive data by side effect: an error tracker that captures a request body, a support desk that receives pasted screenshots, a log platform that stores statements. Each vendor that holds a copy gets a register row with what it contains and when the contract ends. Chapter 12 covers keeping sensitive values out of logs and errors, which is the cheapest way to keep them out of a vendor's store.

Closing the doors at Ledgerly

Consider how Alex would move Ledgerly from scattered copies to a register with owners. As with the read path in Chapter 6, the order matters, and I would take it in six steps.

1. **Inventory the copies.** List backups, snapshots, dumps, exports, staging, BI extracts and vendor files, with storage, readers and an owner. Expect to find copies nobody remembers; the first run of the register is mostly discovery.
2. **Restrict the storage first.** For anything that holds Restricted columns, narrow who can read the storage before deciding what to delete. That is quick and reversible, and it closes the widest exposure while the rest is sorted out.
3. **Build the backup powers.** Give the backup job a write-only identity, put restore behind a request with a second approver, hold the encryp-

tion key apart from the storage administrators, and run one drill through the request path.

4. **Retire the dumps.** Once the real backups exist and one restore has worked, delete the old dumps and record who deleted each. The nightly unmasked restore that feeds `ledgerly_analytics` ends when the masked load of Chapter 12, under `app_analytics_load`, runs in its place; until then it is the largest open door and gets an end date.
5. **Move staging to masked or synthetic data** with outbound messages off by default, and remove the shared staging login.
6. **Put the vendor export on its own login.** Move the export step to `app_vendor_export`, add the manifest and set a file life.

Then connect the register to something that happens anyway. The access review of Chapter 14 asks which copies exist and who can read each, creating a copy needs a register row, and a sweeper removes anything past its ends after a grace period and a notice to the owner. Restore requests and exports should also write to the audit log, which is Chapter 14's subject. Restore and export jobs follow the same execution discipline as the runner that carries approved writes: they run under their own identity, from an approved request, with the record written first, except that an Internal-tier export keeps the self-service rule of the tier table above (Chapter 9 describes that discipline).

Trade-offs and costs

The first cost is speed during an incident. A restore that needs an approver and an isolated target is slower than one done by whoever has the bucket permission, and the time it adds lands on whoever is on call at the worst moment. Rehearse the path and arrange approval in advance (Chapter 15 covers access during incidents), so the control is a procedure people know and not an obstacle they discover.

The second cost is key custody. Splitting the backup key from the storage means a lost key is a lost backup, and the recovery path for the key is one more procedure that has to be written and tested. For a very small team the split may cost more than it returns; one encrypted store, tight storage permissions and a documented key recovery can be a reasonable starting point.

The third cost is a slower debugging loop. When engineers cannot restore production into staging, some bugs take longer to reproduce, because the shape that triggers them is not in the test data yet. The fixture habit pays this back over time and is a real expense at the start. Masked copies add a pipeline that someone owns, and it will break when a column is added, which is why the migration that adds a column is where its class gets decided (Chapter 11).

The fourth cost is retention. Longer retention keeps more history recoverable and keeps more sensitive data in more places; shorter retention reverses both. There is no value that is right for everyone, and a team that has not chosen one has chosen "forever".

Finally, the register and its sweeper are overhead. A sweeper that deletes a copy someone still needed is an incident of its own, so deletion should come after a notice and a grace period, and an owner should be able to extend an end date once with a reason. A register that is never opened protects nothing, and the tie to the access review is what keeps it honest.

When not to build all of this: a small company with one database, a handful of engineers and no Confidential data worth the name does not need tiered export approval or a split key. It still needs the inventory, restricted storage for backups and a rule that nothing from production lands in staging unmasked. Do those three, write down what would make you do more, and review that note when the team or the data changes.

Common mistakes

- **Narrowing access to the primary and ignoring its backups.** Roles, a gateway and an audit log are in place, and the backup bucket is readable by the whole infrastructure group. It is tempting because backups are "operations". The remedy is to treat restore as a read of everything, with its own approver and its own log.
- **Believing "encrypted" means "safe".** Encryption at rest protects the file from people outside the restore path, and the key sits next to the bucket in the same administrator's hands. Hold the key apart and say what it does not protect against.
- **Deleting old dumps before real backups exist.** The dump was someone's only copy of a table, and removing it removed the only way

back. Build and test the backups first, then retire the dumps.

- **Restoring into a network where the old logins work.** The restored copy brings the roles and credentials of its day, including ones retired since. Restore into isolation, re-apply the access definition, and replace the `ledgerly_access_admin` login the backup brought back with a new one instead of keeping either old credential.
- **Treating a masked copy as public.** The values are masked and the copy is shared widely, then free text or a rare combination of fields points at one customer. Handle it as Confidential, with a smaller audience and an end date, until it is reviewed.
- **Refreshing staging from production as a one-time exception.** The masking step failed, the release was waiting and the dump was right there. Make the refresh fail closed, so the old data stays and nothing falls back to a raw source.
- **Letting a BI tool connect with a production role.** One login behind, say, a hundred viewers, with a cache nobody listed. Connect it to the analytics copy with its own role, and register its extracts.
- **Approving every export in seconds.** The approval exists and reduces nothing. Reserve the approver for the tier that deserves a decision, and watch whether anyone ever declines.
- **Giving a copy an owner and no end.** The register is complete and nothing leaves it. Every row needs an `ends` date and a sweeper that acts on it, with a notice first.

Key takeaways

- A side door carries the data without the controls of the system it came from, because access controls live in the database and not in the file.
- The power to restore a backup is the power to read everything in it, so backup access deserves the same review as the most privileged human role.
- Taking, restoring and deleting backups should be held by different principals, and the encryption key should be held apart from the storage administrators, with a tested recovery path.
- A restored copy comes back with the roles and credentials of the day it was taken, so it starts isolated, has the access definition re-applied and

gets its own administrator login.

- A vendor export is controlled by the grant of `app_vendor_export`, not by the query, so the file cannot hold a column the role cannot read, and a manifest lets the file be checked.
- Test data should be masked or synthetic, a masked copy is still handled as Confidential until reviewed, and staging should deny outbound messages by default and fail closed when masking fails.
- Every copy needs an owner and an end date in a register that is reviewed with access, because a door nobody owns is one nobody closes.

Copies, exports and restores are now owned and have an end date, but a design is only as good as the evidence that it is followed. The next chapter turns to auditing: where the audit log lives, what goes into it, and how the access reviews this book keeps mentioning actually run.

PART V

Proving and Operating

Chapter 14 — Auditing and Accountability

Consider Ledgerly on a Tuesday morning. A customer has asked who at the company looked at their payout account last month, and the security contact forwards the question to Alex. It sounds like a lookup. Alex starts with the access gateway's session list, which shows that Robin held a break-glass session on the primary for forty minutes. Then Alex opens the database log, which names a session login but not the reason for the session. Then Alex searches `audit_events`, which records the business actions the application took and says nothing about what a person typed.

The three sources agree on most of the story and disagree on the rest. The gateway's clock is two minutes ahead of the database's. One session login appears in the database log with no gateway entry at all, and nobody can say whether that is a service login that the list forgot or a connection that arrived by a path nobody designed. Alex can say what probably happened, but cannot show it, or show that nobody with a database administrator's rights edited the evidence.

This chapter serves the accountability pillar of the thesis, and it adds the last step of the target state before incidents: access is logged to a store the accessed system cannot rewrite, and reviewed on a schedule. In the five-part model (**Identity, Grant, Path, Execution, Record**), this is the chapter about Record. Identity says who acted; Record is what remains, and whether it ties the statement to the person, the request and the result. The question here is this: what should one statement leave behind, where should it live, and how do you find out that the evidence is incomplete before someone asks?

The problem, and why it is hard

In the starting state the question has no answer. The database log names a shared role and no person, and it sits on hosts the administrators control. The earlier chapters fixed the identity half: a login per session, a request per grant, a runner for approved writes. Each of them appended to the record through the same call, `auditLogAppend`, without saying where the entries go. This chapter answers that.

The difficulty is that Ledgerly already has three records, and each one knows something different. None of them is the audit log as the book defines it: the record of who did what to which data, kept where the actor cannot change it.

Record	What it knows	What it cannot tell you	Who can change it
Gateway session record	The person, the request, the reason, the statement as submitted	What the database actually did; whether another path existed	The platform team, through the gateway's own store
Database log	The login, the time, the statement text if logged, errors	The person behind the login, the request, the reason	Whoever administers the database host
<code>audit_events</code>	The business action, the acting principal, the subject	The statements that produced it; anything the application did not write	Nobody by <code>UPDATE</code> , but <code>ledgerly_owner</code> can alter the table in a migration

The obvious response is to turn up the logging: log every statement in the database and keep it for a long time. That adds a fourth record and answers none of the three gaps. A log the accessed system writes about itself is only as trustworthy as its administrators, and a log nobody reads is theater. A statement log is itself a copy of sensitive data, because people type literals into conditions; Chapter 12 warned about that, and this chapter owns the mechanism.

The record exists to find the gap in the system, not to rank the people who passed through it. When an alert fires on Robin's session, the first question is what the design made necessary, not what Robin did wrong.

One statement, one record

A record is useful only if it can answer a list of questions written down in advance. I would write the list first and derive the fields from it, because an audit store designed from the log lines the gateway happens to produce answers the questions the log lines happen to allow.

The gateway writes two entries per statement, because Chapter 9 requires the intent to be written before the statement runs and the outcome after it. Together they form the **statement record**; the two entries share a `statement_id`, and the second one fills the fields that were unknown at the start. One shape serves for both, and for the records that other principals write.

```

table access_audit_record:
  stream      text not null      // one per producer instance
  sequence    integer not null   // gapless within a stream
  previous_hash text not null    // hash of the previous entry
  entry_hash  text not null
  kind        text not null      // "intent", "outcome", "event"
  action      text null          // events only: grant, revoke,
breakglass, data-fix, contain, and so on
  statement_id text null
  session_id   text null
  login        text not null      // what the database log shows
  principal    text not null      // the person or workload
identity
  on_behalf_of text null          // the human behind an
automation principal
  roles        text not null      // role set at the time of the
statement
  request_id   text null          // always access_request.id; a
data fix has one too, and its fix_id resolves through it
  reason       text null
  approver     text null
  statement_class text null       // the Chapter 9 class the
classifier assigned
  fingerprint  text null          // statement shape, literals
redacted
  literals_hash text null          // keyed hash of the literal
values
  tables       text null          // tables touched
  columns      text null          // columns touched, by data
class
  rows_affected integer null      // outcome only
  rows_returned integer null      // outcome only
  outcome      text null          // ok, refused, error, rolled-
back, row-cap-closed
  started_at   timestamp not null
  ended_at     timestamp null

```

The `event` kind carries everything that is not a statement: a session opening, a grant, a revoke, a rotation, a restore, a migration start. Those are the `auditLogAppend` and `auditOrQueue` calls of Chapters 3, 4, 7, 8, 10, 11, 12, 13 and 15, and the `action` column tells a grant from a revoke or a containment. They use the same chain and the same identity fields, so one query can follow a person from a request to a grant to the statements that used it to the revoke.

The table is only worth building if it answers the questions people ask. The second list is the one I would tape next to the schema.

Question	Fields that answer it
Who ran this statement?	<code>principal</code> , <code>on_behalf_of</code> , <code>login</code>
With which grants and approval?	<code>roles</code> , <code>request_id</code> , <code>approver</code> , <code>reason</code>
What did it ask for?	<code>statement_class</code> , <code>fingerprint</code> , <code>literals_hash</code> , <code>tables</code> , <code>columns</code>
What did it do?	<code>outcome</code> , <code>rows_affected</code> , <code>rows_returned</code>
Did it touch Restricted data?	<code>columns</code> classified by data class, with <code>rows_returned</code>
When, and for how long?	<code>started_at</code> , <code>ended_at</code>
Is anything missing around it?	<code>stream</code> , <code>sequence</code> , <code>previous_hash</code> , the intent without an outcome
Is this entry as it was written?	<code>entry_hash</code> , checked by <code>verifyChain</code> below

Two fields carry most of the weight. `principal` and `on_behalf_of` keep the trail from ending at a machine, as Chapter 9 requires for automation. The `request_id` is the join key to everything Chapters 8 and 10 recorded.

A record without a `request_id` is still valid. A routine `human_readonly` session has no request, and its record says so by leaving the field empty. An empty `request_id` on a session that holds `human_jit_write` triples or `human_breakglass` is a finding, and the checks later in this chapter look for it.

A store the accessed system cannot rewrite

The place where the entries live matters more than their format. Chapter 1 defined observability as a record that sits somewhere the principal cannot change. The principals who matter here are the ones with the most power: the database administrators, the platform team that runs the gateway, and `ledgerly_access_admin`, whose three logins can grant almost anything inside the roles it administers. If any of them can edit the record, the record is evidence only of what they chose to leave.

I would build the store with five properties, in this order.

