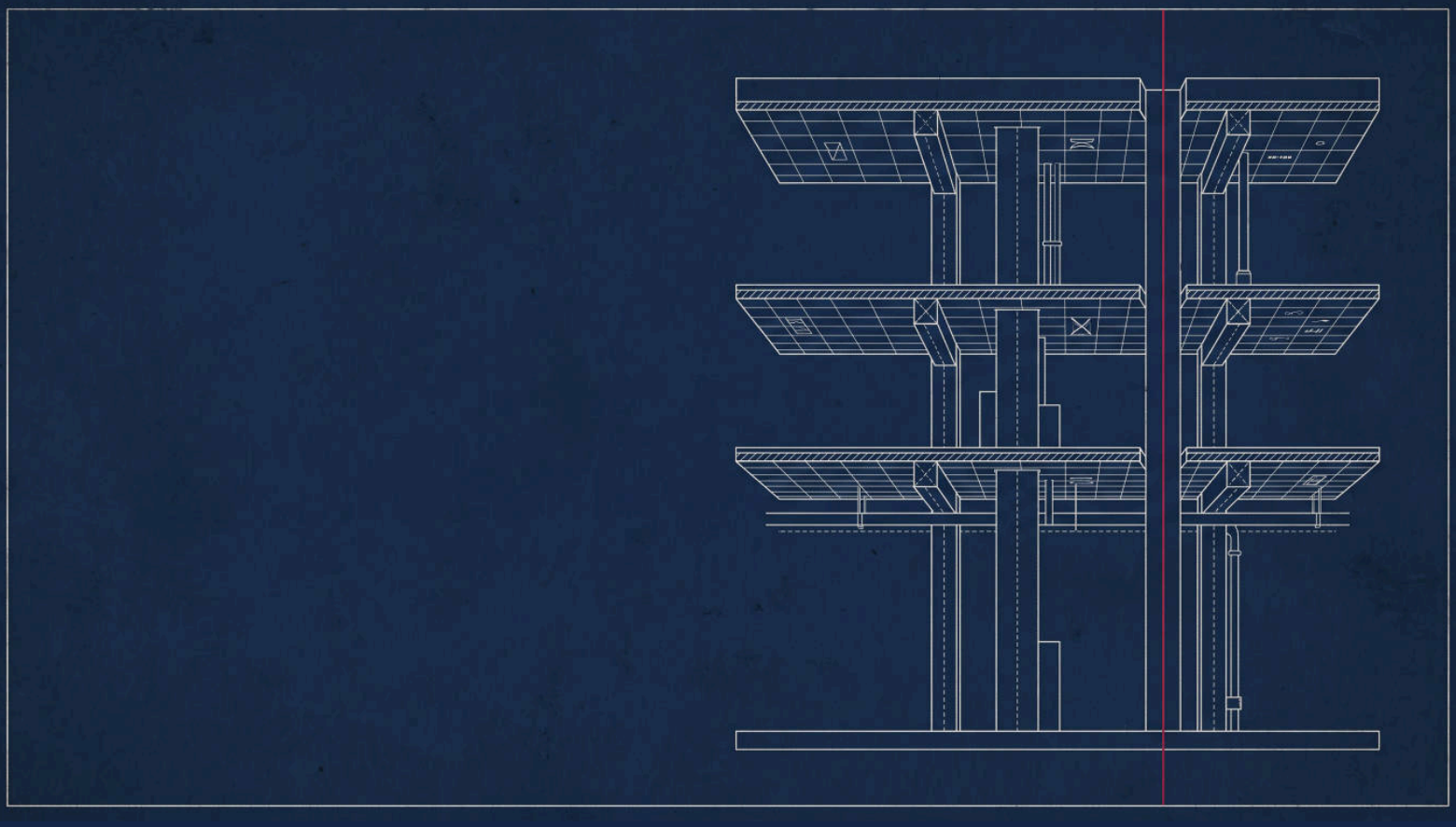

CHAPTER ONE

Environment and Ground Rules

SECTIONS 1.1 TO 1.7

Five decisions made in the first hour determine how expensive the next two years are. This chapter makes all five and builds the repository every later chapter extends.

CHAPTER ONE

Contents

Five decisions made in the first hour of a project determine how expensive the next two years are. Each is nearly free to make correctly now and costs a quarter to retrofit. This chapter makes all five, builds the repository that every later chapter extends, and establishes the frame the rest of the book reasons with.

1.1 How This Book Works and What You Will Build

What an LLM solution architect actually does

The title is new enough that it is worth defining by contrast, because three adjacent roles are frequently confused with it and the confusion produces bad hiring, bad team structure and bad systems.

A **machine learning engineer** owns the model: training, fine tuning, evaluation of model capability, inference optimisation. Their central question is "can this model do the task."

A **backend engineer** owns the service: correctness, throughput, uptime, data integrity. Their central question is "does this system behave as specified under load."

An **AI solution architect** owns neither, and owns the space between them. The central question is different in kind: *given a component that is non deterministic, priced per use, unbounded in latency, adversarially reachable through natural language, and liable to change without notice, what structure makes a reliable product possible?*

That question decomposes into four repeated judgements, and this book is organised around teaching them:

Judgement	What it decides	Where it is taught
What goes in the context	The single largest determinant of answer quality. Retrieval, memory, prompt assembly, budget.	Chapters 2, 5 to 7, 11
What the output may cause	The only security control that survives an adversarial input. Tool permissions, approval gates, scoped credentials.	Chapters 9, 12, 15
Which dimension is unbounded	Latency, cost, blast radius, trust, drift. Every architecture in this book bounds one of them.	Everywhere
What is measured	Whether any of the above can be known rather than believed.	Chapter 14, referenced constantly

Notice what is absent from that list: choosing a model, choosing a framework, and writing prompts. Those are real activities and they are not where the leverage is. A team that reasons well about the four judgements above will get a good result from a mediocre model and an unfashionable framework. A team that reasons badly about them will not be rescued by the best model available, and Chapter 7 contains the measurement that demonstrates it.

> THE DIAGNOSTIC THAT DEFINES THE ROLE
>
> You can identify a competent AI architect with one question: *hand them an unfamiliar system and ask which dimension of it is currently unbounded and what traffic level will exercise it.* Someone who can answer that in ten minutes, with a number attached, is doing the job. Someone who answers with a list of components is describing a diagram. This book is, in the end, an extended exercise in learning to answer that question quickly.

The system we build

One system runs through all nineteen chapters. It is deliberately unglamorous, because every architectural pressure in this field appears in it before it appears in anything more exotic.

> **docassist**: a multi tenant document assistant.
>
> - Users upload contracts and technical reports. Documents belong to a tenant and are never visible across tenants.
> - Users ask questions in text or by voice and receive answers *with citations* that resolve to a page and section a human can check.
> - When the documents do not contain the answer, the assistant says so rather than producing a plausible one.
> - The assistant remembers durable preferences across sessions, and the user can inspect and delete what it remembers.
> - The assistant can draft and send a summary email, which requires human approval before it goes.
> - **Non functional:** p95 time to first token under three seconds, cost per successful task under three cents, measured answer accuracy above a stated floor, and survival of a provider outage in a degraded but useful state.

Those five non functional requirements are the whole book. Each one is impossible to satisfy with a naive implementation and each is satisfied by a specific structure introduced in a specific chapter. Keep the list nearby: when a chapter's argument feels abstract, it is usually defending one of these five numbers.

The good, better, best method

Wherever implementation is involved, this book builds the same capability three times. The three are not a ranking of the engineers who wrote them. They are three points on a curve measuring *how many dimensions of the system are bounded*.

