

A FIELD MANUAL • SECOND EDITION

Engineering **Deterministic** AI Coding Agents

A language model is a probabilistic engine. Almost everything around it can be decided in code: what it retrieves, what stays in its window, which step runs next, what it may do, what it may spend, and what gets recorded. Parts 1 to 13 build that system, with published evidence in each part. Parts 14 to 20 show how to operate the agents you now run. Every part ends with steps for this week and metrics to track.



Mac Anderson

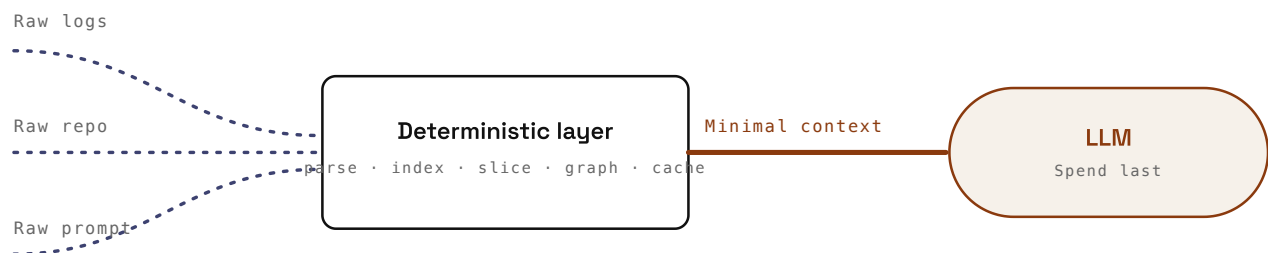
Founder and CEO, Oxagen Inc. • Los Angeles, CA

21 parts **27** cited sources **21** checklists, and a field kit of templates

1 thesis: decide in the system what the system can decide

..... Probabilistic path: you pay per token, and the result varies

——— Deterministic path: you pay once in engineering, and the result repeats





How to use this book

Read the two parts that match your problem this week. Come back for the rest.

You run coding agents, or you are about to, and two questions keep coming up. The first is from your own team: why does this cost so much and behave differently each time? The second is from everyone else: what are these agents allowed to do, what did they spend, and who answers for them? Parts 1 to 13 answer the first question. Parts 14 to 20 answer the second. Part 21 explains why both have the same answer.

How each part is laid out

- **The situation.** Each part opens on a problem you have probably seen.
- **The mechanism.** What to build, in order, with a short sample.
- **Evidence.** A published result, cited. The sources are listed at the back, and the primary papers are better than any summary of them.
- **Counterweight.** Where the argument stops applying. Not every part has one.
- **Do this week.** Three to five steps, cheapest first. Each says what done looks like.
- **Measure it.** The readings that tell you whether the step worked.

Parts 14 to 20 add a short box, "Where Oxagen fits". Oxagen publishes this book and builds a product for that half of the job. The practice in each part stands without the product, and the box says what the product does and where its boundary is. Parts 1 to 13 mention no product.

Three reading paths

If you are	Start with	Then
An engineer building or tuning an agent	Parts 1, 2, and 3	Part 13 to set up measurement, then the rest of 4 to 12 in the build order below
A platform lead or operator answering for several agents	Parts 14, 15, and 16	Parts 17 and 20, then part 13 for the engineering scorecard
A security or finance lead reviewing agent use	Part 16 (security) or part 17 (finance)	Parts 15 and 19, which cover the object you review and the record you review it from

A build order

The parts are numbered by topic, not by the order to build them. If you are starting from an agent that works and costs too much, this order pays back fastest, because each step makes the next one measurable.

Step	Build	Part	Why now
1	Token and cost logging per step, with run, agent, and person	13, 17	Nothing later can be verified without it
2	An inventory of agents, each with an operator	14	One afternoon, and every later step needs the names
3	Trace and log slicing	2	The largest reduction in context for the least code
4	Prompt parsing and a symbol index	3, 4	Replaces the exploratory calls at the start of a run
5	Schema slice and test lookup	8, 10	Removes two common causes of wrong patches

Step	Build	Part	Why now
6	A mandate for each agent, and rules for the top ten write actions	15, 16	By now other teams are asking
7	A workflow skeleton and tiered memory	9, 11	Worth doing once runs are long enough to loop
8	Per-agent tool assignment	18	Needs the offered-against-called data from step 1
9	Domain dossiers and the knowledge graph	5, 12	The largest build. It federates the indexes from steps 4 and 5
10	The run record, chained, and completion checks for bounded tasks	19, 20	Turns the rows you already write into something a reviewer can use

A self-assessment

Answer yes or no. Count a yes only if you could show the evidence today. Each no points at a part.

#	Question	If no, read
1	Can you split one run's tokens into context construction and reasoning?	1
2	Does code, not the model, reduce a stack trace before it reaches the window?	2
3	Are file paths and symbols in a prompt resolved against an index before the first model call?	3, 4
4	Does a new run start from stored knowledge of the repository?	5
5	Can every standing instruction fail a check in CI?	6
6	Does every context assembler take a token budget?	7

#	Question	If no, read
7	Does the agent see the live schema, with lineage, before it writes a migration?	8
8	Is control flow between steps written as code?	9
9	Can you look up the tests that cover a function?	10
10	Does old content leave the window by policy?	11
11	Can you answer a three-hop question about your system with one query?	12
12	Do you report cost per completed task, including retries?	13
13	Can you list every agent that can write to a system, with a named operator for each?	14
14	Is each agent's authority, budget, and equipment written in one reviewed place?	15
15	For a given write action, can you name the rule that allowed it and its author?	16
16	Can you attribute last month's spend to the agent, the run, and the person?	17
17	Do you know how many tools each agent is offered, and how many it calls?	18
18	Could someone outside your team reconstruct one agent action in 30 minutes?	19
19	Are completion checks for bounded tasks fixed before the run starts?	20
20	Does each ongoing agent have a review date on a calendar?	20

Score yourself now and write the number down. Part 21 asks you to do it again.

The print layout is set for US Letter. In the print dialog, choose Save as PDF to keep a copy. The page-flip reader at oxagen.sh/read can also export single parts.

AI agents are not expensive. Bad architecture is.

Most of an agent's token bill comes from the system around the model, not from the model.

Your agent works, and the invoice is bigger than the team that uses it. The first suspects are the obvious ones: the per-token price, the reasoning overhead, the size of the context window. Then you instrument the agent, log every request, and look at where the tokens went. The model did what it was asked to do. The system asked it to do far too much.

Measure where tokens actually go

Run an agent framework against a real repository and log every request. In the runs this book draws on, the dominant cost is not *reasoning about the problem*. It is *rebuilding context the system already had*: re-reading files it read two turns ago, carrying 400 lines of stack trace when 6 frames mattered, pushing the whole conversation history into every tool call, and walking the filesystem with `ls` and `cat` because nothing indexed the repository up front.

The published numbers point the same way. Anthropic's engineering team, describing their production multi-agent research system, reported that agents consume roughly **4× the tokens of a chat interaction**, and multi-agent systems roughly **15×**, and that on their internal evaluations **token usage alone explained about 80% of performance variance**.³ The largest lever they found was not model choice or prompt wording. It was how many tokens the architecture decided to spend.

EVIDENCE • SIMPLICITY WINS THE LEADERBOARD

The clearest data point in this book is **Agentless** (Xia et al., UIUC). Instead of an autonomous agent choosing its own actions with tools, Agentless runs a fixed three-phase pipeline: hierarchical localization, patch generation, validation. On SWE-bench Lite it became the top open-source approach of its time at **27.3% solved for \$0.34 per issue** (later 32.0% at \$0.70), while contemporary agent-based systems spent roughly 10× more per issue for comparable or worse results.¹ The model did not choose the next action. The pipeline chose, and the model filled in the parts that needed judgment.

Xia et al., "Agentless: Demystifying LLM-based Software Engineering Agents," FSE 2025

Context construction against reasoning cost

Split your agent's token bill into two buckets. **Reasoning tokens** are the model thinking about the actual problem: the diff, the root cause, the design decision. **Context-construction tokens** are everything spent getting the model to the point where reasoning can begin: file dumps, search results, log output, retries after the model lost the thread. In an unindexed architecture the second bucket dominates, often by an order of magnitude. Most of what sits in that second bucket is computable without a model. Parsing, indexing, slicing, and graph traversal are deterministic operations. They run on CPU, return the same answer on the same input, and you can test them like any other code.

You cannot split the bill you do not record, so record it at the call site. The caller knows why it is making the call, and the caller is the only part of the system that does.

```
# One row per model call, written when the call returns.
log_call(
    task_id=task.id,
    purpose="context",                # or "reasoning", set by the caller
    source="file_read:billing/processors.py",
    input_tokens=usage.input,
    cached_input_tokens=usage.cache_read,
    output_tokens=usage.output,
)
# context_token_share = input where purpose == "context" / all input
# Group by source to rank what is filling the window.
```

EVIDENCE • COST MUST BE A FIRST-CLASS METRIC

Princeton's "AI Agents That Matter" (Kapoor, Stroebl, Siegel, Nadgir, Narayanan) showed that agent research had been ignoring cost, and that when you evaluate on a joint cost and accuracy Pareto frontier, simple baselines such as retrying a model call matched or beat elaborate agent architectures on HumanEval at a fraction of the price. Their conclusion: accuracy alone cannot identify progress, and state of the art (SOTA) agents are frequently more complex and more costly than the task requires.²

Kapoor et al., "AI Agents That Matter," TMLR 2025

Thinking harder against needing less thinking

The default answer to agent failure is to think harder: a bigger model, a longer reasoning chain, more retries, more sub-agents. Each of those multiplies token spend and adds another probabilistic step, which is another place for variance to enter. The engineering answer runs the other way. Shrink the problem before the model sees it. Every ambiguity you resolve deterministically is reasoning the model no longer has to do, tokens you no longer pay for, and variance you no longer ship. Determinism is not a philosophical preference. It is a cost model.

Attributing that spend across the teams who run the agents is part 17.

DO THIS WEEK

1. **Log usage on every model call.** Write one row per call with `task_id`, `purpose`, `source`, input tokens, cached input tokens, and output tokens. The artifact is a table you can group by, not a log line you can read.
2. **Tag each call reasoning or context.** Set the tag at the call site, where the caller knows why the call is happening. Done looks like every call carrying a tag and none defaulting to unknown.
3. **Rank your context sources.** Group last week's rows by `source` and sum input tokens. The top three sources are your work queue for parts 2 to 4.
4. **Turn on prompt caching for the stable prefix.** Put the system prompt, tool definitions, and repository map ahead of anything that changes per turn, then watch the cache-read share rise.
5. **Put a token budget in code.** Give each task a ceiling on input tokens and fail the run when it trips, so a runaway loop ends in a test failure instead of an invoice.

MEASURE IT

- `context_token_share`: context-construction input tokens divided by all input tokens, per task. Down is good. Above 0.8 means the model is paying to rediscover what your code already knows.
- `tokens_per_completed_task`: input plus output tokens divided by tasks that passed their check. Down is good, and it is the only token metric that survives a change in task mix.
- `cache_read_share`: cached input tokens divided by all input tokens. Up is good. A falling share usually means something volatile moved into the prefix.
- `cost_per_completed_task`: dollars divided by tasks that passed. Down is good, and it is the number to put next to accuracy when you compare two architectures.

TAKEAWAY

Reduce uncertainty before you invoke intelligence. Every token of context your system constructs deterministically is a token the model does not spend rediscovering it probabilistically.

Stop making agents read your entire logs

Logs are structured data, and treating them as prompt text costs tokens you do not have to spend.

Open a run where your agent debugged a failing test and you will find the same four steps every time: `cat` the log file, `grep` for "error", `tail -n 200`, paste the result into context. The most expensive component in the stack is now doing the job of a parser, and doing it probabilistically.

Signal to noise is a system property

A production stack trace from a Python web request can run hundreds of frames, and most of them are framework internals: WSGI plumbing, middleware, object-relational mapper (ORM) dispatch. The frames that matter for the bug are usually the handful that touch *your* code. An irrelevant frame is not neutral padding. It costs tokens on the way in, and it competes for attention once it is there. Chroma's "Context Rot" study evaluated 18 frontier models and found that performance degrades as input length grows, even on simple tasks, and that it degrades faster when the context holds distractors that are related to the target without being the target.⁷ A raw log is a distractor-dense document by construction: hundreds of lines that look like the one line you need.

Retrieval research reports the same shape. Cuconasu et al. ("The Power of Noise," SIGIR 2024) found that adding semantically related but non-answer-bearing documents to a model's context degrades accuracy, and that related noise hurts more than unrelated noise, because the model cannot cheaply dismiss it.⁹ Framework frames in a stack trace are that kind of high-similarity distractor.

Execution slicing: preprocess, do not prompt

The deterministic alternative treats program output as structured data and slices it before inference:

- **Stack trace reduction:** parse the trace, keep frames inside your package roots, collapse framework frames to one-line markers, keep the exception type, the message, and the innermost user frame's local variables.
- **Dynamic trace extraction:** run the failing test under a tracer (`sys.settrace`, `coverage.py`, eBPF, OpenTelemetry) and extract the execution path that actually ran, with the values that flowed through it.
- **Log windowing:** anchor on the failure timestamp or request id and take a bounded, correlated window instead of the file.

Reach for stack trace reduction when

The failure raises. You have a traceback, and what you need is the frames in your own packages, the exception type, and the locals at the innermost user frame.

Reach for dynamic trace extraction when

The test fails without raising, or a legal-looking wrong value arrives somewhere downstream. You need the path that ran and the values that moved along it.

Reach for log windowing when

The failure happened in another process or another service. You have a request id or a timestamp, and you need the correlated window around it, bounded in lines.

```
# Instead of handing the model a shell and hoping:
#   grep -rn "error" logs/ | tail -500   ← the model pays for every candidate

result = run_code_with_trace(           # deterministic, runs in milliseconds
    cmd="pytest tests/test_billing.py::test_refund -x",
    package_roots=["app/"],             # slice to code you own
    capture=["exception", "locals", "executed_lines", "sql"],
    max_tokens=1200,                    # hard budget, enforced by code
)
context = result.to_prompt_block()      # 1.2k tokens of signal
```

The difference is architectural, not cosmetic. `grep | cat | tail` makes the *model* the filter. You pay tokens for every candidate line, and the filtering step itself is probabilistic.

`run_code_with_trace()` makes *code* the filter. Filtering costs no tokens, returns the same slice on the same input, and you can test it like any other function. Bash is a fine escape hatch. It is a poor primary retrieval engine, because each round trip is another model call that carries the whole conversation history with it.

EVIDENCE · THE INTERFACE IS THE BOTTLENECK

SWE-agent (Yang et al., Princeton) showed that agent performance depends heavily on the **agent-computer interface (ACI)**. Replacing raw shell interaction with purpose-built commands that return compact, structured views of files and search results improved resolution rates on SWE-bench.¹⁴ The model did not get smarter. The interface got stricter about what reached the context window. Anthropic's guidance on tool design for agents makes the same point: a tool should return a concise, high-signal representation and enforce token-efficient defaults, because an agent inherits every inefficiency of its tools.²⁰

Yang et al., "SWE-agent," NeurIPS 2024 · Anthropic engineering, 2025

DO THIS WEEK

1. **Write a stack trace reducer.** Parse the traceback with your language's own grammar, keep frames whose file path sits under a configured package root, and collapse the rest to one line each. The artifact is a function that takes a traceback string and returns a block under 1,500 tokens.
2. **Put a token budget on every tool result.** Each tool returns at most N tokens and says how much it dropped and why. Done looks like a truncation counter you can graph, not a silent cut.
3. **Trace the failing test instead of reading its output.** Run it under `coverage.py` or `sys.settrace`, keep executed lines inside your package roots, and hand the model the path rather than the log.
4. **Correlate logs by request id.** Emit a request id on every log line, then serve a bounded window around a failure instead of a file. If the id is missing today, adding it is the whole task.

MEASURE IT

- `context_tokens_per_tool_call`: tokens returned by a tool, at p50 and p99. Down is good. The p99 is where an unbounded `cat` hides.
- `frames_kept_ratio`: frames sent divided by frames in the raw trace. Down is good, as long as the innermost user frame survives every time.
- `tool_calls_per_task`: tool calls between the prompt and the first edit. Down is good, because each one carries the full history with it.
- `truncation_rate`: share of tool results that hit the budget. Watch both directions. Zero means the budget is slack, and a high rate means the slicer is not slicing.

TAKEAWAY

Every line your system filters in code is a line the model does not pay to read and cannot be distracted by. Deterministic preprocessing moves work off the model and onto a CPU you already own.

Parse the user prompt before the model ever sees it

Your prompt already carries structured information, so extract it before you pay for inference.

Someone files a bug against your agent and it reads like this: "The `POST /api/v2/refunds` endpoint throws `DecimalConversionError` in `billing/processors.py` after #4821 merged." That string holds an API route, an exception class, a file path, and an issue reference. Four retrieval keys, sitting in plain sight. Most agent stacks hand the raw string to the model and let it decide what to search for.

