mokata

The Complete Guide

The memory + seatbelt for your AI coding agent.

Spec-driven TDD for Claude Code — knowledge-aware, human-gated, local-first.

mokata — The Complete Guide

Contents

mokata — The Complete Guide

A deep, feature-by-feature tutorial for developers. Every command, gate, layer, and toggle — from `init` to a full governed story, with nothing left out.

⬇ Download this guide as a PDF — generated from this page in CI, so it always matches what you're reading.

> Audience: developers who want to use mokata to its full capacity. This is a **hands-on tutorial** — you'll install the full stack with **every optional dependency wired**, verify each capability resolves to a real provider (not a fallback floor), then take a **guided power tour** that exercises every layer with the commands to run and the output to expect.
>
> **Three ways to run mokata** (set up in §2): (1) the **Claude Code plugin**, (2) **`mokata setup claude`** — the same in-Claude-Code experience without the marketplace, and (3) the **CLI**. The guide is **CLI-led** because the CLI makes every gate visible step by step — but a `pip` CLI install is *terminal-only* (the engine with no LLM); to use mokata *inside* Claude Code you install the plugin or run `mokata setup claude`, and there the LLM drives these same operations. Each step maps to its **`/mokata:<name>`** slash command. Follow it top to bottom with a terminal open; the later Parts double as deep reference. Verified against the **current batch** — spot-run the commands as you go to confirm.
>
> **How to read this:** §0–1 are orientation. **§2 is the guided full-stack install** — do every step. **§3–15 are the deep tour** of each layer (run the commands as you go). **§16 is a complete end-to-end story** that ties it all together. §17–19 are reference you'll come back to.

Table of contents

---

0. Why mokata is different — and when to use it

Before the mechanics, the question that matters for a developer evaluating tools: *what does mokata give me that Cursor, superpowers, spec-kit, Greptile, or a memory tool doesn't?* The honest answer is **not any single capability — it's the integration**. Every individual piece exists somewhere in the market; the combination exists nowhere else.

The white space mokata fills

The AI-coding ecosystem (as of 2026) splits into three camps that **no one has combined**:

1. **Methodology tools** — obra/superpowers, GitHub spec-kit, TDD-Guard — enforce a spec/TDD/review *process*, but have **no codebase graph, no persistent memory, no token governance**.
2. **Context tools** — Cursor, Augment, Sourcegraph, Aider — have graphs/indexes and some memory, but **no opinionated spec-TDD methodology, no deep review**, and the deepest ones are **cloud-only, not local-first**.
3. **Review tools** — Qodo, CodeRabbit, Greptile, plus SAST incumbents SonarQube/Snyk/Semgrep — nail one stage, but **don't verify spec-completeness, aren't TDD-gated, and carry no memory or budget**.

And three whole categories are essentially **unserved by anyone**:

- **Enforced token governance.** Every serious tool *consumes* tokens heavily; the ecosystem only offers passive trackers (ccusage-style). There is **no widely-adopted tool that sets per-task budgets and *gates* execution**. mokata does (Part F + `mokata budget`).
- **Human-gated self-healing memory.** Zep auto-detects contradictions but has *no human gate*; ByteRover has a human gate but *no automatic detection*. **Nobody else ships "contradiction detected → here's the old→new diff → approve / edit / reject."** mokata does (C5).
- **Shared, structured project memory tied to a codebase graph.** Memory today is per-developer and shallow; team-shared, graph-

linked, provenance-carrying memory is open space. mokata targets it (C + B).

mokata's thesis: spec-driven TDD **+** codebase graph **+** shared self-healing memory **+** enforced token governance **+** deep multi-level review — all **human-gated** and **local-first**, on one auditable surface.

How mokata compares, at a glance

Legend: ● strong · ◑ partial/shallow · ○ absent.

Capability	superpowers	spec-kit	Cursor	Augment	Qod
Spec-driven completeness gate	◑				◑
Brainstorm-first (Socratic)		◑			
RED-before-GREEN TDD		◑			◑
Codebase graph			◑		◑
Self-healing memory			◑	◑	
Shared project memory		◑	◑	◑	
Token governance (*enforced*)			◑	◑	
Deep multi-level review (sec/arch/patterns)	◑	◑			
Human-gated writes (universal)			◑	◑	
Local-first / no telemetry			◑		◑
Configurable / composable	◑	◑	◑	◑	◑

No column except mokata's is mostly-● across all rows. The methodology tools cluster at the top; the context/review tools cluster in the middle. The bottom rows — enforced token governance, shared self-healing memory, deep review integrated into the loop, fully local-first — are where the whole field is weakest, and where mokata concentrates its differentiation.

The closest neighbor: mokata vs. superpowers

obra/superpowers is the best **methodology** plugin in the ecosystem and mokata's explicit quality bar — a Socratic brainstorm → spec → RED-GREEN TDD → subagent execution → review lifecycle, delivered

as auto-triggered skills. mokata **inherits that entire methodology clean-room** (nothing imported or copied) and wraps it in the layers superpowers deliberately leaves out:

1. **Navigate by a persistent codebase graph, not by re-grepping every session** — the agent asks *who calls this, what's the blast radius, what implements this* and gets exact identifiers back, instead of dumping files into context. superpowers has no graph or index at all.
2. **Persistent, self-healing, human-gated memory instead of disk files that rot** — decisions and conventions survive across sessions, are shareable across a team, and surface their own contradictions as an old→new diff you approve. superpowers' only durable state is design-doc and plan files on disk.
3. **Token & cost governance that *enforces*, not just observes** — per-task budgets that gate execution, graph/JIT retrieval over file-dumping, sub-agent summaries not transcripts, cache-stable prefixes. superpowers is, by its own architecture, token-heavy with no governance.
4. **Deep, multi-dimensional review tied to the spec** — spec-compliance with static AC→test traceability, security, architecture-fit, CS best practices — under the same gates and audit trail; pluggable review backends. superpowers' review is a fast inline self-review, not a deep spec-tied audit.
5. **One governed surface** — human-gate every durable write (code, memory, *and* config), a full audit ledger ("review every decision"), reversible + resumable, local-first. superpowers gives you a methodology; mokata gives you a methodology you can **govern and audit end to end**.

The one-line version: **superpowers makes the agent follow a good process. mokata makes the agent follow a good process *and* remember, navigate, govern, and prove it — on one governed, auditable surface.**

When *not* to reach for mokata (the honest part)

mokata doesn't claim to beat every tool at every tool's own game. Reach for something else when:

- **You want multi-harness coverage today.** superpowers runs on Claude Code, Cursor, Gemini, Copilot CLI, Codex, and OpenCode right now; mokata is **Claude-Code-first** (broader portability is later, demand-gated). If you live across many CLIs, superpowers fits better today.
- **You want zero-overhead, methodology-only.** If you have no need for a graph, memory, or budget governance, superpowers' lean "skills + disk files" model is lighter to adopt — much of mokata's value won't apply to you.
- **You need the deepest security SAST or a cloud context engine on a huge monorepo.** Semgrep/SonarQube (security depth) and Augment (cloud context scale) win on those specific axes; mokata can *adopt* a best-in-class reviewer as a backend under its gates, but it isn't trying to out-SAST the SAST incumbents.

mokata's defensible position is **integration + governance + local-first**, not any single feature. If what you want is that methodology *fused* with codebase structure, durable team memory, enforced cost

control, and end-to-end auditability — where you can review and walk back every decision the system made — that integration is what mokata is built to be, and nothing else delivers it.

---

1. What mokata actually is

mokata is a **spec-driven, test-driven development engine for Claude Code** with four things bolted in that most tools leave as add-ons: a **codebase knowledge graph**, **persistent self-healing memory**, **active token/cost governance**, and a **human-gated governance + audit layer**. The whole thing is local-first (zero telemetry), pure Python (≥ 3.10, no required runtime dependencies), Apache-2.0, and built clean-room.

The defining behavior: **no code ships until every acceptance criterion maps to a test (RED before GREEN), and every durable write — to code, memory, or config — is human-gated.** The completeness gate *blocks* emit; it never silently passes.

It helps to hold the architecture as twelve lettered Parts — you'll see these referenced throughout, and they map to the source tree under `src/mokata/`:

Part	Area	What lives here
A	Spine	manifest, capability router, tool detection + graceful degradation, bootstrap briefing, `init`
B	Knowledge	adopted code graph + grep floor, five typed queries, incremental index + staleness, drift anchors
C	Memory	persistent / decision / episodic, self-healing by surfacing, pluggable backends, consolidation
D	Engine	7-phase pipeline, provable completeness gate, AC-mapper, pre-mortem, spec-compliance review, dry-run
E	TDD & execution	RED-before-GREEN, model routing, bug/debug/optimize engines, execution-mode selector
F	Token governance	tracker, JIT retrieval, handback caps, output density,

		savings budget, cache-stable prefixes
G	Rules & governance	4-tier rules, taxonomy, sync/async hooks, Karpathy gates, rule-learning, skill authoring
H	Adapters/harness	capability coverage, MCP discovery, the cross-harness boundary
I	Safety & audit	4-layer secret protection, human-gated writes, audit ledger, lethal-trifecta gate, revert, resume
J	Distribution	plugin/marketplace packaging, shareable stack manifests
K	Config	per-layer/tool toggles, profiles, local-first, committed config, trust dial, doctor, reset
L	Composability	standalone commands, mid-pipeline entry, direct skills, catalog, chaining, suggestions

---

Two names to keep straight: **mokata** is the framework; **MoStack** is the brand/marketplace it ships under (`mokata@mostack`).