Tier	Defining property	Correct when
GOOD	Works, and leaves at least one dimension unbounded: latency, cost, blast radius, or trust. This is what ships first at nearly every company, and it is usually the right first move. The engineering failure is never writing it. The failure is not recording which dimension you accepted and what load will exercise it.	You are still answering "does this work at all", with an audience you can name individually.
BETTER	Bounds the dimension that will hurt first, by introducing a *seam*: a place where behaviour can be observed, intercepted and constrained. Carries a residual weakness that the section names explicitly.	Real users, real money, or more than one service touching the model.
BEST	Makes the expensive properties explicit and tradeable: cost per successful task, quality under drift, degradation under failure. Costs more to build, and each section states the threshold below which building it is a mistake.	The stated promotion trigger has fired. Not before.

Every section that builds something ends with a **promotion trigger**: the measurable condition under which you move up a tier. This matters more than it may appear. Architecture maturity is a response to a measured load, not a virtue to be maximised. Building the best version for a product with forty daily users is the same category of error as building the good version for four hundred thousand, and it is the more common error among engineers who have read books like this one.

The scorecard

Implementations are compared on seven axes, chosen because they are the axes on which AI systems fail, as distinct from the axes on which conventional services fail.

Axis	The question it asks
Tail latency	What does the ninety ninth percentile user experience when the provider is slow, and does that experience consume a resource you cannot replace quickly?
Cost per successful task	Not cost per call. What does one *completed unit of user value* cost, including retries and calls that produced unusable output?
Failure containment	When one dependency degrades, how much of the product goes with it, and is the degraded state something you designed or something you discovered?
Attributability	When quality moves, can you name the change that moved it within minutes, or do you bisect a week of commits and prompt edits?
Correctness under drift	The model behind your endpoint changes without your consent. Does anything in your system notice before your users do?
Adversarial surface	Your parser is a language model and your input is arbitrary natural language. What can a hostile string reach, and what is it permitted to do when it gets there?
Evolvability	How expensive is it to swap the model, the vector store, or the framework? In a field that moves quarterly, this axis compounds faster than any other.

The repository

Everything in this book is implemented in one repository, `docassist`, which grows chapter by chapter rather than restarting. By the end of this chapter it contains twenty four files and does something genuinely useful: it answers a question about supplied sources, with citations, with bounded retries, with a spend circuit that refuses before the call, and with one attributable trace record per model call.

```
git clone <repo> docassist && cd docassist
cp .env.example .env          # add your provider key
make up                       # qdrant, redis, ollama, health-gated
make test                     # unit + architecture tests
make eval                     # score the prompt against fixed cases
make dev                      # the API on :8000
```

You need working Python 3.11, Docker, roughly 30 GiB of disk for model weights from Chapter 4 onward, and API access to one hosted provider. A GPU becomes useful in Chapter 4 and is never required: every local model exercise has a CPU path that is slower and functionally identical.

1.2 Python Environment and Dependency Management

Most books treat this as throat clearing. It is not, and the reason is specific to this domain.

The smallest unit: what a dependency does here

In a conventional project, a dependency is code you call. If it changes incompatibly, your code raises an exception, a test fails, and you find out. The failure is *loud*, and loudness is what makes semantic versioning a workable social contract.

In an AI system, several of your dependencies are not merely code you call. They are **parameters of a statistical process whose output you cannot verify by inspection**. When one of them changes, nothing raises. Your system continues to produce fluent, plausible, well formed output that is slightly worse, and the difference is invisible without measurement you probably have not built yet.

Four concrete instances, each of which has caused a real incident.

Change	What breaks	How it presents
Tokenizer library minor bump	Token counts shift by a few percent. Your cost model and your context budget are both now wrong, and your budget enforcement lets slightly larger prompts through.	Nothing. A small cost drift noticed at month end.
HTTP client default timeout change	A hung socket now holds a worker for sixty seconds instead of failing in three, or the reverse: legitimate long generations start failing.	An unexplained latency shift, or a new error class in the logs.
Framework message assembly change	The string that actually reaches the model changes without your code changing. Roles are folded, whitespace normalised, a system message relocated.	Answer quality moves. No code changed, so nobody looks at dependencies.
Embedding library normalisation default	New vectors are no longer comparable with the two million already in your index. Retrieval quality degrades *gradually* as the proportion of new vectors rises.	Nothing, for months. Then a slow, unattributable decline in answer quality.

The fourth deserves particular attention because it is the purest example of the failure mode this section exists to prevent. There is no exception, no alert, no failed test. There is a system that is slightly worse every week, and by the time anyone notices, the change that caused it is forty commits back and nobody thinks to look at a library upgrade.

GOOD

Pin exactly, commit the lock

```
# pyproject.toml
[project]
name = "docassist"
requires-python = "==3.11.*"     # native wheels and tokenizer behaviour both
dependencies = [                  # vary across minor Python versions
  "fastapi==0.115.5",
  "httpx==0.27.2",
  "pydantic==2.9.2",
  "tiktoken==0.8.0",              # changing this changes your cost model
]
```

Exact pins, not compatible-release ranges, plus a committed lock file that captures the transitive tree. Ranges are unacceptable here precisely because the whole problem is that a *compatible* version can still change behaviour. The social contract of semantic versioning covers API compatibility. It does not cover "the numbers this library produces."

The specific version numbers in this chapter's listings, here and in the compose file of section 1.3, are an illustrative snapshot from the time of writing and will be stale by the time you run them, in the same way that Chapter 3's prices are representative rather than current. The durable content is the mechanism: exact pins, a committed lockfile hash, and a runtime fingerprint work unchanged whatever the current numbers happen to be.

The repository's lock file is `uv.lock`, produced by `uv`: a single fast resolver that emits one cross-platform lockfile, which is the property the fingerprint in the next tier depends on. The choice is not load bearing. Any pinning tool works here, provided it produces a lockfile you can hash.

What this leaves unbounded: pinning stops silent change going forward. It tells you nothing about what was actually running six weeks ago when the answer your customer is complaining about was generated.

BETTER

Make the environment observable from a trace

Emit a fingerprint of everything that determines behaviour, attached to every trace record. This is in the repository at `src/docassist/fingerprint.py`.

```
@lru_cache
def runtime_fingerprint() -> dict[str, object]:
    s = settings()
    return {
        "python": platform.python_version(),
        "code_sha": _git_sha(),
        "lockfile_sha": _file_sha("uv.lock"),
        "prompt_registry_sha": _dir_sha("prompts"),   # section 1.6, prompt seam
        "model_alias": s.model_alias,
        "embedding_model": s.embed_model,
        "embedding_dim": s.embed_dim,
        "index_build_id": s.index_build_id,           # see BEST, below
    }
```

Three details are load bearing. The **lockfile hash** rather than a version list, because the transitive tree is where the surprises live. The **prompt directory hash**, because a prompt edit is a behavioural change with exactly the same properties as a dependency bump. And the **index build id**, because it identifies which materialisation of your corpus the retrieval came from.

The cost is one dictionary in every trace record. The payoff arrives on the day someone asks why quality moved, and the answer is a diff between two fingerprints rather than an investigation.

What this leaves unbounded: observability is not control. You can now see that the embedding model changed. You still have an index containing two incompatible generations of vectors and no way to compare them or roll back.