Deterministic extraction is a solved problem

Long before you need a language model, thirty lines of parsing gets you:

- **File paths:** anything matching path grammar, validated against the actual file tree, with near misses fuzzy-matched.
- **Symbols and signatures:** CamelCase and snake_case identifiers, `module.func()` call syntax, function signatures, validated against your symbol index (part 4).
- **Stack traces:** a pasted traceback has a rigid grammar in every language. Parse it fully rather than summarizing it.
- **Issue and PR references:** `#4821`, `JIRA-123`, resolved through an API into titles, diffs, and linked commits.
- **URLs and API endpoints:** a route pattern maps straight onto a router definition in the codebase.
- **Version and config literals:** package names, versions, environment variable names, feature flags.

Every extracted entity is a *verified anchor*. It either resolves against your index or it does not. That check is binary, and it is the check a generative approach cannot give you, so an invented file path or a misremembered function name survives into the next call instead of being dropped at the door.

```
# Thirty lines, no model, runs before the first call.
PATH    = re.compile(r"\b[\w./]+\.(py|ts|go|rs|java)\b")
SYMBOL  = re.compile(r"\b([A-Z][A-Za-z0-9]+|[a-z_][a-z0-9_]{2,})\((?\\)?")
ISSUE   = re.compile(r"#(\d+)\b([A-Z]{2,}\-\d+)\b")

def anchors(prompt, index):
    found, dropped = [], 0
    for raw in PATH.findall(prompt) + SYMBOL.findall(prompt):
        hit = index.resolve(raw)  # None when the tree has no such path or symbol
        if hit:
            found.append(hit)
        else:
            dropped += 1
    metrics.record("anchor_resolution_rate", len(found), dropped)
    return found
```

From entities to a retrieval plan

The output of parsing is not decoration. It is an execution plan that runs *before inference begins*:

```
plan = build_retrieval_plan(prompt)
# {
#   anchors:    [File("billing/processors.py"), Symbol("DecimalConversionError"),
#               Route("POST /api/v2/refunds"), Issue(4821)]
#   expand:     graph_neighbors(anchors, hops=1)          ← Part 4
#   tests:     tests_covering(anchors)                   ← Part 10
#   schema:    tables_touched(anchors)                   ← Part 8
#   budget:    8_000 tokens, ranked by relevance
# }
```

The model's first call now holds the right file, its direct dependencies, the covering tests, and the issue diff, assembled by code, ranked by graph centrality, and cut to budget. Compare that to the common pattern, where the model spends its first five calls, each carrying a growing history, rediscovering what a regex had at millisecond zero.

EVIDENCE • STRUCTURED LOCALIZATION BEATS FREE EXPLORATION

This is the design that let Agentless outperform autonomous agents. Its first phase is **hierarchical localization**, a fixed funnel from repository to files to classes and functions to edit locations, rather than a model wandering with tools. The paper's ablations show that staged narrowing keeps ground-truth edit locations in the candidate set while shrinking the code the model must read, and it is a large part of why the pipeline reached top open-source results at roughly a tenth of the cost of agent baselines.¹ Constrained input, focused attention, fewer irrelevant tokens: the localization step is deterministic scaffolding doing the model's foraging for it.

Xia et al., "Agentless," FSE 2025

○ PROMPT AS OPAQUE STRING

- The model infers what to look for, probabilistically, once per run
- Search terms may be invented, and paths go unverified
- 3 to 8 exploratory tool calls before the real work starts
- Cost scales with how long the model explores

● PROMPT AS STRUCTURED INPUT

- A parser extracts entities the same way on every run
- Every anchor is validated against a real index
- The retrieval plan executes before the first model call
- Cost scales with the complexity of the task

DO THIS WEEK

1. **Write the entity extractor.** Start with four patterns: path grammar, identifiers, issue references, and route patterns. The artifact is a function from prompt string to a list of typed entities, with unit tests over ten real prompts from your own queue.
2. **Validate every entity against an index.** Resolve paths against the file tree and symbols against your symbol index. Drop what does not resolve, fuzzy-match near misses, and count both.
3. **Emit the retrieval plan as an artifact.** Write the plan to the task record before the first model call, with anchors, expansions, and the token budget. When a run goes wrong, the plan is the first thing you read.
4. **Seed the first call from the plan.** Assemble the first message from resolved anchors instead of letting the model open with a search, then compare tool-call counts against the previous week.

MEASURE IT

- `anchor_resolution_rate`: entities that resolved against an index divided by entities extracted. Up is good. A drop usually means the index went stale, not that the prompts got worse.
- `first_call_hit_rate`: share of tasks where the file the final diff edited was already in the first call's context. Up is good, and this is the metric that tells you whether localization works.
- `preflight_tool_calls`: tool calls between the prompt and the first edit. Down is good.
- `unresolved_anchor_count`: entities the model was handed that no index could confirm. Down is good, because each one is a chance to act on a path that does not exist.

TAKEAWAY

Treat the prompt as the first document your system parses, not the first thing your model reads. Extraction is cheap, checkable against an index, and identical on every run. Inference is none of those things.

Why grep is the wrong retrieval engine

Searching is not understanding, and code already carries the structure you need.

You ask your agent to change a function signature, and it starts grepping. Grep answers one question: which lines contain this string? The questions the change actually turns on are structural. Who calls this function? What implements this interface? What breaks if I change the signature? Where does this value come from? Text search approximates those answers over many noisy round trips. A code graph returns them in one query.

Code is already a graph, so stop flattening it

Compilers have modeled code as structure for decades: ASTs, symbol tables, call graphs, dependency graphs, cross-reference indexes. Your IDE resolves "go to definition" in milliseconds with that machinery, and no model is involved. The default agent loop discards all of it and hands the model a shell, which forces it to rebuild structural knowledge out of string matches:

```
# The grep loop (typical agent run, abbreviated):
grep -rn "process_refund" .           # 47 matches, 31 in tests and comments
cat billing/processors.py             # 800 lines into context
grep -rn "RefundProcessor" .         # which of these are real call sites?
cat api/handlers/refunds.py          # another 600 lines
# ... more calls, and the call graph is still not in context

# The graph query:
g.callers("process_refund", depth=2)  # ranked call sites, 0 tokens
g.slice(symbol="process_refund", budget=6000) # minimal closed subgraph
```

One hundred greps approximate what one graph traversal computes. The grep version is not only slower. Each round trip is a model call that re-sends the conversation history, and most of the lines it returns are irrelevant to the change, which part 2 showed is the kind of noise that costs accuracy rather than just tokens. It also gives the model one more chance to take a wrong turn. The graph version is deterministic: the same query against the same index returns the same answer.

The index stack

- **AST layer** (tree-sitter): language-aware parsing, with definitions, references, and signatures for most mainstream languages.
- **Symbol index**: every definition and every reference site, linked in both directions, which is what the Language Server Protocol (LSP) and the SCIP Code Intelligence Protocol (SCIP) already standardize.
- **Dependency graph**: imports, package boundaries, build targets.
- **Call graph**: who invokes whom, with edge weights from reference counts or runtime traces.
- **Centrality ranking**: PageRank over the reference graph tells you which symbols matter most for understanding any region of the code.

EVIDENCE · GRAPH-RANKED CONTEXT IN PRODUCTION TOOLS

Aider's repo map is the clearest proof this works at tool scale. It parses the whole repository with tree-sitter into definitions and references, builds a graph where files are nodes and symbol references are edges, then runs a **PageRank-style algorithm** to pick the most-referenced identifiers that fit a fixed token budget (around 1k tokens for an entire repository by default). The model receives a compressed structural map, with key signatures and the most-referenced symbols, instead of raw file dumps. Aider's own benchmarking found that this improves code-editing performance on large repositories.¹²

AutoCodeRover (Zhang et al., NUS) made the same bet in research. It replaced plain text retrieval with AST-based search APIs (search-class, search-method, search-code-in-file), letting the model query program structure instead of strings, and reported improved issue-resolution efficacy on SWE-bench against text-based navigation.¹³

Aider, "Building a better repository map with tree-sitter" · Zhang et al., ISSTA 2024

Filesystem traversal against graph traversal

The reason graphs win is precision per token. A filesystem is organized for humans and build tools, by directory rather than by meaning. The code relevant to one change is scattered across handlers, models, migrations, and tests, and filesystem traversal makes the agent page through that scatter linearly. Graph traversal follows *edges of actual relationship*, and a two-hop neighborhood around a symbol is close to the minimal sufficient context for editing it. You are not searching a haystack. You are following a wire.

Query the graph when

The question names a symbol and asks about relationships: callers, implementers, overrides, the blast radius of a signature change, or which tests reach this line.

Search text when

The target is a string rather than a symbol: a log message, an error copy string, a feature flag name, a config key, a TODO, or a value in a file no parser understands.

DO THIS WEEK

1. **Index the repository with tree-sitter.** Parse every file into definitions and references, and write them to a table keyed by symbol. The artifact is an index you can rebuild from a clean checkout in one command.
2. **Build the reference graph and rank it.** Files or symbols are nodes, references are edges. Run PageRank over it and emit a repository map that fits a fixed token budget, then put that map in the cached prompt prefix.
3. **Expose three queries as tools.** `callers(symbol, depth)`, `implementers(interface)`, and `slice(symbol, budget)`. Each returns a ranked, budgeted block, not a file.
4. **Keep text search, and scope it.** Leave a search tool for string literals and config values, and say so in its description, so the model reaches for the graph when the question is structural.
5. **Reindex on change.** Update the index from the files a commit touched rather than rebuilding it, and record how old the index was at query time.

MEASURE IT

- `index_staleness_seconds`: age of the index at query time, at p99. Down is good. A stale index returns confident answers about code that moved.
- `context_precision`: lines in the assembled context that the final diff touched, divided by lines sent. Up is good, and it is the number that tells you whether the two-hop neighborhood is the right radius.
- `graph_query_share`: graph queries divided by all retrieval calls. Up is good while `context_precision` holds, and a fall usually means a query the graph cannot answer yet.
- `tokens_to_first_edit`: input tokens spent between the prompt and the first diff. Down is good.

TAKEAWAY

Grep $\times 100$ approximates what one graph query computes. Build the graph once, deterministically, and let the model spend its tokens on the change instead of on reconstructing the compiler's knowledge by string matching.

Build semantic memory once

A run should start from what the last one learned instead of repeating it.

On Monday a run spends about 40,000 tokens learning that your payments code lives in `billing/`, that `LedgerEntry` is the money-movement primitive, and that notifications go through an outbox pattern. On Tuesday a new run spends about 40,000 tokens learning the same three facts. In the traces this book draws on, that rediscovery is most of the startup cost, for knowledge that changes monthly.

Repositories have natural semantic structure

Any codebase that has survived contact with production organizes into domains (Payments, Authentication, Billing, Notifications, Infrastructure) whether or not the directory tree admits it. You can recover those domains automatically. Embed every file and module summary, cluster the embeddings, then cross-check the clusters against the dependency graph from part 4. Community detection over the import graph (Louvain, Leiden) finds the same boundaries from pure structure.

○ CLUSTERS AND GRAPH DISAGREE

- A module imports across what the text says is a boundary
- One directory holds two unrelated vocabularies
- Treat it as a debt list, ranked by how many edges cross

● CLUSTERS AND GRAPH AGREE

- Semantics and structure name the same set of files
- That set is a real architectural boundary
- Give it a dossier and an owner, and retrieve it as one unit

This runs Conway's law backwards: the code's semantic clusters recover the team and domain boundaries that produced it. Once you infer a domain, give it a compact, versioned dossier. Entry points, core types, invariants ("money is integer cents everywhere"), owned tables, and the suites that cover it. Generate it once, refresh it incrementally on merge, and retrieve it at the start of a run by lookup rather than by exploration.

EVIDENCE · INDEX ONCE, QUERY CHEAPLY FOREVER

Microsoft Research's **GraphRAG** shows the economics of amortized semantic indexing. An LLM pass extracts an entity graph from the corpus once, the Leiden algorithm detects semantic communities, and community summaries are pre-generated at index time. At query time, answering corpus-level questions from those summaries won 70 to 80 percent of head-to-head comparisons against naive RAG on comprehensiveness and diversity, while using roughly **2 to 3 percent of the tokens per query** that hierarchical source-text summarization would require, because the expensive understanding was paid for once, up front.¹⁰, ¹¹ The same trade applies to code. Indexing is a fixed cost. Rediscovery inside every run is a tax.

Edge et al., arXiv:2404.16130 · Microsoft Research, 2024

What a dossier looks like

A dossier is the briefing a senior engineer gives a new hire, generated from code and stamped with its commit.

```
domain: billing
sha: 9f31c2
entry_points: [billing/api.py, billing/tasks.py]
core_types: [LedgerEntry, Invoice, RefundRequest]
invariants:
  - money columns are integer cents (lint: no-float-money)
  - outbound mail goes through the outbox, not the mailer
owned_tables: [ledger_entries, invoices, refunds]
covering_tests: [tests/billing/, tests/contract/test_ledger.py]
may_call: [payments, notifications]
# 280 tokens. Rebuilt when any path under billing/ merges.
```

What goes in the memory layer

- **Embeddings:** file, symbol, and doc-chunk vectors for fuzzy entry ("where do we throttle webhooks?") when the prompt carries no hard anchors.
- **Domain dossiers:** the 300-token expert briefing per domain, as above.
- **Architectural boundary map:** which domains may call which, with violations flagged by a check before the model proposes one.
- **Convention registry:** error-handling idioms, naming schemes, "we use the repository pattern here", extracted once and injected only when the diff is in scope.

This memory is infrastructure with a freshness contract, not a cache. It rebuilds incrementally from the merge queue, it carries the commit SHA it was derived from, and the same CI that validates your code invalidates it. That contract is what separates semantic memory from a stale wiki.

DO THIS WEEK

1. **Build the import graph and cut it into communities.** Parse the repository with tree-sitter or your language's own import resolver, load the edges into networkx or python-igraph, and run Leiden. The artifact is `domains.json`, one community id per module.
2. **Cross-check the communities against embeddings.** Summarize each file, embed the summaries, cluster them, and print the modules where the two methods disagree. That list is your boundary debt, and it is worth reading before you write a single dossier.
3. **Generate one dossier for your busiest domain.** Fill the fields above from the index, cap it at 300 tokens, and stamp it with the commit SHA. Done looks like a file a new engineer could read in a minute and act on.
4. **Rebuild dossiers in CI on merge.** Recompute only the domains whose paths changed. Done looks like the dossier SHA matching `HEAD` within one merge.
5. **Make the start of a run a lookup.** Resolve the touched paths to a domain, inject that dossier, and remove the instruction that told the agent to go exploring.

MEASURE IT

- `run_startup_tokens`: tokens consumed before the first file edit. Sum the input tokens of every request that precedes the first write. Lower is better, and the drop after step 5 is the whole return on this chapter.
- `dossier_staleness_commits`: merges between the dossier's stamped SHA and `HEAD` for that domain. Lower is better, with 0 to 1 as the working target.
- `domain_agreement_rate`: share of modules where the embedding cluster and the graph community name the same domain. Higher is better, and the remainder is a ranked debt list.
- `dossier_hit_rate`: share of runs that retrieved a dossier for the domain they actually edited. Higher is better, and a low number usually means your path-to-domain resolver is wrong, not that the dossiers are.

TAKEAWAY

Understanding your repository is a build artifact, not a conversation.
Compute it once, version it, refresh it incrementally, and let every run start from knowledge instead of archaeology.

A rule written in prose cannot fail CI

Constraints belong in a representation a validator can check.

It starts with one file that tells the agent to stop repeating a mistake. Then a second for architecture, a third for conventions, a fourth for the things the first three left out. Six months later every run loads 30,000 tokens of prose, nobody can say which rules are still true, and the agent has started ignoring half of them. What you have is a second codebase written in a language with no compiler, no tests, and no dead-code detection.

Prompt bloat is measurable harm, not just cost

The instinct says instructions are free, and that the worst case is a model skimming the irrelevant ones. The research points the other way. Performance degrades as input grows even when the added material is task-relevant in spirit. Liu et al. showed models lose information positioned in the middle of long contexts,⁶ and Chroma's 18-model study showed degradation with input length across every frontier model tested, at different rates.⁷ Anthropic's context-engineering guidance is direct about the mechanism: attention is a finite budget, every token in context draws it down, and the goal is the *smallest set of high-signal tokens* that achieves the outcome.¹⁹ Thirty thousand tokens of standing instructions are not a safety net. They are attention spent before the task begins, and billed per request.

The caching economics make it worse

Prompt caching looks like it rescues the large standing prompt. Cache reads cost about 10 percent of base input price on Anthropic's API, with a 25 percent premium on writes, cutting cost by up to 90 percent and latency by up to 85 percent for long stable prefixes.¹⁶ The catch is that caching is prefix-exact. Edit one character of an early block and everything after it recomputes

at full price, plus the write premium. A pile of frequently edited prose concatenated into the prompt is the worst shape for this, because each edit invalidates the cache across your whole fleet. Caching rewards small, stable, ordered context. That is an argument for structure, not for volume.