1. **A separate account.** The audit store lives in a cloud account, project or host group that the database administrators, the gateway's opera-

tors and the three `ledgerly_access_admin` logins have no rights in. Someone has to administer it; that is a small group that holds no database administrator rights, and its own use of the store is visible to someone outside it.

2. **A write-only producer.** Each producer (the gateway, the runner, the credential broker, the pipeline, the revoke job, the restore tooling) holds its own credential that can call one operation, append. It cannot read, update or delete. A leaked producer credential can add false entries, which a chain makes detectable, and cannot remove true ones.
3. **An append service that holds the chain.** Producers do not compute hashes. A small service inside the audit account loads the head of the stream, links the new entry and stores both. A producer that computed its own link could also recompute everything after it.
4. **Retention that the account administrator cannot shorten.** Where the storage offers a write-once retention lock, use it for the retention period, so that deleting an entry before its end is not possible for anyone in the account. Where it does not, the honest statement is that the administrator can delete, and the chain and the anchor below are what make a deletion visible.
5. **An anchor.** Once a day the append service sends the head hash of each stream to a second place, owned by a different group: the security contacts' system, or a second store. Without an anchor, a person with full control of the store could rewrite a stream from some point onward, recomputing every hash after it. With one, the rewritten head differs from the anchor.

```

function appendChained(stream, record):
    // runs inside the audit account; producers can call only this
    // one writer per stream at a time: the head is read and locked
    // inside the same transaction that stores the next entry
    transaction:
        head = lockHead(stream)
        entry = { stream: stream, sequence: head.sequence + 1,
                  previous_hash: head.hash, body: record,
                  stored_at: now() }
        entry.entry_hash = hashOf(entry.stream, entry.sequence,
                                   entry.previous_hash, entry.body)
        // one system, so one transaction is allowed: the entry and
the
        // new head exist together or not at all
        storeEntry(entry)
        storeHead(stream, entry.sequence, entry.entry_hash)
        // commit point: the entry is durable and the chain has moved
        return { sequence: entry.sequence, hash: entry.entry_hash }

```

Two sessions can append to the same stream at the same moment, so the head is read under a lock inside the transaction; without it both would read the same head, write the same sequence number and fork the chain, and the verifier would then report tampering that never happened. The rule is one writer per stream at a time, enforced by that lock.

The verifier is a separate job with a read-only credential in the same account. Its work is the part that turns a store into evidence:

```

function verifyChain(stream, from, to):
    previous = loadEntryBefore(stream, from)
    for each entry in entries(stream, from, to):
        if entry.sequence != previous.sequence + 1:
            notify("security", "hole in " + stream + " at " +
entry.sequence)
        if entry.previous_hash != previous.entry_hash:
            notify("security", "broken link in " + stream)
        if entry.entry_hash != hashOf(entry.stream, entry.sequence,
                                       entry.previous_hash, entry.body):
            notify("security", "entry altered in " + stream)
        previous = entry
    // compare at the anchor's own position, not at the end of the
range
    anchor = loadAnchor(stream, atOrBefore: to)
    if anchor is not null and anchor.hash != entryAt(stream,
anchor.sequence).entry_hash:
        notify("security", "head differs from anchor in " + stream)

```

Now the honest limit, which I would put in the design document as a sentence of its own. This design gives evidence, not prevention. It does not stop a statement from running, and it does not stop someone with enough power over the audit account from destroying the store. It makes alteration detectable and makes the people who could alter it a short, separate list. The detection also depends on someone reading `verifyChain`'s output, so the verifier is a scheduled job with a heartbeat of its own, covered in the next section.

The cost is real. The audit account is another system to keep available, and Chapters 8 and 9 already accepted its worst consequence: a grant, a break-glass declaration or a statement is refused when its intent cannot be written. An outage of the audit store therefore blocks privileged work, and it blocks it at the moment someone may be waiting on an emergency. That is the price of a record that does not depend on goodwill.

Warning: Do not place the audit store in the database it describes, even in a schema with its own owner. Whoever can administer that database can alter the entries, and an `INSERT`-only role does not change that. The one structural exception the book already lists, that `ledgerly_owner` can alter `ledger_entries` and `audit_events` in a migration, is the reason `audit_events` is the application's business record and not the audit log.

Finding the gaps

A chain proves that the entries which are present were not altered. It does not prove that every statement produced an entry. The second family of checks looks for what should exist and does not. There are four kinds of gap, and each one has a different meaning, so the check reports them separately.

Sequence holes. A stream is gapless by construction, so a hole means the append service lost an entry or someone removed one. It is the rarest finding and the most serious.

Intents without outcomes. The gateway writes an intent before a statement and an outcome after. An intent with no outcome after the longest statement the timeout allows means the gateway failed between the two,

the outcome is waiting in the retry queue, or the statement is still running. Only the third is harmless, and it should not outlive the timeout.

Sessions on one side only. Every database login should have a known origin. For humans the origin is a gateway session. For services it is a lease from the credential broker of Chapter 4. For migrations it is a run of the pipeline. A login in the database log with none of the three is a path nobody designed, and a gateway session that claims statements the database never saw is a record that describes something that did not happen.

Silent producers. A stream that stops writing may mean a quiet week, or a dead producer, or a producer someone turned off. The check compares each stream with an expected cadence.

```
function detectGaps(window):
  gatewaySessions = sessionsFromAuditStore(window)
  databaseLogins = loginsInDatabaseLog(window)
  for each login in databaseLogins:
    if login not in gatewaySessions
      and not hasBrokerLease(login, window)
      and not isPipelineRun(login, window):
        notify("security", "login with no known origin: " + login)
  for each s in gatewaySessions:
    if s.statements > 0 and s.login not in databaseLogins:
      notify("platform", "gateway session with no database
login: " + s.id)
    for each i in intentsWithoutOutcome(olderThan: maxStatementAge):
      if not inRetryQueue(i):
        notify("platform", "intent with no outcome: " +
i.statement_id)
    for each stream in expectedStreams():
      if now() - lastEntry(stream) > expectedCadence(stream):
        notify("platform", "stream silent: " + stream)
```

The database log reaches the audit account through a shipping agent with its own write-only credential. Someone with host access can stop that agent, which is what the silent-stream check is for: a stopped agent leaves a stream with no entries, and that raises its own finding.

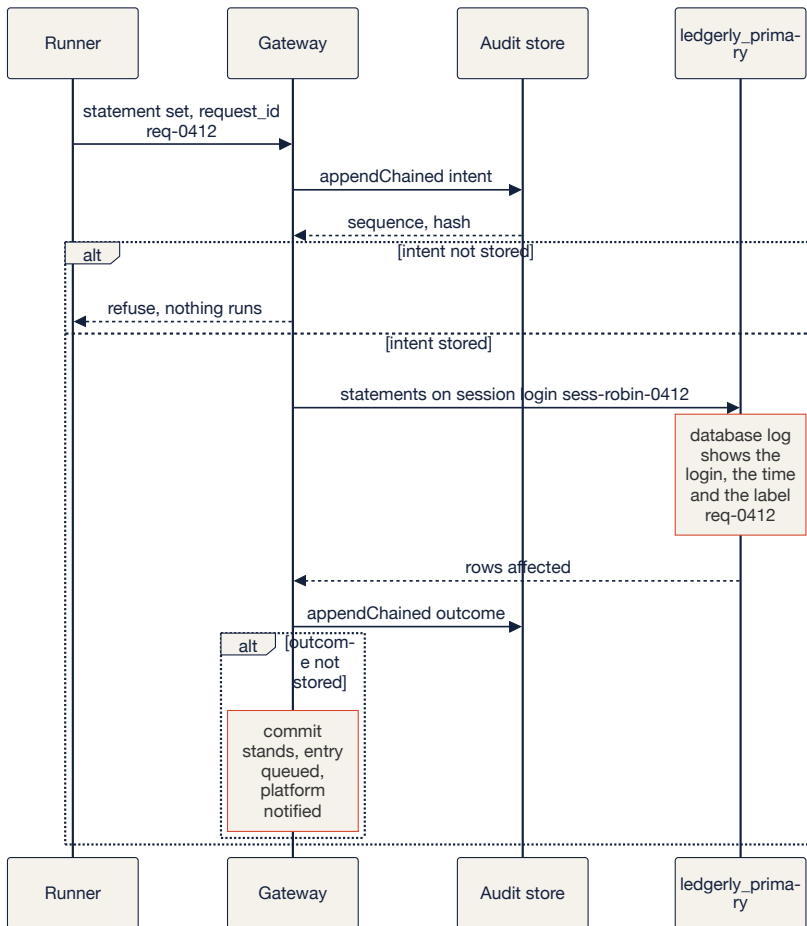
Joining three records

The gateway record, the database log and `audit_events` are produced by three systems that never share a transaction. Joining them is what turns the cases in Alex's morning from guesses into a timeline, and the join keys

have to be chosen before the incident, because they cannot be added afterward.

Three keys do the work. The **login** appears in the database log, and since a session login maps to exactly one person or one workload instance, it ties a log line to a session record. The **request id** is written by the gateway into the audit record and by the runner into the audit store's own `request_id` field; it ties the statement to the approval that allowed it. The runner never writes `audit_events`, which Chapter 9's matrix denies it. The `audit_events` rows that carry the correlation id Chapter 3 introduced come from the product path, where `admin-console` calls `payments-service`, and they join to the same request through that id. The **connection label** from Chapter 5 can carry the request id into the database log for mediated connections. That label is set by whoever opens the connection. In mediated execution the gateway opens it with the request id of the approved request, so it is reliable; in a session handout the person's client could set it to anything, so the login stays the key and the label stays a hint.

The diagram shows one approved data fix and where each record is written; the runner reaches the primary only through the gateway. If the outcome write fails, the commit stands and the gap is detectable.



What the record must not hold, and who audits the auditors

A statement record holds what people typed. A person who looks up a customer by account number has put the account number in the statement, and a log that stores the statement has copied a Restricted value into a place with no column-level protection. Chapter 12's warning applies in full, so the builder of the record has to decide what is kept.

I would store the statement in two parts. The **fingerprint** is the statement with every literal replaced by a placeholder, so `UPDATE invoices SET status = ? WHERE id IN (?)` records the shape and nothing else. The **literals hash** is a keyed hash of the literal values, which lets an investigator test a suspected value against the record without the record holding it.

The key matters: literals such as a status or a short account suffix have low entropy, and an unkeyed hash could be reversed by trying candidates. The key lives with the audit account, apart from the producers, and the investigator's tool asks for a comparison and never for the key.

```
function buildStatementRecord(session, statement, analysis):
    // analysis comes from the gateway's query analysis, not from the
    client
    parsed = parse(statement)
    record = { kind: "intent", session_id: session.id,
               login: session.login, principal: session.principal,
               on_behalf_of: session.on_behalf_of,
               roles: session.roles, request_id: session.request_id,
               fingerprint: redactLiterals(parsed),
               literals_hash: keyedHash(auditKey, literalsOf(parsed)),
               tables: analysis.tables,
               columns: classifyColumns(analysis.columns),
               started_at: now() }
    // a statement that cannot be parsed is recorded as a refusal with
    no
    // text, never stored verbatim as a fallback
    if parsed is null:
        record.fingerprint = "unparsed"
        record.literals_hash = null
    return record
```

The last lines matter most. A fallback that stores the raw statement when the parser fails would keep the one case where a literal is most likely to appear, because unusual statements are the ones a parser fails on. I would store nothing and let the refusal speak. The store itself is classified at least Confidential, because it describes who looks at what, and it has its own reader list and retention. How much a company may record about its own staff is a question for your legal and compliance people.

Who audits `ledgerly_access_admin`

`ledgerly_access_admin` is the role that can create logins and grant anything inside the roles it administers, used through three logins: the pipeline step that applies the access definition (Chapter 3), the credential broker (Chapter 4) and the gateway (Chapter 7). Chapter 3 said every action goes to the audit log; the mechanism is that each of the three is a producer with its own stream, so the record names which login acted. Each `CREATE LOGIN`, `GRANT` and `REVOKE` it issues is an `event` entry, written before the change, with the request or run that caused it.

That gives a check which closes the loop. The catalog of the database is the truth about what grants exist, and the audit store is the truth about what was granted. A grant that appears in the catalog with no matching entry was made outside the three producers, and that is the 2 a.m. grant Chapter 3's drift check was built to find. The two checks are one thing seen from two ends: drift compares the catalog with the code, and this compares the catalog with the record. Neither repairs anything; both report.

Migrations

The migration pipeline writes to the audit store before it runs a migration, as Chapter 11 requires, and again when it finishes or fails. The reason for the external entry is that the migration tool also records what it applied in a table inside `ledgerly_primary`, and `ledgerly_migrator` assumes `ledgerly_owner`, which can alter that table. A record the actor can re-write is a convenience and not evidence. The rule is that a migration counts as applied when the external start entry and the commit hash exist in the audit store, and a row in the migrations table with no matching external entry is reported by the same catalog-versus-record comparison, as a change that bypassed the pipeline.