---

2. Guided setup: from zero to a fully-wired `full` stack

This is the hands-on heart of the tutorial. By the end of §2 you'll have mokata installed, **every optional dependency wired**, the `full` profile initialized, and `mokata status` confirming each capability resolves to a *real* provider rather than a fallback floor — so the power tour in §3–16 runs at full strength.

mokata is designed so **nothing here is mandatory**: the core installs with zero dependencies and every missing tool degrades to a built-in floor. We install everything anyway, precisely so you can *see* the difference the richer providers make.

2.0 What "full" actually wires

The `full` profile declares the richest fallback chain for each capability. Here's every provider, what it gives you, and exactly how mokata detects it (this is the real detection catalog from `src/mokata/profiles.py`):

Capability	Provider (in precedence order)	Gives you	Detected by
`code_graph`	`code-review-graph`	persisted graph: callers/callees/blast-radius, incremental	`code-review-graph` on `PATH`
	`serena`	LSP-backed, type-accurate symbol navigation	`serena` on `PATH`
	`ripgrep`	fast lexical floor (better than plain grep)	`rg` on `PATH`
	`grep`	universal floor — always present	always
`memory_store`	`native-memory`	delegate to Claude Code's native memory	`claude` on `PATH`
	`obsidian`	a human-readable markdown vault you can Git	`~/.obsidian` exists
	`sqlite`	guaranteed stdlib floor	`sqlite3` importable (always)

Plus two Python extras: **`schema`** (richer manifest validation via `jsonschema`) and **`mcp`** (the `mokata-mcp` server that exposes mokata as native Claude Code tools; needs Python ≥ 3.10).

The router binds the **first present** provider in each chain. Our goal in §2 is to make the top of each chain present, so `code_graph → code-review-graph` and `memory_store → native-memory` instead of the floors.

2.1 Step 1 — base system (Python, git, a shell)

```
        python3 --version      # need ≥ 3.10 (the bundled mokata-mcp server
ships by default)
        git --version
```

If Python is older than 3.10, install a newer one (`pyenv install 3.12`, `brew install python@3.12`, or your distro's package) before continuing.

2.2 Step 2 — an isolated environment

The end-user install is plain `pip install mokata` from PyPI — **no clone required**.

```
        python3 -m venv .venv && source .venv/bin/activate   # isolate;
keeps your global env clean
        pip install -U pip
```

2.3 Step 3 — install mokata

`pip install mokata` also pulls the MCP SDK (a default dep), so the bundled `mokata-mcp` server works out of the box.

```
        pip install mokata                # add "mokata[schema]" for richer
```

```
manifest-validation messages
        mokata --version                  # → prints the installed mokata
version (confirms the console script is on PATH)
```

> **Contributors only:** to hack on mokata itself, clone and install editable instead — `git clone https://github.com/JasGujral/mokata-oss.git && cd mokata-oss && pip install -e ".[schema]"`. End users never need this.

> **Checkpoint A.** `mokata --version` prints a version. You now have the engine. Everything below adds *providers* the engine can route to.

2.4 Step 4 — wire the `code_graph` providers

ripgrep (the fast floor — install this at minimum):

```
        # macOS:        brew install ripgrep
        # Debian/Ubuntu: sudo apt-get install -y ripgrep
        # Fedora:       sudo dnf install -y ripgrep
        # Arch:         sudo pacman -S ripgrep
        rg --version        # mokata detects the `rg` executable
```

code-review-graph (the real persisted graph — the top of the chain). Install it per its upstream README so the **`code-review-graph` executable lands on your `PATH`** (that exact command name is what mokata detects). It's MIT, local, SQLite-backed:

```
        # Follow https://github.com/tirth8205/code-review-graph for the
current install method,
        # then confirm the command mokata looks for is resolvable:
        command -v code-review-graph    # must print a path for mokata to
bind it
```

serena (optional, type-accurate navigation via LSP). Install from oraios/serena so the **`serena`** command is on `PATH`:

```
        command -v serena               # optional; sits between code-
review-graph and ripgrep in the chain
```

> **Why bother, when grep works?** The graph turns "who calls this / what's the blast radius / what implements this interface" from a fuzzy lexical guess (`degraded=True`) into exact, edge-accurate answers — which is what makes brainstorm and review *grounded in real structure* and what powers JIT retrieval's 60%+ token savings (§8, §11). The floors keep mokata working everywhere; the graph makes it strong.

2.5 Step 5 — wire the `memory_store` providers

native-memory — detected by the **`claude`** CLI being on `PATH` (it delegates persistence to Claude Code's own memory). If you use Claude Code, you already have it:

```
        command -v claude               # present → memory_store can bind
native-memory
```

obsidian — a human-readable markdown vault mokata can write memory into, which you can diff and Git. mokata detects Obsidian from its **real per-OS config locations** (macOS `~/Library/Application`

`Support/obsidian`, Linux `~/.config/obsidian` + Flatpak, Windows `%APPDATA%\obsidian`) **or** a vault you point it at via config:

```
        # install the Obsidian app (it creates the config dir), or point
mokata at an existing vault:
        mokata config set tools.obsidian.config.vault ~/Documents/MyVault
# a configured vault that exists counts as "present"
```

sqlite — the guaranteed floor. Nothing to install: Python ships `sqlite3` in the stdlib, so this is always available as the bottom of the chain.

> **Checkpoint B.** `command -v rg code-review-graph claude` resolve, and `~/.obsidian` exists. The top of both capability chains is now present on this machine.

2.6 Step 6 — initialize the `full` profile (human-gated)

Now point mokata at a project (your own repo, or a fresh `mkdir demo && cd demo` to follow along safely):

```
        mokata init --profile full
```

`init` **detects** your installed tools, picks the `full` chains, and writes `.mokata/manifest.json` + `.mokata/constitution.md`. It's a **human-gated write** — it shows a preview of exactly what it will create and which tools it detected, then waits for your confirmation. (Add `--yes` to skip the prompt, `--force` to overwrite an existing manifest.)

2.7 Step 7 — verify the stack resolves to real providers

This is the payoff of all the wiring. Run the four inspection commands:

```
        mokata validate     # the committed manifest parses + validates
(jsonschema pass too, now)
        mokata detect       # tool-presence across the whole catalog
(present/absent)
        mokata status       # ← the key check: what each capability resolves
to RIGHT NOW
        mokata route        # the full attempted fallback chain + the reason
it bound where it did
```

`mokata status` is the proof. On a fully-wired machine you want to see `code_graph` resolving to **`code-review-graph`** (not `grep`) and `memory_store` resolving to **`native-memory`** (not `sqlite`). If you see a floor instead, the provider above it isn't detected — run `mokata route code_graph` to see the chain and fix the missing tool (re-check `command -v …`).

```
        mokata coverage     # capability coverage + any unmet gaps + role
overlaps (resolved by precedence)
        mokata doctor       # missing providers, broken adapters, role
conflicts, bad trust, oversized rule tiers
```

> **Checkpoint C — you are fully wired.** `mokata status` shows real providers at the top of both chains, `mokata doctor` reports no errors. From here, every command in this guide runs at full strength.

2.8 Step 8 — drive it from Claude Code (the primary surface)

The CLI you just verified is the engine's mechanics. For day-to-day building you'll drive the same engine from **inside Claude Code**. (The canonical pip-first setup is in Getting started; this section shows it in context.)

Recommended — wire it in with one command (`mokata setup claude`): on your existing Claude Code sign-in, no API key:

```
        cd /path/to/your/project
        mokata setup claude          # --profile / --scope options; reverse
with `mokata unsetup claude`
        # restart Claude Code, then:
        mokata mcp status            # expect: mokata-mcp: CONNECTED ✓
```

`mokata setup claude` runs `init` if needed, copies the slash commands into `.claude/commands/`, installs the Agent Skills, registers the `mokata-mcp` server in `.mcp.json`, wires the SessionStart + secret-guard hooks into `.claude/settings.json`, and wires the status-line badge. JSON files are **merged, never clobbered**, and it's idempotent (re-running syncs and prunes old skills on update). `--scope user` installs into `~/.claude` for every project; `--no-hooks` wires only commands + MCP.

Restart Claude Code. You now have the workflow slash commands (`/mokata:brainstorm`, `/mokata:refine`, `/mokata:spec`, `/mokata:test`, `/mokata:develop`, `/mokata:review`, `/mokata:debug`, `/mokata:optimize`, `/mokata:bug`, and `/mokata:init`), the SessionStart briefing hook, and the secret-guard hook — all automatic. Typing `/mokata` lists them all.

> You don't even have to set up first: on a fresh repo mokata's SessionStart briefing **offers to initialize it for you** (once), and you can run **`/mokata:init full`** from inside Claude Code — human-gated, no terminal trip.

> **Pending Claude plugin-directory approval.** A one-click Claude Code **plugin** is **planned, not yet available** — mokata isn't registered on any Claude Code marketplace. The supported way to run mokata inside Claude Code today is the pip-first path: `pip install mokata` → `mokata setup claude` (see Getting started). *(This notice auto-flips once the listing is approved — single source: `scripts/directory_listing.py`.)*

(An experimental manual `/plugin marketplace add ~/path/to/mokata-oss` route from a local clone exists for advanced users — see Install the plugin.)

Any other harness, or none. The `mokata` CLI is harness-agnostic and has **no LLM of its own** — invoke as `mokata <command>` or `python -m mokata <command>` from any shell, script, CI, or shell-capable assistant (Gemini, Codex). Almost every command accepts **`--path PATH`** (the repo root; defaults to cwd). Commands that need an initialized repo exit non-zero with a clear error if `.mokata/` is missing.