○ RULES AS PROSE

- Loaded whole, in every run, relevant or not
- No validation, so rules rot silently as the code moves
- Conflicting rules resolved probabilistically by the model
- Every edit invalidates the prompt cache prefix
- Grows monotonically, and nobody dares delete a line

● RULES AS DATA

- Retrieved selectively, only the rules matching the touched domain
- Machine-checkable, because rules reference real symbols and run in CI
- Conflicts detected at build time
- Stable structured core caches cleanly, variable tail stays small
- Dead rules detected the way dead code is

What replaces the prose

- **JSON Schema or typed config** for anything that is a constraint: allowed dependencies, naming patterns, review requirements. A constraint in schema form is enforced by a validator rather than suggested to a model.
- **Tool metadata:** capabilities, costs, and preconditions declared on the tools themselves, the direction MCP standardizes, so capability discovery is a lookup instead of a paragraph.
- **Rule objects with scopes:** `{rule, applies_to: "billing/**", verified_by: "lint:no-float-money", since: "a1b2c3"}`. The retrieval layer injects the three rules that bear on this diff, not the three hundred that exist.
- **Generated docs:** the part 5 dossiers, derived from code, carrying their source SHA.

One rule, both ways

```
# before: a line in conventions.md, loaded in full in every run
# "Money is cents. Do not use floats for money."

# after: a rule object, injected when the diff touches billing/
{"id": "no-float-money",
 "applies_to": "billing/**",
 "verified_by": "lint:no-float-money",
 "since": "a1b2c3",
 "text": "Money columns and money variables are integer cents."}
# 40 tokens in the prompt, and CI fails on a float literal under billing/.
```

The second form costs less, arrives only when it applies, and has a witness. If the lint rule stops firing because the code moved, you learn that the rule is dead instead of carrying it for another year.

Order the prompt so the cache survives your edits

```
prompt = [
  schema_snapshot,    # stable for days
  domain_dossier,     # stable until that domain merges
  matched_rules,      # 3 of 300, changes per task
  task_slice,         # changes per task
]
# Edits land in the tail. The cached prefix survives them.
```

Keep prose for what prose is good at: intent, history, and taste. Anything that functions as a rule belongs in a representation you can validate, scope, version, and retrieve selectively. If a rule cannot fail CI, it is a preference.

DO THIS WEEK

1. **Count what you are loading.** Run every standing prose file through a tokenizer (tiktoken, or your provider's token-count endpoint) and print a table sorted by tokens. The number at the bottom is what each run pays before it reads any code.
2. **Sort each line into constraint, context, or history.** Constraints move to a rule file. Context moves to a generated dossier. History stays in prose and leaves the prompt.
3. **Give your three most repeated constraints a check.** An ESLint or Ruff rule, a JSON Schema validator, or a dependency-cruiser rule. Done looks like the check failing on a violation you introduce on purpose, then passing when you revert it.
4. **Inject rules by glob.** Match `applies_to` against the paths in the diff and pass only the matches. Done looks like a prompt carrying 3 rules on a billing change rather than the full registry.
5. **Reorder the prompt.** Stable structured blocks first, task slice last, so an edit lands in the tail and the cached prefix holds.

MEASURE IT

- `prompt_tokens_static`: tokens loaded before the task-specific slice. Read it from the request log. Lower is better, and it should fall in steps as files move out.
- `rules_injected_per_task`: rules in the prompt divided by rules in the registry. Lower is better, and a ratio near 1 means your scoping is not working.
- `rule_enforcement_rate`: share of rules whose `verified_by` check runs in CI. Higher is better. Rules without a check are the ones that rot.
- `cache_read_ratio`: cached input tokens divided by total input tokens, taken from the provider's usage fields. Higher is better, and a sudden drop means somebody edited an early block.

TAKEAWAY

Natural-language configuration is configuration without a type system. Move constraints into structured, validated, selectively retrieved data, and let prose go back to being prose.

Context compression is worth more than a bigger model

Removing irrelevant information beats buying a larger model.

An agent underperforms on your repository, and the available levers look like this: a bigger model, a longer window, a larger reasoning budget. All three cost money and none of them change what you are sending. The question that predicts success is not how much the model can hold. It is what fraction of what it holds bears on the task, and that is a question you can answer with instrumentation you already own.

The Shannon view of a context window

Shannon's framework ¹⁸ gives the right vocabulary. The task has an intrinsic information requirement, the minimal set of facts needed to produce the correct patch. Everything else in the window is noise relative to that task, and attention is the finite channel the signal has to pass through. Two consequences follow. First, mutual information matters more than volume: a 4k-token context where every line bears on the task carries more usable information than 100k tokens where the answer is diluted 25 to 1. Second, compression is safe when it is lossless with respect to the task, which is what deterministic slicing (parts 2 to 4) gives you and what generic truncation does not. A stack-trace parser that keeps your frames and drops framework frames is a task-aware compressor with a retention property you can state. `head -c` has no such property.

Four results point the same way

U-curve

LIU ET AL.:
ACCURACY COLLAPSES
FOR FACTS MID-
CONTEXT, LONG-
CONTEXT MODELS
INCLUDED⁶

18 / 18

CHROMA: EVERY
FRONTIER MODEL
TESTED DEGRADES AS
INPUT GROWS, EVEN
ON TRIVIAL TASKS⁷

-dozens of pts

NOLIMA: WHEN
MATCHES ARE
SEMANTIC RATHER
THAN LITERAL, MOST
MODELS FALL FAR
BELOW SHORT-
CONTEXT ACCURACY
BY 32K⁸

Related > random

POWER OF NOISE:
SEMANTICALLY
RELATED
DISTRACTORS HURT
ACCURACY MOST⁹

Read the four together for coding agents specifically. A repository dump is long, which triggers length degradation. It is full of near-duplicate distractors, which is the most harmful noise type the fourth result identifies. Using it requires semantic rather than lexical matching, the NoLiMa condition where the drop is steepest. A raw-context design sits at the intersection of all three.

COUNTERWEIGHT · LONG-CONTEXT IS IMPROVING

Be honest about the trend. Newer models post high scores on simple needle-in-a-haystack retrieval at extreme lengths, and vendors are attacking positional bias directly. The gains are strongest on literal retrieval, the easiest case, while semantic reasoning over long, distractor-dense context remains the weak point, which is the regime coding agents live in. And a model that holds its accuracy at length still bills you for every irrelevant token. Compression wins on cost even where it stops winning on accuracy.

Compression as an engineering discipline

- **Retrieval precision over recall by default:** start from verified anchors (part 3) and expand along graph edges (part 4) instead of dumping candidates.
- **Representation changes:** signatures instead of bodies (a repo map is roughly a 100 to 1 compressor¹²), schemas instead of migrations, sliced traces instead of logs.
- **Budgets enforced in code:** every context assembler takes a token budget and ranks content to fit it, and going over budget fails the build.
- **Context utilization as a metric:** what fraction of the provided tokens does the final patch depend on? In the traces this book draws on it often sits below 10 percent, and it is the clearest single indicator of wasted spend (part 13, and part 17 for attributing that spend to the

team that caused it).

A budget the assembler cannot talk its way out of

```
def assemble(task, budget=8000):
    blocks = rank(candidates(task))          # anchors first, then graph neighbors
    kept, used = [], 0
    for b in blocks:
        if used + b.tokens > budget:
            break
        kept.append(b)
        used += b.tokens
    missing = required(task) - {b.id for b in kept}
    if missing:                               # a required anchor did not fit
        raise BudgetError(task, missing, used, budget)
    return kept
# Over budget is a failure with a name, not a truncation nobody sees.
```

The failure matters more than the cap. When a required anchor does not fit, you want a stack trace naming the task and the anchor, because that is a retrieval bug. Silent truncation turns the same bug into a wrong patch and a confusing review.

DO THIS WEEK

1. **Record what you sent and what got used.** For each run, log the ids and token counts of every block you provided, then log the files the accepted patch touched. One JSON line per run is enough to compute utilization.
2. **Replace one file dump with signatures.** Parse with tree-sitter and emit declarations rather than bodies for anything outside the edit site. The artifact is a repo map you can diff against the old prompt, token for token.
3. **Give the assembler a budget.** Add the cap and the named failure above. Done looks like a test that asks for an over-budget assembly and asserts `BudgetError`.
4. **Drop near-duplicates before they ship.** Run minhash or simhash over retrieved chunks and collapse anything above your similarity threshold. Related distractors are the ones the fourth result says cost you most.
5. **Re-run last week's failures at the new budget.** If accuracy holds with 40 percent fewer tokens, you have found the model upgrade you were about to buy.

MEASURE IT

- `context_utilization`: tokens of blocks the accepted patch depended on, divided by tokens provided. Higher is better. Below 10 percent means most of your bill is noise.
- `tokens_per_accepted_patch`: total input tokens across the run, divided by patches a human accepted. Lower is better, and it is the number to quote when somebody proposes a larger window.
- `distractor_rate`: share of retrieved chunks the final patch does not reference. Lower is better, and a rising value usually means recall crept back into your retriever.
- `budget_overflow_rate`: assemblies that raised `BudgetError`, divided by assemblies. Lower is better, and each one is a retrieval bug with an address.

TAKEAWAY

A bigger model reads your noise at a higher price. A better system removes the noise. Only one of the two compounds: every point of retrieval precision you engineer carries over to every model you run after this one.

Agents guess at schema they were not shown

A model adds a duplicate column when the context did not carry the existing one.

Every team running coding agents against a real product meets the same bug report. The agent added a `user_email` column to a table that already had `email`. Or wrote a migration duplicating an index. Or built a query against a column renamed two quarters ago. The postmortem instinct is to call it a hallucination. The mechanism is duller: the model was not shown the schema, so it did what an engineer with no view of the database would do, which is guess.

Why schema is the worst-served context type

Schema knowledge resists text retrieval. The live truth is not in any one file. It is the fold of an ordered migration history: 400 migration files where the current shape of `orders` is the sum of 23 of them, interleaved with everything else. Grep for the table name and you get all 23 plus every query that touches it, which is a pile of maximum-similarity distractors, the exact noise profile part 7 showed is most damaging. Object-relational mapper (ORM) models drift from the database. Column semantics (`amount` is integer cents, not a float) live in tribal memory. Telling the agent to read the migrations folder does not fix this, because what it needs is a computed state rather than a document.

Schema becomes another graph node

The deterministic fix treats schema the way part 4 treats code, as structure to be indexed rather than text to be searched.

- **Schema index:** introspect the live database (or replay migrations) into a canonical, versioned snapshot of tables, columns, types, constraints, indexes, and foreign keys. This is a solved problem, and every ORM ships the machinery.
- **Migration graph:** each column carries its lineage, created in `0042`, renamed in `0187`, backfilled in `0201`. "Why is this nullable?" becomes a graph query with commit-linked answers.
- **Code and schema edges:** static analysis of ORM models and query sites links `LedgerEntry` to `ledger_entries` to the 14 call sites that write it. "What breaks if I drop this column?" becomes a traversal rather than a guess.
- **Domain mapping:** tables attach to the part 5 domains, so a billing task retrieves billing tables and only billing tables.

```
ctx = schema.slice_for(anchors=["RefundProcessor"])
# orders(id, user_id→users.id, amount_cents INT NOT NULL, status ENUM,...)
# refunds(id, order_id→orders.id, amount_cents INT, reason TEXT,...)
# lineage: refunds.amount_cents renamed from amount in 0187 (PR #3122)
# invariant: money columns are integer cents (lint: no-float-money)
# ~600 tokens. Verified against production at SHA 9f31c2.
```

Six hundred tokens of verified schema beat sixty thousand tokens of migration files. Because the slice comes from the live database, the duplicate-column failure also gets a second line of defence. The deterministic layer diffs the proposed migration against the index and rejects collisions before a human reviews them. That is the pattern of this series in miniature. Do not ask the model to remember reality. Show it reality, then check its output against reality.

The check that catches the duplicate column

```
$ schema-check migrations/0203_add_user_email.py
# reject: users.user_email collides with users.email (created 0042)
# lineage: users.email created 0042, indexed 0091, no rename since
# hint: 3 call sites already read users.email
# 1 collision, 0 warnings. exit 1
```

Run it in the same CI job that lints migrations. The failure names the column, the migration that created it, and the call sites that depend on it, which is enough for the agent to correct itself on the next turn.

Introspect the database, or replay the migrations

- **Introspect a restored copy** when production is the source of truth and hand-run DDL exists. You get the real shape, including the index somebody added at 2am. The cost is a restore in CI.
- **Replay migrations into a scratch database** when the migration history is authoritative and complete. It is cheaper and reproducible from the repository alone, but drift between the files and production stays invisible.

Pick one and stamp the snapshot with the SHA it came from. An unstamped snapshot becomes the stale wiki of part 5, with worse consequences.

EVIDENCE • GROUNDING BEATS RECALL

The general principle holds even where the schema-specific study does not exist yet. Retrieval grounding reduces fabrication compared with parametric recall, the precision of what is retrieved dominates its volume (the distractor results in parts 2 and 7^{7, 9}), and structured representations of a system outperform prose descriptions of it for machine consumption (parts 4 to 6^{12, 13}). Schema is the highest-stakes instance. It is the context type where a fabrication becomes a migration, and a migration becomes an outage.

Synthesis. See refs 7, 9, 12, 13

DO THIS WEEK

1. **Produce one snapshot.** Run your ORM's introspection (SQLAlchemy reflection, Django `inspectdb`, Prisma or Drizzle introspect) against a restored copy or a replayed scratch database. The artifact is `schema.json` with tables, columns, types, constraints, indexes, foreign keys, and the SHA it was built from.
2. **Slice it by anchor.** Given a symbol or a path, return the tables that symbol touches plus their direct foreign keys. Done looks like the 600-token slice above for your busiest table.
3. **Add the collision check.** Diff any proposed migration against the snapshot and fail on a column whose name, or an obvious variant of it, already exists on that table. Done looks like the check rejecting a migration you write on purpose.
4. **Link code to tables.** A static pass over ORM model definitions and raw query sites gives you table to symbol to call site. That index answers the drop-column question without anyone reading 400 files.
5. **Refresh the snapshot where migrations run.** Same job, same commit, so the stamp stays honest.

MEASURE IT

- `schema_slice_tokens`: tokens of schema context per task. Lower is better, with a few hundred as the working target. Compare it against the token count of the migration files the agent used to read.
- `schema_snapshot_age`: migrations applied since the snapshot's stamped SHA. Lower is better, and 0 is reachable once step 5 lands.
- `collision_rejections`: proposed migrations the check rejects per week. Watch the direction. A count that falls while agent volume holds steady means the slice is doing its job upstream.
- `unmapped_query_sites`: query sites the static pass could not link to a table. Lower is better, and each one is a place where the drop-column answer is still a guess.

TAKEAWAY

Your database's current shape is computable, versionable, and sliceable. If the agent guesses about schema, that is a missing index in your system rather than a failure of the model.

Why coordinator agents don't scale

Most multi-agent systems pay a coordination tax and call it architecture.

You split the work. A manager agent decomposes the ticket, three specialists execute, and the manager stitches the results back together. Then the token bill climbs, the wall clock stretches, and when the patch comes back wrong you read four transcripts to find which hand-off dropped the constraint. The org chart is not an architecture.

The tax, itemized

- **Token duplication.** Each sub-agent needs enough context to act, so you re-send and re-bill the shared background once per agent, then re-serialize every result back through the coordinator's window.
- **Latency amplification.** Coordinator → worker → coordinator turns serialize model calls. A hand-off costs seconds, not microseconds.
- **Lossy hand-offs.** Each hand-off compresses intent into natural language. Errors compound across hops.
- **Conflicts found at merge time.** Parallel agents cannot see each other's decisions. You pay for both branches, then pay a model call to adjudicate the collision.
- **Fan-out with no budget.** A misbehaving branch multiplies cost recursively, and an instruction written in prose does not stop it.

EVIDENCE · THREE INDEPENDENT SOURCES, ONE CONCLUSION

Anthropic runs multi-agent systems in production, so it is not a hostile witness. Its report: multi-agent systems consume **~15× the tokens** of chat, are economical only where task value justifies it, and are explicitly "not a good fit" for domains with tight dependencies between agents. Their example of a poor fit is coding.³

Cognition (Devin) published "Don't Build Multi-Agents." In their production experience, parallel sub-agents with fragmented context make conflicting assumptions, and reliability comes from a single agent with continuous, fully-shared context. Context engineering, not agent proliferation.⁴

Berkeley (Cemri et al., NeurIPS 2025) annotated 1,600+ traces from 7 popular multi-agent frameworks. Performance gains over single agents were often minimal. The team catalogued **14 recurring failure modes** in 3 classes: specification failures, inter-agent misalignment, and verification failures. Most stem from *system design*, not model capability, and better models will not fix them.⁵

Anthropic engineering, 2025 · Cognition, 2025 · Cemri et al., MAST (the Berkeley failure taxonomy), NeurIPS 2025

Deterministic orchestration, the boring alternative

Look at what the coordinator agent does all day: routing, sequencing, retrying, aggregating. None of those four needs judgment. All four have mature, deterministic machinery.