Alerts that someone will answer

A record answers questions after the fact. A few of those questions are worth asking continuously, and an alert is the form that does it. The test I would apply to every rule is whether somebody is named to answer it and knows what to do.

Ledgerly starts with three rules, each of which exploits something the record knows and the other sources do not.

```

function evaluateAlertRules(window):
    for each r in outcomeRecords(window):
        // 1. large reads outside the person's working hours
        if isReadClass(r.statement_class) and isHuman(r.principal)
            and (r.rows_returned > nightRowThreshold or r.outcome ==
"row-cap-closed")
            and isOutsideHours(r.started_at, r.principal):
                notify("security", "large read outside hours: " +
r.statement_id)
        // 2. break-glass in the role set without a declared request
        if "human_breakglass" in r.roles and not
hasDeclaredRequest(r):
            notify("security", "break-glass role without a
declaration: " + r.session_id)
        // 3. Restricted columns returned to a human principal
        if touchesRestricted(r.columns) and r.rows_returned > 0
            and isHuman(r.principal) and not isDeclaredBreakGlass(r):
            notify("security", "Restricted data read by a human: " +
r.statement_id)
        for each s in sessionsOpened(window):
            if "human_breakglass" in s.roles:
                notify("security", "break-glass in use: " + s.id)

```

The thresholds are illustrations; the right `nightRowThreshold` is whatever puts a handful of events a week in front of a person. The second rule is the independent check on Chapter 8's declaration: that chapter alerts when a person declares the emergency, and this rule finds a session holding the role with no declaration, which is a grant that bypassed it. The third rule should almost never fire. `human_readonly` cannot read Restricted columns and a JIT request cannot name one, so a human read of Restricted data is either a declared break-glass session or a grant that does not match the matrix, and the second is a design finding.

Each rule needs a named owner and a first action: the security contact asks what the task was, the on-call platform engineer confirms the declaration and opens the review item, the platform team revokes a bypassed grant. A rule that is acknowledged without a look is security theater.

The access review

Alerts watch events. The **access review** is the scheduled check that each principal's access is still needed, and it is the control that catches the slow failure that no event shows: access that was right when granted and has quietly stopped being right. People change teams, a service stops reading a

table, a role keeps a privilege its workload has not used in months. Chapter 3 said that a wrong grant, correctly declared, passes the drift check, and that the access review is where that judgment belongs.

The review compares what is granted with what has been used. The record supplies the second list, and that is what makes it a decision and not a form to sign.

```
function runAccessReview(period):
  proposals = []
  for each grant in declaredGrants():
    // role to table privileges, identity provider group to role,
    // and the logins of ledgerly_access_admin
    use = lastUse(grant, inRecords: true)
    if use is null or use.at < now() - period:
      proposals.append({ grant: grant, finding: "unused",
                        proposal: "revoke" })
    if grant.owner is null or not isActivePrincipal(grant.owner):
      proposals.append({ grant: grant, finding: "no owner",
                        proposal: "assign or revoke" })
  for each r in breakGlassUses(period):
    if reviewFor(r.id) is null:
      proposals.append({ grant: r, finding: "no review",
                        proposal: "review now" })
  // the review proposes; a named reviewer decides, and is never
  // the person whose access is under review
  return assignReviewers(proposals)
```

Three rules govern how I would use its output. First, the function proposes and never revokes, for the reason Chapter 3 gave: an automatic revoke of a grant someone depends on is a new way to break production. Second, the `period` has to be longer than the longest cycle of the workload. A grant used by a quarterly report is unused in most months, so a one-month review would propose revoking it in two months out of three. Third, use is matched by privilege and not only by table. A role that selects from a table every night and has never updated it is a candidate for narrowing from `S, U` to `S`, and the record has what is needed because each record carries the statement class and the tables it touched.

The review has a second input the record cannot supply: things that are not grants. Chapter 13's register of copies belongs on the agenda, and so do the list of who may approve JIT requests and the list of who can read

the audit store. The question for each is the same: does the person or system still need this, and does someone still own it? The Appendix has a template.

Evidence, not compliance

Data-protection and payment-card regimes tend to ask a family of questions about access: who can reach the data, who did what, how long access lasts, where copies are, and what happens in an emergency. An auditor working under one of these regimes will usually want to see that a control exists, that it ran, and that its results were looked at. Your legal and compliance people own the interpretation of any regime. This section claims no more than the engineering above can deliver, which is evidence: records that answer questions, and a way to show they were not altered.

Nothing in this chapter makes a company compliant with anything. The table lists the questions an auditor tends to ask, the control that answers each, the artifact it leaves and the chapter that builds it; the Appendix repeats it as a crosswalk.

Question	Control	Artifact	Chapter
Who can reach production data, and by which path?	Gateway as the only door, role schema as code, drift check	Network rule list, role definitions, drift report	3, 7
Who ran a given statement, with which grants and approval?	<code>access_audit_record</code> with request id and approver	The record and the <code>verifyChain</code> report	14
How long does elevated access last, and who approved it?	JIT request with approver and <code>expires_at</code> , revoke job	Request records, revoke log, overdue-grant check	8
How is sensitive data identified and limited?	Classification in the schema, masking, withheld columns	The catalog, the masked-load verification	12
Where are copies of production data, and who owns each?	Register with owner and end date, sweeper	Register, sweeper log	6, 13

Question	Control	Artifact	Chapter
Who changes the schema, and who reviewed it?	Migration pipeline, protected branch, external start entry	Migration records matched to commits and audit entries	11
Is access still needed?	Access review with named reviewers	Completed review with decisions	14
What happens in an emergency?	Break-glass with alert and mandatory review	Review records, alert history	8, 15
Who administers access itself?	<code>ledgerly_access_admin</code> logins as separate producers	Their streams and the catalog-versus-record comparison	3, 14

Trade-offs and costs

The record has a bill, and most of it is paid in two places.

Storage and attention. A record per statement, twice, grows with traffic, and the store needs its own retention, backups and reader list. The larger cost is attention. A store nobody queries is expensive security theater, so the weekly sample of elevated sessions from Chapter 7, the alerts and the review are part of the price and not extras.

A new dependency on the critical path. Because the intent is written before the statement runs, an audit outage blocks privileged work. The alternative, running statements and logging later, makes the record optional exactly when it matters. I would keep refusal for grants, break-glass and writes, and consider a queued write for routine `human_readonly` reads, with a cap on how long the queue may grow, if the team decides a read outage costs more than a gap. That is a decision for the platform team to make once and write down.

When not to build all of it: a company with a handful of engineers, one database and no gateway yet should not start with an append service, an anchor and a separate account. Start with three things: log each person's session under their own login, ship the database log off the host to a place the administrators cannot delete from, and put a named person on a monthly review. Add the chain and the anchor when the number of people who could alter the log makes "trust" an inadequate answer.

Common mistakes

- **Treating the database log as the audit log.** It is the most complete source and it sits under the administrators' control. Ship it elsewhere and use it as one input to the join, not as the record.
- **A hash chain without a separate store.** The chain looks like proof and is recomputable by whoever can write the store. Put the store in another account, hold the chain in the append service and anchor the head.
- **Storing the raw statement.** It is tempting because it makes investigation easy, and it copies Restricted values into a store with no column protection. Store the fingerprint and a keyed hash, and refuse the fall-back to raw text.
- **Alerts without an owner.** A channel nobody is named to answer becomes noise. Give every rule an owner and a first action.
- **Auditing everything except the auditors.** The role that grants access is the one whose actions matter most. Give it its own producers and compare its work with the catalog.

Key takeaways

- The audit log is the record of who did what to which data, kept where the actor cannot change it, and the gateway record, the database log and `audit_events` are three inputs to it, not the thing itself.
- A statement record is built from a list of questions written first, and its identity fields end at a person even when a machine acted, through `principal` and `on_behalf_of`.
- Tamper evidence comes from a separate account, write-only producers, an append service that holds the chain and an anchor owned by someone else, and it gives evidence and never prevention.
- A chain shows that present entries were not altered, so separate checks look for sequence holes, intents without outcomes, database logs with no known origin and producers that went silent; the three records join on the login and the request id, and the connection label is only a hint.
- The record must never become a copy of the data it describes, so statements are stored as fingerprints with a keyed hash of the literals, and

the store is classified and restricted like the data.

- Alerts need an owner and a first action, and the access review compares grants with use over a period longer than the workload's cycle, so that revocations are proposed from evidence.
- Evidence is not compliance: the crosswalk of question, control, artifact and chapter shows what the system can prove, and your legal and compliance people own what that proof means under any regime.

The record now exists, is joined and watched. What remains is the day it is needed under pressure, when the person asking is no longer an auditor but an on-call engineer with a leaked credential. The next chapter turns to access during incidents, and uses the record built here to find out what a compromised principal actually touched.

Chapter 15 — Access During Incidents

Imagine a Saturday afternoon at Ledgerly. Sam, from support, writes in the on-call channel: a work laptop was taken from a parked car, and a gateway session may have been open on it. Robin is on call and reads the message twice. Nothing is known to be wrong, but "may have been open" is the kind of sentence that has to be answered, not weighed.

Robin needs three things, and the order matters. The first is to stop whatever that session, and any login behind it, can still do. The second is to learn what it did while it could. The third is to keep the business running while the first two happen, which means other people still need the database, some of them urgently, and the usual approvers are at home.

This chapter treats incident access as something to design and rehearse. It exercises all three pillars at once: authorization when grants are removed or added in a hurry, execution when repairs have to run under pressure, and accountability when the record must say what happened. In the five-part model, Identity finds the logins, Grant and Path cut them, Execution carries the emergency writes, and Record measures the damage. This chapter adds the last step of Ledgerly's target state: incident access has a rehearsed path, and revocation is a routine. The question is how Ledgerly stops access, measures what was reached and keeps working, without re-opening the doors the earlier chapters closed.

The problem, and why it is hard

An incident is the moment when every earlier control looks like an obstacle. The approval that was reasonable on a quiet Tuesday is a wait on a Saturday, the gateway that records everything is one more thing that can be down, and the cleanest shortcut in the room is a login that skips all of it. The usual response is to build that login in advance "for emergencies". It is the shared admin password again, with a better name.

Three things differ under pressure. Time is short, so steps that take a person to remember get skipped. Knowledge is partial, so the first scope estimate is wrong in one direction or the other. And the actors are unusual: people who rarely touch the database are suddenly asked to, and people who touch it daily are tired.

The design question is therefore which controls relax and which do not. I would hold one rule: urgency changes speed, never identity, never the statement that runs, and never the record. Waiting periods and approver ordering may shrink. A principal acting under its own login, an approval tied to a statement hash, and an entry in the audit log may not. If a control cannot be shortened without removing one of those three, the control is wrong, not the rule.

The speed has to be built. A revocation that takes a dozen manual steps will be done in the wrong order or half done. The order is code, the scope query is code, and the reconciliation that says whether the incident left access behind is code. A runbook that points at those is a procedure; one that replaces them with judgment is a hope.

Containing a compromise

Containment here means stopping one principal's access, whether the cause is a lost laptop, a leaked credential or a workload that behaves in a way nobody expected. It does not decide the cause. Chapter 4 separated two questions that panic tends to merge: how to stop a credential from working, and what it did while it worked. Containment answers the first, and the order below keeps the second answerable.

The order is: revoke grants, close sessions, rotate credentials, scope the damage from the audit log, then record and notify. Each step is placed where it is for a reason.

Revoking grants comes first because a session with no privileges can do nothing, and a reconnect gets nothing. Closing sessions comes second, for the in-flight work and the open connections that outlive a revoked grant (Chapter 4 made the same point about expired credentials). Rotation comes third because it stops new logins and does nothing for the ones al-

ready open. Scoping is read-only and needs the record, not the access, so it waits until the access is gone. Notification is last, but the record of each step is not: every step is appended as it finishes.

```
function containCompromise(incident, principal, since):
    unfinished = []
    // every step runs even if the one before failed: a stuck revoke
    must
    // not leave sessions open and the credential valid
    for each step in ["revoke-grants", "close-sessions", "rotate-
credentials"]:
        outcome = "done"
        try:
            runContainmentStep(step, principal)
        on StepFailed:
            outcome = "failed"
            unfinished.append(step)
            notify("platform", "containment step failed: " + step + "
" + incident.id)
    // the log never delays containment; an unreachable log queues
    the entry
    auditOrQueue({ action: "contain", actor: incident.commander,
                    subject: principal, request: incident.id,
                    step: step, outcome: outcome })
    // read-only and last: it needs the record, not the access
    scope = blastRadius(principal, since)
    saveIncidentRecord(incident.id, { principal: principal, since:
since,
                                scope: scope, unfinished:
unfinished })
    notify("security", "contained, scope attached: " + incident.id)
    return unfinished
```

Two details deserve a note. The function returns the unfinished steps and does not raise, because a half-contained principal is a state to be tracked, and an exception would end the one process that knows about it. And `auditOrQueue` is the same helper as in Chapter 7: a log outage delays the entry and never the removal of access. The asymmetry is the one Chapter 8 chose for revocation, a grant without a log entry being an exposure and a revoke without one being a gap that can be filled later.