> **A `pip` CLI install is terminal-only.** It does **not** put mokata inside Claude Code (no slash commands, no hooks, no LLM driving the gates). For the in-Claude workflow, run **`mokata setup claude`** (above). See How mokata uses an LLM: harness vs CLI.

> **The rest of this guide leads with the CLI** so each gate is visible as a discrete step. The mapping is one-to-one: inside Claude Code the slash commands and the LLM run these same operations. Where a step has a slash-command equivalent, it's called out.

2.9 The 60-second setup recap

```
        git clone https://github.com/JasGujral/mokata-oss.git && cd mokata-
oss
        python3 -m venv .venv && source .venv/bin/activate && pip install -U
pip
        pip install -e ".[schema]"                    # engine + extras
([postgres] too for a hosted store); MCP SDK comes by default on ≥
3.10
        brew install ripgrep                          # (or apt/dnf/pacman)
— the rg floor
        #  …install code-review-graph + serena per their README so both
commands are on PATH
        #  …install Obsidian (or `mokata config set
tools.obsidian.config.vault <path>`) for that backend
        command -v rg code-review-graph serena claude  # confirm providers
are present
        cd /path/to/your/project
        mokata init --profile full                    # human-gated
scaffold
        mokata status && mokata doctor                # verify real
providers, no errors
        mokata setup claude                           # (optional) drive it
from Claude Code
```

---

3. Anatomy of a stack: the `.mokata/` directory

Everything mokata creates as its own data lives under `.mokata/`, with a clear **committed vs. transient** split inside it (everything mokata owns stays here — nothing scattered at the repo root).

Committed config — at the `.mokata/` root, diffable in code review:

Path	What it is
`manifest.json`	the stack manifest — profile, layers, capabilities, tools, settings
`constitution.md`	governing articles, read before non-trivial work
`.gitignore`	ships with `init`; ignores `temp_local/` (below)
`mokata-stack.json`	*(optional)* a stack you chose to `export` here
`mcp.json`	*(optional)* a list of external MCP servers for mokata to discover and route to
`steering.md`	*(optional)* a free-form steering rules tier

`lat.md`	*(optional)* the registry of concepts for drift anchors

Transient runtime — under `.mokata/temp_local/` (gitignored, safe to delete, regenerated as you work):

Path	What it is
`temp_local/state/`	pipeline state (JSON) — handoffs and resume checkpoints
`temp_local/audit/ledger.jsonl`	the append-only audit ledger (every gate decision, tool call, write)
`temp_local/memory/memory.db`	the SQLite memory backend (plus `memory/vault/` for the Obsidian backend)

Inside `temp_local/state/` you'll find files that make the pipeline durable and inspectable:

- `approved_approach.json` — the brainstorm handoff (the approach you approved)
- `approved_refinements.json` — the `refine` handoff (the scoped set you approved)
- `emitted_spec.json` — the spec that passed the completeness gate
- `memory_stats.json` — the read/write ratio instrumentation
- `knowledge_index.json` — the per-file freshness index
- `story_analysis__<id>.json` — per-story analysis bridge
- `undo_log.json` — prior values for reversible writes
- `pipeline_run__<id>.json` — resume checkpoints (one per passed gate; the run-progress tracker reads these)

So the **committed config** is diffable in code review (you see exactly what the agent may do), while the **runtime stores** stay out of version control. A user-set backend path (`tools.<id>.config.path` / `.vault`) can point a store elsewhere — your explicit choice overrides the default. (Harness wiring from `mokata setup claude` — `.claude/`, `.mcp.json` — is **Claude Code's** config at its required paths, not mokata data, so it lives there by necessity.)

---

4. Part A — The spine: `init`, detection, routing, bootstrap

The spine is the machinery that detects what tools you have, decides which one answers each need, and produces the config and the session briefing.

4.1 `mokata init` — scaffold a stack (human-gated)

```
        mokata init                          # default profile: standard
        mokata init --profile full           # wire every known graph +
memory provider
        mokata init --profile minimal --yes  # engine only, non-interactive
        mokata init --profile full --preview # dry-run: print the plan,
write NOTHING (exit 0)
```

`init` detects your installed tools, picks a profile, and writes `.mokata/manifest.json` + `.mokata/constitution.md` (+ a committed `.mokata/.gitignore`). It's a **human-gated write**: it shows a preview of exactly what it will create and which tools it detected, then waits for confirmation.

Flag	Meaning
`--profile {minimal,standard,full,custom}`	starting profile (default `standard`)
`--yes`	non-interactive; skip the write prompt
`--force`	overwrite an existing manifest
`--preview`	print the plan and exit **without writing** (the dry-run the `/mokata:init` command uses)

> **No terminal needed.** Inside Claude Code you can run **`/mokata:init full`** (or `standard`/`minimal`) — pip-free, on the bundled engine: it previews, asks you to approve, then sets the profile. And on a brand-new repo, mokata's SessionStart briefing **proactively offers to initialize it** for you (once — never a nag once `.mokata/manifest.json` exists). Approval is always required; auto-engaging only *starts* the conversation.

4.2 `mokata detect` — what's on this machine

```
mokata detect      # tool-presence for the whole catalog (present/absent), no manifest needed
```

Detection backs the graceful-degradation guarantee: a tool is found via one of four `detect.type` strategies — `command` (on PATH), `python_module` (importable), `path` (a file exists), or `always` (builtins like grep). A tool that isn't present is simply skipped and the router falls through to the next provider.

4.3 `mokata route` — resolve a capability to a tool

```
mokata route                # resolve every declared capability
mokata route code_graph     # resolve one, showing the fallback chain + reason
```

This is the **capability router (H6)**, the heart of the spine. A *capability* (a "need" like `code_graph` or `memory_store`) declares an ordered `fallback` list of provider ids. The router walks that list and binds the first present, enabled provider — and **that ordering *is* the precedence**. `route` shows you the attempted chain and why it landed where it did.

4.4 `mokata status` — the one-line stack summary

```
mokata status      # version, profile, and what each capability resolves to right now
```

This is your quickest "where am I" check. On `standard`, you'll typically see `code_graph → grep` and `memory_store → sqlite` unless richer tools are installed.

4.5 `mokata validate` — does the committed manifest parse?

```
mokata validate    # parse + validate the manifest; exit non-zero on an invalid manifest
```

Validation is a built-in structural pass; if `jsonschema` is installed (`pip install -e ".[schema]"`) it adds a richer schema pass, and its absence is degraded over (never fatal). This is the command to wire into CI.

4.6 `mokata bootstrap` — the SessionStart briefing

```
mokata bootstrap                # print the briefing
mokata bootstrap --show-tokens   # also print the token estimate + budget check to stderr
```

`bootstrap` prints the compact briefing the SessionStart hook injects: **which stack you're in, which capabilities are live, and the inviolable gates** — capped at a **2,000-token budget**. With `--show-tokens` it reports the estimate and exits non-zero if over budget. In the plugin this runs automatically at the start of each session; on the CLI it's the same content on demand.

> **Slash-command mapping:** there's no slash command for `init`/`detect`/`route` — in Claude Code the plugin install (or `mokata setup claude`) does the scaffolding, and the SessionStart hook surfaces the bootstrap briefing for you. Run these CLI commands any time to inspect the spine.

---

5. Part K — Profiles & configuration

mokata is **fully configurable and composable**: switch any layer or tool on or off, pick a profile, set per-adapter trust, and everything is a toggle in the committed manifest.

5.1 The four profiles

Profiles are deterministic "enabled sets" — they decide which layers are on and what each capability's fallback chain looks like.

Profile	Layers	`code_graph` chain	`memory_store` chain	Network
`minimal`	engine, governance	—	—	**zero egress**
`standard` *(default)*	all four	ripgrep → grep	sqlite	local-only
`full`	all four	code-review-graph → serena → ripgrep → grep	native-memory → obsidian → sqlite	only present tools, all gated
`custom`	all four	full chains (hand-tune)	full chains (hand-tune)	—

`grep` is the universal floor for `code_graph`; `sqlite` (stdlib) is the guaranteed floor for `memory_store`. **Every richer provider degrades to its floor when absent — never a hard failure.** Pick a profile at init:

```
        mokata init --profile minimal    # just the governed TDD engine,
zero network egress
        mokata init                      # standard — knowledge + memory on
lean local defaults
        mokata init --profile full       # every known graph/memory provider
wired
        mokata init --profile custom     # full chains wired as a starting
point to hand-tune
```

5.2 The manifest, field by field

`.mokata/manifest.json` is the single source of configuration truth:

```
        {
          "manifest_version": 1,
          "mokata": { "version": "0.0.0" },
          "profile": "full",
          "layers": {
            "engine":     { "enabled": true },
            "knowledge":  { "enabled": true },
            "memory":     { "enabled": true },
            "governance": { "enabled": true }
          },
          "capabilities": {
            "code_graph": {
              "description": "structural code queries",
              "layer": "knowledge",
              "fallback": ["code-review-graph", "serena", "ripgrep", "grep"]
            },
            "memory_store": {
              "description": "durable memory backend",
              "layer": "memory",
              "fallback": ["native-memory", "obsidian", "sqlite"]
            }
          },
          "tools": {
            "grep": {
              "provides": "code_graph",
              "kind": "builtin",
              "version": null,
              "enabled": true,
              "detect": { "type": "always" }
            }
          },
          "settings": { }
        }
```

- **`layers.<name>.enabled`** — toggle a whole layer (`engine`, `knowledge`, `memory`, `governance`). Disable a layer and its capabilities drop out of the router entirely.
- **`capabilities.<need>.fallback`** — the ordered provider precedence the router honors. `layer` ties the capability to a layer; it's routable only while that layer is enabled.
- **`tools.<id>`** — `provides` (which capability it serves), `kind` (`mcp` / `cli` / `library` / `builtin` / `external` — `mcp` and `external` are the network-

capable kinds for local-first accounting), `enabled` (per-tool toggle — a disabled tool is treated as absent), and `detect`.