○ COORDINATOR AGENT

- Control flow decided per run by inference, so it differs every time
- State lives in a context window, and a crash loses it
- Retries, timeouts, and budgets improvised in prose
- Debugging means reading several transcripts and reconstructing the order

● STATE MACHINE / WORKFLOW ENGINE

- Control flow is code: explicit states, typed transitions, reviewable in a PR
- State is durable and replayable (Temporal-class engines, LangGraph-style graphs)
- Retries, timeouts, budgets, and circuit breakers are first-class primitives
- Debugging means inspecting a state machine's history

The pattern that works is *deterministic skeleton, probabilistic muscles*. A workflow engine owns the graph, localize → retrieve → patch → test → review, and calls the model only inside the nodes where judgment is required. Event-driven orchestration handles fan-out where sub-tasks are independent, the one regime where even Anthropic's data says parallelism pays.³ Where sub-agents do earn their keep, the honest framing is Cognition's: the reliable ones are mostly read-only context gatherers, closer to expensive tool calls than to collaborating colleagues.⁴

Written down, the skeleton fits on a page. Every line below is code you can read in a diff, and the budget is a number the engine enforces rather than a sentence the model may ignore.

```
workflow fix_ticket:
  budget tokens=120_000  wall_clock=8m  attempts=3

  localize : deterministic    # parser and code graph, no model call
  retrieve  : deterministic    # graph traversal, sliced to budget
  patch    : model             # judgment lives here
  test     : deterministic     # test runner, structured failure
  review   : model             # judgment lives here

  on test.fail  → retry patch with the sliced failure, up to attempts
  on budget.out → halt, persist state, page a person
```

That leaves one question worth asking per node: does this step need a model at all?

Use a workflow node when	Use a sub-agent call when
The step has a decidable rule (route by file type, retry on exit code 1)	The step needs judgment over ambiguous text
The step's output is typed and checkable	The output is prose a person or a test will grade
Later steps depend on this one's exact result	The sub-task is independent and read-only
You want the same answer twice	You can afford variance and will verify the result

DO THIS WEEK

1. **Draw the graph your coordinator improvises.** Take ten recent multi-agent runs, write down the node sequence each one actually followed, and count how many distinct sequences you get. More than two or three for the same task class tells you the routing is inference, not design.
2. **Move routing into code.** Replace the manager's decomposition prompt with a function that maps task type to a fixed node list. Done looks like a run whose node sequence you can predict before you start it.
3. **Port one task class to a workflow engine.** Pick a durable engine (Temporal-class) or a graph library (LangGraph-style) and encode localize, retrieve, patch, test, and review as typed states. Done looks like a replayable run history you can open after a crash.
4. **Put the budget in the engine.** Set a token ceiling, a wall-clock ceiling, and an attempt ceiling per run, and make the engine halt on breach. Done looks like a halted run with persisted state, not a surprise invoice.
5. **Demote the survivors.** Any sub-agent still standing should be read-only. If it writes files or decides control flow, it is a node in disguise.

MEASURE IT

- `coordination_token_share`: tokens spent on hand-off messages and re-sent background, divided by total tokens for the run. Tag each model call with its node, then sum the routing and aggregation nodes. Down is good.
- `handoff_count`: model calls whose input is mostly another model's output. Count them per completed task. Down is good, and a run with zero is a run with no telephone game in it.
- `path_entropy`: the number of distinct node sequences observed across runs of the same task class. Log the sequence on every run. Down is good, and 1 means the control flow is code.
- `budget_halt_rate`: runs the engine stopped on a ceiling, divided by all runs. You want this small and nonzero. Zero usually means no ceiling is set.

TAKEAWAY

Coordination is a solved, deterministic problem: state machines, workflow engines, queues. Spend inference on judgment inside the nodes, and keep it off the edges between them.

Tests should be first-class retrieval objects

Tests describe behavior in executable form, and a test runner's verdict is computed rather than inferred.

Your agent writes a patch, then runs the suite to find out whether it broke something. The tests that would have told it what the function was supposed to do were sitting in the repository the whole time, and it read none of them before it started. Most stacks treat tests as a thing you run after generating a patch, not a thing you retrieve before. That is backwards. The test suite is the best-maintained behavioral specification your repository has.

Behavioral indexing

Make tests queryable along the axes that matter:

- **Coverage mapping.** Run the suite under coverage once per merge and invert the result: for every function, the exact tests that execute it. `tests_covering("process_refund")` becomes a deterministic lookup.
- **Contract extraction.** Parse assertions into behavior statements. "A refund of a settled order returns `RefundResult(status=PENDING)`. A refund exceeding the order total raises `InvalidRefund`." Signatures told the model the shape (Part 4). Assertions tell it the meaning.
- **Fixture graph.** Fixtures encode how valid domain objects are constructed. Link them into the code graph and the model gets canonical object construction for free.

Now the retrieval plan from Part 3 gains a behavioral layer. Touching `process_refund` pulls its three covering tests, a precise executable account of current behavior, usually far smaller than the docs that would attempt to describe it.

The inversion is a few lines in CI, and the artifact it produces is a map from symbol to tests.

```
# once per merge, in CI
pytest --cov=app --cov-context=test --cov-report=json

# invert contexts: {symbol: [tests that executed it]}
index = defaultdict(list)
for fn, arcs in coverage["files"].items():
    for line, contexts in arcs["contexts"].items():
        for test in contexts:
            index[symbol_at(fn, line)].append(test)

tests_covering("process_refund")
→ ["test_refund_settled_order",
    "test_refund_exceeds_total",
    "test_refund_is_idempotent"]
```

EVIDENCE · TESTS AS SELECTION MACHINERY

Agentless does more than run tests at the end. Tests are load-bearing in its pipeline. It generates *reproduction tests* from the issue, uses existing *regression tests* to filter candidate patches, and re-ranks the survivors, a deterministic selection stage that the paper's ablations credit for a substantial share of final accuracy.¹ SWE-bench makes the broader point: the industry's shared yardstick for agent capability defines "solved" as nothing more or less than "the fail-to-pass tests now pass."¹⁵ If tests are how you judge agents, they should be how agents see.

Xia et al., FSE 2025 · Jimenez et al., ICLR 2024

What an assertion says that a docstring does not

Contract extraction is worth a worked example, because the output is smaller than the input and carries more.

- **THE DOCSTRING**

- "Processes a refund for an order."
- Written once, at creation
- Silent about the settled case, the over-total case, and idempotency
- No signal when it drifts from the code

- **THE EXTRACTED CONTRACT**

- settled order →
`RefundResult(status=PENDING)`
- amount > order.total → raises
`InvalidRefund`
- same idempotency key twice → one ledger entry
- CI reruns it, so a drift turns the build red

Execution-driven, contract-first loops

Tests also give the workflow skeleton (Part 9) its cheapest validators. A contract-first loop runs like this: retrieve the covering tests, have the model write or extend the failing test first so the intent becomes executable, have it patch, run the deterministic runner, then feed the structured failure (sliced per Part 2) into the next attempt. A compiler and a test runner grade each iteration. Their verdicts are computed from the code that ran, not inferred from a diff by another model emitting an opinion. The loop converges on green, and green is a fact about a process exit code.

DO THIS WEEK

1. **Turn on per-test coverage contexts.** Run the suite once with coverage.py recording a context per test (or your language's equivalent) and write the JSON to a build artifact. Done looks like one file that records which test touched which line.
2. **Invert it into a symbol index.** Map each covered line to its enclosing symbol with tree-sitter and store `{symbol: [tests]}` next to your code graph. Done looks like `tests_covering("process_refund")` answering in milliseconds.
3. **Put covering tests in the retrieval bundle.** When a node targets a symbol, attach its covering tests ahead of any prose documentation, under the same token budget. Done looks like a bundle you can diff against the old one and see docs give way to tests.
4. **Extract assertion contracts for your ten hottest symbols.** Parse the assert statements into one-line behavior statements and store them as graph properties. Done looks like ten symbols whose contracts a reviewer agrees with.
5. **Make the loop write the test first.** Change the patch node to emit a failing test before it emits a patch, and reject attempts that skip it. Done looks like every merged agent patch carrying a test that failed before it.

MEASURE IT

- `symbol_test_coverage_index_freshness`: hours since the coverage inversion last ran, compared against commits merged since. Down is good. A stale index points an agent at tests that no longer exist.
- `covering_tests_in_bundle`: share of retrieval bundles for an edit task that include at least one covering test for the target symbol. Compute it from the bundle logs. Up is good.
- `first_attempt_green_rate`: patches that pass the suite on attempt 1, divided by all patches. Up is good, and it moves when retrieval improves rather than when the model changes.
- `test_first_compliance`: agent patches that shipped with a test that failed before the patch, divided by all agent patches. Up is good.

TAKEAWAY

Index your tests the way you index your code. They are the documentation in your repository that executes, and the ones running in CI are current as of the last green run. That makes them the highest-signal retrieval objects you own.

Agents need working memory, not bigger context windows

You do not reread every book you own before fixing a bug, and your agent should not either.

At turn 60, your agent's transcript still carries the file it read at turn 3, the dead-end approach it abandoned at turn 12, and the stack trace it resolved at turn 20. You are paying for all of it on every subsequent call, and it is competing for attention with the two hundred lines that matter now. The window is being asked to be four things at once: attention, notebook, library, and diary. Human cognition separates those, with a small working set over vast indexed stores and forgetting as a feature. The append-only transcript is the opposite design.

A memory hierarchy for agents

- **Working memory** (the window). Only what the current step needs: the task frame, the active files, the last failure. Small by policy, not by accident.
- **Task memory**. Durable structured state for this job: the plan, the decisions made ("chose option B because of migration risk"), the files touched, the budget consumed. It lives in the workflow engine's state (Part 9), survives a crash, and pages in selectively.
- **Episodic memory**. Records of past runs. "We fixed a similar `DecimalConversionError` in March, and the fix was in the serializer config." Indexed by embeddings and entities, retrieved when relevant, absent otherwise.
- **Semantic memory**. The durable knowledge layer: the code graph (Part 4), domain dossiers (Part 5), the schema index (Part 8), test contracts (Part 10).

EVIDENCE · THE OS ANALOGY IS A WORKING ARCHITECTURE

MemGPT (Packer et al., Berkeley) made this concrete. Treat the context window as RAM and give the system OS-style virtual memory, paging information between the window and external storage with explicit memory-management operations. The approach let bounded-context models sustain coherent long-horizon behavior, including document analysis and multi-session conversation beyond the raw window, by managing what is resident rather than growing it.¹⁷ The same conclusion arrives from the failure side. Chroma's results imply that even within a large window, keeping stale content resident harms performance,⁷ and Anthropic's context-engineering guidance formalizes the remedies, naming compaction, structured note-taking outside the window, and just-in-time retrieval as the standard toolkit for long-horizon agents.¹⁹

Packer et al., arXiv:2310.08560 · Chroma, 2025 · Anthropic engineering, 2025

Context aging, eviction, retrieval scheduling

Once memory is tiered, management becomes ordinary systems engineering with ordinary policies. **Aging:** a file read 30 turns ago decays to its signature, and a resolved sub-task collapses to a one-line decision record. **Eviction:** superseded attempts and dead-end explorations leave working memory, recoverable from task memory if needed, no longer taxing attention. **Retrieval scheduling:** each workflow node declares its context contract, so the patch node gets code and contracts, the migration node gets schema and lineage, and the review node gets the diff and the invariants. Nothing rides along just in case, because just in case is the distractor mass the research says hurts most.

A context contract is a few lines of configuration per node. It is worth writing down because it turns an argument about what the model needs into a diff a reviewer can read.

```

nodes:
  patch:
    include: [task_frame, target_symbols, covering_tests, last_failure]
    max_tokens: 24000
  migrate:
    include: [task_frame, schema_slice, column_lineage, prior_migrations]
    max_tokens: 16000
  review:
    include: [task_frame, diff, invariants, covering_tests]
    max_tokens: 12000

on_overflow: drop lowest rank first, then fail the node

```

Aging needs a policy table rather than a paragraph, because each rule has to hold for a run you did not watch.

Item in working memory	Policy	Where it goes
File read, untouched for 20 turns	Decay to signature and docstring	Full text stays in semantic memory
Resolved sub-task	Collapse to a one-line decision record	Full transcript stays in task memory
Abandoned approach	Evict, keep the reason it failed	Task memory, one line
Superseded stack trace	Evict on the next green run	Run log, outside the window
Active target symbol	Pin until the node completes	Stays resident

Every one of those policies is deterministic. What pages in, what decays, and what leaves are decidable by code against declared contracts. The window stops being a place things accumulate and becomes the small, hot set of what this step needs. A structured memory of entities, relations, and decisions is also most of the way to a knowledge graph, which is where this series goes next.

DO THIS WEEK

1. **Measure what is resident.** Instrument one long run to log, per turn, the token count by category: task frame, active files, stale files, resolved failures. Done looks like a chart showing how much of turn 60 is turn 3.
2. **Write a context contract for one node.** Pick your patch node, declare its include list and a token ceiling, and reject anything outside the list. Done looks like a node whose input you can predict from its config.
3. **Add an aging rule.** Decay any file untouched for N turns to its signature, keeping the full text one lookup away. Done looks like the same run finishing with a smaller peak window and the same outcome.
4. **Move decisions into task memory.** When a sub-task resolves, write a one-line record to the workflow engine's state and evict the transcript from the window. Done looks like a crashed run resuming from its decisions rather than replaying the conversation.
5. **Add episodic recall behind a relevance gate.** Index past run summaries by entity and error type, and retrieve one only when the current error matches. Done looks like recall firing on a repeat incident and staying quiet otherwise.

MEASURE IT

- `resident_token_p95`: the 95th percentile window size across turns in a run. Log the prompt size per call. Down is good while outcomes hold.
- `stale_token_share`: tokens in the window belonging to items untouched for 20 or more turns, divided by total window tokens. Tag each item with its last-touch turn. Down is good.
- `context_contract_violations`: node inputs containing an item the node's include list does not name. Count them per run. Down is good, and zero means the contracts are enforced rather than advisory.
- `resume_success_rate`: crashed or halted runs that resume from task memory and finish, divided by all crashed runs. Up is good, and it is the clearest proof state lives outside the window.

TAKEAWAY

Scale memory, not context. The window is RAM: small, fast, and expensive. Everything else belongs in indexed storage with deterministic paging policies.

Knowledge graphs beat prompt engineering

Knowledge should be connected, not concatenated.

Your system prompt has grown to four thousand words because every time the agent missed a relationship, someone added a sentence describing it.

"`RefundProcessor` writes `ledger_entries`, which `ReconciliationJob` reads nightly, which the finance dashboard queries." In prose, that chain is three sentences the model has to re-derive by attention on every call. In a graph, it is three edges the system traverses before the model wakes up.

One ontology, many sources

The previous eleven parts each built a specialized index. The knowledge graph is where they federate into a single ontology over your engineering reality: **code graphs** (symbols, calls, imports, Part 4), **schema graphs** (tables, lineage, Part 8), **documentation graphs** (architecture decision records, or ADRs, and dossiers linked to the entities they govern, Parts 5 to 6), **test and contract graphs** (behaviors linked to covered symbols, Part 10), and **runtime graphs** (services, traces, and incidents from your tracing and incident systems). Entity relationships cross the boundaries: this function ↔ writes this table ↔ governed by this ADR ↔ covered by these tests ↔ implicated in that incident. No single tool holds those edges today, which is why an incident review opens with an hour of archaeology.

Why graph retrieval reduces reasoning complexity

When an answer requires multi-hop connection, flat retrieval makes the model do the hops. It retrieves chunks, holds them in attention, and infers the links: probabilistic work, paid in tokens, degraded by every distractor (Part 7). Graph retrieval does the hops in the system by

deterministic traversal, then hands the model a pre-connected subgraph. You have converted reasoning, which is expensive and variable, into lookup, which is cheap and exact. That is the thesis of this series applied to knowledge itself.

EVIDENCE • STRUCTURE WINS WHERE CONNECTION MATTERS

Microsoft's **GraphRAG** is the flagship result: an LLM-extracted entity graph, Leiden community detection, and pre-computed community summaries. On global sensemaking questions over corpora of roughly 1M tokens, it beat vector RAG with **win rates of about 70 to 80% on comprehensiveness and diversity**, and it matched full hierarchical source-text summarization while using roughly **2 to 3% of the tokens per query**, because connection-finding moved from query time to index time.¹⁰, ¹¹ Vector search finds things that sound like the question. Graph traversal finds things related to the answer.

Edge et al., arXiv:2404.16130 · Microsoft Research, 2024

COUNTERWEIGHT, GRAPHS ARE NOT FREE

Honest engineering requires the caveat. Naive graph retrieval can inflate prompts instead of shrinking them. Comparative studies measured some graph-RAG variants stuffing 40k to 100k tokens per query, orders of magnitude above vector baselines, and found that piling on more retrieved subgraph stopped improving answers while adding noise.²¹ Indexing also costs real money up front and has to be maintained. The lesson is not that graphs win everywhere. It is that a graph is an index for precise traversal, not a license to dump neighborhoods. Budgets and slicing (Part 7) apply to subgraphs the way they apply to logs. For code you are in the luckiest domain: unlike a prose corpus, this graph does not need LLM extraction, because compilers, parsers, and migrations give you most of the edges deterministically and nearly free.