What each step means depends on what the principal is. The table lists the four cases Ledgerly has.

Principal	Revoke grants	Close sessions	Rotate credentials
Person, session logins	Remove the JIT triples, <code>human_breakglass</code> and <code>human_readonly</code> from every session login the person holds	<code>endSession</code> for each login; disable the person in the identity provider so no new session opens (Chapter 7)	Reset the person's identity-provider credentials and factors; session logins are dropped, so there is no database password to rotate
Service, dynamic credential	End the lease; the login goes with it (Chapter 4)	The broker closes the sessions as part of ending the lease	Nothing by hand; the next request gets a new value
Service, static credential in an A/B pair	Remove the <code>app_*</code> role from the leaked login at once, without waiting for the old login to drain	Close its remaining sessions	Publish a new value to the idle login and switch the pointer; restart services that cannot re-read their secret
<code>ledgerly_access_admin</code> login	Remove its privileges, which stops provisioning for whichever principal used it	Close its sessions	Issue a new credential to that principal; the scope question becomes which grants and logins no request explains (the drift check of Chapter 3)

The static row is the one that matches Chapter 4. A leaked static credential is not a planned rotation, and the leaked login is cut before its replacement is ready, which costs an outage in the services that use it. Chapter 4 already accepted that cost, and I would keep the acceptance: a leaked credential that keeps working is the larger problem. The services that need a restart should be on a list before the day it matters. In the starting state, where every service shares one admin login, step one is the whole company's outage, which is the strongest argument for building routine revocation first.

The last row is the heavy one. The login that administers access can grant anything inside the roles it administers, so containing it stops the gateway, the pipeline or the broker from provisioning anything until a new credential exists. That is the price of cutting the login with the widest reach, and a team should decide in advance who may pay it.

Scoping the blast radius from the record

Once access is gone, the question becomes what it did. The answer is one query over `access_audit_record`, whose design Chapter 14 describes, run with the fields that chapter defines. The query is the blast radius of one principal since the time T when its access may first have been mis-used: every table it touched, how many rows came back, how many were changed, and which of those tables hold Restricted data.

```
// illustrative notation; field names as in Chapter 14
SELECT t.table_name,
       classOf(t.table_name)           AS data_class,
       COUNT(*)                       AS statements,
       SUM(r.rows_returned)           AS rows_returned,
       SUM(r.rows_affected)           AS rows_affected,
       SUM(CASE WHEN r.outcome = "refused" THEN 1 ELSE 0 END) AS
refused,
       MIN(r.started_at)               AS first_seen,
       MAX(r.ended_at)                AS last_seen
FROM access_audit_record r
  EXPAND r.tables AS t                // one row per table
touched
WHERE r.kind = "outcome"              // one row per statement;
intent rows have no counts
  AND r.login IN :logins
  AND r.started_at >= :since
GROUP BY t.table_name
ORDER BY (data_class = "Restricted") DESC, rows_returned DESC;
```

Choosing `:logins` and `:since` is where most of the judgment sits. For a person, the gateway creates a new login for each session, so the set is every session login the principal has had since T, not one. For T, use the earliest moment the access could have been misused, such as the time the laptop was last known to be in its owner's hands, and not the time somebody noticed. A window that is too narrow produces a clean answer to the wrong question.

The result is only as good as the record, so three checks come before anyone reads it.

1. `verifyChain` passes over the window, as Chapter 14 defines it. If it fails, the result is evidence of something other than the incident, and the incident gets a second record.
2. The window has no sequence holes and no gateway session without a database session, or the reverse, which is Chapter 14's gap detection. A hole inside the window is a stretch where the answer is unknown.
3. The principal's side paths are checked separately: restores, exports and copies it requested (Chapter 13), found through the register of copies and not through this query.

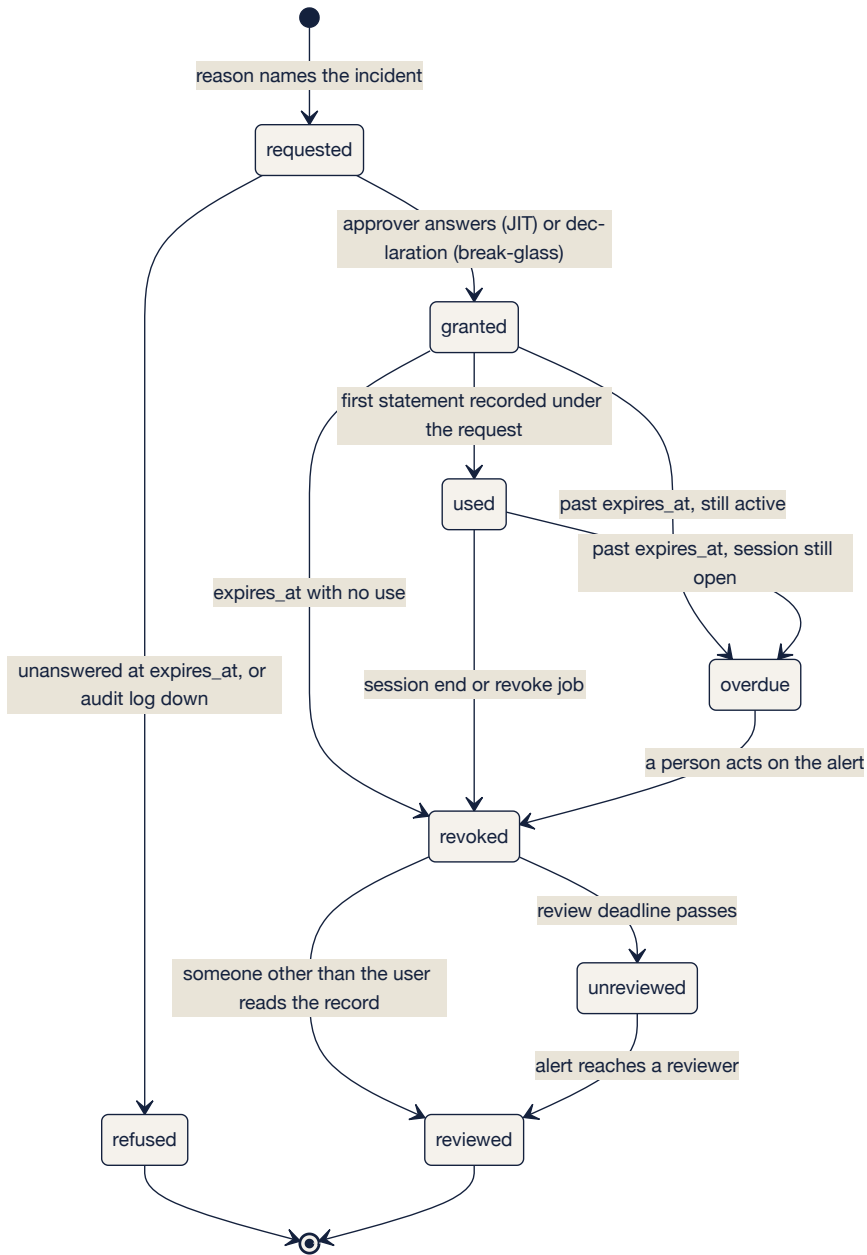
If any check fails, the result is a lower bound, and I would say so in the incident record in those words. The query also has limits of its own.

`rows_returned` is a count, not a list: it tells you that a Restricted table returned some number of rows in the window, and it cannot tell you which customers they were. The `fingerprint` and `literals_hash` fields let you ask whether a known statement was repeated, but the literals are redacted by design. So the record bounds the exposure and does not enumerate it. A nonzero `rows_returned` on a Restricted table is the line the security contacts and the reader's legal and compliance people need, and the interpretation of what it means, including any notification duty, is theirs.

The life of incident access

Containment removes access. An incident also needs access added: someone has to look, and someone may have to repair. The risk in that direction is the grant that was created in a hurry and is still there on Monday. The cure is a lifecycle with a visible end, in which every piece of access made for an incident passes through the same five states.

The diagram is a view over the records Chapter 8 already keeps, not a new status field. `granted` is `approved` in the request record, `used` means a statement was recorded under the request, and `reviewed` means someone other than the user read the record. Look for the two exits that do not reach `reviewed`: a grant that stays past its end, and a review that never happens.



For this to work, every request made during an incident carries the incident's identifier in its reason. The gateway's form can require it, and a request without one in an open incident is refused. That one convention is what lets a job ask "everything this incident granted".

```

function reconcileIncidentAccess(incident):
    // watches the incident's own requests; never changes access
    (Chapter 8:
    // an automatic revoke of unknown grants is a way to break
    production)
    for each r in requestsWithReason(incident.id):
        if r.status == "approved" and r.expires_at < now() -
        graceMinutes:
            notify("platform", "incident grant past expiry: " + r.id)
            if r.status == "revoked" and reviewFor(r.id) is null
                and now() - r.expires_at > reviewDeadline:
                    notify("security", "incident access not reviewed: " +
                    r.id)
            // the database side: grants that no request explains
            for each g in unexplainedGrants(since: incident.opened_at):
                notify("platform", "grant with no request: " + g.login)

function closeIncident(incident):
    reconcileIncidentAccess(incident)
    // an incident is not over while any access it created is still
    open
    if openFindings(incident.id) > 0:
        raise IncidentHasOpenAccess
    markClosed(incident.id)

```

This job does not replace the overdue check and the drift check of Chapters 8 and 3. It looks at the same facts from the incident's side, and it gives the incident a gate: nobody can mark it closed while a grant is still live or a break-glass use is unreviewed. A team that uses a ticket system can wire `closeIncident` to the ticket, which turns "we forgot to revoke" into a visible blocker.

Reviewed

The last state is the one most often skipped, because the incident feels over. The review is the one Chapter 8 describes for break-glass, extended to every grant the incident made. It is blameless and asks about the system: was the access the smallest that could do the job, and was there a faster path that should have existed? The Appendix has a post-incident template. The finding that pays is usually a missing path.

Writes during an incident

Repair is the most tempting place to relax a control, because the harm is visible and the data is wrong right now. Chapter 10 settled that a repair is a data fix, and Chapter 8 settled that break-glass is not a route for repairs. Under pressure I would keep both decisions and make the compliant path fast, instead of making the wrong path available.

The first choice is which path the need belongs to. The order of the checks runs from the narrowest door to the widest.

```
function chooseIncidentWritePath(need):
  // first match wins; Restricted values are never written by a
  person
  if need.touches_restricted_column or need.is_owner_link_change:
    return "product path: admin-console asks payments-service
    (Chapters 9, 10)"
  if need.is_schema_change:
    return "pipeline, expedited lane (Chapter 11)"
  if need.stops_ongoing_harm and need.is_small_limiting_action:
    // pausing, disabling or flagging: never the repair itself
    return "break-glass session, reviewed afterward (Chapter 8)"
  return "data fix: JIT request, runner, expedited review (Chapters
  8, 10)"
```

Break-glass is reachable only for the small limiting action: a flag that stops a job writing bad rows, a status that holds a batch. The lasting repair of the data is always a data fix, and on the same reasoning as before: a repair run in a raw break-glass session skips the one control that reads the state-ment before it runs, and it runs where an unbounded `UPDATE` is still possible (Chapter 8 names that as an accepted risk). During an incident the temptation is stronger, and so is the cost of one wrong write.

The data fix keeps its shape. The request names a reviewed fix and carries an `approved_hash` taken from the merge record, the runner refuses a script whose hash differs, the grant is derived from the fix, and the count guard and the snapshot still apply (Chapters 9 and 10). What the incident changes is the review.

```

function expediteFixReview(fix, incident, reviewer):
    // relaxes: the waiting period, and who may be asked first
    // never relaxes: a reviewer who is not the author, the hash, the
runner
    if incident.status != "open":
        raise NoOpenIncident
    if reviewer == fix.author:
        raise SelfApprovalNotAllowed
    mergeRecord = mergeFix(fix.id, reviewedBy: reviewer, expedited:
true,
                        incident: incident.id)
    // the hash is read from this record by the gateway, not from the
runner
    queueFollowUpReview(fix.id, reviewer: "platform", due:
nextWorkingDay)
    notify("platform", "expedited fix merged: " + fix.id)
    return mergeRecord

```

This mirrors the expedited lane that Chapter 11 gives migrations: same job, same credential, same record, different timing. The follow-up review is where the shortcut is paid back, and it is also what turns an expedited path into a counted event. If a team finds that most fixes in a quarter use it, the normal lane is too slow for the work, and the right response is to speed the normal lane, not to stop counting.

Warning: A fix that "needs" break-glass is a signal about something else, as Chapter 10 says: a ceiling that lacks a column, an approver pool too small for the hour, or a Restricted column that belongs on the product path. In an incident that signal is easy to ignore and expensive to ignore twice.