5.3 The `settings` block — the generic toggle store

`settings` is an open-ended key/value block. The keys mokata reads:

Key	Shape	Default	Feature
`memory`	`{persistent: bool, decision: bool, episodic: bool}`	all on	per-type memory toggles (C9)
`governance.output_density`	bool	`false`	output-density compression (F4)
`governance.karpathy.<id>`	bool per gate id	all on	Karpathy gate toggles (G3) — ids `think-first`, `simplicity`, `surgical-scope`, `verify`
`trust.<tool>`	`"read-only"` / `"propose-only"` / `"gated-write"`	`gated-write`	per-adapter trust dial (K3)
`execution.default`	`"ask"` / `"sequential"` / `"parallel"`	`ask`	per-run execution mode — ask once, or honor a saved choice (Stage 25)
`brainstorm.auto`	`"on"` / `"off"` / `"ask"`	`on`	auto-engage brainstorm when exploring (Stage 29)

It’s intentionally open-ended so future settings read from it the same way.

5.4 The trust dial (K3)

Per-adapter, you decide how much an external tool is allowed to do:

- **`read-only`** — the adapter can never write.
- **`propose-only`** — every action is surfaced for approval; nothing is ever auto-approved.
- **`gated-write`** *(default)* — writes go through the standard human gate.

Adopted external tools are treated as untrusted input; the trust dial is how you scope them.

5.5 Backend paths & the `config` command

Each tool can carry a `config` block so you point a backend wherever you want, and `mokata config get/set` edits the manifest for you (human-gated — preview → confirm; secrets are a hard block):

```
        mokata config set tools.sqlite.config.path ~/data/mokata-memory.db
# custom SQLite path
        mokata config set tools.obsidian.config.vault ~/Documents/MyVault
# external Obsidian vault
        mokata config get tools.sqlite.config.path
# read it back
```

A **hosted Postgres** backend is opt-in and local-first-safe: the DSN is referenced by an **env var name**, never inline (the manifest is committed; an inline credential is hard-blocked by the secret-guard). Install the optional driver with `pip install "mokata[postgres]"`:

```
        export
MOKATA_PG_DSN="postgresql://user:pass@db.example.com:5432/mokata"
        mokata config set tools.postgres.config.dsn_env MOKATA_PG_DSN
```

Postgres **degrades to the SQLite floor** if the env var is unset, `psycopg` is absent, or the database is unreachable — never a hard failure. Defaults are unchanged when no `config` block is set. See Configure storage backends & paths.

5.6 Verify your configuration

```
        mokata validate    # manifest parses + validates
        mokata doctor      # missing providers, broken adapters, role
conflicts, bad trust, oversized rule tiers
        mokata coverage    # which capabilities are covered + any gaps/role
overlaps
        mokata status      # what each capability resolves to right now
```

Run these four after any manifest edit. `doctor` exits non-zero if there's an error, so it's CI-usable too.

---

6. Part D — The 7-phase pipeline & gates

The engine is a **7-phase pipeline**. Each phase consumes the prior phase's handoff; three phases carry **gates** that must hold before the run proceeds.

```
brainstorm → analysis → strawman → pre_mortem → probes →
completeness_gate → emit
```

These are the canonical `PIPELINE_PHASES`. You can run the whole thing (`mokata playbook`) or enter at any phase (`mokata enter <phase>`).

6.1 The phases

1. **brainstorm** — Socratic, *one question at a time*; surfaces 2–3 real

approaches with tradeoffs. **HARD-GATE**: no spec proceeds until exactly one approach is explicitly approved. The approved approach is persisted to `state/approved_approach.json` and becomes a downstream constraint.
2. **analysis** — grounds the approved approach in the codebase (structural facts from the knowledge layer) plus the answered questions; produces components/notes.
3. **strawman** — a first-cut design mapping the approach to each acceptance criterion.
4. **pre_mortem** — derives adversarial *risk probes* from the approved approach: each declared downside becomes a probe, plus standard failure/scale/rollback angles.
5. **probes** — checks the spec addresses each derived probe.
6. **completeness_gate** — the **provable-completeness blocker**: `emit` is refused until every acceptance criterion maps to a test (RED-before-GREEN traceability). It reads the brainstorm handoff so the approved approach stays in view.
7. **emit** — produces durable output; the write is **human-gated**.

6.2 The three pipeline gates

Phase	Gate id	Kind	Blocks on
brainstorm	`approach-approval`	human	no approved approach
completeness_gate	`completeness`	check	any acceptance criterion with no mapped test (or an empty spec)
emit	`emit-approval`	human	un-approved durable output

The remaining phases are advisory. The completeness gate **never silently passes**: an empty spec blocks, and *any* unmapped AC blocks.

6.3 Brainstorm — the HARD-GATE (slash: `/mokata:brainstorm`)

```
        mokata brainstorm            # launch the Socratic pre-spec
exploration
        mokata brainstorm --status   # report whether an approved approach
is persisted
```

The brainstorm protocol drives a one-question-at-a-time exploration, proposes 2–3 real approaches grounded in your actual codebase, and **refuses to let a spec proceed until you explicitly approve one**. This is the single most distinctive gate in the tool — it forces design thinking before any spec exists.

> **mokata engages this on its own when you're exploring.** Beyond `/mokata:brainstorm`, the skill is *model-invocable*: inside Claude Code it auto-activates when you're weighing options or describing a new problem before code (you'll see the banner `mokata · brainstorm (engaged)`). It's proactive, not intrusive — it only *starts* the conversation; the HARD-GATE still holds, and it

> won't hijack a direct command. Turn it off or make it ask first: `mokata config set settings.brainstorm.auto off` (`on` | `off` | `ask`, default `on`).

6.4 Dry-run preview — see the plan with zero side effects (E7)

```
        mokata preview                                  # the whole
pipeline
        mokata preview --start pre_mortem --to completeness_gate   # a slice
```

`preview` lists the planned actions, the gate at each phase, and the files each phase *would* touch — with **no writes and no ledger entries**. Run it before any real pipeline run to know exactly where things can block (the answer is always `completeness_gate` and `emit`).

6.5 Mid-pipeline entry (L2)

```
        mokata enter completeness_gate          # run just the gate on a
hand-written spec
        mokata enter strawman --to probes       # run a slice of phases
```

`enter <phase> [--to <phase>]` runs a slice. The gates of the phases you run **still apply**; upstream phases aren't forced, and the skip is **reported explicitly** — never silent. This is what lets you reach for one part of the engine (e.g. just the completeness gate on an existing spec) without running the whole story.

6.6 Drive the whole story (slash workflow: `/mokata:brainstorm → /mokata:spec → /mokata:test → /mokata:develop → /mokata:review`)

```
        mokata playbook                 # sequential
        mokata playbook --parallel      # parallel subagents (degrades to
sequential without a harness)
        mokata playbook --parallel --fanout   # + concurrent fan-out
```

`playbook` runs the complete flow — brainstorm → completeness gate → tests → RED-before-GREEN implement → review — with knowledge and memory active, printing PASS/FAIL per checkpoint:

```
brainstorm_approved … gate_blocked_initially …
gate_passed_after_tests …
red_before_green … review_passed … memory_written … RESULT: PASS
```

Under the hood: the completeness gate first **blocks** emit (no tests mapped), then **passes** once every AC maps to a test; RED-before-GREEN is enforced (implementing a test that hasn't failed first is blocked); the two-stage review runs; and on `standard`/`full`, memory records the decision.

6.7 Run-progress tracker — always know where you are

A multi-phase run is legible, not opaque. `mokata progress` (and the `progress` MCP tool inside Claude Code) prints a **read-only** tracker derived from the persisted run-state (`temp_local/state/pipeline_run__*` checkpoints), so it can't drift from what the engine actually did:

```
        mokata progress            # the active/most-recent run
        mokata progress --ascii    # plain [x]/[>]/[ ] glyphs instead of ✓/▶/○
```

```
mokata · run  [3/7 done]
  ✓ brainstorm        approach-approval passed
  ✓ analysis
  ✓ strawman
  ▶ pre_mortem        ← you are here
  ○ probes
  ○ completeness_gate
  ○ emit
next: probes     ·     pending: 4/7
```

Each phase is marked **done / current / pending** with a `[done/total]` count and what's next. Inside Claude Code the pipeline skills also print this at the start and end of each phase, plus a one-line banner naming what's running (`mokata · develop (running)` → `mokata · develop (done)`) — so you never wonder whether mokata is working or which part. With no active run it says so cleanly (never an error).

---

7. Part L — Composability: skills, standalone runs, chaining, mid-pipeline entry

Every capability is also a **standalone command** — no full-pipeline prerequisite. You can reach for one piece (generate tests, debug a failure, review a diff) and enter the pipeline wherever you need.