A small ontology beats a large prompt

The practical migration path is short. Define a small ontology. Populate it from the deterministic sources you already built in Parts 4, 5, 8, and 10. Add embedding-based entry points for fuzzy queries, but make edges rather than similarity the backbone of expansion. Retrieval becomes

anchor (parsed, Part 3) → traverse (typed edges, bounded hops) → rank (centrality plus task relevance) → slice to budget. Every step is deterministic except the final generation.

```
entities: Function, Table, Service, ADR, Test, Incident, Domain
edges:    calls, writes, covered_by, governed_by, caused

anchor("process_refund")           # parsed from the ticket
  .out("writes")                   hops=1       → ledger_entries
  .in("reads")                     hops=1       → ReconciliationJob
  .out("governed_by")              hops=1       → ADR-031 (money is integer cents)
  .out("covered_by")               hops=1       → 3 tests
  .rank(centrality + task_relevance)
  .slice(budget=8000)              # subgraphs get budgets too
```

The difference that buys is worth stating as a comparison, because it is the reason the token bill moves.

Question: what breaks if I change the refund amount type?	Flat retrieval	Graph traversal
Finding the writer of <code>ledger_entries</code>	Hope a chunk mentions both	One <code>writes</code> edge
Finding the nightly reader	A second similarity query, unlinked	One <code>reads</code> edge
Finding the rule that governs the column	The ADR ranks low on lexical overlap	One <code>governed_by</code> edge
Who does the joining	The model, in attention, per call	The index, once, at write time
Cost of a wrong answer	A silent miss you find in production	A missing edge you can go add

DO THIS WEEK

1. **Write the ontology on one page.** Name seven entity types and five edge types, and reject anything you cannot populate deterministically this quarter. Done looks like a schema file, not a diagram.
2. **Load the two cheapest sources.** Emit `calls` and `imports` edges from tree-sitter or your compiler's index, and `writes` edges from your object-relational mapper's (ORM) introspection or from migration history. Done looks like a graph you can query for one real symbol.
3. **Link the governing documents.** Add a front-matter field to each ADR listing the entities it governs, and load `governed_by` edges from it. Done looks like a traversal that returns the integer-cents rule without anyone naming it.
4. **Attach your tracing and incident systems.** Map service names from OpenTelemetry resource attributes to graph nodes, and add a `caused` edge from each incident's postmortem to the symbols it names. Done looks like an incident reachable in one hop from the function that caused it.
5. **Put a budget on the traversal.** Cap hops, cap returned nodes, and slice to a token ceiling before the subgraph reaches the model. Done looks like a subgraph whose size you can state in advance.

MEASURE IT

- `edge_coverage`: symbols with at least one outbound typed edge, divided by all symbols in the repository. Up is good, and a low number tells you the graph is a demo rather than an index.
- `hops_to_answer`: the traversal depth at which the node the patch actually needed appeared. Log the anchor, the path, and the edited symbols. Down is good, and a rising number means an edge type is missing.
- `subgraph_tokens_per_query`: tokens of retrieved subgraph handed to the model per call. Down is good, and the counterweight above is the reason to watch it rather than assume it.
- `graph_query_share`: graph queries divided by graph queries plus exploratory file reads. Up is good, because it means the agent looks things up instead of foraging.

TAKEAWAY

Concatenation forces the model to rediscover relationships you already know. Store knowledge as a graph and retrieval becomes traversal, which moves connection-finding out of the token bill and into the index.

Measuring agent intelligence

Benchmarking on coding challenges misses what matters in production.

Someone asks how good your agent is, and the number you have is a benchmark score. It tells you one thing about one distribution of tasks under lab conditions. It does not tell you what a task costs, how long it takes, how much collateral change it makes, or whether it will do the same thing twice. Benchmarks measure capability ceilings. Production needs efficiency, precision, and repeatability, and those take different instruments.

EVIDENCE • ACCURACY-ONLY EVALUATION MISLEADS

The Princeton team showed this formally. Without cost as a co-equal axis, leaderboards reward scientifically meaningless spend, including retry loops and complexity that buy accuracy at any price, and the community drew wrong conclusions about why agents improved. Their prescription, jointly optimizing on a **cost-accuracy Pareto frontier**, revealed simple designs matching complex ones at a fraction of the cost.² Agentless made the same point empirically by reporting % *resolved, average cost, and average tokens together*, and separately audited SWE-bench Lite itself, finding problem-quality issues (such as issues whose text leaks the solution) that inflate naive readings of benchmark scores.¹ Anthropic's variance analysis completes the picture. If token usage explains ~80% of performance variance,³ then a score reported without token counts is mostly measuring budget, not intelligence.

Kapoor et al., TMLR 2025 · Xia et al., FSE 2025 · Anthropic, 2025

The production scorecard

files_touched / lines_changed

Surgical or scattered. Two agents both solve the ticket. One edits 2 files, the other rewrites 14. Blast radius is review cost, merge risk, and regression surface.

tokens_consumed_per_task

The direct cost of thinking. Tracked per phase (retrieval, generation, repair), it tells you which part of the architecture is wasteful.

tool_calls and graph_queries

The exploration ratio. A high tool-call count with a low graph-query count means the agent is foraging probabilistically for what an index should hand it.

time_to_green

Wall clock from task start to passing CI. The metric people feel, and the one a latency-amplifying architecture (Part 9) quietly destroys.

test_pass_rate and first_attempt_rate

Not just eventually green but green in how many attempts. Each repair loop multiplies cost and proxies for context quality.

cost_per_completed_task

The number finance cares about, inclusive of retries, abandoned runs, and human rework. Report it like cost of goods sold (COGS), because it is.

retrieval_precision

Of the context supplied, how much did the final patch depend on? Intersect provided spans with edited and read spans. The best single health metric for your deterministic layer.

context_utilization

The inverse view: tokens supplied against tokens that mattered. Utilization under 10% means you are paying attention-degradation costs (Part 7) for nothing.

variance / repeatability

Run the same task 10 times. Determinism-heavy systems cluster tightly on cost and outcome. Agent-heavy systems scatter. Variance is risk, and risk is cost.

These do not replace benchmarks. They are the instrument panel a benchmark cannot be. A capability score tells you which model to buy. The scorecard tells you whether your system is improving: whether this quarter's retrieval work moved precision from 8% to 40%, whether graph queries are displacing exploratory tool calls, whether cost per task is falling while pass rates hold. Optimizing these numbers is engineering. Optimizing a single leaderboard score tells you about the model you bought, not the system you built.

The scorecard on one page

Copy this into your runbook. Each row is a metric, the computation behind it, and the failure it usually points at.

Metric	How to compute it	What a bad reading usually means
<code>files_touched</code>	Count distinct files in the final diff, per completed task	Retrieval is too broad, so the agent edits what it read rather than what the ticket named
<code>tokens_per_task</code>	Sum prompt plus completion tokens per run, tagged by phase	One phase dominates. Repair usually means bad context, retrieval usually means no budget
<code>graph_query_ratio</code>	Graph queries divided by graph queries plus exploratory file reads	The index is missing edges, so the agent forages instead of looking up
<code>time_to_green</code>	Wall clock from task start to the first passing CI run	Hand-offs between model calls dominate, which is the Part 9 tax
<code>first_attempt_rate</code>	Tasks green on attempt 1 divided by all completed tasks	The context bundle lacks the contract, so the model guesses the behavior
<code>retrieval_precision</code>	Supplied spans that the final patch read or edited, divided by all supplied spans	The retriever is shipping neighborhoods rather than slices

Metric	How to compute it	What a bad reading usually means
<code>cost_per_completed_task</code>	Total spend including retries and abandoned runs, divided by tasks merged	Abandoned runs are uncouneted, and the true figure is higher than the one you quote
<code>outcome_variance</code>	Run one task 10 times, take the spread of cost and of pass or fail	Control flow is decided by inference, so the same input takes different paths

Cost per task is the number that travels furthest outside engineering, and it raises a question this part does not answer: which agent, which run, and which person does a given dollar belong to. Part 17 takes that up.

DO THIS WEEK

- Emit one span per run.** Wrap each agent run in an OpenTelemetry span carrying task id, model, prompt tokens, completion tokens, tool calls, and outcome. Done looks like a trace you can group by task class.
- Log the bundle and the diff together.** Record the spans you supplied and the spans the patch read or edited, so `retrieval_precision` becomes an intersection rather than an estimate. Done looks like one precision number for last week.
- Count abandoned runs in the cost.** Add halted, timed-out, and human-rewritten runs to the denominator of `cost_per_completed_task`. Done looks like a figure higher than the one you were quoting, and true.
- Run a repeatability probe.** Pick three representative tasks, run each 10 times on the same commit, and record the spread of cost and outcome. Done looks like a variance number you can put next to your pass rate.
- Publish the eight-row table weekly.** Post it where the team reads it, with last week's values beside this week's. Done looks like an argument about one row instead of an argument about the agent.

MEASURE IT

- `retrieval_precision`: supplied context spans the final patch depended on, divided by all supplied spans. Up is good, and it moves when your index improves rather than when the model changes.
- `cost_per_completed_task`: all spend, including retries and abandoned runs, divided by tasks merged. Down is good. Compute it weekly, because a monthly figure hides a bad week.
- `first_attempt_rate`: tasks green on the first attempt, divided by all completed tasks. Up is good, and each repair loop you remove pays twice, in tokens and in wall clock.
- `outcome_variance`: the spread of cost and pass or fail across 10 runs of one task on one commit. Down is good, and a wide spread tells you inference is deciding something code should decide.

TAKEAWAY

A benchmark score measures the model's ceiling. Tokens, tool calls, blast radius, time to green, and retrieval precision measure your architecture. One of those is under your control, so instrument it.

Give every agent its own identity

You cannot set authority for, bill, or review something you cannot name.

A pull request lands overnight. The commit author is a shared bot account that six agents and two scripts use. The change is fine, but the review question is not about the diff. It is: which agent did this, who started it, and what was it allowed to do? If the answer takes an afternoon of reading logs, the agents are doing work that nobody in particular is accountable for.

What parts 1 to 13 left open

The first thirteen parts moved decisions out of the model and into code: what to retrieve, what to keep in the window, which step runs next. One decision is still open, and it is the one a second team will ask about. What may this agent do in systems other people own? The same principle answers it. Decide in the system, before the model is involved, and keep the decision where a person can read it. That starts with a name.

The familiar model and what it cannot answer

The familiar model is a shared credential. One token carries everything any agent might need, and every agent that holds it looks the same to the system on the other end. It is quick to set up. It also means three questions have no answer in the data:

- **Which agent acted?** The target system saw the token, not the agent.
- **On whose behalf?** The person who started the task is not in the call.
- **Under what authority?** The token's scope is the union of every task anyone ever planned, so it says nothing about this task.

Zero trust architecture gives the general form of the fix. NIST's definition assumes no implicit trust from network location or asset ownership, and it evaluates each request against policy for the specific subject and resource.²² An agent is a subject. It needs to be one the policy can tell apart from the other agents.

What an agent identity holds

An identity is a record, not a credential. It answers who this is and who answers for it:

Field	What it says	Example
<code>agent_id</code>	A stable principal, one per agent, never reused	<code>agt_refund_triage</code>
<code>operator</code>	The person accountable for this agent and its runs	Dana Okafor, payments platform
<code>harness</code>	What runs it, and at which version	Claude Code 2.x, wrapped
<code>purpose</code>	One sentence a reviewer can check a run against	Triage refund failures and open a fix PR
<code>environment</code>	Where it may work	staging, <code>billing/**</code>
<code>status</code>	Enrolled, held, or retired, with a date	enrolled 2026-08-04

Three properties matter more than the fields. The identity is **separate from any connection credential**, so rotating a GitHub token does not change who the agent is. It **carries the initiator through**: each run records the person who started it, so "on whose behalf" is data, not inference. And it can be **revoked alone**: holding one agent does not stop the other five.

○ ONE SHARED CREDENTIAL ○ The target system sees one actor for every agent ○ The initiator is not recorded ○ Scope is the union of all planned tasks ○ Revoking it stops everything that uses it	● ONE IDENTITY PER AGENT ● Each call names the agent that made it ● Each run names the person who started it ● Authority attaches to the identity (part 15) ● One agent can be held while the others work
--	---

EVIDENCE · EXCESS AUTHORITY IS A DESIGN PROPERTY

OWASP's Top 10 for LLM applications lists excessive agency as its own category. It breaks the cause into three parts: excessive functionality, excessive permissions, and excessive autonomy. Each is a property of how the system around the model was built, not of the model.²³ The Berkeley failure taxonomy from part 9 points the same way: most recorded multi-agent failures traced to system design, including specification and verification, and the authors do not expect better models alone to fix them.⁵

OWASP, Top 10 for LLM applications · Cemri et al., NeurIPS 2025

Start with an inventory

Most teams cannot list their agents, because agents arrive one team at a time. The inventory is a spreadsheet before it is a system. One row per agent, the six fields above, and two more columns: which credentials it can reach today, and who would notice if it stopped. The rows with no operator are the first finding. The rows that share a credential are the second.

WHERE OXAGEN FITS

In Oxagen each enrolled agent is a registered principal with its own identity and an accountable operator. That identity is distinct from any connection credential. Agents built by different groups, on different harnesses, appear on one Fleet page with their open requests, their spend, and their last run. Oxagen does not run the agent. A supported wrapper sits beside the harness, and the agent keeps running where it runs today.

DO THIS WEEK

1. **List every agent that can write to a system.** Ask each team. Include scheduled scripts that call a model. Done is one row per agent with an operator's name in it.
2. **Mark each shared credential.** For every credential, list the agents that can reach it. Any credential with more than one agent is a row in your backlog.
3. **Give each agent a stable id and a one-sentence purpose.** Put the id in the commit trailer, the user agent string, or the request header, whichever the target system records.
4. **Record the initiator.** Pass the id of the person who started the task into the run, and write it next to the agent id on every outbound call you log.
5. **Test a single revocation.** Hold one agent in staging. Confirm that it stops and that nothing else does.

MEASURE IT

- `agents_with_operator`: agents with a named, current operator, over all agents. Anything under 1.0 is an agent nobody answers for.
- `agents_per_credential`: the largest number of agents that share one credential. The target is 1.
- `attributable_actions`: logged write actions that carry both an agent id and an initiator, over all logged write actions. Track it weekly. It should rise as wrappers roll out.

TAKEAWAY

Authority, budget, and review all attach to a name. Give each agent its own identity and a person who answers for it, and keep both apart from the credentials it works through.

Write the mandate

Four decisions govern an agent. Most teams have made all four. Few have them in one place.

Ask what one of your agents is allowed to do and you will get four answers from four places. Security points at an IAM role. Finance points at a cloud budget alert. Engineering points at a prompt file and a tool list. The record of what happened is in a log platform under a fourth team's account. Each answer is right. None of them is about the agent, so nobody can read the whole picture, and nobody can change one part without asking who owns the others.

One object per agent

Set an agent's authority, budget, tools, and skills in its mandate. A mandate is one object per agent with four clauses. Three of them are written by the people who already own that decision. The platform keeps the fourth.

Clause	Who sets it	What it says
Access	Security	The identity the agent acts as, which systems it may request, which data it may read, and which actions it may request
Budget and rules	Finance	What it may spend, under which commercial terms, and the rules it must obey: approval thresholds, allowed vendors, decision rules

Clause	Who sets it	What it says
Equipment	Engineering	The tools and skills it may use, the business context it may read, and the steering it runs under
Record	The platform	What each run read, what it changed, what it cost, and which rule answered each request

The three writers can be three teams or one person with three hats. The structure does not assume an org chart. What it assumes is that each clause has one owner who can change it without a meeting, and that a reader can see all four at once.

Why it is one object and not four

Part 6 argued that a rule which cannot fail CI is a hope. The same holds here. Four settings in four systems cannot be checked against each other. A tool in the equipment clause that writes to a system the access clause does not list is a contradiction, and only a single object makes it visible. A budget with no rule for what happens at the limit is incomplete, and only a schema can say so.

A mandate is typed configuration. It lives in version control, it changes by pull request, and a validator runs on every change.

```

# mandates/agt_refund_triage.yaml
agent: agt_refund_triage
operator: dana.okafor

access:                                # owner: security
  acts_as: agt_refund_triage
  may_request:
    - system: github
      scope: "repo:acme/billing"
      actions: [read, open_pull_request]
    - system: postgres_staging
      scope: "schema:billing"
      actions: [read]

budget_and_rules:                      # owner: finance
  monthly_limit_usd: 400
  at_limit: hold_and_notify_operator
  rules: [no-push-main, release-branch-needs-owner]

equipment:                             # owner: engineering
  tools: [run_code_with_trace, schema_slice, tests_covering]
  skills: [billing-conventions]
  context_scope: ["domain:billing"]

record:                                # kept by the platform, not writable here
  retention_days: 400

```

The validator checks what a reviewer would otherwise check by eye: every tool's target system appears under `may_request`, every rule named exists, the operator is a current employee, and the budget has an `at_limit` action. Those are four deterministic checks, and none of them needs a model.

Lead with the clause the reader owns

A mandate is read by different people for different reasons. The security lead reads access first and wants to know which rule answers a request. The finance lead reads the budget and wants the number attributed. The engineering lead reads equipment, then the record. The operator reads all four, because the operator is the one who gets asked. Write each clause so its owner can review it without reading the other three, and so the operator can read them together.