The cases earlier chapters deferred

Three situations were passed forward by earlier chapters. The table is the decision up front; the subsections give the reasoning.

Situation	What stops	What does not change	Decided in advance
The gateway is down	Human data access (no second door)	Services still run on their own credentials	Who restores the gateway, and when the sealed instrument may be used

Situation	What stops	What does not change	Decided in advance
A restore is needed during the incident	Nothing, if the path was rehearsed	Approver, isolated target, register entry, scheduled deletion	A named approver, a prepared target, a restore runbook
Approvals are slow at night	A repair that is not an emergency	Self-approval is still refused	An escalation order and who may widen the pool

The gateway is down

Chapters 7 and 8 decided that there is no second door: humans reach the database only through the gateway, so a gateway outage is also a break-glass outage. The reasoning there was that a credential that works without the gateway also works without the logging, the identity and the expiry. I would hold to that on the worst night, and the first move in an outage is to notice what has not stopped. The services connect with their own credentials and keep running, so many incidents in which the gateway happens to be down do not need a person in the database at all. Roll back the deploy, pause the job, flip the feature flag, scale the service: none of these go through the gateway.

If the incident does need data access, the incident is now two incidents, and the first is the gateway. Restoring it comes before the data question, and the people who restore it are the platform team, using the deploy path and the redundancy that Chapter 7 asked for.

If every instance is down, no human data path exists, and the one instrument left is the superuser that Chapter 3 treated as sealed. It is an engine-level tool, not a data path: it bypasses the append-only and Restricted protections, and its network route is closed to people. I would limit its use to restoring the gateway or what the gateway depends on, never to reading or writing business data, and I would write the rule down.

Warning: Do not use the sealed instrument to look at data because the gateway is down. It bypasses the append-only and Restricted protections, and the access it gives is the access every earlier chapter was built to prevent. If data must be read, restore the gateway first.

Its use under the rule has a mechanism. Retrieval of the credential leaves a record, a second person is present and named, opening the network route is a change with an owner and an end time, and the credential is rotated afterward because a person has seen it. The gateway is not recording during this window, so the engine log is the only trail. After the gateway is back, the platform team appends an entry by hand, like the external audit entry Chapter 14 describes for migrations, and the use gets the review that every break-glass use gets.

A restore during an incident

Chapter 13 made restore a request: an approver who is not the requester, an isolated target, a register entry, a scheduled deletion and a closed start. The cost named there was speed during an incident, and its answer was to rehearse and to arrange approval in advance. That is the whole mechanism, so it is worth spelling out what "in advance" means.

Before the incident, the runbook names who can approve a restore at night, the target is prepared and isolated, and the restore request has a form that takes an incident identifier. During the incident, the request goes through unchanged: the approver is not the requester, the register entry comes first, the scheduled deletion is set before the copy exists, and logins are disabled and the access definition is applied before the copy opens. The incident adds the reason, not a shortcut.

What the copy is for matters more than how it is made. During an incident a restored copy is a read source, usually for old values: the state of a column before a wrong write, or rows removed by mistake. Its readers are the people on the request. The repair that follows runs against the primary as a data fix, with the values read from the copy as input to a reviewed script, and Chapter 13's `reapplyDeletions` runs before the copy opens, so the copy does not bring back data that was deleted on purpose. Replacing the primary with a restored copy is a recovery decision, and it belongs to the platform team and the incident commander. The access question inside it is the one above: who reads the copy, and when it is deleted.

Slow approvals at night

Chapter 8's night scenario ended in a decision that this chapter keeps: if the approver pool does not answer and the need can wait, leave a note and stop. That is a good outcome and one the old Ledgerly could not produce. For a need that cannot wait, the escalation is built into the request, so a page that goes unanswered moves down the pool inside the same request.

```
function escalateApproval(request, pool):
    // one request, one record: the page moves, the request does not
    restart
    for each approver in pool.inOrder():
        if approver == request.requester or approver ==
request.fix_author:
            continue
        page(approver, request.id)
        if waitForAnswer(request.id, within: escalationMinutes):
            return
    // pool exhausted, window still open: the incident commander
    decides
    notify("platform", "approval unanswered, pool exhausted: " +
request.id)
    // never widens the pool or approves by default; the request stays
    pending
    // until expires_at, when it is refused
```

Two rules sit around this. First, the pool is widened only by a person who owns it, and the widening is a record: during a declared incident the incident commander may add a named approver for the incident, with an end. Second, an automatic approval for whoever is on call is still not built, because a control that grants every request within seconds is no control. If approvals regularly outlast the grant window, the finding is about the rota, the pool size or the hour, and the remedy is to add people to the pool, not to take the approver out of the loop.

Rehearsing the paths

A path that has never run is untested. The whole chapter is built from steps that nobody wants to discover for the first time during an incident, so each needs a rehearsal, called a game day here: a scheduled exercise with a scenario, an expected outcome and a check on the record. Four sce-

narios cover what this chapter built. They are defensive by design, each starting from a trigger the monitoring already knows how to see, and each carries an observable pass condition.

Scenario	Trigger	Expected behavior	Pass condition	Parts exercised
Credential leak	A drill principal's credential is declared exposed	<code>containCompromise</code> in order; scope query attached to the incident	No live login for the principal, unfinished is empty, scope names every table the drill touched	Identity, Grant, Record
Gateway down	One then all gateway instances stopped in a non-production copy	Human access stops; services keep running; the restore path runs	Platform team restores the gateway using the runbook; sealed instrument unused, or used under its rule with a record	Path, Record
Suspicious export	An alert on large <code>rows_returned</code> , from <code>evaluateAlertRules</code> (Chapter 14)	Contain the principal; find the destination in the register of copies; delete it	The copy is found, ended and recorded; the scope query agrees with the register	Identity, Record, Execution
Mistaken bulk <code>UPDATE</code>	A write that fits the row cap and changes the wrong rows, in a rehearsal environment	Stop further writes; compute the reversal from the snapshot; run it as an expedited data fix	Reversal merged by a reviewer who is not the author; follow-up review queued	Execution, Grant, Record

A game day passes only on the observed result, never on someone remembering what to do. If a drill finishes because somebody knew a shortcut, it failed, and the shortcut becomes the thing to build. Write down the times each step actually took; those numbers belong to your company, and I would not give a number that does not.

Use a non-production copy where possible, and synthetic principals in production exercises, so no real customer data is read to practice reading it. Drill copies are registered and closed like any other copy (Chapter 13).

Trade-offs and costs

The first cost is the rehearsal itself. Game days take engineers away from product work, and a drill needs an owner and a calendar. For a team of three, a single drill a year that covers the credential leak is a reasonable start, and the other three scenarios can wait until the team grows.

The second cost is the outage that containment causes. Revoking a service's credential at once is an outage in that service, and containing `ledgerly_access_admin` stops provisioning. Both are choices to make in advance, with the people who bear them, because the person deciding at 3 a.m. should not also be learning what the choice costs.

The third cost is the incident-identifier convention. It adds a field to every request and one more reason for a form to be refused, and it is only as good as people's willingness to fill it in. The reconciliation that depends on it fails quietly when it is not filled in, so the gateway should enforce the field and not rely on habit.

The fourth cost is honesty about scope: some incidents end with a bound larger than the truth, and I would report the bound rather than claim a precision the record lacks.

Do not apply the full machinery to a system that has no audit log yet. `blastRadius` needs the record, and `reconcileIncidentAccess` needs the request records; a team without them should build those first (Chapters 8 and 14), and in the meantime write a short runbook that names who revokes, who closes sessions and who owns the question of what was touched.

Common mistakes

- **Rotating without revoking.** A rotation changes the secret and leaves the open session and the granted privileges in place. It is tempting because rotation is a routine the team already has. Revoke first, then close sessions, then rotate.
- **Scoping before containing.** The urge to understand comes before the urge to stop, and a team can spend an hour on the query while the access stays live. Contain first. The record survives containment because it lives in a store the access cannot touch.
- **Trusting the log without checking it.** A scope query over a window with a gap reads as a clean answer. Check the chain and the sequence first, and report a lower bound when either fails.
- **Break-glass as the repair lane.** It is fast and it is available, and it has no statement review. Use it for the small limiting action and run the repair as an expedited data fix.
- **Leaving incident access open.** The incident feels over when the cause is fixed, and the grants stay. Tie closing the incident to the reconciliation, so the ticket cannot close while access is open.

Key takeaways

- Urgency changes the speed of a control, never the identity that acts, the statement that runs or the record that is left.
- Containment has an order: revoke grants, close sessions, rotate credentials, scope from the audit log, then record and notify, and every step runs even if the one before it failed.
- Revocation is not rotation: a rotation stops new logins and does nothing for the ones already open, which is why a leaked static credential is cut first and replaced after.
- The blast radius comes from one query over `access_audit_record`, and it is a lower bound whenever the chain or the sequence has a gap.
- Incident access passes through requested, granted, used, revoked and reviewed, and an incident cannot be closed while any grant it created is still open.

- Repairs during an incident are data fixes with an expedited review, and break-glass stays for the small action that stops harm.
- A path that has not been rehearsed is an assumption, so the credential leak, the gateway outage, the suspicious export and the mistaken bulk UPDATE each get a game day with an observable pass condition.

Chapter 16 gathers the controls of the whole book into a checklist and a maturity ladder, and returns to the sentence that every chapter has served.

Chapter 16 — An Access Checklist and Closing Thoughts

Consider Ledgerly a year after the security questionnaire of Chapter 1. Another prospective customer sends the same two lines: "List every person and system that can connect to your production database" and "How do you know what they did?" Alex opens the access inventory instead of the password manager, reads the list of doors, and sends it back in an afternoon. Nothing in the answer is impressive. It is short, every row has an owner and an end, and none of it required digging.

Robin, who is on call that week, asks over chat: "We have a gateway, a runner, a pipeline, an audit store and three review jobs. Which of these did we need, and which did we build because the book said so?" Alex has no tidy reply. Some pieces earned their place the first time a grant expired on its own; others have never been used in anger, and Alex cannot say whether that means they work or sit idle.

This chapter serves all three pillars at once: authorization, execution and accountability. It adds no new control. It adds the stages of the target state in the order a team can reach them, the patterns scored side by side, and a checklist whose lines are observations instead of opinions. The question it answers is the one Robin asked: how does a team decide what to build next, and how does it know a control works?

The problem, and why it is hard

Fifteen chapters leave a reader with many controls and no order. Each one was introduced with its own reasoning, and each reasoning was sound in its chapter. Read together they look like a project of a year or more, and a team facing that list tends to do one of two things. It starts with whatever is cheapest and stops when the work gets political, or it starts with whatever is most frightening and builds an elaborate piece of the design while the shared password is still in five configuration files.

The difficulty is that "done" is hard to see in access work. A role schema that exists on paper and one that is enforced look the same in a review meeting, so teams measure by what was installed, and installed is the wrong unit. The useful unit is a test someone can run: a statement they can trace, a connection that is refused, a grant that is gone.

It is also a matter of how the system is built rather than what the policy says. A policy states what should happen; a design determines what can happen and what leaves a record when it does. Each line below can be answered by looking at a system, a log or a table, and a line that can only be answered by asking someone what they intend is not on the list.

The thesis, landed

The book argued one sentence, and this is where it is checked against the chapters:

Production database access is not a credential-distribution problem; it is an authorization, execution, and accountability problem.

The sentence has three pillars, and each pillar stands on specific chapters. Authorization is grant plus path, decided per request. Execution is what happens to each statement. Accountability is identity plus record. A password touches only the first of these, and only at the door.

Pillar	Parts of the model	Chapters that build it	The question a reader can now answer
Authorization	Grant, Path	3, 5, 6, 8, 12	What may this principal do, where, for how long, and with which columns hidden?
Execution	Execution	7, 9, 10, 11, 13	How did this statement reach the data, who or what inspected it, and what bounded it?
Accountability	Identity, Record	4, 7, 14, 15	Which person or workload instance acted, on whose request, and what evidence ties it to the result?

Chapters 1 and 2 frame all three, and Chapter 15 exercises them under pressure. Chapter 7 appears twice because the access gateway is both the path and the place where a person becomes a session login a log can

name; Chapter 13 sits under execution because the export step of `reporting-job` runs a statement set, and the other side doors are routes around the same controls.

Why the credential work was necessary and was not the answer

Chapter 4 is about credentials, and it is the chapter most readers expect to be the center of the book. It is necessary. A service that reads its password from a shared file has no identity of its own, and without distinct identities the record cannot name anyone. Short-lived and dynamic credentials make revocation a routine step. Workload identity removes the file.

None of that decides what a principal may do once it is in, how a statement is checked, or what is written down. A team could finish Chapter 4 perfectly and still have a role that owns every table, a laptop session that commits every statement as typed, and a log in which every line has a distinct name and none has a reason. A password store supplies the distinct names the Identity part needs and, through revocation, a piece of Grant; a password touches only the first pillar, and only at the door. The other parts are built elsewhere, and the rest of the book is where.