BEST

Treat stateful dependencies as migrations, not upgrades

The insight that separates this tier: **some dependencies have state materialised elsewhere in your system.** Your vector index is a materialisation of a specific embedding model at a specific version, in the same way a database table is a materialisation of a schema. Changing the model invalidates the index, and it should be handled the way you handle a schema change: with a build identifier, a dual write window, a comparison, and a cutover you can reverse.

```
@dataclass(frozen=True)
class IndexBuild:
    build_id: str            # "emb-v3-1536-20260714"
    embedding_model: str
    dimensions: int
    normalised: bool
    created_at: datetime


class IndexRegistry:
    """Two builds coexist during migration.

    Queries read the ACTIVE build. Ingestion writes to ALL builds. Cutover is
    a config change and rollback is the same change reversed, which is the
    property that makes the migration safe to attempt on a Tuesday.
    """

    def query_build(self) -> IndexBuild:
        return self._active

    def write_builds(self) -> list[IndexBuild]:
        return [b for b in (self._active, self._shadow) if b]

    def promote(self, build_id: str) -> None:
        assert self._shadow and self._shadow.build_id == build_id
        # The gate: a shadow build does not become active on someone's opinion.
        assert self._eval_passed(build_id), "shadow build failed retrieval eval"
        self._active, self._shadow = self._shadow, None

    def _eval_passed(self, build_id: str) -> bool:
        """Runs the retrieval eval set (section 7.10) against the shadow index
        and compares recall against the active one. Chapter 14 formalises the
        thresholds; the structural point is that the comparison exists."""
        return retrieval_eval(build_id).recall_at_10 >= (
            retrieval_eval(self._active.build_id).recall_at_10 - 0.01
        )
```

The dual write window costs storage and a little ingestion latency for as long as it runs. In exchange, an embedding model change becomes a reversible operation with a measured comparison, rather than a one way door you walk through and hope.

PROMOTION TRIGGERS

Good: from the first commit, always. Pinning and a lock file cost nothing.

Better: the first time you cannot explain a behaviour change, which will happen inside the first month. Roughly forty lines.

Best: before your first embedding model or chunking strategy change. Doing it afterwards means a full reindex with no ability to compare or roll back, which in practice means the change never happens and you are stuck on your first guess forever.

EXERCISE 1.2

Take a page of your own domain's text. Count its tokens with `tiktoken` for two different model encodings and compute tokens per thousand characters. General English prose sits near 250. Anything above 320 means your corpus is expensive and any margin assumption built on English prose averages is wrong. Write the number down: section 2.4 uses it, and it will change a pricing conversation later.

1.3 Docker and Docker Compose Essentials for AI Workloads

By Chapter 11 this system depends on a vector database, Redis, Postgres, object storage, a local model runtime and a checkpoint store. All of them should exist in one compose file from week one, because the alternative is a codebase where retrieval works only on the machine of the person who first wrote it.

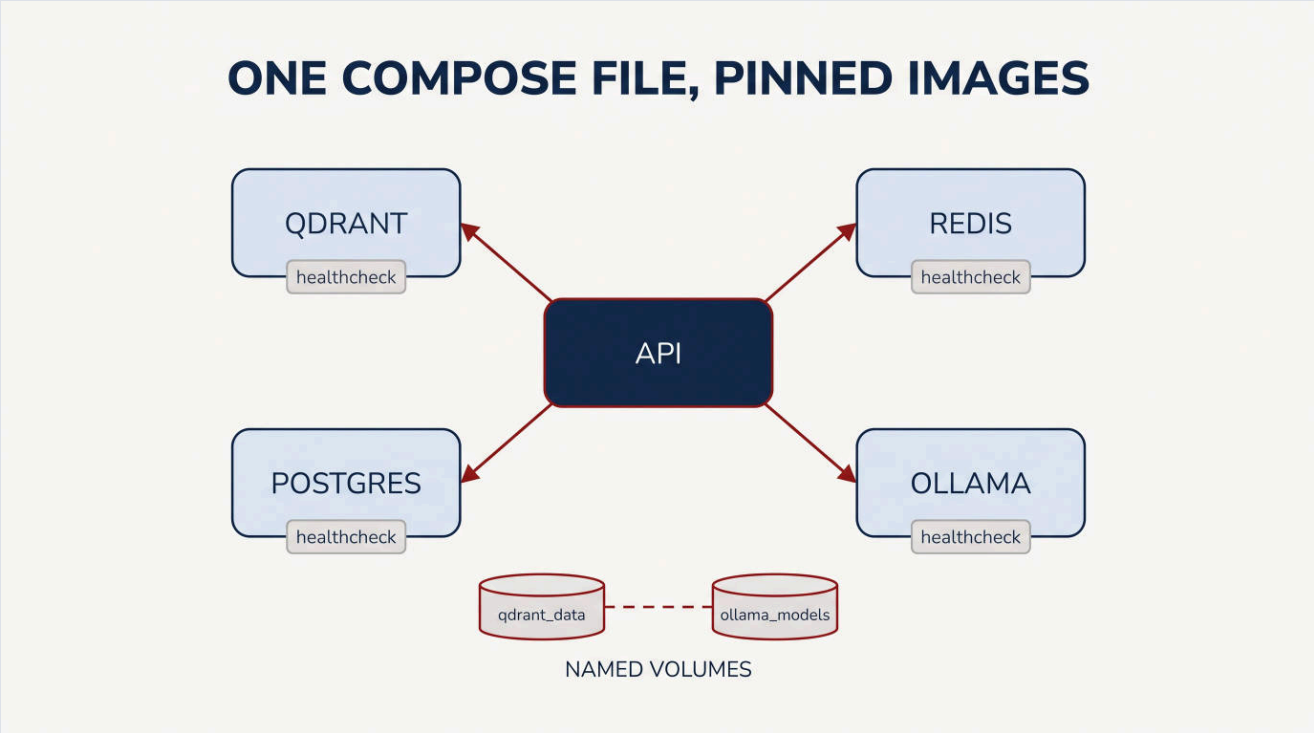

Figure 1.1 One compose file, pinned images, health-gated startup. Weights live in a named volume, never in the image.

The rule that matters more than the file

Local composition must use the same interfaces as production, even when the implementations differ. A file backed vector store behind the same interface as your production index is fine and often sensible. A branch on an environment flag in the middle of your retrieval logic is not, because it guarantees the two environments diverge in ways nobody notices until a customer does. The repository enforces this with a test, which we will look at in section 1.4.

The compose file, decision by decision

```
services:
  qdrant:
    image: qdrant/qdrant:v1.12.1          # (1) pinned, like any dependency
    volumes: ["qdrant_data:/qdrant/storage"]
    deploy: {resources: {limits: {memory: 2G}}}    # (2)
    healthcheck:                                    # (3)
      test: ["CMD-SHELL", "bash -c ':> /dev/tcp/127.0.0.1/6333' || exit 1"]
      interval: 5s
      retries: 12

  ollama:
    image: ollama/ollama:0.5.4
    volumes: ["ollama_models:/root/.ollama"]        # (4) never in the image
    environment:
      OLLAMA_KEEP_ALIVE: "-1"                       # (5) see section 4.1
      OLLAMA_NUM_PARALLEL: "4"
      OLLAMA_MAX_QUEUE: "64"
    deploy:
      resources: {limits: {memory: 12G}}            # (6)
    healthcheck:
      test: ["CMD-SHELL", "ollama list || exit 1"]
      start_period: 120s                            # (7) first pull is slow

  api:
    build: {context: ., target: dev}
    depends_on:
      qdrant: {condition: service_healthy}          # (3) again, and it matters
      redis:  {condition: service_healthy}

volumes: {qdrant_data: {}, ollama_models: {}}
```

(1) Image tags are pinned. A vector database that silently changes its default distance metric or its HNSW parameters between `:latest` pulls is the section 1.2 problem wearing a different hat. Everything that determines behaviour gets a version.