COUNTERWEIGHT · SCOPE IS PART OF THE CLAIM

A mandate governs the actions that pass through the layer that checks it. An agent with a second path to a system, such as a token in an environment variable, is not governed on that path, whatever the mandate says. State the boundary when you describe the control: "for actions routed through the gateway." If a mode only records calls and does not block them, say recorded, not enforced. An auditor tests the claim as it is written, so write it as it applies.

WHERE OXAGEN FITS

The mandate is the unit Oxagen manages. Identity, knowledge scope, permitted action, commercial terms, and the audit record are one typed contract. For actions routed through Oxagen, that contract is checked when the agent calls, not reconstructed after an incident. An agent outside that boundary, or a call that does not pass through Oxagen, is not governed by it. Observe mode is recorded, not enforced.

DO THIS WEEK

1. **Pick one agent and fill in the template.** The field kit has a blank mandate. Write what is true today, not what should be true. Done is four clauses with an owner's name on each.
2. **Find the contradictions.** List every tool the agent can call and the system it touches. Any system missing from the access clause is either an access gap or a tool to remove.
3. **Add an `at_limit` action.** Decide what happens when the budget is spent: hold the agent, notify the operator, or continue and flag. Any of the three beats finding out from the invoice.
4. **Put it in version control.** One file per agent, a `CODEOWNERS` entry per clause, and a schema check in CI.
5. **Write down the boundary.** One sentence: which paths to which systems this mandate governs, and which it does not.

MEASURE IT

- `agents_with_mandate`: agents with all four clauses filled and owned, over all agents from the part 14 inventory.
- `mandate_contradictions`: validator failures on the current set of mandates. The target is 0, and a rising number after a tooling change is the early signal.
- `ungoverned_paths`: known routes from an agent to a system that bypass the checking layer. Count them and list them, even if you cannot close them yet.

TAKEAWAY

Access, budget and rules, equipment, and the record are four clauses of one object. Give each clause an owner, keep them in one validated file per agent, and state which actions the mandate covers.

The agent asks, a rule decides

Authority is a decision at the moment of use, written down by the team that owns the system.

An agent needs to push a fix to a release branch at 2 a.m. Someone decided, at some point, whether that is fine. Where is that decision? If it is "the token has write scope," then the decision was made once, for every branch, for every task, by whoever created the token. When a team adds an agent, where does its authority get set, and who sets it?

A request, not a standing grant

The alternative to a broad credential is a request at the moment of use. When a task needs a system, a scope, or an action, the agent asks for it. The request carries four facts, and each one is data your system already has:

- **Who started the task.** The initiator from part 14.
- **Which agent is asking.** Its identity.
- **What it wants.** The tool and the action: `github.push`.
- **What the action would reach.** The resource: `acme/billing`, branch `release/2026.09`.

This is the prompt parsing of part 3 applied to authority. The request is structured input. A rule is a deterministic function of it. No model decides whether the agent may act, in the same way no model decides which files to retrieve.

Three answers

The team that owns the system writes the rule, and a rule has three possible answers.

Answer	What happens	Example
Allowed	The action proceeds inside the same call. The record keeps the rule id.	Read a public repository
Denied	The action does not reach the code that would have done it. The agent gets a reason it can act on.	Push to <code>main</code>
Routed to a person	The run waits. A named person answers. The run resumes with the answer in the record.	Push to a release branch

A rule that allows everything is still a rule. Someone wrote it, someone can read it, and someone can change it. That is the difference between an allowed request and a standing grant: the first has an author and a row.

```
# rules/release-branch.yaml          owner: release engineering
id: release-branch
when:
  action: github.push
  resource.branch: "release/*"
effect: route
route_to: role:release-owner
timeout: 4h
on_timeout: deny

# rules/no-push-main.yaml
id: no-push-main
when: { action: github.push, resource.branch: main }
effect: deny
reason: "Push to a branch and open a pull request."
```

Write the denial for the agent

A denied request is input to the next step, so write the reason as an instruction. "Request denied by rule `no-push-main`. Push to a branch and open a pull request." An agent that receives that sentence opens a pull request. An agent that receives `403` retries, then tries another path. Part 2 made the same point about tools: what the interface returns decides what the model does next.²⁰

The default is a decision too

Every rule set has a request that matches nothing. Decide what happens then, and say it on the same page as the rules. "When no rule matches, deny" and "when no rule matches, identity alone answers" are both defensible. An unstated default is the one to avoid, because the first person to learn it will be learning it during a review.

Where the credential lives

For a connection the platform mediates, the agent does not need the connection credential at all. The platform holds it, checks the request, and makes the call on the agent's behalf. The agent still has its own identity and whatever reaches its context, so this is not a promise about leaks. It is a statement about custody: the secret for that connection stays in one place, and each use of it is a row.

EVIDENCE · PER-REQUEST DECISIONS ARE THE ESTABLISHED PATTERN

NIST's zero trust architecture places a policy decision point and a policy enforcement point between every subject and every resource. Access is granted per request, under dynamic policy, and limited to what the task needs.²² OWASP's guidance for excessive agency recommends the same controls for LLM applications: limit the permissions an extension holds, execute actions in the context of the specific user, and require human approval for high-impact actions.²³ The three answers above are those recommendations in one mechanism.

NIST SP 800-207 · OWASP, Top 10 for LLM applications

COUNTERWEIGHT · ROUTING HAS A COST

Every routed request makes a person the latency. Route too much and approvals become a reflex, which is worse than a written rule. Review the routed requests monthly. A request the same person approves every time, without changes, is a candidate for an allow rule with a narrower condition. A request that is often rejected is a candidate for a deny rule with a better reason.

WHERE OXAGEN FITS

In Oxagen a decision rule names a capability, a condition, and one effect: allow, deny, or require approval. For governed calls, the kernel checks the rule after identity and entitlement and before the handler runs, so a denied action does not reach the code that would have done it. A routed request pauses the run. The person answers from the Access page, and the run resumes with the answer in the record. For mediated connections, Oxagen uses the connection credential on the agent's behalf and the agent does not receive it. When a workspace has no rules, identity alone answers requests, and the Access page says so.

DO THIS WEEK

1. **List the ten write actions your agents take most.** Pull them from the logs of part 14. For each, write the answer you want: allowed, denied, or routed, and to whom.
2. **Write the three rules that matter most as files.** Use the shape above. Give each an id, an owner, and a reason.
3. **State the default.** One sentence at the top of the rules directory that says what happens when nothing matches.
4. **Rewrite one denial message.** Pick the denial your agents hit most and make its text the next step.
5. **Route one action to a named person.** Set a timeout and an on-timeout answer. Run it once in staging and time how long the run waits.

MEASURE IT

- `requests_by_answer`: allowed, denied, and routed, per agent, per week. A sudden rise in denials after a deploy is a tooling change nobody told security about.
- `time_to_answer_p50` and `p95`: how long routed requests wait. This is the latency you are adding to `time_to_green` from part 13.
- `unmatched_requests`: requests answered by the default. Each one is a rule nobody has written yet.
- `retries_after_denial`: calls an agent makes after a denial before it changes course. A good reason brings this toward 0.

TAKEAWAY

Do not hand your agents the keys. Let the agent ask at the moment of use, let a rule the owning team wrote answer allowed, denied, or routed to a person, and keep the answer.

Spend you can attribute

A total is not an answer. The answer is which agent spent what, and on whose behalf.

The model invoice arrives and the number is larger than last month. You can see the total by API key and by day. The question from finance is different: which agent spent it, for which person, on which task, under which budget? Can you attribute last month's agent spend to the agent, the run, and the person who started the run? If the honest answer is an estimate built from a spreadsheet, the data to answer it was never recorded.

Why the invoice cannot answer

A provider bills the account that made the call. It has no field for your agent, your run, or your colleague, because you never sent them. Attribution cannot be reconstructed afterward from a total. It is recorded at the call, by the layer the call passes through, or it does not exist.

Part 13 made cost per completed task a first-class metric, and the Princeton work behind it showed why accuracy reported without cost misleads.² That chapter measured the architecture. This one measures the organization: the same rows, with four more keys on each.

The meter is a row

Write one row per priced step. A step is one model call or one tool call. Each row carries the keys that every later question groups by.

Column	Why it is there
person	Who started the task. The "on whose behalf" column.

Column	Why it is there
<code>agent</code>	The identity from part 14.
<code>run</code> , <code>turn</code> , <code>step</code>	Where in the work the cost fell. A run is one agent on one task. A turn is one prompt through to the point the agent stops. A step is one call.
<code>model</code> , <code>input_tokens</code> , <code>cached_tokens</code> , <code>output_tokens</code>	The quantities the price applies to. Cached reads price differently, so they need their own column. ¹⁶
<code>price_version</code>	Which price list was applied. Prices change, and a rate negotiated in March should not reprice February.
<code>cost_usd</code> , <code>cost_basis</code>	The amount, and whether it was <code>measured</code> , <code>reported</code> , or <code>estimated</code> .
<code>rule_id</code>	The rule that answered the request behind this step, from part 16.

The `cost_basis` column carries more weight than it looks. Measured means your layer saw the tokens and applied the price. Reported means the harness or the provider told you a figure. Estimated means you inferred it. A finance lead can work with all three as long as each one is labelled. A blended number with no label is the one that fails review.

Budgets belong beside the agent

A budget in a cloud console alerts someone after the money is spent, and it alerts on the account, not the agent. A budget in the mandate is checked before the next step and names the agent it stops. Three settings are enough to start:

- **A monthly limit per agent**, in currency, set by the person who owns the number.
- **A per-run ceiling**, so one run that loops cannot spend the month. Part 9 called this a circuit breaker, and this is where it gets its number.
- **An `at_limit` action**: hold the agent and notify the operator, route the next step to a person, or continue and flag.

The finance lead and the operator then read the same rows. One reads them grouped by cost center, the other grouped by run.

EVIDENCE · ALLOCATION COMES BEFORE OPTIMIZATION

The FinOps Foundation's framework treats allocation as a core capability: assign cost to the teams and workloads that incur it, using consistent metadata, so that owners are accountable for their own usage.²⁴ The order matters. A saving is hard to verify when nobody can say whose spend fell. Anthropic's production report gives agent teams a reason to care about the same order: on their internal evaluations, token usage alone explained about 80% of performance variance.³ Spend is a description of what the architecture did. Attributed spend says which part did it.

FinOps Foundation, FinOps Framework · Anthropic engineering, 2025

A worked example

One agent, one week, read three ways from the same rows:

Grouped by	What it shows	Who acts
agent	agt_refund_triage spent \$212 of a \$400 limit	The operator: on pace, no change
run	Two runs cost \$61 together. The median run costs \$1.90.	The engineer: both runs looped on a failing migration, so set a per-run ceiling and fix the schema slice from part 8
person	One person started 70% of the runs	Finance: charge the payments cost center, not platform

The figures are illustrative. The point is that three people took three different actions from one table, and none of them needed the invoice.

COUNTERWEIGHT · WHAT THIS DOES NOT CLAIM

Attribution is not savings. Recording who spent what does not lower the bill. It shows where the bill comes from, which is the precondition for any change you can verify afterward. If you later claim a reduction, state the workload, the baseline, the model and price versions, the sample size, the quality measure, and the costs including retries. Without those, the claim you can support is visibility.

WHERE OXAGEN FITS

In Oxagen every governed action is priced and attributed to the person, the agent, the run, the turn, and the step. The Spend page and the Run page read the same rows, and measured, reported, and estimated costs are labelled as such. The budget an agent runs under, what it has spent this month, and the rule that stops it sit beside the agent. This covers calls that pass through Oxagen. A harness whose model calls do not pass through the gateway reports its spend, or is not metered, and the page says which.

DO THIS WEEK

1. **Add four keys to your model-call log.** `person`, `agent`, `run`, and `step`. If you proxy model calls, add them there. If you do not, add them in the wrapper that starts the run.
2. **Separate cached input tokens.** Log them in their own column and price them at the cached rate.
3. **Version your price list.** A small table keyed by model and effective date. Stamp each row with the version used.
4. **Label the cost basis.** Go through each source of cost data and mark it measured, reported, or estimated.
5. **Set one per-run ceiling.** Pick the agent with the widest spread between median and maximum run cost. Set the ceiling at ten times the median and route to the operator when a run reaches it.

MEASURE IT

- `attributed_spend_ratio`: spend carrying person, agent, and run, over total model spend on the invoice. The gap is what you cannot explain.
- `cost_per_run_p50` and `p99`, per agent. A wide spread points at loops, and part 9 and part 11 are where to look.
- `unpriced_steps`: steps with tokens and no price, usually a new model missing from the price list.
- `budget_holds`: times an `at_limit` action fired, per agent, per month.

TAKEAWAY

Record the person, the agent, the run, and the step on every priced call, label how each cost was obtained, and keep the budget beside the agent it limits. Then the question of who spent what has a query for an answer.

Equip agents on purpose

A tool in the catalog is not a tool in the agent's hands. Assign equipment the way you assign access.

Your platform team has built the tools this book describes: a trace slicer, a code graph, a schema index, a test lookup. They are registered once and every agent gets all of them, along with forty others that accumulated. The refund agent can see the deploy tool. The docs agent can see the database tool. Nobody decided that. It is what happens when the catalog and the assignment are the same list.

The tool list is context, and it is the largest part

Every tool an agent can see is sent with every request: its name, its description, and its input schema. Part 7 showed what irrelevant context costs, and tool definitions are context like any other. One measurement from Oxagen's own in-app agent, taken by hand because nothing reported it: 271 tools came to 45,007 tokens, which was 92.4% of the cacheable prefix. Providers also cap the count. OpenAI's limit is 128 tools per request, and a request over the cap fails at the provider with an error about a payload nobody on your side can inspect.

Anthropic's guidance on writing tools makes the design point: more tools do not always lead to better outcomes, and it recommends building a few tools aimed at the workflows that matter most.²⁰ The assignment is an engineering decision. Make it in the mandate, where it can be reviewed.

Three kinds of equipment

Kind	What it is	Assigned by
Tools	Callable actions with typed inputs: <code>schema_slice</code> , <code>tests_covering</code> , <code>open_pull_request</code>	A list in the equipment clause
Skills	Packaged instructions for one kind of task, loaded when that task comes up, not on every request	A list in the equipment clause
Business context	The knowledge the work needs: domain dossiers from part 5, the graph from part 12, conventions, decisions	A scope, such as <code>domain:billing</code>

Skills follow the argument of part 6. A rule that applies to billing migrations should reach the agent when it touches a billing migration, and stay out of the window otherwise. A skill is that rule with a trigger.

Context scope follows the argument of part 12. A graph makes retrieval exact, and it also makes permission exact. If knowledge is nodes with domains attached, then "this agent may read `domain:billing`" is a filter on a traversal. The same filter serves relevance and authority, and you maintain it once.

Assignment is not permission

Equipment and access are different clauses for a reason. Assigning `open_pull_request` to an agent means the tool is in its hands. Whether a particular pull request may be opened is still a request that a rule answers (part 16). Keep both checks. The first keeps the window small and the agent on task. The second decides each action at the moment of use.

```

equipment:                                # owner: engineering
  tools:
    - run_code_with_trace                 # part 2
    - schema_slice                         # part 8
    - tests_covering                       # part 10
    - open_pull_request                   # each call still meets the rules
  skills:
    - billing-conventions                 # loads when a diff touches billing/**
    - migration-review
  context_scope:
    - "domain:billing"
    - "adr:*"
  tool_budget:
    max_tools: 24                         # a build failure over this, not a warning
    max_tokens: 6000

```

EVIDENCE · INTERFACES DECIDE OUTCOMES

SWE-agent's central result was that the agent-computer interface changes what the same model can do: purpose-built commands with compact, structured output raised resolution rates on SWE-bench over raw shell access.¹⁴ Anthropic's tool-writing guidance adds the operating advice: choose tools deliberately, namespace them, return high-signal output, and keep token use low by default, because an agent inherits the inefficiency of its tools.²⁰ The Model Context Protocol moves in the same direction by having each server declare its tools, their schemas, and their descriptions, so that discovery is a lookup.²⁵ A declared catalog is what makes a per-agent assignment possible.

Yang et al., NeurIPS 2024 · Anthropic engineering, 2025 · Model Context Protocol specification

Check that the equipment reached the work

An assignment is a claim until a run shows it. For each run, record which tools were offered, which were called, which skills loaded, and which context scopes were read. Two findings come out of that data quickly. Tools that are offered on every run and called on none are candidates to remove. Tools that an agent asks for and does not have show up as failed attempts or as workarounds through a shell.

COUNTERWEIGHT • CONTEXT IS SCOPED, NOT UNIVERSAL

Giving an agent business context does not mean it recalls it correctly, and it does not mean every agent should have it. Storing a fact is not the same as retrieving it at the right moment, which is why parts 3 to 5 exist. Scope context to what the agent's work requires and what its mandate permits, and measure retrieval precision for the context layer the same way you measure it for code.