When a proposal says "we will fix access by improving X", place X on the model before approving it:

```
function placeControl(proposal):
    parts = partsOfModelTouched(proposal) // Identity, Grant, Path,
    Execution, Record
    if "Execution" not in parts and "Record" not in parts:
        // a control that touches only the door leaves two of three
        pillars as they were
        note("improves how access is obtained; says nothing about what
        it does or what remains")
        for each part in ["Identity", "Grant", "Path", "Execution",
        "Record"]:
            if part not in parts:
                note("unchanged: " + part)
    return notes
```

The function is not a gate. It names which parts a proposal leaves as they were, because an empty part is a finding and not an omission.

A catalogue of patterns, scored by pillar

Chapter 1 drew the range of ways in, and Chapter 7 compared the concrete paths. The table below puts the five patterns of that range side by side and scores each one against the three pillars. The scores are my judgment, not a measurement. They say how much of a pillar the pattern can evidence by itself, with the grants and logging of the earlier chapters held equal. Read "strong" as "the pattern is built to prove this", "partial" as "it proves some of it, or only if something else is added", and "none" as "the pattern is silent here".

Pattern	Authorization	Execution	Accountability
Direct access (client to database)	Partial: grants only, reach is whatever the network allows	None	Partial at best: per-person logins if someone provisions them, otherwise none
Bastion	Partial: narrows the network path	None: it forwards, it cannot tell a <code>SELECT</code> from a <code>DROP</code>	Partial: who logged in to the host, not what ran
Session proxy or access gateway	Strong: per-session logins, expiry, per-request grants	Partial: bounded by grants and guarded settings, statements are typed freely	Strong on who and when, partial on what
Statement-mediating executor	Strong	Strong: inspected before it runs, bound to an approved hash	Strong: statement, approver and result in one record
Product path (no SQL)	Strong: the operation is the permission	Strong: the operation is fixed	Partial: the person is in the tool's audit trail, the database log shows the service login

Direct access

- **Proves:** that a login exists and that its grants were what they were.
- **Cannot prove:** which person used a shared login, or what a statement did.
- **Cost:** none to build, which is the attraction. Every cost arrives later, as a week of digging.
- **Failure mode:** the shared admin password of Chapter 1.

- **Do not use when:** more than one person or workload can reach the login. It is tolerable for a single-engineer system whose only consumer is its owner, and I would write down what would make that stop being true.

Bastion

- **Proves:** that a connection came through one controlled host, and, if accounts on it are per person, who logged in there.
- **Cannot prove:** what any statement was, since the host does not speak the database protocol.
- **Cost:** a host to patch and keep available, in return for a single place to turn the network path off.
- **Failure mode:** a shared bastion account turns the host into the shared password again, one hop earlier.
- **Do not use when:** you need statement-level evidence. It remains useful as network reach in front of something that does record.

Session proxy or access gateway

- **Proves:** that a named person held a specific session login, between two timestamps, under specific grants.
- **Cannot prove:** that a statement typed in that session was a good idea. The session is a handout (Chapter 9) and anything the grants allow can be typed.
- **Cost:** a new system that must be available when someone needs it, and the approval path that comes with it (Chapters 7 and 8).
- **Failure mode:** a gateway with a second door beside it, which is a shared password with a better name.
- **Do not use when:** the tasks are repeated and have a fixed shape. They belong on a product path, where nobody needs a session.

Statement-mediating executor

- **Proves:** that this exact statement set, identified by its hash, was approved by a named person, was inspected, ran within a bound, and produced a counted result.

- **Cannot prove:** that the analysis understood the statement. Query analysis reduces mistakes and is not the authorization boundary, which stays with the grants (Chapter 9).
- **Cost:** every task waits for review, nothing is interactive, and someone has to build and keep the runner.
- **Failure mode:** an approval that is always granted within seconds, which is security theater with extra steps.
- **Do not use when:** the work is exploratory. A person who is trying to understand a state needs a read session first, and that is what `human_readonly` is for.

Product path

- **Proves:** that a defined operation ran, for a role in a tool, with the tool's own audit trail.
- **Cannot prove:** anything about the tasks nobody built a feature for. Free-form questions have no path.
- **Cost:** a feature per task, and the slowest change cycle of the five.
- **Failure mode:** the product path that accumulates a "run any SQL" escape hatch because the first urgent case had no feature.
- **Do not use when:** the task has not repeated yet. Chapter 10 describes the signal for turning a repeated fix into a feature, and a task that has happened once is not that signal.

The rightmost pattern is not always the best, and a statement-mediating executor is more than a question about one customer's invoices needs. I would choose per task, with the cheapest pattern that still evidences the pillars the task puts at risk:

```

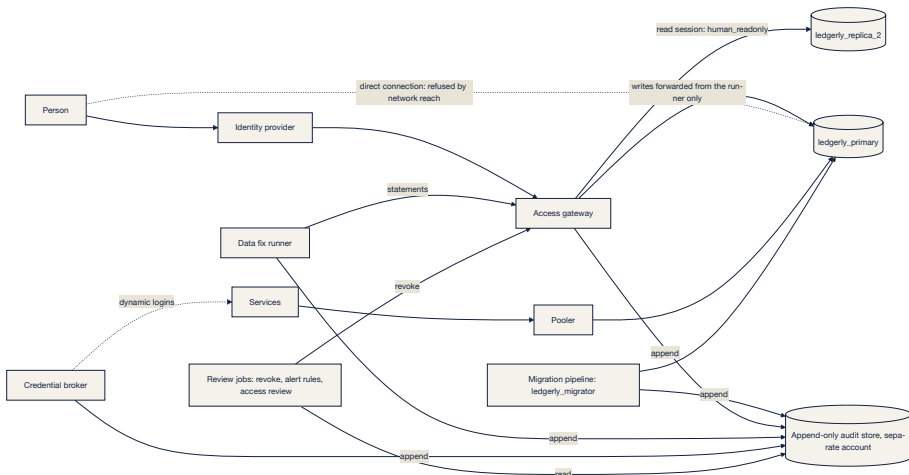
function choosePattern(task):
    // a person who never opens a session cannot leave one open
    if task.repeats and task.hasFixedShape:
        return "product path"
    // a schema change takes the migration pipeline, never the runner
    (Chapter 11)
    if task.changesSchema:
        return "migration pipeline"
    // Restricted columns and owner links are changed by the product,
    not by SQL
    if task.writesRestrictedColumn or task.changesOwnerLink:
        return "product path"
    if task.writesData:
        // reviewed statement set, approved hash, runner executes
        return "statement-mediating executor"
    if task.readsData:
        return "access gateway: human_readonly on ledgerly_replica_2"
    // anything else is an exception with a named owner and an end
    return "exception: owner, reason, ends_on"

```

The last line is the point. A route that does not fit goes into the inventory with an owner and an end date, as Chapter 1's inventory block asks, and an exception without both is a leftover.

The reference architecture

The diagram shows Ledgerly's target state with every control of the book in one picture. Look at three things: that no human arrow reaches a data-base except through the gateway, that every write from a person passes through the runner, and that the audit store sits in an account the data-base administrators cannot write to.



The picture simplifies: the application read replicas of Chapter 6 share one box with their writers, and the masked analytics load, the vendor export and the backups (Chapter 13) are left off, because each is a copy with an owner and an expiry and a drawing with every copy on it would have no readable center.

Each box maps onto the model: the identity provider and broker supply Identity; roles, session logins and the expiry job supply Grant; the network rule behind the dotted arrow and the gateway supply Path; the runner and pipeline supply Execution; the audit store and review jobs supply Record. A box that supplies no part is a candidate for removal, and Robin's question deserves that test.

A maturity ladder with exit tests

The target state is a direction and not a finish line, so the book's steps are grouped into five stages here. The grouping is mine. It follows how much one team can change before the next change becomes easier, and different organizations will reorder it; the one ordering I would defend is that the inventory comes first, because without it nobody knows what each later step has to cover. Record-keeping is also best started early in simple form, since every later stage is easier to judge with evidence.

A stage is finished when its exit test passes, and "we installed it" is not a test. Each test is an observation someone can repeat.

Stage	What it adds	Exit test	Why teams stall here
1. Know the doors	Inventory of principals, doors, copies (Chapters 1, 2, 13)	One document lists every door with an owner and an end, and a new engineer can answer "who can reach the primary?" from it alone	The inventory is treated as a project, so it is never finished and never trusted
2. Name every principal	Own login and role per principal, dynamic credentials, per-role budgets (Chapters 3, 4, 5)	A day of the database log contains no shared login, and no service connects as the owner or the admin	Cutover looks like an outage risk for each service, so the shared password stays "for now"

Stage	What it adds	Exit test	Why teams stall here
3. One path, bounded	Gateway, replicas, JIT and break-glass, classification and masking (Chapters 6, 7, 8, 12)	A direct connection to the primary from a laptop fails, a JIT grant disappears at <code>expires_at</code> without anyone acting, and a read session returns no Restricted column	Fear of the gateway as a single point of failure, and approvers who cannot answer at night
4. Mediated and written	Execution policy, runner, reviewed fixes, pipeline-only migrations (Chapters 9, 10, 11)	A write from a person's own client is refused, and the runner refuses a statement set whose hash differs from <code>approved_hash</code>	The review becomes the bottleneck, so exceptions pile up faster than product features
5. Provable and rehearsed	Hash-chained audit store, alert rules, access review, game days (Chapters 14, 15)	An unannounced statement picked from yesterday can be traced to a person, a request and a result without asking anyone; <code>verifyChain</code> passes; a game day ends with revoked access confirmed from the record	The log exists and nobody reads it, and the game day is the first thing cut when a release is late

A stage can fail its test after it was declared finished, because configuration drifts; a test rerun on a schedule shows the drift while it is small.

Note: A small team with one database and four engineers might reasonably stop at stage three and write down what would make it continue. The discipline is the written reason, not the height.

The checklist

The checklist is grouped by pillar, and every line is an observation. The convention for using it is the same as for the exit tests: a "no" is a finding with an owner and a date, not a failure to be explained away.

```

function runChecklist(items):
    findings = []
    for each item in items:
        answer = observe(item.check)    // read a system, a log or a
        table, never ask what is intended
        if answer != "yes":
            findings.add({ item: item.id, owner: nominate(item), due:
            dateByWhichToDecide })
        // a finding may end in "accepted, with reason, until a date"; it
        may not end in silence
    return findings

```

Authorization

- ☐ Every table is owned by `ledgerly_owner`, and no person or service holds its login.
- ☐ Each service connects with a login that is a member of its own `app_*` role, and no application role holds DDL.
- ☐ No standing write access exists for any person; `human_readonly` is the default and `human_jit_write` is a ceiling and not a granted role.
- ☐ Every JIT and break-glass grant has an `expires_at`, and the revoke job and its independent overdue check both ran recently.
- ☐ A direct connection to the primary from a laptop is refused, and the replicas reject the admin login.
- ☐ Restricted columns appear in no JIT request, no read session and no analytics copy, and the classification catalog lists every column that exists.
- ☐ Every backup, dump, export and staging copy has a named owner and an expiry.

Execution

- ☐ Statements from people and from automation reach the data through the execution policy, and the deny / gate / allow matrix is in version control with an owner.
- ☐ Write sessions run with auto-commit off and a row cap, and guarded settings cannot be changed from inside a session.
- ☐ A JIT write names a reviewed fix, and the runner refuses any statement set whose hash differs from `approved_hash`.

- ☐ Owner-link changes and writes to Restricted columns go to the product path in every lane.
- ☐ Schema changes run as `ledgerly_migrator` from the pipeline, and no person holds that login.
- ☐ The export step runs with a narrow role, and its file has a named owner.

Accountability

- ☐ Each login maps to exactly one person or one workload instance, and the database log shows it.
- ☐ Every write and every elevated session record carries the request, the approver, the fingerprint and the rows returned, and the literals are redacted.
- ☐ The audit store sits in an account the database administrators and `ledgerly_access_admin` cannot write to, and `verifyChain` passes.
- ☐ A sequence gap, a gateway session without a database session, or the reverse, raises a finding.
- ☐ `evaluateAlertRules` and `runAccessReview` ran in the last cycle, and the revokes they proposed were applied or accepted with a reason.
- ☐ `containCompromise` has been rehearsed in a game day, and the blast-radius query was answered from the record.

Culture: the safe path has to be the easy path

Every control in the book is a cost on the person using it, and that person decides, on the worst night of the week, whether to pay it. A slow or often-down control is bypassed because the design made the safe route the hard one, and an incident does not wait. The useful response is to treat the bypass as a defect in the design.