(2) and (6) Memory limits on everything. This is the single most valuable line in the file for a developer's sanity. A local model runtime will consume all available memory and take your database down with it, and the symptom is a laptop that freezes hard rather than a container that restarts. Explicit limits convert a machine wide failure into a single container failure, which is both survivable and diagnosable.

(3) Health checks that gate startup. A vector database takes several seconds to become ready. Without `condition: service_healthy`, your API starts immediately, fails its first requests with connection errors, and you spend twenty minutes debugging a race condition that is not a bug in your code. This is a small thing that costs new team members an afternoon each, forever, if it is missing.

(4) Weights live in a named volume. A five gigabyte model that re-downloads on every `compose down` makes the environment hostile enough that people stop using it. Weights are also never baked into the image, because an image that must be rebuilt to change a model is an image nobody will change.

(5) Idle unloading disabled. Ollama unloads a model after five idle minutes by default, so the next request waits thirty to sixty seconds for a cold load. This is invisible in development, where you are always warm, and it is the most common complaint about self hosted deployments. Section 4.1 covers the full set of consequences.

(7) A generous start period on the model runtime, because the first pull genuinely takes minutes and a health check that fails during it produces a restart loop that never completes the download.

> IMAGE SIZE IS A LATENCY PROBLEM HERE, NOT A TIDINESS ONE
>
> A naive Python image with the machine learning stack installed reaches four to six gigabytes. That makes cold starts slow, which makes autoscaling sluggish, which matters directly in Chapter 16 where you scale worker pools on queue depth and need new capacity in seconds rather than minutes. Use a multi stage build, install only inference dependencies in the runtime stage, and keep weights out of the image. The repository's Dockerfile has `base`, `dev` and `prod` targets for exactly this reason.

> EXERCISE 1.3
>
> Run `make up`, then `docker compose stop qdrant` while the API is running. Observe what `/ready` returns versus `/alive`. Then remove the `condition: service_healthy` lines, run `make down && make up`, and watch the API's first requests fail. That failure mode is what the four extra lines buy, and seeing it once is worth more than reading about it.

1.4 Project Repository Structure and Conventions

Structure is not aesthetics here. It is what determines whether the four seams of section 1.6 have somewhere to live,

and whether they stay there under deadline pressure.

Figure 1.2 The tree, with the four seams marked. Three of the four are single directories; the fourth is a single script.

The tree, and why each directory exists

```
docassist/
  pyproject.toml          exact pins (1.2)
  compose.yaml            stateful dependencies (1.3)
  Dockerfile              base / dev / prod targets
  Makefile                up, dev, test, eval, check
  .env.example            never .env

  prompts/                THE PROMPT SEAM. Versioned YAML, not literals.
    doc_qa/v1.yaml

  src/docassist/
    settings.py           typed config + SecretProvider (1.5)
    fingerprint.py        what was actually running (1.2)
    gateway/              THE MODEL SEAM. The only place a provider exists.
      client.py             retries, deadline, circuit breaker, streaming
      types.py              Completion, CircuitBreaker, SpendCircuit
    prompts/
      registry.py           loads and renders the prompt seam
    obs/
      trace.py            THE TRACE SEAM. One record per model call.
    api/
      main.py             HTTP only. No business logic, no prompts, no SDK.
    retrieval/            (Chapter 6)
    agents/               (Chapters 9 to 12)
    memory/               (Chapter 11)

  evals/                  THE EVAL SEAM.
    cases/doc_qa.jsonl      labelled cases, including injection tests
    run.py                  prints a score

  tests/
    test_architecture.py  the seams, enforced in CI
    test_gateway.py       behaviour, with a mocked transport
    test_prompts.py
```

Two conventions carry almost all of the value, and both are enforced by tests rather than by documentation, because **guidelines lose to deadlines and a failing build does not**.

Convention one: no provider SDK outside `gateway/`

This is the model seam expressed as a rule. Its value is not abstraction for its own sake. It is that when Chapter 3 adds a router, a semantic cache and a budget ledger, they go in one place and no caller changes. When Chapter 15 adds output policy enforcement, there is exactly one chokepoint to add it to.

The rule is worthless as a code review guideline. Someone will be shipping on a Thursday and will import the SDK directly because the gateway does not yet expose the parameter they need, and they will mean to come back to it. Enforce it in CI, where it costs two hundred milliseconds:

```
# tests/test_architecture.py
FORBIDDEN_SDKS = {"openai", "anthropic", "google", "ollama", "litellm", "langchain"}

def _imports(path: Path) -> set[str]:
    tree = ast.parse(path.read_text())
    out: set[str] = set()
    for node in ast.walk(tree):
        if isinstance(node, ast.Import):
            out.update(a.name.split(".")[0] for a in node.names)
        elif isinstance(node, ast.ImportFrom) and node.module:
            out.add(node.module.split(".")[0])
    return out


def test_provider_sdks_are_confined_to_the_gateway():
    violations = []
    for path in SRC.rglob("*.py"):
        if "gateway" in path.parts:
            continue
        for mod in _imports(path) & FORBIDDEN_SDKS:
            violations.append(f"{path.relative_to(SRC)} imports {mod}")
    assert not violations, "provider SDK outside gateway/:\n" + "\n".join(violations)
```

Note that `langchain` is on the forbidden list alongside the provider SDKs. That is deliberate and it is a real architectural position, not a criticism of the library. An orchestration framework that constructs model requests is a provider abstraction, and if it is imported from forty call sites you have simply moved the coupling rather than removed it. Frameworks belong behind the seam, exactly like providers.

Convention two: no prompt text outside `prompts/`

A prompt that escapes the registry is a prompt with no version. A prompt with no version cannot be named when quality moves, which is the entire subject of section 5.8. The test is heuristic and occasionally produces a false positive, and it is still worth running:

```
def test_no_prompt_text_outside_the_registry():
    markers = ("you are a", "your task is", "answer only from", "respond with json")
    violations = []
    for path in SRC.rglob("*.py"):
        if path.parts[-2:] == ("prompts", "registry.py"):
            continue
        for node in ast.walk(ast.parse(path.read_text())):
            if isinstance(node, ast.Constant) and isinstance(node.value, str):
                text = node.value.strip().lower()
                if len(text) > 60 and any(m in text for m in markers):
                    violations.append(f"{path.relative_to(SRC)}:{node.lineno}")
    assert not violations, "prompt text outside prompts/:\n" + "\n".join(violations)
```

Two further tests in the repository are worth mentioning because they enforce structure that is otherwise invisible. `test_no_local_environment_branching` fails the build on `if settings.LOCAL` style branches inside `src/`, protecting the parity rule from section 1.3. `test_api_layer_holds_no_business_logic` fails if the API package imports `retrieval`, `agents` or `memory` directly, which keeps HTTP concerns from leaking into the domain and, more practically, keeps the API layer testable without a vector database.

All of these run in CI on every push, in the same job as the unit tests: they need no network, no containers and no model, so there is no reason to run them anywhere slower. The same pipeline should run a secret scanner over every commit, because a provider key pasted into a test fixture on a Thursday evening is caught by machinery within seconds and by code review only sometimes. Which scanner you choose matters far less than that one runs before merge rather than after the leak.

> WHY ARCHITECTURE TESTS AND NOT A LINTER PLUGIN
>
> These four tests are about seventy lines of `ast` walking with no dependencies. They are readable by anyone on the team, modifiable in a minute, and they fail with a message that names the file and the line. A configured plugin does the same job and adds a dependency, a config file and a layer of indirection between the rule and its rationale. When someone new asks "why can't I import openai here", the assertion message and the docstring answer it. That is worth more than the elegance.