7.1 The skill catalog (progressive disclosure)

```
        mokata skills           # list — names + one-line summaries (cheap)
        mokata skills test      # reveal one skill's full prompt + gate (details on demand)
```

The shipped `/<name>` slash commands under `templates/commands/` are **generated from this same registry**, so the command and the CLI never drift.

7.2 The skills and their gates

Skill	Slash	Gate id	Kind	What it doe
`brainstorm`	`/mokata:brainstorm`	`approach-approval`	human	Socratic pre-spec exploration (f a *new* problem); HARD-GATE - no spec until one approach is approved
				deep, user-steerable review of *existing* code

`refine`	`/mokata:refine`	`refinement-approval`	human	prioritized refinements; HARD-GATE - no spec until a scoped set is approved, the hands off to `spec`
`spec`	`/mokata:spec`	`completeness`	human	turn the problem into testable acceptance criteria, each mapped to a test
`test`	`/mokata:test`	`red-before-green`	check	write failing tests first (RED); no implementatio here
`develop`	`/mokata:develop`	`no-code-without-failing-test`	check	implement the minimum to turn a failing test green
`review`	`/mokata:review`	`spec-then-quality`	human	two-pass review — against the spec, then quality
`debug`	`/mokata:debug`	`repro-first`	check	reproduce firs root-cause (N- strikes escalation), then fix
`optimize`	`/mokata:optimize`	`measure-first`	check	measure before/after; keep only proven, behavior-preserving wins
`bug`	`/mokata:bug`	`reproducer-required`	check	start from a reproducer + failing test, then fix; label progression

Two gate **kinds**: **human** (requires explicit approval — it surfaces, you decide) and **check** (a verifiable condition, e.g. a failing test must exist before implementation).

> **`refine` vs `review`** — opposite ends of the pipeline. `refine` is a **front-end** (like `brainstorm`, but for code you already have): review → propose → approve a scoped set → hand off to `spec` → `test` → `develop` → `review`. `review` is the **back-end** check that

> verifies a diff against its spec. Behavior-preserving by default: refactors are pinned by characterization tests written RED before the change. See Refine existing code.

7.3 Run a skill standalone

```
        mokata run review      # run a skill with no pipeline prerequisite
(grounding degrades cleanly)
        mokata run test        # works even with no init
```

`run <name>` executes one skill and applies **only its own gate**. `name` is one of the eight skills.

7.4 Chain skills (gates never bypassed)

```
        mokata chain spec test            # plan a manual chain — each step
keeps its own gate
        mokata chain test develop review  # gates are never bypassed in a
chain
```

7.5 Suggestions — recommend, never run

```
        mokata suggest --fresh            # suggest a command for a fresh
start
        mokata suggest --failing-test     # …for a failing test
        mokata suggest --diff             # …for a diff to review
```

`suggest` recommends a relevant command for the context and **only suggests — it never runs anything**. The boolean flags describe your situation: `--fresh`, `--spec`, `--failing-test`, `--implementation`, `--diff`, `--bug`, `--stacktrace`, `--perf`.

7.6 Authoring your own skill (test-first — G6)

Skills are authored **RED-GREEN-REFACTOR-for-docs**: declare the doc requirements, watch them fail, write content until they pass, then promote to a registry `Skill`.

```
        from mokata.govern import SkillDraft
        from mokata.skills import Gate

        draft = (SkillDraft("clarify")
                 .require("gate section", "## Gate")
                 .require("trigger", "Use when"))

        draft.check().passed          # False — RED (no content yet)
        draft.status                  # "red"

        draft.write("## Gate\nUse when the spec is ambiguous; ask one
question at a time.")
        draft.check().passed          # True — GREEN

        skill = draft.to_skill("clarify ambiguous specs",
                               Gate("clarify-gate", "ask before assuming"))
```

Register it by adding the `Skill` to `src/mokata/skills.py`, then regenerate its slash-command template so the shipped command and the CLI stay in sync:

```
        from mokata.skills import command_markdown, get_skill
        open("templates/commands/clarify.md",
"w").write(command_markdown(get_skill("clarify")))
```

---

8. Part B — The knowledge layer

mokata **orchestrates** a codebase graph — it never builds a parser. Structural queries return one typed shape regardless of which backend answers, the backend is chosen through the router, and stale results are surfaced rather than served silently.

8.1 The five typed queries (B2)

Each query returns a `QueryResult` (`kind`, `target`, `references[]`, `backend`, `degraded`, `note`), where each `Reference` is `(path, line, snippet, symbol)`:

Kind	Question
`callers`	who calls this symbol?
`callees`	what does this symbol call?
`implementers`	which classes subclass/implement this?
`imports`	where is this module/symbol imported?
`blast_radius`	transitive callers up to `--depth` hops (the impact surface)

```
        mokata query callers compute
        mokata query callees myFunction
        mokata query implementers BaseHandler
        mokata query imports utils.parsing
        mokata query blast_radius helper --depth 3   # --depth defaults to
2, applies to blast_radius
```

8.2 Backend selection — one detection path (B1/B3)

The layer resolves `code_graph` through the router (`code-review-graph → serena → ripgrep → grep`) and uses the first present provider:

- A real graph tool (`code-review-graph` / `serena`) → the adopted **graph backend (B1)**, which delegates all graph work to the external tool via an injected client. No in-house parser, no in-house graph.
- Otherwise the **grep floor (B3)** — a dependency-free lexical implementation of the same five queries. Results are marked `degraded=True` (approximate, but always available).

If the graph backend errors mid-query, the layer **degrades to grep** rather than failing.

8.3 Incremental index + staleness (B4)

```
        mokata index    # build/refresh the per-file freshness index; report
added/changed/removed + stale files
```

`index` builds a per-file fingerprint index (content hash + mtime + size) and re-indexes **only what changed**. When a query touches a file that changed since indexing, the result's `note` is annotated with a `STALE: …` warning — **staleness is surfaced, never served silently**.

8.4 Drift anchors / `lat-check` (B5)

Optional `@lat: <concept>` comments tie code to concepts registered in a `lat.md`. `mokata lat-check` flags drift in two directions — anchors pointing at unknown concepts (orphans) and registered concepts with no anchor — and **degrades cleanly** (inactive, exit 0) when there are no anchors and no registry.

```
mokata lat-check    # exit 1 on drift (so it's usable as a review gate), exit 0 when clean or inactive
```

8.5 The per-story bridge (B6)

A story's queries are recorded and can be persisted via the state surface (`story_analysis__<id>.json`), so the analysis phase enriches a durable layer instead of recomputing structural facts each run.

---

9. Part C — Memory (default-on, self-healing)

Memory is a **native part of the framework — on by default on `standard`/`full`, not an add-on you wire up.** There are three individually-toggleable types, every durable write is human-gated, and it heals itself by *surfacing* contradictions, never by silent rewrite.

9.1 The memory triad (C1/C2/C3)

Type	What it holds
`persistent`	project facts / conventions
`decision`	project decisions ("why we did X")
`episodic`	past conversation turns (searchable)

Each item carries provenance, a TTL (`expires_at` / `valid_for`), and `supersedes` / `depends_on` edges; status is `active` / `superseded` / `stale`.

9.2 Inspect (read-only)

```
mokata memory    # active items, the read/write ratio, and any pending self-healing proposals
```

This commits nothing. The read/write ratio is the **instrumentation (C8)**: if writes ≫ reads, the feature is failing (you're storing more than you use), and mokata tells you.

9.3 Pluggable backends — storage only (C4)

Chosen through the router (`memory_store`): **SQLite** (default, stdlib, the guaranteed floor), **Obsidian** (a markdown vault under `memory/vault/`), or **native-memory** (an adapter delegating to an injected client). These are *storage only* — the memory *logic* is mokata's own. When native-memory has no client wired, selection degrades to the SQLite floor.

9.4 Recording facts and decisions (human-gated — C6)

Nothing reaches a backend without approval. The write API takes a `confirm` callback or an `assume_yes` flag; a declined write commits nothing.

```
        from mokata.config import Surface
        from mokata.memory import MemoryStore, MemoryItem, DECISION

        store = MemoryStore.from_surface(Surface.load("."))
        store.remember(MemoryItem.create("db.engine", "postgres"),
assume_yes=True)
        store.remember_decision("api.style", "REST", assume_yes=True)
```

9.5 Self-healing by surfacing (C5)

mokata **detects** contradictions (two active items, same subject, different value) and staleness (elapsed TTL), and **surfaces each as an old → new diff** for you to **approve / edit / reject**. It **never silently rewrites**; the default is no change. Detection writes nothing — only an explicit, gated `apply` changes anything.

```
        for p in store.detect_issues():          # read-only: detects,
writes nothing
            print(store.render_proposal(p))      # old → new diff
            store.apply_proposal(p, "approve", assume_yes=True)   # or
"edit" / "reject"
```

9.6 Consolidation — proposal-only (C7)

A pass that **proposes** merges (duplicates), summaries (episodic clusters), and prunes (already-stale items) — and **never auto-applies**. Each proposal is an old → new diff, human-gated like C5, and logged to the audit ledger.

```
        for p in store.propose_consolidations():
            store.apply_consolidation(p, "approve", assume_yes=True)
```

9.7 Episodic search (C3)

```
        from mokata.memory import EpisodicMemory
        epi = EpisodicMemory(store)
        epi.record("session-1", "we chose postgres as the database engine",
assume_yes=True)
        epi.search("which database did we choose")   # embeddings optional;
lexical fallback
```

9.8 Per-type toggles (C9)

`settings.memory.{persistent,decision,episodic}` toggle each type independently (default on). A disabled type is **refused on write and never surfaced on read**. Disabling the whole `memory` layer turns all three off.