I would put one rule at the centre of the culture, and make it measurable:

```
function reviewBypass(signal):
    // signal: someone used a second door, a shared login, or an
    // unplanned break-glass
    cause = ask("what made the intended path unusable at that
    moment?")
    if cause in ["slow approval", "gateway unavailable", "missing
    product feature", "grant too narrow"]:
        // a design defect: repair the path, and note the date by
        // which it should be easy
        openFinding(cause, owner: "platform", due: nextReview)
    else if cause == "unknown":
        // an unexplained bypass stays open until it is understood
        keepOpen(signal)
    // nothing here assigns the signal to a person
    return cause
```

Two habits support the rule. Count how often the break-glass role is declared, and ask each time whether a JIT path or a product feature should have served instead; a rising count says the normal path is too slow, and a count of zero for a year may mean the role works or that nobody believes it will be allowed, which a game day settles. And give approvers something they can answer at night: an approval that takes hours teaches requesters to ask for more than they need, and one that is always granted within seconds teaches nobody to read it (Chapter 8 covers the approver's checklist).

Trade-offs and costs

A closing chapter that only recommends would repeat the mistake the book warns about, so here is what the whole design costs. The system in the reference architecture has many parts, and every one of them must be available at the moment someone needs it. The gateway, the runner, the pipeline, the broker, the audit store and the review jobs each have an owner, an on-call path and a failure mode. A small platform team can find that it spends its time keeping the controls running, and that is a real cost to set against the exposure it removes.

The checklist and the ladder have costs of their own. A checklist read quarterly becomes a ritual unless findings are closed, and a ladder turns into a status report when teams optimize for the stage name; the exit test

is the antidote, because it is cheap to rerun and hard to fake. A score of "strong" says what a pattern is built to prove, not that a given deployment of it does.

When not to use this chapter's apparatus: a single-engineer system with one database and no vendor needs an owner login nobody uses, a read-only role, a backup that restores, and a one-page inventory. The book is for the point at which the number of principals or the cost of an error outgrows that, and I do not know where that point is for your company; it depends on how many people can reach the data, how much of it is Restricted, and how long a mistake stays invisible.

Common mistakes

- **Treating the checklist as a certificate.** It is tempting to finish a pass and file the result. The checklist shows where evidence is missing on the day it was read. Rerun it on a schedule, and keep the findings open until each one is resolved or accepted with a reason and a date.
- **Building the rightmost pattern first.** A statement-mediating executor is attractive because it looks like the most control. Without the inventory and distinct principals beneath it, it mediates the statements of a shared login. Climb the ladder in order, or write down why a different order suits you.
- **Adding a second door for emergencies.** It looks like resilience and rebuilds the shared password. If a team needs one while the gateway matures, it needs an owner, an alert on use and a retirement condition (Chapter 8).
- **Reading a bypass as a conduct problem.** When someone routes around a control, the design made that cheaper than the intended path. Ask what the intended path cost at that moment, and change that.

Key takeaways

- Production database access is an authorization, execution and accountability problem, and a password touches only the first of these, and only at the door.

- Chapter 4's credential work is necessary because identity makes the record possible, and it is not sufficient because it says nothing about what a principal may do or what a statement does.
- Choose among direct access, a bastion, an access gateway, a state-mediating executor and a product path per task, with the cheapest pattern that still evidences the pillars the task puts at risk.
- A maturity stage is finished when its exit test passes, and a test someone can rerun is the only form of "done" that survives drift.
- Every line of the checklist is an observation of a system, a log or a table; a "no" is a finding with an owner and a date, never a verdict on a person.
- The safe path has to be the easy path, and every bypass is information about where the design made the intended route costly.
- Controls reduce blast radius and make behavior visible, and they cost availability, attention and waiting, so name those costs before adopting them and again when they are paid.

Alex answers Robin's question the next morning with a test and not a defense: a piece stays if it supplies a part of the model that nothing else supplies and its exit test passes. The questionnaire will come again, from another customer or another auditor, and the same two lines will be on it. What the book can leave you with is a system that answers the first line from a document, answers the second from a record, and lets the person on call do the right thing without a heroic effort.

Appendix

This appendix is reference material for the three pillars the book argues from: authorization, execution and accountability. It holds six parts: a glossary, a quick reference to the patterns scored in Chapter 16, the deny / gate / allow matrix of Chapter 9, a crosswalk from auditor questions to artifacts, and two templates, one for the access review of Chapter 14 and one for the post-incident review of Chapter 15.

Nothing here is new. Where a table repeats one from a chapter, the cells are the chapter's own, and if the two differ, the chapter is the source. Your legal and compliance people own the interpretation of any regime; the crosswalk lists engineering artifacts and claims no more.

A. Glossary

Terms are listed alphabetically, with the chapter that builds each. This is the book's final reference for its vocabulary: each entry is the short form, and the chapter named holds the full account.

Term	Meaning
access gateway	An identity-aware intermediary that authenticates a human, issues a short-lived credential and records the session. Not a bastion, and not a database proxy (Chapter 7).
access review	A scheduled check that each principal's access is still needed. It compares what is granted with what has been used, and a named reviewer decides (Chapter 14).
access_audit_record	The entries written for each statement, one of kind intent and one of kind outcome, holding the principal, login, request, statement <code>fingerprint</code> , touched tables and columns, and, on the outcome, row counts and result. It is the Record part of the model (Chapter 14).
ad-hoc query	A query a person writes and runs once, outside any application (Chapter 10).
allowlist, denylist	A list of what is permitted, and a list of what is refused. Never "whitelist" or "blacklist" (Chapter 7).

Term	Meaning
analytics copy	The <code>ledgerly_analytics</code> store. It is a copy of production data, not a replica (Chapter 6).
approved hash	The <code>approved_hash</code> of an access request: the hash of the approved state-ment set, taken from the merge record of the reviewed fix or the ap-proved request record, never from the runner's input (Chapters 9, 10).
audit log	The record of who did what to which data, kept where the actor cannot change it. The <code>audit_events</code> table is the application's business record, not the audit log (Chapter 14).
automation principal	A principal that is not a person: a job, a runbook, a ChatOps command, an AI agent. It has a workload identity and a read-only default, and the record names both it and the person who triggered it (Chapter 9).
bastion	A hardened host that is the single network path to a protected system. It forwards traffic and does not understand the database protocol (Chapter 7).
blast radius	How much can be affected if one principal, credential or action goes wrong. Always stated about a specific thing (Chapters 1, 2).
break-glass	Emergency access with heavy logging and mandatory review. The role is <code>human_breakglass</code> , and the request is self-declared (Chapter 8).
ceiling	The most a JIT request may ask for, defined by <code>human_jit_write</code> . Never a Restricted column (Chapters 3, 8).
column-level grant	A grant of a privilege on named columns of a table. Where the engine lacks it, Restricted columns live in separate tables outside the ceiling, and JIT grants drop to table level (Chapters 3, 8).
containment	Stopping one principal's access in a fixed order: revoke grants, close ses-sions, rotate credentials, scope the damage, record and notify (Chapter 15).
credential	What a principal presents to authenticate: a password, a certificate, a to-ken. A credential is one kind of secret (Chapter 4).
data classes	The four named classes of data: Restricted (exposure is hard to undo, such as <code>iban</code> and <code>token_value</code>), Confidential (personal or commercial data about customers and contacts), Internal (operational data without personal content), Public (safe to publish). Capitalized when used as class names (Chapters 2, 12).
data classification	Sorting data into the named classes that decide handling rules (Chapter 12).
data copy	Any place production data has been duplicated: replicas, the analytics copy, backups, dumps, exports, staging data (Chapters 6, 13).
data fix	A change to production data made by a person outside the normal appli-cation path, to repair a state the application cannot repair (Chapter 10).

Term	Meaning
database proxy	A protocol-aware intermediary that pools, routes or filters database connections. It is called a pooler when the topic is pooling only (Chapters 5, 7).
deny / gate / allow	The execution policy's answers, always in this order. Deny never runs through this path. Gate runs only after extra approval, or only through the product path. Allow runs within the session's grants (Chapter 9).
dynamic credential	A short-lived credential generated on request for one principal, so two requests never get the same value (Chapter 4).
engine-independent	The book's word for notation and advice that belongs to no database engine (Preface).
execution	How a statement reaches the data: who or what submits it, whether anything inspects it before it runs, what bounds its effect, and how it is committed (Chapters 1, 9).
four-eyes	Approval in which a second person reviews before a risky action runs. It says who approves, not what is approved (Chapter 8).
grant	Giving a privilege to a role, a role to a login or another role, or, for JIT only, a privilege on named columns directly to a session login. Also the resulting record (Chapter 3).
guarded settings	Session settings a principal cannot change from inside a session: statement timeout, row cap, auto-commit mode, read-only default, role. Called non-overridable only where a mechanism enforces it (Chapters 5, 9).
hash chain	A record format in which each entry carries the hash of the one before it, so a changed or removed entry breaks <code>verifyChain</code> . It gives evidence of tampering and does not prevent it (Chapter 14).
identity	The verified claim that a principal is who it says it is. A role is not an identity (Chapter 1).
just-in-time (JIT) access	Access granted for a specific task and a limited time (Chapter 8).
least privilege	Granting only what a task needs, for as long as it needs it, with the cost of doing so counted (Chapters 1, 3).
login	The database-level account a principal connects with (Chapters 1, 3).
masking	Replacing or obscuring values so the real value is not visible while the shape stays usable. Static masking changes a copy, dynamic masking changes what a query returns (Chapter 12).
mediated execution	A mode in which the principal never holds a connection: it submits a statement, the gateway or a runner analyzes it, and the runner executes it (Chapter 9).

Term	Meaning
migration	A versioned schema change run by <code>ledgerly_migrator</code> . "Schema change" is the general idea, "migration" the artifact (Chapter 11).
minimization	Not collecting, copying or keeping data that nobody needs (Chapter 12).
owner-link change	An <code>UPDATE</code> of a column that links a row to its owning customer, such as <code>customer_id</code> on <code>bank_accounts</code> . Gated and routed to the product path (Chapter 9).
PII	Personally identifiable information. "Sensitive data" is the umbrella term for Restricted and Confidential data (Chapter 2).
primary, replica	The writable database and its read copies. The analytics copy is not a replica (Chapter 6).
principal	Anything that authenticates: a person, a service, a job, a vendor integration (Chapter 1).
privilege	A specific thing a grant allows, such as <code>SELECT</code> on a table (Chapter 1).
product path	A defined operation in a tool, such as <code>admin-console</code> asking <code>payments-service</code> to change a Restricted column, with no SQL for the person (Chapters 9, 10).
query analysis	Inspecting a statement before it runs: its kind, the tables and columns it touches, whether a write has a bounded row guard, and the expected row count. It reduces mistakes and is not the authorization boundary (Chapter 9).
record	The part of the model that asks what evidence remains and whether it ties the statement to the person, the request and the result (Chapter 14).
redaction	Removing a value entirely from an output or a log (Chapter 12).
revocation, rotation	Revocation makes a credential or grant stop working now. Rotation replaces a credential on a schedule or on suspicion, and does not close sessions that are already open (Chapters 4, 15).
role	A named bundle of privileges in the database. A role is never a person (Chapter 3).
runbook	A written, followed procedure for a repeatable task (Chapter 10).
runner	The reviewed data fix runner. It executes a statement set only if its hash equals <code>approved_hash</code> , and the gateway forwards JIT writes only from it (Chapter 10).
secret	Any value that grants access if disclosed (Chapter 4).
security theater	A control that adds friction without reducing exposure (Chapters 2, 8).

Term	Meaning
session	One authenticated connection or gateway-brokered interaction by a principal (Chapters 5, 7).
session handout	A mode in which the gateway gives a principal a session login and a connection, and the principal's own client sends statements. The grants on the login bound everything (Chapter 9).
short-lived credential	A credential that expires on its own within a short, defined time (Chapter 4).
side door	A data copy or export path that bypasses the access controls of the primary (Chapters 6, 13).
standing access	Access that exists continuously whether or not it is being used (Chapter 1).
statement-bound approval	An approval attached to the exact statement set that will run, identified by <code>approved_hash</code> , rather than only to a time window or a privilege set (Chapters 8, 9, 10).
static credential	A credential that stays the same until someone changes it (Chapter 4).
tokenization	Replacing a value with a surrogate that only a controlled service can map back. It is neither encryption nor masking (Chapter 12).
workload identity	An identity the platform gives to a running workload so it can authenticate without a stored password (Chapter 4).

Several of these pairs get confused, and a short rule settles each one before an argument starts.

If the question is	Use	Not
A host that forwards network traffic, with no knowledge of statements	bastion	access gateway
An intermediary that issues per-person session logins and records sessions	access gateway	database proxy
Pooling or routing connections, nothing about people	pooler, or database proxy	access gateway
The person's own client sends statements over a connection the gateway gave them	session handout	mediated execution
The person submits a statement and never holds a connection	mediated execution	session handout
Stopping a credential from working now	revocation	rotation

If the question is	Use	Not
Who approves a risky action	four-eyes	statement-bound approval
What exactly was approved, by hash	statement-bound approval	four-eyes
The application's business record	<code>audit_events</code>	audit log

B. Pattern Quick Reference

Chapter 16 scores five patterns against the three pillars. The scores are the book's judgment and not a measurement: "strong" means the pattern is built to prove the pillar, "partial" that it proves some of it or only with something added, "none" that it is silent. The cost column is the one Chapter 16 gives, shortened.