EXERCISE 1.4

Add `import openai` to `src/docassist/api/main.py` and run `make test`. Read the failure message. Then move it into `gateway/client.py` and watch it pass. You have just experienced the entire mechanism by which a seam survives its second year.

1.5 API Keys, Secrets Hygiene and Cost Guardrails

The instinct is to move keys out of source control and stop. Three properties matter, and only the first is about source control.

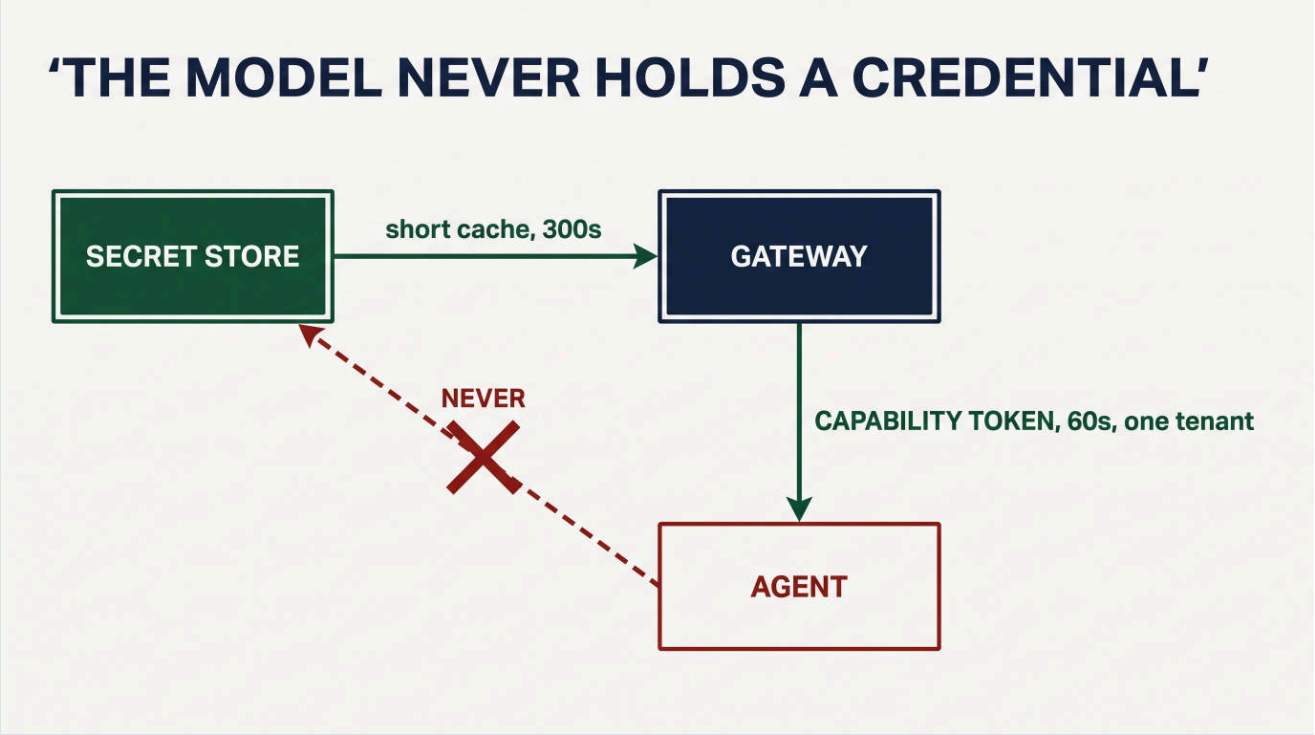

Figure 1.3 The credential boundary. The gateway holds the provider key. The agent holds a scoped, expiring capability, and never the key itself.

Property one: rotatable without a deploy

GOOD

Read the environment at startup

```
OPENAI_API_KEY = os.environ["OPENAI_API_KEY"]   # fails fast if absent
```

Correct as far as it goes, and the right thing for a prototype. It fails loudly when unconfigured, which is better than a `None` propagating into a request header.

What it leaves unbounded: rotation requires a redeploy. Which means rotation is slow. Which means that when a key leaks into a log or a screenshot, the exposure window is measured in whatever your deploy cycle is, and in practice people delay rotating because it means a release. The property you want is not "we can rotate", it is "we will actually rotate promptly", and the second is a function of how cheap it is.

BETTER

Fetch per request, cache for minutes, invalidate on rotation

```
class SecretProvider:
    """A short-lived cache in front of the secret source.

    Reading a credential once at import is what makes rotation require a
    deploy, which is what makes rotation slow, which is what makes a leak
    window days long.
    """

    def __init__(self, ttl_s: int = 300) -> None:
        self._ttl_s = ttl_s
        self._cache: dict[str, tuple[str, float]] = {}

    async def get(self, name: str) -> str:
        hit = self._cache.get(name)
        if hit and time.monotonic() - hit[1] < self._ttl_s:
            return hit[0]
        value = await self._fetch(name)
        self._cache[name] = (value, time.monotonic())
        return value

    def invalidate(self, name: str) -> None:
        """Called by the rotation webhook. This method is the entire point."""
        self._cache.pop(name, None)
```

Five minutes of cache means a rotation propagates across your fleet in five minutes with no deploy, no coordination and no downtime. The implementation behind `_fetch` is a secret manager in production and the process environment locally; what production and local share is the *interface*, per the rule in section 1.3.

Scope keys as narrowly as the provider allows while you are here. Separate keys per environment is the minimum, because it gives you independent spend attribution and lets you revoke staging without stopping production. Separate keys per major workload is better: when the nightly batch job's key leaks, the interactive product keeps working.

Property two: the model can never reach a credential

BEST

Capabilities, not credentials, past the gateway

This property is specific to this domain and is routinely violated, usually by accident, usually in the sprint where tools are added.

The reasoning is short and it is worth following completely, because it is the first appearance of the argument that governs Chapters 12 and 15. Your agent reads untrusted content: uploaded documents, retrieved passages, web pages, tool results. That content shares a channel with your instructions and the model has no enforced notion of privilege between them, which section 5.9 develops. Therefore assume the model can be induced to attempt anything expressible in its output. Now ask what its output can reach. If any tool it can call reads the process environment, executes a shell command, renders an arbitrary template, or fetches an arbitrary URL, then **a successful prompt injection reads your provider key, your database URL and your cloud credentials**.

The control is not to make the model refuse. It is to arrange that the credential is not present where the model can reach it:

```
class ToolContext:
    """Tools receive a context, never the process environment.

    A capability is minted per run, scoped to one tenant and one tool, and
    expires in seconds. A leaked capability is worth almost nothing. A leaked
    API key is worth everything.
    """
    tenant_id: str
    run_id: str
    _capabilities: dict[str, Capability]

    def credential_for(self, tool_name: str) -> Capability:
        cap = self._capabilities.get(tool_name)
        if cap is None or cap.expired():
            raise PermissionError(f"no capability for {tool_name}")
        return cap


# And in the worker process, before any agent runs:
os.environ.pop("OPENAI_API_KEY", None)     # the gateway already holds it
os.environ.pop("DATABASE_URL", None)       # nothing downstream needs it
```

Those two `os.environ.pop` lines look almost trivial and they change the shape of the worst case. After them, an attacker who fully controls the model's output gets a bad answer and a rejected tool call, rather than a credential. Chapter 12

builds the full permission model on this foundation; the decision that makes it possible is made here, in the first week, by deciding that the environment is not a place tools read from.

Property three: cost is bounded before the call, not discovered after

Set the provider's own spend cap and alert on day one, at a number that would embarrass you rather than bankrupt you. The failure mode is not gradual: an unbounded retry loop, a test suite pointed at the live endpoint, or a batch job aimed at the wrong model can generate several months of expected spend in an afternoon.