WHERE OXAGEN FITS

In Oxagen, tools and skills reach an agent through its mandate. A tool in the catalog does not grant every agent permission to use it, and each request still meets the decision rules. Business context is served within the scope the mandate permits. Oxagen wraps four harnesses: Claude Code, Codex, Cursor, and Stella. What a wrapper can gate and meter differs by harness, and the product says which.

DO THIS WEEK

1. **Count the tools and the tokens.** For one agent, serialize the tool list exactly as it is sent and count both. Done is two numbers you did not have yesterday.
2. **Log offered against called.** Thirty days of runs is enough. Sort tools by calls per run.
3. **Remove the tools with no calls.** Take them out of that agent's assignment, not out of the catalog. Watch `first_attempt_rate` from part 13 for a week.
4. **Move one always-loaded rule into a skill.** Pick the longest block of instructions that applies to one kind of task. Give it a trigger and load it on demand.
5. **Add a tool budget to CI.** Fail the build when an agent's assignment exceeds a count or a token limit, or a provider's cap.

MEASURE IT

- `tool_tokens_per_request`: tokens spent on tool definitions, per agent. Compare it with the agent's median task context.
- `tool_use_ratio`: distinct tools called over tools offered, per run. A low ratio means the assignment is wider than the work.
- `skill_load_rate`: runs in which a skill loaded, over runs where its trigger matched. Under 1.0 means the trigger is wrong.
- `out_of_scope_reads`: context requests outside the agent's scope. Each one is either a scope to widen or a task the agent should not have.

TAKEAWAY

Give each agent the tools and skills its work requires, and the business context its mandate permits. Keep the catalog and the assignment as separate lists, and check the assignment against what the runs used.

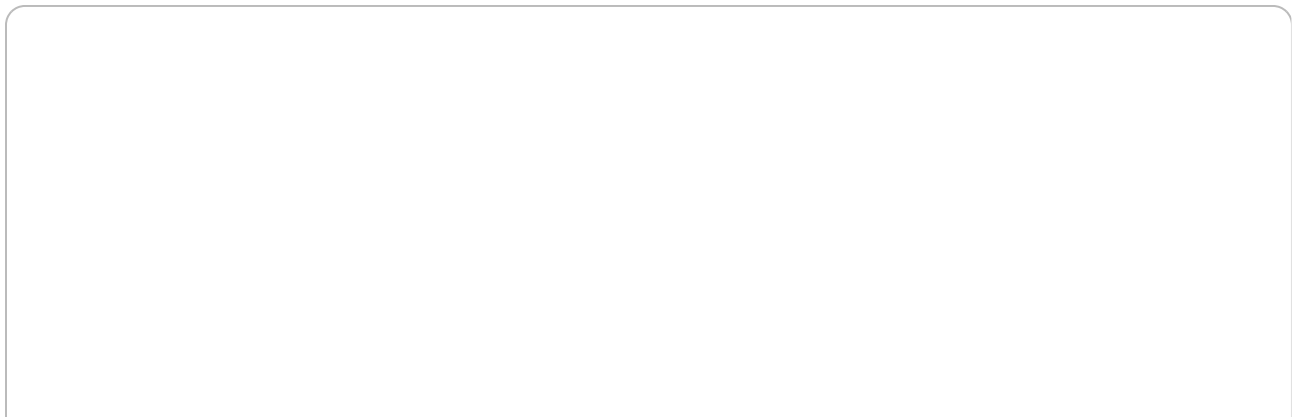
Keep a record another person can read

A log answers the engineer who wrote it. A record answers the person who was not there.

Something an agent did is being reviewed. It might be an incident, an audit sample, or a customer question. The person reviewing was not on the run and does not know your logging conventions. They ask four things: what did the agent read, what did it change, who or what allowed it, and what did it cost? When an agent requests an action in a system another team owns, which rule answers it, and where is that answer recorded? If those answers live in four tools and a chat transcript, the review becomes an investigation.

Logs and records are different artifacts

Logs are for debugging. They are verbose, unstructured at the edges, sampled under load, and deleted after a few weeks. All of that is correct for debugging. A record has a different reader and a different job. It is complete for the actions it covers, structured so a query can answer a question, ordered so the sequence is not in doubt, and kept as long as the accountability lasts.



○ A LOG

- Written for the author of the code
- Free text with some fields
- Sampled and rotated
- Answers "why did this break?"

● A RECORD

- Written for a reader who was not there
- One typed row per event, with stable keys
- Complete for governed actions, and retained
- Answers "what happened, and under whose authority?"

What a run's record holds

Parts 14 to 18 each produced rows. The record is those rows in one sequence, keyed by the run.

Event	Fields	From
Run opened	agent, operator, initiator, task, mandate version	Parts 14 and 15
Context read	what was retrieved, from which scope, at which source commit	Parts 3 to 5 and 18
Request	tool, action, resource, the answer, the rule id, and the approver when routed	Part 16
Step	model or tool call, tokens, cost, cost basis	Part 17
Change	files, rows, or resources written, with a diff or a reference	The tool that made it
Run closed	outcome, totals, and the checks that ran if the task was bounded	Part 20

The important property is that the request, the rule that answered it, and the cost of the step sit in the same row family under the same keys. A reviewer follows one action back to its authority without joining systems by timestamp.

Order you can check

A record that can be edited quietly is an account, not evidence. Two inexpensive mechanisms raise the bar. First, chain the rows: each row stores a hash of the row before it, so a removed or altered row breaks the chain at a known point. Second, hash a canonical form. Two serializers can emit the same JSON object with keys in a different order, and the hashes will differ. RFC 8785 defines a canonical JSON serialization for this purpose, so that the same data yields the same bytes and the same digest everywhere.²⁶

```
row = {
  "run": "run_7f3a", "seq": 41, "kind": "request",
  "agent": "agt_refund_triage", "initiator": "dana.okafor",
  "action": "github.push", "resource": "acme/billing@release/2026.09",
  "answer": "routed", "rule": "release-branch", "approver": "priya.n",
  "prev": "sha256:9c1e..."           # digest of row 40
}
digest = sha256(canonicalize(row))    # RFC 8785, then hash
# At close, sign the last digest. A reviewer with the export can
# recompute the chain without access to the system that wrote it.
```

This shows integrity, and only integrity. A valid chain means the rows are the rows that were written, in that order. It does not mean the agent's work was correct, and it does not cover an action that never passed through the recording layer.

EVIDENCE · A SHARED WRITTEN RECORD CHANGES WHAT A REVIEW CAN DO

Google's site reliability practice treats the written postmortem as the unit of learning from an incident: a record of the impact, the actions taken, the root causes, and the follow-up, kept blameless so that people contribute what they know.²⁷ The practice depends on a timeline that people accept as accurate. With agents, the timeline is the hard part, because the actor cannot be interviewed and the transcript is long. Part 13 instrumented the run to improve the architecture. The same instrumentation, kept and ordered, is what a review reads.

Beyer et al., Site Reliability Engineering, chapter 15

Write the record for its three readers

- **The operator** wants the run as a sequence: what it is waiting on now, and what it did before that.
- **The security reviewer** wants requests grouped by answer and by rule, with every routed request showing who signed.
- **The finance reader** wants the same rows grouped by person and cost center.

One set of rows serves all three when the keys are consistent. Three separate exports, built at three different times, will disagree with each other, and the review will be about the disagreement.

COUNTERWEIGHT · A RECORD HOLDS WHAT PASSED THROUGH IT

State what the record covers. It holds governed activity: the calls that went through the layer that writes it. It does not hold what an agent did on a path around that layer, and it records an observe-mode call without having enforced anything on it. It also holds data that may be sensitive. Decide what is stored in full, what is stored as a digest, who can read which rows, and how long each kind is kept, before the first review asks.

WHERE OXAGEN FITS

For governed activity, Oxagen keeps what the run read, what it changed, what it cost, and which rule answered each request. Every governed request is a row: who started the task, which agent asked, which tool it wanted, what data it would reach, which rule answered, and who signed. It is the same row the meter prices. Oxagen calls each recorded event in a run a frame. Every frame is hash-chained to the one before it, and the seal signs the close of the run, so a reviewer with the export can check its integrity outside the product.

DO THIS WEEK

1. **Run a tabletop review.** Pick one agent action from last week. Give a colleague who was not involved 30 minutes to answer the four questions in the opening. Write down where they got stuck.
2. **Pick the run id and thread it through.** One id, created when the run opens, present on every log line, request, and cost row.
3. **Separate the record from the logs.** Give it its own store, schema, and retention, even if the first version is one append-only table.
4. **Chain the rows.** Add `seq` and `prev`, and canonicalize before hashing. Write the 20-line verifier at the same time, and run it nightly.
5. **Write the coverage sentence.** "This record covers X. It does not cover Y." Put it at the top of the export.

MEASURE IT

- `time_to_answer_review`: minutes for someone outside the team to answer the four questions for a sampled action. Measure it quarterly. This is the number the whole part exists to lower.
- `record_coverage`: write actions present in the record, over write actions seen by the target systems' own audit logs. The gap is the ungoverned path count from part 15, observed.
- `chain_breaks`: verifier failures per night. The expected value is 0, and any other value is worth a person's morning.

TAKEAWAY

Keep one ordered, typed record per run that holds the request, the rule that answered it, the change, and the cost under the same keys. Make its order checkable, and say what it covers.

Bounded tasks and ongoing work

Some work has an endpoint. Other work continues. Manage each in its own way.

You operate two kinds of agent and they tend to get one kind of management. The first fixes a bug, runs a migration, or writes a report. It starts, works, and stops. The second triages an inbox, watches a queue, or keeps dependencies current. It has no last step. Ask "is it done?" of the first and the question has an answer. Ask it of the second and the honest reply is that the question does not apply.

Tell the two apart first

	Bounded task	Ongoing responsibility
Shape	Has a start and an end state you can describe	Runs until someone stops it
Examples	Fix issue 4821, migrate a table, draft the quarterly report	Triage refunds, keep dependencies patched, answer tier-1 tickets
The question	Did the checks hold?	Is it inside its authority and budget, and what is it waiting on?
Managed through	Completion criteria set before the work starts	Authority, budget, and review points
Reviewed	At the end of the run	On a schedule, and when a threshold trips

The common mistake runs in one direction: inventing a finish line for ongoing work so that it fits a task tracker. An inbox agent with a "done" state will report done. That tells you nothing about the inbox.

Bounded tasks: define completion before the work starts

Part 10 made tests the ground truth of behavior and had the agent write the failing test first. Generalize that. Before a bounded run begins, write down the checks that decide completion, and fix them so the run cannot revise them.

- **A command that must pass.** `pytest tests/test_billing.py` exits 0.
- **A file or a diff condition.** The migration exists. No file outside `billing/**` changed.
- **A human decision.** The release owner approves the pull request.

Fixing the checks first matters for a plain reason. An agent that judges its own work can stop early or skip verification. The Berkeley taxonomy names this family of failures: premature termination, and no or incomplete verification, both recorded across the frameworks they studied.⁵ A check the run cannot edit removes the agent's opinion from the verdict. The verdict becomes a function of recorded results, and another person can recompute it from the record (part 19).

```
# done/issue-4821.yaml      written before the run, hashed when it opens
task: "Fix DecimalConversionError on POST /api/v2/refunds"
checks:
  - run: "pytest tests/test_billing.py::test_refund -x"
  - run: "ruff check billing/"
  - diff: { only_paths: ["billing/**", "tests/**"] }
  - human: { role: release-owner, question: "Safe to ship in 2026.09?" }
on_fail: { retries: 3, then: route_to_operator }
```

COUNTERWEIGHT • WHAT A PASSING VERDICT MEANS

A passing verdict means the specified checks held. It does not establish that every requirement was captured or that every action was correct. Your team decides whether those checks are enough for the task, and a thin set of checks passes thin work. When checks fail, the useful record is not just "failed". It is the reasons, and whether the agent continued, waited for a person, or reached its retry limit.

Ongoing work: authority, budget, and review points

An ongoing agent is managed the way you manage a standing responsibility held by a person. You do not ask whether they are done. You look at what they are allowed to do, what they have spent, what they are waiting on, and what they did since you last looked.

- **Authority.** The mandate from part 15, reviewed on a date, not only when something goes wrong.
- **Budget.** The monthly limit and the per-run ceiling from part 17, with the `at_limit` action stated.
- **Open requests.** The routed requests from part 16 that a person has not yet answered. This is the list an operator reads first each morning.
- **Review points.** A schedule (weekly for a new agent, monthly once it is stable) and a few thresholds that trigger a review early.

Review trigger	Reading	Usual cause
Denials rise week over week	<code>requests_by_answer</code>	A tool or a task changed and the rules did not
Run cost p99 leaves its band	<code>cost_per_run_p99</code>	A loop, often on a failing check or a stale index
Routed requests wait longer	<code>time_to_answer_p95</code>	The named approver changed roles, or the rule routes too much
Tool use ratio falls	<code>tool_use_ratio</code>	The work drifted away from the equipment

Review trigger	Reading	Usual cause
The operator changes	Inventory	Nobody has re-read the mandate since

The operator's week

Put together, the operating job is small and regular. Daily, answer what is waiting. Weekly, read the spend and the denials for each agent, and review the runs that tripped a threshold. Monthly, re-read each mandate with its three owners, retire the agents nobody can justify, and turn repeated approvals into narrower rules. The field kit has this as a one-page checklist. None of it requires reading transcripts, which is the test of whether parts 14 to 19 are in place.

WHERE OXAGEN FITS

Mission Control is where that week happens. Every agent enrolled in Oxagen is on the Fleet page with its mandate, its open requests, its spend, and its last run. An operator answers routed requests from the Access page, and an ongoing agent shows its authority, its open requests, and its spend this month with no finish line. Oxagen treats completion checks as an optional control for bounded tasks. They are not the definition of the product, and ongoing work does not need them.

DO THIS WEEK

1. **Sort your agents into the two columns.** Use the inventory from part 14. Anything you cannot place is doing both jobs and should be split.
2. **Write completion checks for one bounded task type.** Start with the one that reaches review most often. Three checks are enough: a command, a diff condition, and a human decision if the change ships.
3. **Fix the checks before the run.** Hash the file when the run opens and record the digest as the first row. Reject a run whose checks changed midway.
4. **Set two review triggers for one ongoing agent.** Pick from the table. Send the alert to the operator by name.
5. **Put the monthly mandate review on a calendar.** Thirty minutes, three owners, one agent at a time.

MEASURE IT

- `checks_held_first_attempt`: bounded runs whose checks held without a retry, over all bounded runs. This is `first_attempt_rate` from part 13, against criteria the run could not edit.
- `runs_closed_without_checks`: bounded runs that ended with no completion criteria at all. Each is a verdict by assertion.
- `open_requests_age`: the oldest unanswered routed request, per operator.
- `days_since_mandate_review`: per agent. Over 90 is a review that is due.

TAKEAWAY

Define completion when the work has an endpoint, and fix the checks before the run starts. When the work continues, do not invent a finish line. Manage its authority, its requests, and its spend on a schedule.

Better systems, not better models

You rent the model. You own the system around it, and the way you operate it.

A new model ships and your team spends a week deciding whether to switch. The benchmark says it is better. Whether it is better for you depends on things the benchmark did not measure: how much context you send, how you verify a patch, what each run costs, and what the agent is allowed to touch. If those are engineered, switching is a configuration change and a week of comparing scorecards. If they are not, switching means re-tuning prompts and hoping.

Where the advantage moved

Capability gaps between frontier models have tended to close within months. Prices per token have fallen, and open-weight models keep climbing the same benchmarks. Whatever you gain from using the strongest model today, a competitor can rent with one configuration change. Model choice is becoming procurement. What compounds is the work in this book:

- **Retrieval.** Parsed prompts, code graphs, schema indexes, and test contracts (parts 3, 4, 8, and 10). Each point of precision carries over to every model you run later.
- **Orchestration.** A deterministic skeleton with the model inside the nodes (part 9), and budgets and circuit breakers as primitives.
- **Memory.** Tiered, indexed, and paged by policy (parts 5 and 11), so knowledge accumulates across runs.
- **Tooling.** Interfaces that return sliced signal (part 2), assigned per agent (part 18).
- **Measurement.** The part 13 scorecard on every run, with cost attributed to the agent, the run, and the person (part 17).

- **Authority.** An identity per agent, a mandate with four owned clauses, and a rule that answers each request (parts 14 to 16).
- **The record.** One ordered account per run that another person can read (part 19).

Two halves of one argument

Parts 1 to 13 took decisions away from the model because code makes them cheaper and the same way each time: what to retrieve, what to keep, what runs next. Parts 14 to 20 took a second set of decisions away from the model for a different reason. What an agent may do, what it may spend, and whether it is finished are not the agent's decisions to make. They belong to the teams accountable for it. Both halves use one method. Make the decision in the system, before the model is involved, and keep it where a person can read it.

The asymmetry is what makes this a durable strategy. Model improvements reach everyone at once, your competitors included. System improvements are yours. They encode your repository, your schema lineage, your test contracts, your rules, and your history of runs. They also carry over: when a cheaper or stronger model ships, a system built this way swaps it in and keeps everything else.