Pattern	What it solves	Cost	Authorization	Execution	Accountability	Chapters
Direct access (client to database)	A login exists and its grants were what they were	None to build; every cost arrives later, as a week of digging	Partial: grants only, reach is whatever the network allows	None	Partial at best: per-person logins if someone provisions them, otherwise none	1
Bastion	One controlled host, one place to turn the network path off	A host to patch and keep available	Partial: narrows the network path	None: it forwards, it cannot tell a <code>SELECT</code> from a <code>DROP</code>	Partial: who logged in to the host, not what ran	7
Session proxy or access gateway	A named person held a specific session login, between two time-stamps, under specific grants	A new system that must be available when someone needs it, and the approval path with it	Strong: per-session logins, expiry, per-request grants	Partial: bounded by grants and guarded settings, statements are typed freely	Strong on who and when, partial on what	7, 8

Pattern	What it solves	Cost	Authorization	Execution	Accountability	Chapters
Statement-mediating executor	This exact statement set, by hash, was approved, inspected, bounded and counted	Every task waits for review, nothing is interactive, someone builds and keeps the runner	Strong	Strong: inspected before it runs, bound to an approved hash	Strong: statement, approver and result in one record	9, 10
Product path (no SQL)	A defined operation ran, for a role in a tool	A feature per task, and the slowest change cycle of the five	Strong: the operation is the permission	Strong: the operation is fixed	Partial: the person is in the tool's audit trail, the database log shows the service login	9, 10

Choosing among them is a per-task decision, and the rule is the one Chapter 16 gives: the cheapest pattern that still evidences the pillars the task puts at risk.

```
function choosePattern(task):
    // a person who never opens a session cannot leave one open
    if task.repeats and task.hasFixedShape:
        return "product path"
    // a schema change takes the migration pipeline, never the runner
    (Chapter 11)
    if task.changesSchema:
        return "migration pipeline"
    // Restricted columns and owner links are changed by the product,
    not by SQL
    if task.writesRestrictedColumn or task.changesOwnerLink:
        return "product path"
    if task.writesData:
        // reviewed statement set, approved hash, runner executes
        return "statement-mediating executor"
    if task.readsData:
        return "access gateway: human_readonly on ledgerly_replica_2"
    // anything else is an exception with a named owner and an end
    return "exception: owner, reason, ends_on"
```

C. Dangerous-Operations Matrix

The matrix below is the execution policy of Chapter 9, repeated with its cells unchanged. Rows are the statement classes the classifier produces. Columns are the lanes a statement can arrive in. Each cell has one of three

answers: deny (never runs through this path), gate (needs approval it does not yet have, or belongs on a different route), allow (runs within the loop's limits, and where a cell says "bound", only inside the bound from the source named).

Statement class	Human read session	JIT write	Break-glass	Data-fix runner	Migration
Read, within the row cap	Allow	Allow	Allow	Allow	Allow
Bounded write, within the approved count	Deny	Allow, bound to the fix (runner only)	Allow, bound to a row cap (note required)	Allow, bound to the approved hash and count	Allow, bound to the back-fill rule
Unbounded write	Deny	Deny	Deny	Deny	Gate: named approver
Owner-link change	Deny	Gate: product path	Gate: product path	Gate: product path	Gate: named approver
Write to a Restricted column	Deny	Gate: product path	Gate: product path	Gate: product path	Gate: named approver
Export	Gate: by tier	Deny	Deny	Deny	Gate: named approver
Schema change (DDL)	Deny	Deny	Deny	Deny	Allow: reviewed lane
Privilege change	Deny	Deny	Deny	Deny	Gate: platform review
Write to ledger_entries or audit_events	Deny	Deny	Deny	Deny	Gate: platform review

Three reading rules save most lookups.

- **A write in a person's hands is denied or bound.** The human read session has no write path, JIT writes come only from the runner, and break-glass is the one lane where a person writes directly, under a row cap and with a note.

- **A gate names where the statement goes next.** "Product path" means `admin-console` asks `payments-service`, a second person approves and the change is recorded in `audit_events`. "Named approver" and "platform review" mean the migration lane flags the statement for a person who has to say yes.
- **The matrix is not the authorization boundary.** Query analysis reduces mistakes. The grants on the login stay the boundary, and a statement the classifier misreads is still limited by them (Chapter 9).

The decision for one statement, written as a rule:

```
function answerFor(statementClass, lane):
    // the matrix above is the only source; no lane has its own rule
    set
        cell = matrix[statementClass][lane]
        if cell starts with "Deny":
            refuse and record the refusal
        else if cell starts with "Gate":
            // the text after the colon names the route or the approver
            hold the statement; route it as the cell says; do not run it
    as sent
        else:
            // "bound" cells compare the real count with the bound before
    COMMIT
        run within the loop's limits
```

D. Compliance Crosswalk

Data-protection and payment-card regimes tend to ask a family of questions about access. An auditor working under one of these regimes will usually want to see that a control exists, that it ran, and that its results were looked at. This crosswalk lists the question types, the control that answers each, the artifact it leaves and the chapter that builds it. It is the table of Chapter 14, repeated, and it makes no claim that any control satisfies any obligation.

Question	Control	Artifact	Chapter
Who can reach production data, and by which path?	Gateway as the only door, role schema as code, drift check	Network rule list, role definitions, drift report	3, 7

Question	Control	Artifact	Chapter
Who ran a given statement, with which grants and approval?	<code>access_audit_record</code> with request id and approver	The record and the <code>verifyChain</code> report	14
How long does elevated access last, and who approved it?	JIT request with approver and <code>expires_at</code> , revoke job	Request records, revoke log, overdue-grant check	8
How is sensitive data identified and limited?	Classification in the schema, masking, withheld columns	The catalog, the masked-load verification	12
Where are copies of production data, and who owns each?	Register with owner and end date, sweeper	Register, sweeper log	6, 13
Who changes the schema, and who reviewed it?	Migration pipeline, protected branch, external start entry	Migration records matched to commits and audit entries	11
Is access still needed?	Access review with named reviewers	Completed review with decisions	14
What happens in an emergency?	Break-glass with alert and mandatory review	Review records, alert history	8, 15
Who administers access itself?	<code>ledgerly_access_admin</code> logs as separate producers	Their streams and the catalog-versus-record comparison	3, 14

A control that exists but never ran, or ran and was never looked at, answers only a third of the question. The check below is the one to run before handing an artifact to anyone who asked for it.

```
function evidenceStatus(row):
    // three things an auditor tends to want, from Chapter 14
    if row.artifact is null:
        return "control not evidenced: build or find the artifact"
    if row.artifact.lastProducedAt is null:
        return "control exists, no evidence it ran"
    if row.artifact.lastReviewedBy is null:
        return "control ran, results not looked at"
    return "evidenced: attach artifact and its review"
```

E. Access Review Template

This is the template behind Chapter 14's `runAccessReview`. Copy it for each review cycle. The review proposes and a named reviewer decides; nobody reviews their own access, and the period is longer than the longest cycle of the workload being reviewed (a one-month review would propose revoking a grant that a quarterly report uses in two months out of three).

Header

Review cycle:

<period start> to <period end>

Reviewer(s):

<names; none is the owner of an item they review>

Record queried on:

<date>, verifyChain result: <passed | failed: details>

Period chosen and why:

<longest workload cycle covered?>

Part 1: declared grants against use

Grant (role, privilege, table) or login	Owner	Last use in the record	Finding	Decision	Decided by	Done on
<role>, <privilege>, <table>	<named principal>	<date or none>	unused / no owner / use narrower than grant / fine	keep / narrow to <privilege> / revoke / assign owner	<reviewer>	<date>

Match use by privilege and not only by table. A role that selects every night and has never updated is a candidate for narrowing from `S`, `U` to `S`. Include the identity provider group to role mappings and the logins of `ledgerly_access_admin`.

Part 2: things that are not grants

Item	Question	Answer	Owner	Decision
Register of data copies (Chapter 13)	Does each copy still have an owner and a future end date?	<yes / no, which>	<name>	<keep / end by date>
Who may approve JIT requests	Is each approver still in a role that justifies it?	<yes / no, which>	<name>	<keep / remove>

Item	Question	Answer	Owner	Decision
Who can read the audit store	Does each reader still need it?	<yes / no, which>	<name>	<keep / remove>
Automation principals (jobs, runbooks, AI agents)	Does each still run, with a workload identity and a read-only default?	<yes / no, which>	<name>	<keep / retire>

Part 3: break-glass uses in the period

Request id	Used by	Incident or reason	Reviewed by	Review date	Finding about the system
<id>	<principal>	<reason>	<other than user>	<date or MISSING>	<for example: missing path>

A row with no review date is itself a finding.

Sign-off rules

```
function closeAccessReview(review):
  for each row in review.rows:
    // a row without a decision is not a review, it is a form
    if row.decision is null or row.decidedBy is null:
      raise ReviewIncomplete
    if row.decidedBy == row.owner:
      raise ReviewerIsOwner
    // decisions become tracked work with an owner and an end date
    for each row in review.rows where row.decision != "keep":
      createChange(row, owner: row.owner, endsOn:
review.nextCycleStart)
markClosed(review.id)
```

F. Post-Incident Review Template

This template follows Chapter 15. It is blameless: each question is about the system that allowed an action or made it invisible, and none asks who should have known better. Fill it in from the record, not from memory. Where the record cannot answer a question, write that, because the gap is a finding. Times are the ones the record shows; do not estimate.

1. Summary

Incident id: <id carried in every request reason>
 Opened / closed: <timestamps from the incident record>
 Trigger: <alert, report or declared exposure>
 Principal(s) in scope: <logins, services, automation principals>
 Pillars touched: <authorization / execution / accountability>

2. Containment, in order

Step	Done at	Outcome	Notes on what got in the way
Revoke grants	<timestamp>	done / failed	
Close sessions	<timestamp>	done / failed	
Rotate credentials	<timestamp>	done / failed	
Scope the damage from the audit log	<timestamp>	done / failed	
Record and notify	<timestamp>	done / failed	

Questions: Did every step run even if an earlier one failed? Were any steps left unfinished, and were they tracked? Was the log reachable, and if not, did entries queue without delaying removal of access?

3. Blast radius

Query run: <everything the principal touched since T, Restricted rows per table>
 Tables touched: <list, with class>
 Restricted columns: <touched or not, rows returned per table>
 Agreement check: <does the scope agree with the register of copies and the gateway session list?>

4. Incident access lifecycle

Request id	Kind	Requested	Granted	First use	Revoked	Reviewed by
<id>	jit / breakglass	<time>	<time>	<time or none>	<time>	<other than user, or MISSING>

Questions: Did any grant stay past `expires_at`? Was any access granted that no request explains? Did `closeIncident` find open access, and was it closed before the incident was?

5. Writes during the incident

Write	Route (runner with expedited review, product path, raw break-glass)	Reviewer who was not the author	Approved hash matched
<description>	<route>	<name>	yes / no

Question: Was raw break-glass used for a write where the runner would have served, and why?

6. Deferred cases, if they occurred

- **Gateway unavailable:** how long human access was stopped, whether services kept running, and whether the restore runbook worked as written.
- **Restore during the incident:** whether the restored copy was registered with an owner and an end date, and ended.
- **Slow approvals at night:** which request waited, for whom, and what the rule for an unanswered request did.

7. Findings and follow-up

Finding about the system	Part (Identity, Grant, Path, Execution, Record)	Change	Owner	Ends on
<for example: a path that should have existed>	<part>	<change>	<name>	<date>

Close with one check: is every finding a change to a design with an owner and an end date, and not a reminder to be more careful?

THE UNRELIABLE SYSTEMS TRILOGY

ABOUT THE BOOK

Production database access is not a credential-distribution problem. It is an authorization, execution and accountability problem.

A better password vault only changes how a secret reaches a person. It does not decide what that person may do, what actually runs against the data, or who answers for it afterwards. This book follows Ledgerly from a shared admin password to an access model designed on purpose: least-privilege identities, just-in-time and break-glass access, mediated execution for dangerous operations, and an audit trail that can say who did what, and why. The patterns are engine-independent, and each one is given with the cost it carries.

ABOUT THE AUTHOR

Muhammet Şafak has been building software since 2008. Over many years Muhammet has held active roles in the healthcare, finance and telecommunications domains, and argues that development should focus on the problem, not the technology. Since 2024 Muhammet has been designing and building high-traffic systems that scale end to end.

Muhammet mentors small and mid-sized software development teams, leads open-source projects, and writes decisions down with their reasons and their cost.

Reliable systems are not built from reliable parts. They are built around the parts that fail.

I

ACCESS

II

EVENTS

III

AGENTS