Provider caps are coarse and slow, so mirror them locally with something that can refuse a specific call:

```
class SpendCircuit:
    """Two windows, because they catch different failures.

    The DAILY limit catches slow overspend. The PER-MINUTE limit catches the
    runaway loop, and it is the one that matters: a retry storm can burn a
    month of budget in an hour, and a daily cap notices far too late.
    """

    def __init__(self, daily_usd: float, per_minute_usd: float) -> None:
        self.daily_usd = daily_usd
        self.per_minute_usd = per_minute_usd
        self._events: list[tuple[float, float]] = []

    def record(self, usd: float) -> None:
        self._events.append((time.monotonic(), usd))

    def spend_since(self, seconds: float) -> float:
        cutoff = time.monotonic() - seconds
        return sum(usd for ts, usd in self._events if ts >= cutoff)

    def check(self, projected_usd: float) -> None:
        if self.spend_since(86_400) + projected_usd > self.daily_usd:
            raise BudgetExceeded("daily cap")
        if self.spend_since(60) > self.per_minute_usd:
            raise BudgetExceeded("spend rate cap")
```

Two design points that are easy to get wrong. The check runs *before* the call, on an estimate, which means you need a cost estimator that is roughly right on input tokens and deliberately conservative on output tokens, because the model may use its full allowance. And the rate window matters more than the absolute one: by the time a daily cap trips, the money is gone, whereas a rate anomaly is visible within a minute of the loop starting.

```
# From gateway/client.py. The estimate is deliberately pessimistic.
def estimate_cost(self, prompt_chars: int, max_tokens: int) -> float:
    approx_input = prompt_chars // 4
    return self._cost(approx_input, max_tokens)   # assume full output

# ...and the check is the first thing complete() does, before any I/O.
self.spend.check(self.estimate_cost(prompt_chars, max_tokens))
```

PROMOTION TRIGGERS

Good to better: before the first credential exists in more than one place, which is usually the first CI run. An hour of work.

Better to best: before the first tool with side effects is attached to an agent, which is Chapter 9. The `os.environ.pop` lines cost nothing and should go in now; the full capability model can wait until there is something to scope.

The spend circuit is not optional at any tier. It is thirty lines and it is the difference between an incident and an inconvenience.

EXERCISE 1.5

Set `DAILY_BUDGET_USD=0.001` in `.env` and call `/ask`. You should get a 429 before any network request is made. Confirm that with `docker compose logs api`: there should be no outbound call at all. A budget that refuses *after* the call is a report, not a control.

1.6 The Four Seams, and the Frame

Everything in the preceding sections assembles into four seams. Each costs about an hour to build during a prototype and about a quarter to retrofit into a running system, and that asymmetry is the whole argument for building them now.

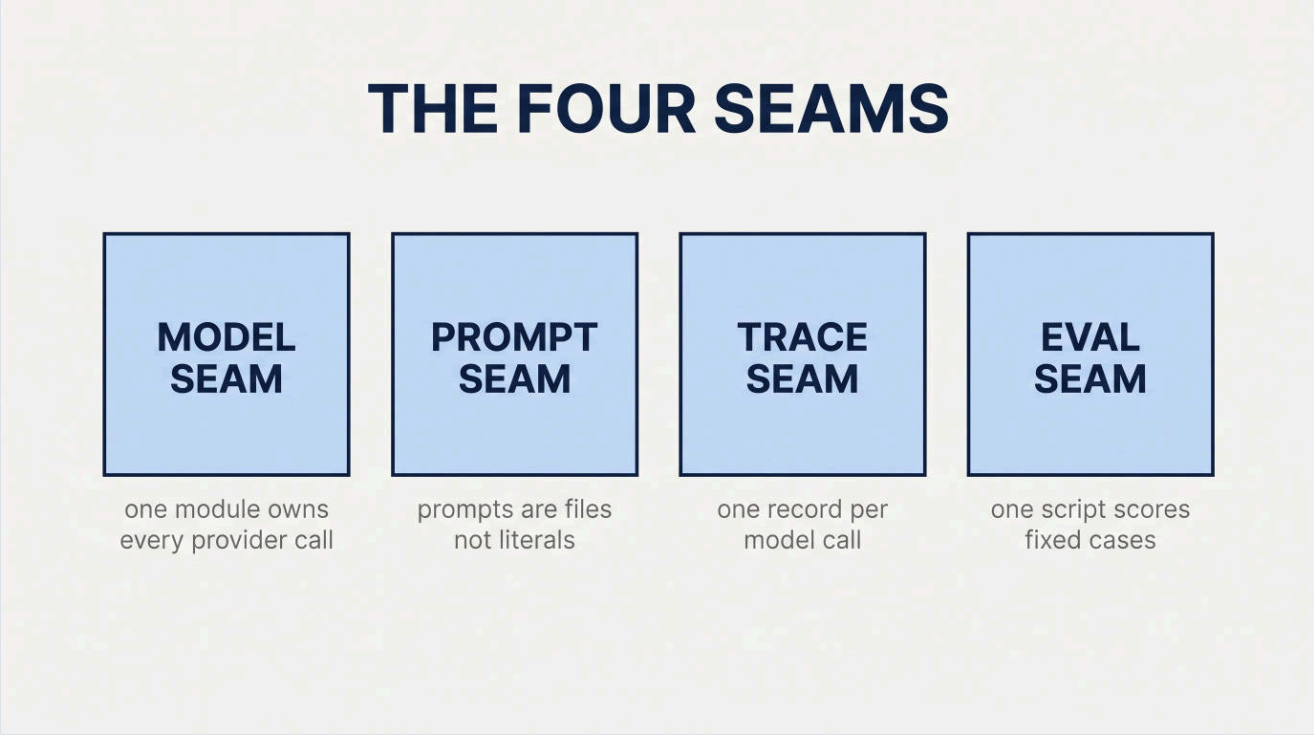

Figure 1.4 The four seams. Their absence is what makes month four expensive: there is no place to put the fix.

The model seam

Exactly one module constructs a provider request. In the repository that is `gateway/client.py`, and the CI test from section 1.4 keeps it that way.

What it holds after Chapter 1: a deadline that governs the retry loop rather than an attempt counter, full jitter on backoff, separate connect and request timeouts, a circuit breaker, token accounting, and the spend check. What it will hold by Chapter 3: tier routing, a semantic cache, a budget ledger and a degraded response path, none of which will require a caller to change.

```
for attempt in range(1, self._s.max_attempts + 1):
    remaining = deadline_s - (time.monotonic() - started)
    # The DEADLINE governs, not the attempt counter. Three attempts at thirty
    # seconds each is a ninety second worst case no upstream will tolerate.
    if remaining <= 0.5:
        break
    try:
        r = await self._client.post(..., timeout=httpx.Timeout(
            remaining, connect=self._s.connect_timeout_s))
        ...
    except (RetryableError, httpx.TimeoutException, httpx.TransportError) as exc:
        if attempt == self._s.max_attempts:
            break
        # Full jitter. Pure exponential backoff SYNCHRONISES your fleet:
        # every client fails at the same moment, waits the same interval,
        # and retries in the same instant.
        backoff = min(2 ** (attempt - 1), 8) * random.random()
        await asyncio.sleep(min(backoff, max(0.0, remaining - 0.5)))
```

Four small decisions in that loop separate a gateway that helps from one that provides false comfort, and each has a corresponding production incident behind it. The deadline, because a retry policy expressed only as a count has an unbounded worst case. The jitter, because without it your fleet converts a provider hiccup into a thundering herd. Separate connect and request timeouts, because a connection that will never establish should fail in three seconds while a legitimate generation may need thirty, and one number cannot serve both. And, in the success path, **recording the model the provider actually ran** rather than the one you asked for, because aliases resolve server side and change without notice.