EVIDENCE · THE NUMBERS FROM THIS BOOK, SIDE BY SIDE

A fixed pipeline solved SWE-bench Lite issues at \$0.34 each while agent-based systems of the same period spent roughly ten times as much for comparable results.¹ On Anthropic's internal evaluations, token usage alone explained about 80% of performance variance.³ Evaluated on cost and accuracy together, simple baselines matched elaborate agent architectures at a fraction of the price.² Each result points at the system, not at the model inside it.

Xia et al., FSE 2025 · Anthropic engineering, 2025 · Kapoor et al., TMLR 2025

A precedent

Databases stopped competing on raw storage and started competing on query planners. Networks moved from bandwidth to protocols. Compute moved from clock speed to scheduling. In each case the commodity layer kept improving, and the value moved to the systems that used it most efficiently and could account for what they did. Inference looks to be following the same curve.

COUNTERWEIGHT • MODELS STILL MATTER

None of this says the model is irrelevant. A stronger model raises the ceiling of what any system can do, and some tasks are out of reach below a capability threshold. The claim is narrower. Between two teams with the same model, the published results above favor the one with the better system, and that team can also say what its agents did.

DO THIS WEEK

1. **Score yourself.** Take the self-assessment in "How to use this book" again. Compare it with your first pass.
2. **Pick one part from each half.** One from parts 2 to 12 that lowers tokens per task, and one from parts 14 to 20 that answers a question someone outside your team has asked.
3. **Make the model a configuration value.** If switching models means editing prompts, list the prompts. Each one is a decision that belongs in the system.
4. **Book the 30, 60, and 90 day reviews.** The field kit has the plan. Put the three dates on a calendar now.

MEASURE IT

- `model_swap_cost`: engineer-days to evaluate and adopt a new model. It falls as decisions move out of prompts.
- `cost_per_completed_task` and `retrieval_precision` from part 13, tracked quarter over quarter. These two show whether the system is improving independent of the model.
- `time_to_answer_review` from part 19. It shows whether you can account for what the system did.

TAKEAWAY

Models are rented. Systems are owned. Build the layer that makes any model cheaper and more precise, and operate it so that you can say who did what, under which rule, at what cost.

Closing thought

Intelligence is expensive.

Determinism is cheap.

Spend the first only where the second cannot do the job.

Twenty-one parts, one discipline. Parse before you prompt. Index before you search. Traverse before you reason. Validate before you accept. Decide authority before the agent asks. Record what happened in a form someone else can read. Measure all of it.

The model is the most capable component in the stack and the most expensive one to call. Treat each call the way you treat any scarce resource: schedule it deliberately, feed it only what it needs, and account for what it did.

If this book changes one habit, make it this one. Before each model call your system makes, ask what part of it code could have done. Before each action your agent takes, ask who decided it may. In both cases the answer should already be written down.



Field kit

Templates and worksheets to copy. Print this section and fill it in with the people who own each line.

Every template here is referenced from a part of the book. They are starting points. Change the fields to match your systems, and keep the owners.

SHEET 1 • AGENT INVENTORY (PART 14)

One row per agent that can write to a system. A row without an operator is the first thing to fix.

Agent id	Purpose, one sentence	Operator	Harness	Bounded or ongoing	Credentials it can reach	Shared with
_____	_____	_____	_____	_____	_____	_____
_____	_____	_____	_____	_____	_____	_____
_____	_____	_____	_____	_____	_____	_____

SHEET 2 • MANDATE (PART 15)

One file per agent. Each clause has one owner who can change it by pull request.

```
agent:
operator:
boundary: "Governs actions routed through _____. Does not govern _____."

access:                                # owner:
  acts_as:
  may_request:
    - system:
      scope:
      actions: []

budget_and_rules:                      # owner:
  monthly_limit_usd:
  per_run_ceiling_usd:
  at_limit:                            # hold_and_notify | route_to_operator | continue_e
  rules: []

equipment:                             # owner:
  tools: []
  skills: []
  context_scope: []
  tool_budget: { max_tools: , max_tokens: }

record:
  covers:
  retention_days:
last_reviewed:                         # date, and the three owners who were present
```

SHEET 3 · DECISION RULE (PART 16)

```

id:
owner:
when:
  action:
  resource:
effect:                # allow | deny | route
route_to:              # a role or a named person, when effect is route
timeout:
on_timeout:            # deny | allow
reason: ""             # written as the agent's next step

# At the top of the rules directory:
default_when_no_rule_matches:  # deny | identity_alone

```

Write action	Calls per week	Answer you want	Routed to	Rule id
_____	_____	_____	_____	_____
_____	_____	_____	_____	_____
_____	_____	_____	_____	_____

SHEET 4 · SCOPED RULE OBJECT (PART 6)

```

{
  "rule": "Money is integer cents. No float arithmetic on amounts.",
  "applies_to": "billing/**",
  "verified_by": "lint:no-float-money",
  "since": "a1b2c3",
  "owner": "payments-platform"
}

```

A rule with no `verified_by` is a preference. Keep it in prose, or write the check.

SHEET 5 · DOMAIN DOSSIER (PART 5)

```
domain: billing
source_sha:           # the commit this was generated from
entry_points: []      # routes, jobs, and consumers
core_types: []        # with one line each
invariants: []        # each links to the check that holds it
owned_tables: []
covering_tests: []
may_call: []          # other domains
must_not_call: []
recent_decisions: []  # ADR ids
budget_tokens: 300
```

SHEET 6 · COST ROW (PART 17)

```
person, agent, run, turn, step,
model, input_tokens, cached_tokens, output_tokens,
price_version, cost_usd, cost_basis,      # measured | reported | estimated
rule_id, started_at
```

SHEET 7 · PRODUCTION SCORECARD (PART 13)

Metric	This quarter	Last quarter	Owner
cost_per_completed_task	_____	_____	_____
tokens_consumed per task, by phase	_____	_____	_____
retrieval_precision	_____	_____	_____
first_attempt_rate	_____	_____	_____
time_to_green	_____	_____	_____
files_touched per task	_____	_____	_____
Variance across 10 repeats of one task	_____	_____	_____

Metric	This quarter	Last quarter	Owner
attributed_spend_ratio	_____	_____	_____
agents_with_mandate	_____	_____	_____
time_to_answer_review	_____	_____	_____

SHEET 8 • COMPLETION CHECKS FOR A BOUNDED TASK (PART 20)

```
task: ""
checks:
  - run: "" # a command that must exit 0
  - diff: { only_paths: [] }
  - file: { exists: "" }
  - human: { role: , question: "" }
on_fail: { retries: , then: route_to_operator }
# Hash this file when the run opens. Record the digest as the first row.
```

SHEET 9 • THE OPERATOR'S WEEK (PART 20)

When	Do	Read
Daily	Answer routed requests. Look at anything held at a budget limit.	open_requests_age , budget_holds
Weekly	Read spend and denials per agent. Review runs that tripped a threshold.	requests_by_answer , cost_per_run_p99
Monthly	Re-read each mandate with its owners. Turn repeated approvals into narrower rules. Retire agents nobody can justify.	days_since_mandate_review , tool_use_ratio
Quarterly	Run a tabletop review with someone outside the team. Fill in the scorecard.	time_to_answer_review , sheet 7

By day	Engineering half	Operating half	You can now answer
30	Per-step token and cost logging. Trace slicing in one agent.	Inventory complete. Every agent has an operator and an id.	Where do the tokens go, and who answers for each agent?
60	Prompt parsing, a symbol index, a schema slice, and a test lookup.	One mandate per agent. Rules for the top ten write actions. A stated default.	What is each agent allowed to do, and which rule said so?
90	A workflow skeleton with budgets. Per-agent tool assignment. The scorecard filled in twice.	Attributed spend. A chained run record. Completion checks for one bounded task type. Review dates booked.	What did it cost, per agent and per person, and can someone else verify what happened?

ONE TASK THROUGH THE WHOLE SYSTEM

To see how the parts connect, follow one prompt. "The `POST /api/v2/refunds` endpoint throws `DecimalConversionError` in `billing/processors.py` after #4821 merged."

The run opens. The record stores the agent, the operator, the initiator, and the mandate version (parts 14, 15, and 19). The completion checks are hashed (part 20).

Code parses the prompt into a route, an exception class, a file path, and an issue, and resolves each against an index (part 3).

The graph expands the anchors one hop. The schema index adds the two tables. The test lookup adds three covering tests. The billing dossier adds its invariants. Everything is cut to an 8,000 token budget (parts 4, 5, 7, 8, and 10).

The workflow engine runs the failing test under a tracer, and the slicer returns about 1,200 tokens of frames and locals (parts 2 and 9).

The model writes the patch. This is the first step where judgment is needed, and the first large model call.

The runner executes the checks. On a failure, the sliced result feeds one more attempt, and working memory drops the superseded one (parts 10 and 11).

The agent requests `github.push` to a release branch. The rule routes it to the release owner, who approves. The push goes through a mediated connection (part 16).

Every step wrote a cost row with the person, the agent, and the run (part 17). The run closes, the checks held, and the chain is signed (parts 19 and 20).

The scorecard updates: tokens by phase, retrieval precision, attempts, time to green, and cost (part 13).

The model made one kind of decision in that sequence: what the patch should be. Code, rules, and people made the rest, and each of those decisions is in the record.



Glossary

The words this book uses in a fixed sense.

Term	Meaning here	Part
Agent	A program that calls a model in a loop and takes actions through tools. In parts 14 to 20, a registered principal with its own identity.	14
Agent-computer interface	The commands and output formats an agent works through. SWE-agent's term.	2
Anchor	An entity parsed from a prompt and resolved against an index: a path, a symbol, a route, an issue.	3
Bounded task	Work with an end state you can describe before it starts.	20
Check	One completion criterion for a bounded task: a command, a file or diff condition, or a human decision.	20
Clause	One of the four parts of a mandate, owned by one team.	15
Connection	A system reached on the agent's behalf. For a mediated connection, the platform holds the credential and the agent does not receive it.	16
Context utilization	Tokens the final output depended on, over tokens supplied.	7, 13
Cost basis	How a cost figure was obtained: measured, reported, or estimated.	17
Domain dossier	A short, generated briefing on one area of a codebase, stamped with its source commit.	5

Term	Meaning here	Part
Equipment	The tools, skills, and permitted business context assigned to an agent through its mandate.	18
Governed action	A call that passed through the layer that checks the mandate and writes the record.	15, 19
Identity	The agent's own principal, with an accountable operator. Separate from any connection credential.	14
Initiator	The person who started the task a run is working on.	14, 17
Mandate	The one object an agent works under: access, budget and rules, equipment, and the record.	15
Observe mode	Calls are recorded and not blocked. Recorded, not enforced.	15
Ongoing responsibility	Work with no last step, managed through authority, budget, and review points.	20
Operator	The person accountable for an agent and its runs.	14
Record	The ordered, typed account of a run, kept for a reader who was not there.	19
Request	An agent asking for a system, scope, or action at the moment of use.	16
Retrieval precision	Of the context supplied, the share the final patch depended on.	13
Rule	What answers a request: allowed, denied, or routed to a person.	16
Run	One agent, under one operator, on one task.	13
Skill	Packaged instructions for one kind of task, loaded when that task comes up.	18
Step	One model call or one tool call.	17
Turn	One prompt through to the point the agent stops.	17

Term	Meaning here	Part
Working memory	What is in the context window for the current step, kept small by policy.	11

About the author



Mac Anderson

Founder and CEO, Oxagen Inc. • Los Angeles, California

Oxagen is Mission Control for agent operators: assign identities, set authority, equip agents with tools and skills, and oversee their work through a shared control plane.

Oxagen does not run agents. It decides what the agents you run may do for the actions routed through it, hands them what they need, and keeps the record.

Mac has built coding agents since the first release of OpenAI's Codex. The first thirteen parts of this book come from that work and from the published research it cites. The operating parts come from a later lesson: once agents work, the questions change from how well they code to who answers for them.

oxagen oxagen.sh



Sources

Every empirical claim in this book traces to one of the sources below. Read the primary literature. It is better than any summary of it, including this one. Sources 1 to 21 support parts 1 to 13. Sources 22 to 27 were added for the operating parts in the second edition.

1. **Xia, Deng, Dunn & Zhang**. "Agentless: Demystifying LLM-based Software Engineering Agents." FSE 2025. [arXiv:2407.01489](https://arxiv.org/abs/2407.01489) · github.com/OpenAutoCoder/Agentless
2. **Kapoor, Stroebl, Siegel, Nadgir & Narayanan** (Princeton). "AI Agents That Matter." TMLR 2025. [arXiv:2407.01502](https://arxiv.org/abs/2407.01502)
3. **Anthropic Engineering**. "How we built our multi-agent research system." June 2025. anthropic.com/engineering/multi-agent-research-system
4. **Cognition (Walden Yan)**. "Don't Build Multi-Agents." June 2025. cognition.ai/blog/dont-build-multi-agents
5. **Cemri et al.** (UC Berkeley). "Why Do Multi-Agent LLM Systems Fail?" (MAST). NeurIPS 2025 Datasets & Benchmarks. [arXiv:2503.13657](https://arxiv.org/abs/2503.13657)
6. **Liu, Lin, Hewitt, Paranjape, Bevilacqua, Petroni & Liang**. "Lost in the Middle: How Language Models Use Long Contexts." TACL 2024. [arXiv:2307.03172](https://arxiv.org/abs/2307.03172)
7. **Hong, Troynikov & Huber** (Chroma). "Context Rot: How Increasing Input Tokens Impacts LLM Performance." July 2025. research.trychroma.com/context-rot
8. **Modarressi et al.** (LMU Munich & Adobe). "NoLiMa: Long-Context Evaluation Beyond Literal Matching." 2025. [arXiv:2502.05167](https://arxiv.org/abs/2502.05167)
9. **Cuconasu et al.**. "The Power of Noise: Redefining Retrieval for RAG Systems." SIGIR 2024. [arXiv:2401.14887](https://arxiv.org/abs/2401.14887)
10. **Edge et al.** (Microsoft Research). "From Local to Global: A Graph RAG Approach to Query-Focused Summarization." 2024. [arXiv:2404.16130](https://arxiv.org/abs/2404.16130)

11. **Microsoft Research**. "GraphRAG: New tool for complex data discovery." July 2024. [microsoft.com/research blog](https://microsoft.com/research/blog)
12. **Aider (Paul Gauthier)**. "Building a better repository map with tree-sitter" & repo-map docs. 2023 onward. aider.chat/2023/10/22/repomap.html · aider.chat/docs/repomap.html
13. **Zhang, Ruan, Fan & Roychoudhury (NUS)**. "AutoCodeRover: Autonomous Program Improvement." ISSTA 2024. [arXiv:2404.05427](https://arxiv.org/abs/2404.05427)
14. **Yang, Jimenez et al. (Princeton)**. "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering." NeurIPS 2024. [arXiv:2405.15793](https://arxiv.org/abs/2405.15793)
15. **Jimenez et al.**. "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" ICLR 2024. [arXiv:2310.06770](https://arxiv.org/abs/2310.06770)
16. **Anthropic**. "Prompt caching with Claude" (up to 90% cost and 85% latency reduction, and a cache read costs about 0.1 times the input price). 2024. anthropic.com/news/prompt-caching
17. **Packer et al. (UC Berkeley)**. "MemGPT: Towards LLMs as Operating Systems." 2023. [arXiv:2310.08560](https://arxiv.org/abs/2310.08560)
18. **Shannon, C.E.** "A Mathematical Theory of Communication." Bell System Technical Journal, 1948. The original framework for signal and noise
19. **Anthropic Engineering**. "Effective context engineering for AI agents." 2025. anthropic.com/engineering
20. **Anthropic Engineering**. "Writing effective tools for agents." 2025. anthropic.com/engineering
21. **Zhu et al.**. "When to use Graphs in RAG: A Comprehensive Analysis for Graph Retrieval-Augmented Generation." 2025. [arXiv:2506.05690](https://arxiv.org/abs/2506.05690)
22. **Rose, Borchert, Mitchell and Connelly (NIST)**. "Zero Trust Architecture." NIST Special Publication 800-207, 2020. doi.org/10.6028/NIST.SP.800-207
23. **OWASP**. "Top 10 for LLM Applications 2025," LLM06: Excessive Agency. genai.owasp.org
24. **FinOps Foundation**. "FinOps Framework," the Allocation capability. finops.org/framework
25. **Model Context Protocol**. Specification: servers, tools, and schemas. modelcontextprotocol.io/specification
26. **Rundgren, Jordan and Erdtman**. "JSON Canonicalization Scheme (JCS)." RFC 8785, 2020. rfc-editor.org/rfc/rfc8785

27. **Beyer, Jones, Petooff and Murphy (eds.).** "Site Reliability Engineering," chapter 15: Postmortem culture, learning from failure. O'Reilly, 2016. sre.google/sre-book
-

How to use this book. Each part stands alone, and the whole reads in order. Share the file, or excerpt a part with its evidence block and takeaway intact. The numbers cited reflect the referenced publications. Check them against the primary source before you quote them, because sources change. Figures in worked examples are illustrative and are labelled as such.

© 2026 Mac Anderson. Published with Oxagen Inc., Los Angeles. Second edition. Each edition is a single HTML file with no tracking. The field manual prints on US Letter, and the reader exports any part as a PDF.