---

10. Part E — Execution modes & the depth engines

At the start of **every implementation** — the `develop` skill, `playbook`, `exec` — mokata asks which execution mode to use: **parallel subagents vs. the sequential gated flow**. It never fans out without your pick. The default is the sequential gated flow; parallel is the governed opt-in.

10.1 The selector (E8)

```
        mokata exec                          # ask once (honors a saved
preference); default sequential
        mokata exec --parallel               # parallel subagents
        mokata exec --parallel --isolation # explicit fresh-context
isolation + two-stage review
        mokata exec --parallel --fanout    # concurrent fan-out (tasks at
once)
```

It’s asked **once per run** (not per sub-task), with a sensible default (sequential = lowest cost), and a saved preference so power users aren’t re-prompted — `mokata config set settings.execution.default sequential` (or `parallel`, or `ask`, the default). When parallel is offered, mokata shows a **token/cost estimate** first; a harness without subagents degrades to sequential with a clear message. With no choice (non-interactive), the default is **sequential**.

10.2 Sequential gated flow

The default and lowest-cost path: mokata processes tasks in-loop, no subagent runner required. This is the floor that always works.

10.3 Parallel subagents (E2/E3)

- **Fresh-subagent-per-task isolation (E2)** — each task is given *only* its own context (clean per task); the handback is a **summary, not raw context**.
- **Two-stage review (E3)** — when isolation is on, each task result is reviewed in two passes: **spec-compliance**, then **code-quality**.
- **Concurrent fan-out** — tasks run at once (a thread pool).

Parallel runs **surface a token/cost estimate before running**, stay inside the existing gates + audit ledger + token budget, **log every subagent decision**, and **degrade to the sequential flow** when subagent execution is unavailable — never a hard failure.

10.4 Per-task model routing (E4)

`ModelRouter` picks the **cheapest capable** model for a task and **escalates on a BLOCKED signal** to the next stronger tier. The model set is a pluggable policy (generic `fast` / `balanced` / `deep` tiers by default — override with your own); cost is computed through the same `TokenTracker`.

10.5 The depth engines (E5/E6)

These are the `/mokata:bug`, `/mokata:debug`, and `/mokata:optimize` skills, each with a discipline-enforcing gate:

- **`/mokata:bug` (E5)** — capture a **reproducer first**, then fix; labels progress `reported → reproduced → fixing → verified`; reproducer-before-fix is gated (`reproducer-required`).
- **`/mokata:debug` (E6)** — **root-cause-before-fix** with **N-strikes escalation**: after N ruled-out hypotheses, escalate the model (`repro-first`).
- **`/mokata:optimize` (E6)** — **measure-first**: no change before a baseline; an optimization is kept only when a before/after measurement shows it faster with behavior preserved (`measure-first`).

---

11. Part F — Token & cost governance

mokata measures and reduces context spend in-loop. Everything builds on one token view (the `TokenTracker`); there is no parallel token machinery.

11.1 The token / cost tracker (F1)

A conservative, dependency-free estimator (~4 chars/token). `TokenTracker.add(...)` accumulates input/output tokens per call; `cost()` applies per-1k rates (illustrative, configurable); `report()` prints totals. This is for **in-loop governance** — "are we spending more than the work is worth" — **not billing**.

11.2 JIT graph-backed retrieval (F2)

Instead of dumping whole files into context, `jit_retrieve(layer, identifiers)` pulls the relevant **reference snippets** for a set of identifiers via the knowledge layer, and reports the reduction vs the file-dump baseline (`tokens_retrieved` vs `tokens_if_dumped`, `saved`, `saved_pct`). On a small sample this is routinely a **60%+ reduction**. This is *why* the knowledge layer matters for cost, not just navigation.

11.3 Sub-agent handback cap (F3)

Heavy sub-agent work returns a compact **summary**, not raw context. `cap_summary(text, cap_tokens)` bounds the handback to a token cap; it's wired into the parallel path via `run_tasks(..., handback_cap=N)` so a parent context never absorbs a sub-agent's full working set.

11.4 Output-density mode (F4)

An **optional**, toggleable terse-output compressor (`settings.governance.output_density`, default **off**): collapses blank-line runs and duplicate lines, squeezes whitespace, and can cap noisy tool output. Default behavior is unchanged unless you enable it.

11.5 Savings budget + statusline (F5)

```
mokata budget    # live savings readout (aggregated from the ledger)
+ a one-line statusline
```

`SavingsTracker.record(label, baseline, actual)` logs each governance win to the audit ledger; `mokata budget` aggregates them into a report plus a statusline like `mokata · saved N tok (P%)`.

11.6 Prompt-cache awareness (F6)

`stable_prefix_for(surface)` composes a deterministic prefix from slow-changing sources (manifest identity + always-on rules + constitution), excluding volatile per-run content, so the prefix stays **byte-identical across runs** and keeps hitting a prompt cache. `is_cache_stable(a, b)` verifies stability by fingerprint.

---

12. Part G — Rules & governance

12.1 4-tier rules + constitution (G1)

Tier	Source	Cap
`always_on`	reflex rules injected each session	**≤ 60 lines**
`agent_memory`	per-agent `MEMORY.md`	**≤ 200 lines**
`steering`	optional `.mokata/steering.md`	—
`articles`	the constitution's governing articles	—

```
mokata rules    # show the tiers and their line budgets; exit non-
zero if a tier is over cap
```

The line caps are a deliberate token-governance discipline: the always-on tier is what gets injected every session, so it's kept tiny.

12.2 The rules-vs-gates-vs-hooks taxonomy (G2)

A rule is **advisory** (stays prose), **blocking** (make it a gate), or **event-driven** (make it a hook). The guiding principle: *"checkable → a gate or a hook, not prose."* If a rule can be mechanically verified, it shouldn't live as a sentence the model might ignore.

12.3 Karpathy gates (G3)

Four engine checks, each registered/toggleable/audited through the rules layer (reusing the shared gate type), firing at their pipeline point:

Gate	Phase	Checks
`think-first`	analysis	a plan/approach exists before

		code
`simplicity`	strawman	complexity under a cap
`surgical-scope`	emit	touched files under a cap
`verify`	completeness_gate	success criteria defined + verified

Toggle via `settings.governance.karpathy.<id>` (default on); a disabled gate does not fire and is not audited.

12.4 Hooks (G4)

- **Sync** hooks block **only for security** (exit code 2) — a non-security sync hook is rejected by design.
- **Async** hooks **observe and never block** (exceptions are captured).

The shipped sync security hook is `hooks/secret_guard.py` (PreToolUse, blocks a write/edit/command that would commit or send a secret — un-overridable). The shipped async hook is `hooks/session_start.py` (injects the sub-2k-token briefing). Both are declared in `hooks/hooks.json`.

---

13. Part I — Safety & audit

Everything mokata does is reviewable and gated. It is **local-first**: nothing leaves the machine unless you wire it; there is no telemetry.

13.1 Secret protection — 4 layers (I1)

`scan(text, path, for_send)` runs four independent layers, catching secrets before they're written, committed, or sent:

1. **signature** — known credential patterns
2. **entropy** — high-entropy tokens
3. **path** — `.env`, `id_rsa`, `*.pem`, …
4. **egress** — any secret in outbound content is **fatal**

13.2 Human-gated writes + the trust dial (I2/K3)

Every durable write goes through the `WriteGate`: **secret scan (an un-overridable security block) → human approval → commit → logged**. The trust dial (`settings.trust.<tool>`) layers on top: `read-only` (cannot write), `propose-only` (always surfaced, never auto-approved), or `gated-write` (default).

13.3 The audit ledger (I3)

```
mokata audit    # print the append-only ledger
```

`.mokata/temp_local/audit/ledger.jsonl` records **every gate decision, tool call, hook, write, savings event, subagent decision, healing/consolidation decision, and more** — each with a monotonic `seq`. This is the single source of truth for "what did the agent do and why."

13.4 Reversibility & resume (I5/I6)

- **Reversibility (I5)** — `ReversibleStateStore` records each write's prior value to a durable undo log (`state/undo_log.json`); `revert` restores it.
- **Resume (I6)** — `PipelineCheckpoint` persists each passed gate (`state/pipeline_run__<id>.json`) so an interrupted run **resumes from the last passed gate** — a crash never loses state.

13.5 The lethal-trifecta gate (I4)

When **system access + private data + an outbound action** coexist, the outbound action is **gated behind explicit human approval** (and logged). When the trifecta isn't active, no gate is imposed. This is the specific defense against the classic prompt-injection exfiltration pattern.

---

14. Part J — Distribution: sharing a governed stack

A mokata stack — which tools are wired, the profile, toggles, trust dials — is a committed, reviewable manifest. Share it so a teammate adopts the same governed setup in one step.

```
        mokata export                       # writes <path>/mokata-
stack.json
        mokata export team-stack.json       # custom destination
        mokata import team-stack.json --yes        # validate + apply
(human-gated)
        mokata import team-stack.json --yes --force  # overwrite an existing
config
```

Imports are **validated before they apply** — an invalid manifest is rejected (exit 1, nothing written) — and applying is **human-gated**, so sharing a stack never silently reconfigures someone's repo. `--force` is required to overwrite an existing `.mokata/manifest.json`.

Programmatic equivalents: `export_manifest(surface, dest=...)`, `validate_shared(data)`, `apply_manifest(root, data, assume_yes=..., force=...)`.