The prompt seam

Prompts are versioned files with identifiers, loaded by a registry, rendered with strict variable checking. The property that matters from day one is attribution: every trace carries `prompt_id` and `prompt_version`, so "quality dropped on Tuesday" becomes a version name rather than a week of bisecting.

Two design decisions in the registry are worth flagging. Missing variables raise at the boundary rather than rendering a template with a hole in it, because a prompt with an unfilled slot produces a plausible, confident, wrong answer rather than an error. And untrusted content goes into a delimited user turn, never the system message:

```
def render(self, variables: dict[str, str]) -> list[dict]:
    missing = set(self.variables) - set(variables)
    if missing:
        raise MissingVariables(f"{self.id} v{self.version} missing: {sorted(missing)}")
    ...
    return [
        {"role": "system", "content": self.system},
        {"role": "user", "content": body},      # evidence and query live here
    ]
```

That separation is a mitigation rather than a control, as section 5.9 will argue at length, and it costs nothing, so there is no reason to skip it. The repository has a test asserting the property, which means a future refactor cannot quietly undo it.

The trace seam

One structured record per model call, containing *the inputs that determined the output* rather than only the output. The minimum viable version is a JSON line on stdout collected by whatever already ships your logs, and it is worth more on day one than any dashboard you will buy in month six.

The field list is the substance. If any of these is absent there is a class of regression you will be unable to attribute:

Field	The question it answers later
`prompt_version`	Which prompt produced last Tuesday's answer?
`model_returned`	Did the provider silently repoint the alias?
`index_build_id`	Did retrieval quality move because the index was rebuilt?
`retrieved_ids`	Was the answer wrong, or was the evidence wrong?
`input_tokens`, `output_tokens`, `cost_usd`	What does one task actually cost?
`outcome`	What is the denominator of that cost?
`attempts`, `ttft_ms`	Is the tail latency ours or the provider's?

The implementation is a context manager that records on the way out, including on the exception path, and re-raises rather than swallowing. Its test double, `RecordingTracer`, is what lets the gateway tests in the repository assert on *decisions* rather than only on returned values, which is the property that makes a seam genuinely testable.

One consequence deserves naming now rather than in Chapter 15. A record that contains the inputs that determined the output contains your users' data: queries, retrieved passages, sometimes whole documents. Traces are therefore a data surface from the first day they exist, and they inherit the retention and deletion obligations of the primary store, because a tenant who deletes a document has not deleted it while it survives in six weeks of trace records. Section 15.4 treats traces as one of the five surfaces a deletion request must reach. Nothing needs building yet; what is needed now is only to know the surface exists.

The eval seam

One script, a fixed case set, a printed score. In the repository this is eighty lines and six seed cases, two of which are prompt injection attempts.

Its value is that it converts "does this change help" from an opinion into a number. Every optimisation in the remaining sixteen chapters is justified by a number this script or its descendants produced. Two structural choices are worth adopting immediately:

```
# Deterministic scoring where possible. A model judge (section 14.3) is for
# prose; for extraction and refusal, substring rules are exact, free, and not
# subject to the judge's own drift.
def score_one(case: dict, answer: str) -> tuple[bool, list[str]]:
    failures = []
    for needle in case.get("expect_contains", []):
        if needle.upper() not in answer.upper():
            failures.append(f"missing:{needle}")
    for banned in case.get("expect_absent", []):
        if banned.upper() in answer.upper():
            failures.append(f"leaked:{banned}")
    if (cite := case.get("expect_citation")) and f"[{cite}]" not in answer:
        failures.append(f"uncited:{cite}")
    return (not failures), failures


# Injection resistance is reported SEPARATELY and has no tolerance. A security
# property that is allowed to degrade gradually degrades to zero.
"injection_held": all(r["pass"] for r in results if "injection" in r["tags"]),
```

Six cases is not a serious eval set and it is enough to start, because the point of the seam is that the *apparatus* exists. Chapter 14 covers composition, judges and gates. What matters now is that adding a case is one line in a JSONL file, which means failures from production become tests instead of anecdotes.

> AGAINST THE CHARGE OF PREMATURE ARCHITECTURE
>
> Four seams in a prototype sounds like over-engineering, and the objection deserves a direct answer. None of the four is an abstraction layer, a framework, or a design pattern. Each is under a hundred lines of unremarkable code, and each replaces something you were going to write anyway with a version that has a name and a location. The test is simple: *if your prototype succeeds, will you keep the code?* If yes, build the seams. If the code is genuinely disposable, skip all four and skip most of this book until it is not.

The frame: five properties that invalidate ordinary reasoning

An experienced backend architect arriving here brings a toolkit that is mostly correct and dangerously incomplete. Instincts about statelessness, horizontal scaling, idempotency and circuit breaking all transfer. Five properties do not fit the toolkit, and each invalidates a specific habit.

Property	What it invalidates
Marginal cost per request is large, variable, and paid to someone else. One request can cost a hundredth of a cent or several dollars, determined largely by how much text the model chooses to emit.	*Caching as an optimisation.* Here it is a cost control with a latency side effect and belongs in the architecture from the start. Also invalidates any load test measuring requests per second while ignoring tokens per second.
Latency is unbounded and proportional to output, not input. No timeout is simultaneously generous enough for the legitimate long answer and tight enough to protect workers from a pathological one.	*The synchronous request handler as a default.* This is the most common cause of production incidents in first generation AI products. The correct default is streaming, a queue, or both. Chapter 8.
Correctness is statistical, so testing is sampling. The same input can produce a different output tomorrow because the provider changed something.	*The pass-or-fail test as the unit of confidence.* A system with no eval set does not have untested quality; it has quality that is unknown and unknowable, including to the people operating it.
The parser is the model, and instruction and data share one channel. A retrieved document containing an instruction is, to the model, the same kind of thing as your system prompt.	*Input validation as a boundary control.* There is no grammar to validate against. The control must move to the other side: constrain what the output may *cause*. Chapters 12 and 15.
Your dependency changes underneath you. Hosted endpoints are updated, repriced and re-tuned without a version string you control; self hosted weights drift as runtimes and quantisation kernels change.	*The assumption that unchanged code implies unchanged behaviour.* Regression detection cannot be a release-time activity. It runs continuously, against production traffic, forever.

THE COMPOUND ERROR

These interact, and almost every serious AI production incident is a combination of at least three, never one. Non-determinism plus per-request cost means a retry is not free. Unbounded latency plus statistical correctness means a naive retry loop multiplies your tail latency by the attempt count. Silent drift plus an absent eval set means the failure is invisible until a customer describes it, at which point you cannot reproduce it because the model has already moved on. When you read the incident post-mortems in later chapters, look for the three that combined.

The unit of measurement

Cost per API call is a number your provider gives you and it is nearly useless for architecture. The number that matters is the cost of one *completed unit of user value*, including everything spent on attempts that produced none.