---

15. Part H — Integrating other harnesses & MCP servers

15.1 The harness boundary

```
        mokata harness    # show the harness boundary's capabilities for the
reference Claude Code harness
```

mokata's engine runs through a thin **harness boundary** that defines how four things map to a host: `commands`, `hooks`, `context_injection`, and `subagents`. The reference implementation targets Claude Code. On a harness that lacks a capability (e.g. a host without subagents), mokata

degrades with a clear message instead of failing — which is what lets the same workflow extend to Codex or OpenCode without rebuilding the engine.

15.2 Orchestrating external MCP servers (H4)

```
mokata mcp    # discover MCP servers from .mokata/mcp.json and map them to roles
```

Drop a `.mokata/mcp.json` listing the servers you use (`{name, provides, command}`), and `mokata mcp` enumerates them and maps them to stack roles/capabilities — then mokata orchestrates them through its own gates and audit trail. Discovery is pluggable and **degrades cleanly**: with no config present, the registry is empty and nothing errors.

> mokata both **ships its own MCP server and consumes external ones.** The bundled **`mokata-mcp`** server (registered by the plugin / by `mokata setup claude`) exposes mokata's operations — `query`, `recall`, `doctor`, `coverage`, `budget`, `audit`, `status`, `preview`, `progress`, and the human-gated writes `remember` / `import_stack` / `reset` / `apply_proposal` / `init` / `memory_export` / `memory_import` / `vault_push` / `spec_check` — to Claude Code as native tools. Separately, `mokata mcp` **discovers and routes to** the external MCP servers you list in `.mokata/mcp.json`. (Write tools are propose-only unless you pass `approve=true` — `confirm` is a deprecated alias; secrets are a hard block.)

15.3 Capability coverage (H/A6)

```
mokata coverage    # capability coverage + unmet gaps + role overlaps (resolved by precedence)
```

15.4 Wiring into another harness

The plugin is just three portable artifacts: prompt templates (`templates/commands/*.md`), the `mokata-mcp` server, and hook scripts. To integrate another harness, point it at those three using its own command/MCP/hook conventions — that's exactly what `mokata setup claude` automates for Claude Code. When a full harness integration isn't available, the CLI works anywhere a shell does: have your assistant run `mokata query callers foo`, `mokata preview`, or `mokata doctor` and read the output.

What stays true on every path: local-first (nothing leaves the machine unless you wire an external tool), every durable write human-gated, and a full audit trail. Adopted external tools are treated as untrusted input — gated and permission-scoped via the trust dial.

---

16. A guided power tour: a complete story, end to end

Everything above introduced the layers one at a time. This section is the **payoff run** — a single story driven all the way through, where you'll *watch* the HARD-GATE refuse a spec, the completeness gate block and then pass, RED-before-GREEN refuse premature code, the two-stage review run, memory capture a decision, and the audit

ledger record every move. Run each block in your fully-wired `full` stack from §2 and read the **"what just happened"** note before moving on.

Assumes you've completed §2 (so `mokata status` shows real providers). If you only did the lean install, swap `--profile full` for `--profile standard` — the flow is identical; only the providers differ.

Step 0 — confirm you're at full strength

```
mokata status
#   mokata <version> · profile: full
#   code_graph   → code-review-graph        (not grep — the real graph is bound)
#   memory_store → native-memory            (not sqlite — real memory is bound)
```

> **What just happened:** the router bound the *top* of each chain. If you see `grep`/`sqlite`, a provider isn't detected — revisit §2.4–2.5. The rest of the tour assumes real providers so you can see their effect.

Step 1 — see the plan before touching anything (zero side effects)

```
mokata preview
#   phase            gate                kind    would touch
#   brainstorm       approach-approval   human   state/approved_approach.json
#   analysis         (think-first)       check   —
#   strawman         (simplicity)        check   —
#   pre_mortem       —                   —       —
#   probes           —                   —       —
#   completeness_gate completeness+verify check  state/emitted_spec.json
#   emit             emit-approval+scope human   <your source files>
```

> **What just happened:** a **dry run** listed all 7 phases, the gate at each, and the files each *would* write — with **no writes and no ledger entries**. Note where things can block: `completeness_gate` and `emit`. This is your map for the rest of the run.

Step 2 — brainstorm, and feel the HARD-GATE (slash: `/mokata:brainstorm`)

```
mokata brainstorm
#   Q1: What's the smallest change that delivers the outcome?  …(you answer)
#   Proposed approaches:
#     A) … (tradeoffs)   B) … (tradeoffs)   C) … (tradeoffs)
#   → Approve exactly ONE approach to proceed. (no spec is allowed until you do)
mokata brainstorm --status
#   approved_approach: A  (persisted → state/approved_approach.json)
```

> **What just happened:** this is mokata's signature gate. It asked **one question at a time**, proposed 2–3 *real* approaches grounded in your codebase (the graph from §2 makes this

> grounding exact, not a guess), and **refused to let any spec proceed until you explicitly approved one**. The approved approach is now a durable constraint the completeness gate will check downstream. Try running a spec without approving — it blocks. That's `approach-approval`, the HARD-GATE.

Step 3 — drive the whole pipeline and watch the gates fire

```
        mokata playbook
        #   brainstorm_approved ........... PASS
        #   gate_blocked_initially ........ PASS   ← completeness gate
REFUSED emit (no tests mapped yet)
        #   gate_passed_after_tests ....... PASS   ← every AC now maps to a
test → emit allowed
        #   red_before_green .............. PASS   ← implementing before a
failing test was blocked
        #   review_passed ................. PASS   ← two-stage review: spec-
compliance, then quality
        #   memory_written ................ PASS   ← the decision was
captured (human-gated)
        #   RESULT: PASS
```

> **What just happened — the core guarantee, made concrete:** the **completeness gate first BLOCKED** emit because acceptance criteria had no mapped tests, then **PASSED** only once every AC mapped to a test (RED-before-GREEN traceability). In between, **RED-before-GREEN** refused to let implementation run against a test that hadn't first failed. Then the **two-stage review** checked the result against the spec *and* for quality. This is "no code ships until every acceptance criterion maps to a test" — not a slogan, an enforced sequence. **The same flow in Claude Code is:** `/mokata:brainstorm` → approve → `/mokata:spec` (blocked until ACs map to tests) → `/mokata:test` → `/mokata:develop` (RED-before-GREEN) → `/mokata:review`.

Step 4 — prove the knowledge layer is doing real work

```
        mokata index                              # build the per-file
freshness index
        mokata query callers <a function in your repo>
        #   backend: code-review-graph   degraded: false        ← a REAL
graph answered
        #   path:line  symbol  snippet …                          (exact
callers, not lexical guesses)
        mokata query blast_radius <that function> --depth 3
        #   transitive impact surface up to 3 hops — what a change here
would touch
```

> **What just happened:** on the wired stack, `backend: code-review-graph` and `degraded: false` mean these are **edge-accurate** answers, not grep approximations. On the lean stack you'd see `backend: grep, degraded: true` — same questions, approximate answers, still always available. `blast_radius` is the query reviewers reach for: *before* I change this, what's the impact

> surface? Edit a file, re-query, and watch the result's `note` carry a `STALE: …` warning until you re-`index` — staleness is **surfaced, never served silently**.

Step 5 — see memory capture a decision, and self-heal by surfacing

```
mokata memory
#   active items: N    read/write ratio: R    pending heal proposals: M
#   - decision  api.style = REST        (from this story)
#   - persistent db.engine = postgres
#   [heal] CONTRADICTION  db.engine: postgres → mysql   (approve / edit / reject)
```

> **What just happened:** the `memory_written` checkpoint from Step 3 landed here — a durable **decision** the next session (or a teammate, if shared) inherits instead of relearning. If two active items disagree, mokata **surfaces the contradiction as an old→new diff** for you to approve, edit, or reject — it **never silently rewrites** (default is no change). That surface-and-approve loop is the thing no other tool ships. The **read/write ratio** is the honesty check: if writes ≫ reads, memory is a write-only graveyard and mokata tells you.

Step 6 — see token governance pay off

```
mokata budget
#   mokata · saved 12,480 tok (63%)
#   - jit_retrieve(callers)     baseline 4,200 → actual 380   saved 91%
#   - subagent handback cap     baseline 9,100 → actual 1,900 saved 79%
```

> **What just happened:** every governance win this run was **measured** (baseline vs actual) and logged to the ledger; `budget` aggregates them. The biggest lever is **JIT retrieval** — pulling the exact reference snippets a task needs via the graph instead of dumping whole files (the §2 graph is *why* this is large). This is the enforced-cost-control layer no methodology-only tool has.

Step 7 — try the parallel path (degrade-safe)

```
mokata playbook --parallel
#   (with a subagent harness) isolates each task's context + runs the two-stage review
#   (without one)             "subagents unavailable → running sequential flow"  … RESULT: PASS
```

> **What just happened:** parallel surfaced a token/cost estimate first, stayed inside the same gates + ledger + budget, and — crucially — **degraded to the sequential flow instead of crashing** when no subagent runner was present. Never a hard failure; just fewer features with a clear message.

Step 8 — inspect everything (the governance payoff)

```
mokata audit       # every gate decision, tool call, hook, write,
```

```
healing/subagent decision — with seq
        mokata rules       # the 4-tier rules + their line budgets (over-cap
tiers flagged)
        mokata budget      # the savings readout again
        mokata memory      # decisions + pending heal proposals
```

> **What just happened:** `mokata audit` is the hero promise made literal — an append-only `.mokata/temp_local/audit/ledger.jsonl` where **every** decision the run made is recorded with a monotonic `seq`, in order. You can review and (via the undo log) walk back every action. Nothing the agent did is opaque.

Step 9 — enter mid-pipeline (the composability escape hatch)

```
        mokata enter completeness_gate        # run JUST the gate against a
hand-written spec
        mokata enter strawman --to probes     # run only a slice of phases
        mokata run review                     # run one skill standalone —
applies only its own gate
        mokata chain spec test                # a manual chain — each step
keeps its gate
```

> **What just happened:** you don't have to run the whole story. `enter` runs a slice (only those phases' gates apply; skipped upstream phases are reported explicitly), `run` executes a single skill standalone, and `chain` sequences skills with **every gate still enforced**. This is the "reach for one capability, enter the pipeline anywhere" promise — useful for existing code (jump straight to `/mokata:test`) or for running the completeness gate on a spec you wrote by hand.

What you just proved

In one story you exercised all twelve Parts: the **spine** (status/route), **knowledge** (query/index/staleness), **memory** (capture + self-healing), the **engine** (7 phases + 3 gates), **execution modes** (sequential + degrade-safe parallel), **token governance** (measured savings), **governance** (rules + Karpathy gates), **safety & audit** (secret-guarded writes + the ledger), and **composability** (enter/run/chain). And every durable write along the way was **human-gated** and **local-first** — nothing left your machine, and you can review every decision it made.

If a criterion ever has no test, the completeness gate blocks `emit`, the audit ledger records the block, and you fix the mapping (write the test) and re-run. That refusal *is* the product working as designed.

---

17. Full command reference

Every CLI command, grouped by Part. All accept the shared `--path PATH` (repo root; default cwd).

Spine (Part A) & lifecycle (Part K)

Command	Purpose	Key flags

`mokata init`	scaffold `.mokata/` (human-gated)	`--profile`, `--yes`, `--force`, `--preview`
`mokata setup <harness>`	wire mokata into a harness without the plugin	`--scope {project,user}`, `--profile`, `--no-hooks`, `--yes`, `--force`
`mokata unsetup <harness>`	reverse `setup` (leaves `.mokata/` intact)	`--scope`, `--yes`
`mokata detect`	tool-presence for the catalog (no manifest needed)	—
`mokata route [need]`	resolve a capability → tool, with the fallback chain	—
`mokata status`	one-line stack summary	—
`mokata validate`	parse + validate the manifest	—
`mokata bootstrap`	print the SessionStart briefing (≤ 2k tokens)	`--show-tokens`
`mokata coverage`	capability coverage + gaps + role overlaps	—
`mokata doctor`	diagnose manifest/config errors	—
`mokata config get/set <key> [value]`	read/update a manifest key (human-gated; secrets hard-blocked)	`--yes`
`mokata reset`	remove `.mokata/` state (human-gated)	`--keep-config`, `--backup DIR`, `--yes`

Engine & pipeline (Part D) + execution (Part E)

Command	Purpose	Key flags
`mokata brainstorm`	Socratic pre-spec exploration (HARD-GATE)	`--status`
`mokata enter <phase>`	enter the pipeline at a phase	`--to <phase>`
`mokata preview`	dry-run; zero side effects	`--start <phase>`, `--to <phase>`
`mokata progress`	run-progress tracker (done/current/pending); read-only	`--run <id>`, `--ascii`
`mokata playbook`	run the full story end-to-end	`--parallel`, `--fanout`
`mokata exec`	show/select the execution mode	`--parallel`, `--isolation`, `--fanout`

Composability (Part L)

Command	Purpose	Key flags
`mokata skills [name]`	list the catalog; reveal one skill's prompt + gate	—
`mokata run <name>`	run one skill standalone	—
`mokata chain <skill> …`	plan a manual chain (gates kept)	—
`mokata suggest`	recommend a command (never runs)	`--fresh`, `--spec`, `--failing-test`, `--implementation`, `--diff`, `--bug`, `--stacktrace`, `--perf`

Knowledge (Part B)

Command	Purpose	Key flags
`mokata query <kind> <target>`	structural query (graph or grep floor)	`--depth N` (for `blast_radius`)
`mokata index`	build/refresh the freshness index	—
`mokata lat-check`	scan `@lat` anchors, flag drift (exit 1 on drift)	—

`<kind>` ∈ `callers`, `callees`, `implementers`, `imports`, `blast_radius`.

Memory (C), token (F), governance (G), audit (I)

Command	Purpose
`mokata memory`	read-only: active items, read/write ratio, pending heal proposals
`mokata rules`	4-tier rules + line budgets (exit non-zero if over cap)
`mokata budget`	token savings readout + statusline
`mokata audit`	the append-only audit ledger

Adapters & distribution (Part H/J)

Command	Purpose	Key flags
`mokata mcp`	discover MCP servers from `.mokata/mcp.json`, map to roles	—
`mokata harness`	show the harness boundary's capabilities	—
`mokata export [file]`	export the manifest as a shareable stack	—
	validate + apply a	

`mokata import <file>`	shared stack (human-gated)	`--yes`, `--force`

18. Gate, settings & state reference

Every gate, in one place

Gate id	Where	Kind	Blocks on
`approach-approval`	brainstorm / `/mokata:brainstorm`	human	no approved approach (the HARD-GATE)
`refinement-approval`	refine / `/mokata:refine`	human	no approved scoped set of refinements (the HARD-GATE for existing code)
`completeness`	completeness_gate / `/mokata:spec`	check	any AC with no mapped test, or an empty spec
`emit-approval`	emit	human	un-approved durable output
`red-before-green`	`/mokata:test`	check	implementing before a test has failed
`no-code-without-failing-test`	`/mokata:develop`	check	code without a failing test to satisfy
`spec-then-quality`	`/mokata:review`	human	review not done in two passes
`repro-first`	`/mokata:debug`	check	a fix attempted before root cause
`measure-first`	`/mokata:optimize`	check	a change before a baseline measurement
`reproducer-required`	`/mokata:bug`	check	a fix before a reproducer exists
`think-first`	analysis (Karpathy)	check	no plan/approach before code
`simplicity`	strawman (Karpathy)	check	complexity over a cap
`surgical-scope`	emit (Karpathy)	check	touched files over a cap
`verify`	completeness_gate (Karpathy)	check	success criteria undefined/unverified

Every `settings` key

Key	Shape	Default
`memory.persistent` / `.decision` / `.episodic`	bool each	all on
`governance.output_density`	bool	off

`governance.karpathy.{think-first,simplicity,surgical-scope,verify}`	bool each	all on
`trust.<tool>`	`read-only` / `propose-only` / `gated-write`	`gated-write`
`execution.default`	`ask` / `sequential` / `parallel`	`ask`
`brainstorm.auto`	`on` / `off` / `ask`	`on`

Per-tool backend config (Stage 24A): `tools.sqlite.config.path`, `tools.obsidian.config.vault`, `tools.postgres.config.dsn_env` (env-var name — never an inline DSN).

`temp_local/state/` files

`approved_approach.json` (brainstorm handoff) · `approved_refinements.json` (refine handoff) · `emitted_spec.json` · `memory_stats.json` · `knowledge_index.json` · `story_analysis__<id>.json` · `undo_log.json` (revert) · `pipeline_run__<id>.json` (resume checkpoints; the progress tracker reads these). All transient/runtime — under `.mokata/temp_local/` (gitignored); committed config (manifest, constitution, `.gitignore`) stays at the `.mokata/` root.

---

19. Troubleshooting & FAQ

`.mokata/` is missing / a command exits non-zero immediately. Commands that need an initialized repo load the `Surface` and fail fast if `.mokata/` isn't there. Run `mokata init` (or `mokata setup claude`) first.

The completeness gate keeps blocking emit. That's the design. It blocks on an empty spec and on *any* acceptance criterion without a mapped test. Run `mokata enter completeness_gate` to see which AC is unmapped, write the test, and re-run. The block is recorded in `mokata audit`.

A graph/memory query says `degraded`. You're on the grep floor (no `code-review-graph`/`serena`) or the SQLite floor (no richer memory backend) — results are approximate but always available. Install a richer provider and re-run `mokata status` to confirm the router picked it up. Nothing ever hard-fails for a missing provider.

`mokata index` then a query still warns `STALE`. A file changed since the last index. That warning is intentional — staleness is surfaced, never served silently. Re-run `mokata index` to refresh.

Parallel didn't actually parallelize. Without a subagent-capable harness, `--parallel` degrades to the sequential flow and says so. That's expected on the bare CLI.

A write was blocked and I can't override it. If it tripped the secret-guard (the 4-layer secret scan), the block is **un-overridable** by design — a secret in outbound content is fatal. Remove the secret (e.g. move it to `.env`, which the path layer already protects) and retry.

I want to undo something mokata wrote. Durable writes record their prior value to the undo log; use the revert path (`ReversibleStateStore`). To wipe state entirely, `mokata reset` (use `--keep-config` to keep the manifest, `--backup DIR` to make it reversible).

Is anything leaving my machine? No — local-first, zero telemetry. The `minimal` profile performs **zero network egress**; on `standard`/`full`, only tools you explicitly wire (kinds `mcp`/`external`) can touch the network, and every such action is gated and audited.

How do I confirm exactly what the agent did? `mokata audit` — every gate decision, tool call, hook, write, savings event, and subagent decision, each with a monotonic `seq`, in `.mokata/temp_local/audit/ledger.jsonl`.