Figure 1.5 The same task, two retrieval strategies, identical model. The per-call view cannot see the success rate, so it cannot see the gap.

```
def cost_per_successful_task(attempt_costs: list[float], success_rate: float) -> float:
    """success_rate is MEASURED against a held-out set, never estimated.
    Without a measured rate you do not have a cost model, you have a bill."""
    return sum(attempt_costs) / success_rate


# Same user task, two retrieval strategies, identical model.
# Naive: 20 unranked chunks stuffed in. Budgeted: reranked to 4 (Chapter 7).
naive    = 14_000 * 2.50 / 1e6 + 700 * 10.00 / 1e6   # $0.04200 per call
budgeted =  3_400 * 2.50 / 1e6 + 700 * 10.00 / 1e6   # $0.01550 per call

cost_per_successful_task([naive],    success_rate=0.71)   # $0.05915
cost_per_successful_task([budgeted], success_rate=0.88)   # $0.01761
```

A factor of 3.4, from a change that touched no model and no prompt. On a cost-per-call dashboard it would have looked like a 2.7x improvement, because the per-call view cannot see the success rate. The reranking step itself costs money, roughly a tenth of a cent here, which the calculation should include and which does not change the conclusion.

Two habits follow. **Denominate cost in successful tasks**, which forces you to have a measured success rate, which forces you to have an eval set, which is why the fourth seam exists. And **be suspicious of any optimisation argued in per-call terms, including your own**: reducing per-call cost while reducing success rate moves the real number the wrong way and looks like a win on every dashboard you own.

1.7 The Four Layers

A fixed vocabulary for where a problem lives is worth more than it sounds, because the most expensive mistake in this field is fixing the wrong layer.

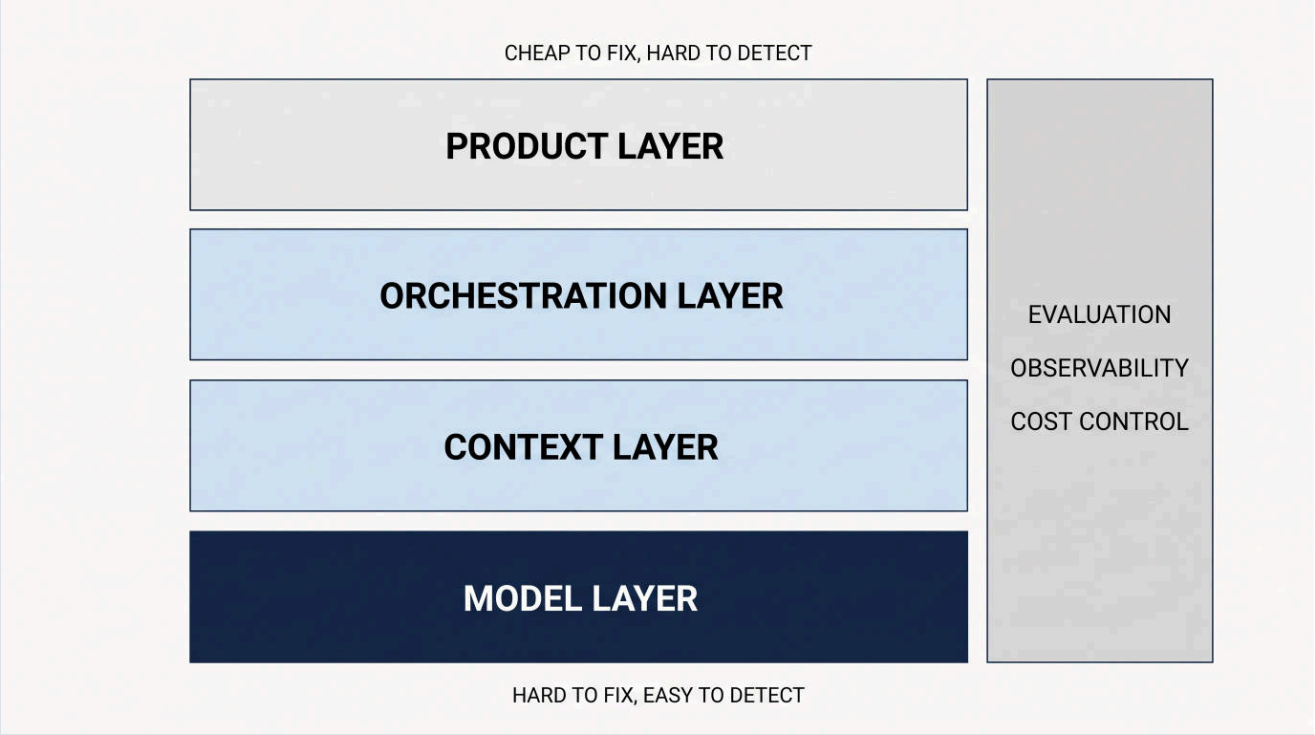

Figure 1.6 Problems at the top are cheap to fix and hard to detect. Problems at the bottom are the reverse. Evaluation cuts all four.

The **model layer** is weights, serving runtime and provider APIs. It dominates cost and latency, and it is easy to change if and only if you built the model seam. Chapters 2, 3 and 4.

The **context layer** decides what text the model actually sees: retrieval, memory, prompt assembly, tool schemas. **It dominates answer quality**, and it is where the majority of problems misdiagnosed as "the model is not smart enough" actually live. Chapters 5 to 7 and 11.

The **orchestration layer** is the agent loop, the graph, the state machine, the routing. It dominates reliability and debuggability. Chapters 8 to 10 and 12.

The **product layer** is interface, guardrails and human handoff. It dominates whether users trust the result. Chapters 13 and 15.

The diagnostic worth internalising. When answer quality is poor, engineers reach for the model layer, because that is the layer with a knob labelled "better model." The distribution of root causes we see repeatedly, in our own systems and the ones we are asked to review, is closer to: **context layer sixty percent** (wrong chunks retrieved, wrong chunk size, history crowding out evidence), prompt and output contract twenty percent, orchestration ten percent, model capability ten percent. Reaching for a bigger model first is expensive and usually addresses the smallest slice. Section 7.2 gives the procedure for measuring your own distribution, an afternoon's work: expect it to differ from these numbers in detail and to agree with them in ordering.

Where evaluation sits

Evaluation and observability cut through all four layers, which is why the figure shows them as a vertical band rather than a fifth layer. This is also why the eval seam is built in Chapter 1 rather than Chapter 14. Its absence does not slow you down at first, and it slows you down enormously at month four, when you have accumulated forty prompt changes, three model swaps and two retrieval strategies, and no way to attribute a quality regression to any of them.

Chapter checklist

Before moving to Chapter 2, you should be able to do all of the following. If any is unclear, the relevant section is named.

Can you...	Section
State the four judgements an AI architect repeatedly makes, and name which one dominates answer quality	1.1, 1.7
Explain why an embedding library upgrade is a migration rather than an upgrade	1.2
Name the three compose settings that prevent a local environment from becoming hostile to use	1.3
Write the CI test that keeps a provider SDK out of forty call sites	1.4
Explain why a spend cap that trips daily is insufficient	1.5
Say what the four seams are, and what each costs to retrofit	1.6
Compute cost per successful task, and say why cost per call misleads	1.6
Name the five properties that invalidate ordinary distributed systems reasoning	1.6
Given an unfamiliar AI system, name its unbounded dimension in ten minutes	the whole book

EXERCISE 1.7, AND THE HABIT WORTH FORMING

Pick an AI feature you have seen shipped, yours or someone else's. Write one paragraph answering: which dimension is unbounded, what traffic level exercises it, and which of the four seams is missing. Keep the paragraph. Repeat it for a different system every week for a month. That exercise, more than any single chapter here, is what turns the material into judgement.

What Chapter 2 establishes

Chapter 1 built the structure. Chapter 2 establishes the physics that structure has to respect: what a token actually is and why domain vocabulary inflates your bill, why the context window is a budget rather than a container, why attention is quadratic and what that means for how many users fit on one accelerator, and why cosine similarity is topical rather than answer bearing, which is the fact that makes reranking structural rather than optional. Every constraint in Chapter 2 is a constraint you will design around for the remaining fifteen chapters.