AI agents, architected for production

Voltade's engineering blueprint for agentic AI in production.

Voltade
Singapore
May 2026 • v0.1

Abstract

Agentic AI is what you get when a language model becomes a *colleague*. The kind that reads transcripts, drafts replies, writes notes, schedules follow-ups, and hands off to humans when the situation warrants it. Done well, it removes hundreds of hours of repetitive work from a team's month. Done badly, it embarrasses the brand, leaks data, costs you years of customer trust, and the team finds out a week after the damage.

Most teams who have tried to put an agent in production have lived the gap between the two. The demo looked clean. The pilot worked on a Tuesday. By the time real customer messages started arriving, the agent had said something it shouldn't have, written into a field it wasn't authorised to touch, or quietly stopped working in a way nobody noticed for a week.

Voltade's agentic platform was built to close that gap. It rests on three commitments. **Firstly, every action the agent can take is a typed function in our codebase, with no hidden orchestration layer underneath. Secondly, the agent's view of the world is a filesystem of plain-text markdown files any human on the team can read, edit, and version-control in a normal text editor. Thirdly, every customer-affecting change flows through a propose-review-apply pipeline with full history; nothing important happens without a record of who decided what, and why.**

The result is an agent that's bounded by design, reversible by default, and audited to the row. Those properties are the difference between an experiment and a production system. The rest of this document is how we got there.

1. The trust gap

There's a quiet truth in the agentic-AI conversation that the demos don't acknowledge. The hard part isn't building an agent that *can* do the work. The hard part is building one your operations director will let near a paying customer.

The same pattern recurs across most agentic projects that are already in flight when we first see them. A team builds something impressive on a generic LLM. It passes the show-and-tell. Then someone asks the questions that decide whether it ever gets switched on:

- Can the agent send a refund without a person looking at it first?
- If it answers a question wrongly, can we tell which knowledge source it pulled from?
- When the customer comes back two weeks later, does the agent remember what it told them?
- When a staff member is already replying on WhatsApp, does the agent stay quiet?
- If the agent makes a mistake, what's the smallest blast radius?
- Who has access to what the agent has access to, and how do we change that without rewriting the agent?

Reasonable questions. They are the questions any operations team would ask of a new hire on day one. Most agentic-AI architectures cannot answer them without a long conversation about *prompts*, *context windows*, and *vector stores*, none of which a customer ever sees, and none of which constitute a real answer.

The trust gap is the distance between a working demo and a production system. It's where projects go to die. The architecture in this document is what we built to close it.

2. Three properties required for production deployment

Three properties separate the agents that survive contact with real customers from the ones that don't. Every architectural decision is in service of one of them. None are optional.

Bounded

The agent can only do what we let it do. A clever input cannot talk it into more.

A generic LLM with tool-use is a powerful thing. By default it's also a system where the agent's capabilities are bounded only by what someone can convince the model to attempt. That's the wrong default for production. Our agents don't have *capabilities* in the abstract. They have a specific, enumerated set of typed functions, each of which routes through a single service call with an input schema we wrote. The agent cannot run an action that isn't already in the codebase.

Anyone can open the file `modules/messaging/tools/` and see every single action the agent can take in a conversation, in two screens. No hidden orchestration, no emergent behaviour. The map is the territory.

Auditable

Every change the agent makes is reviewable, traceable, and explainable.

Some of our agents handle small writes, like appending a note to a customer's memory or flagging a conversation as stale. Some handle larger ones, like drafting a refund confirmation or updating a contact's tier. Either way, every customer-affecting write either auto-applies with a recorded reason, or queues for staff review with a diff that shows exactly what would change. The agent never updates *state* and leaves the team to discover it a week later.

When something goes wrong, and at some point something will, the question "what did the agent actually do?" has an answer that takes seconds to produce. *Every* mutation is a row in a change-history table with a before-state, an after-state, a rationale, a decider, and a timestamp.

Reversible

When the agent is wrong, the undo is cheap.

The first two properties prevent the worst outcomes. The third is what happens when one slips through anyway. Our change-history pipeline records the prior state of every write the agent makes. A revert is a single write of that prior payload. No forensic exercise, no manual reconstruction, no phone call to engineering at 11pm on a Sunday.

We design for the assumption that the agent will eventually be wrong. The architecture is calibrated for that day, and only secondarily for the day of the demo.

3. Agent-native architecture

Most agentic AI in the market today is what we'd call *AI-bolted-on*. A team built a CRM, a helpdesk, a customer-service platform, and then, in the last eighteen months, bolted an LLM on top that talks to it. The LLM lives in a sidebar. Behind the scenes it's plugged into the same internal APIs the human staff use, often with the same level of access. It writes into the same fields. It calls the same endpoints. The company hopes very much that the prompt is enough to keep it inside the lines.

It almost never is. We've taken over more than one project where the previous vendor's agent could, by design, update any contact field in the CRM from any conversation, with no audit trail beyond a generic "AI assistant edit" tag. That's what AI-bolted-on looks like in practice, and it's why those agents stay confined to read-only demos, never quite trusted with anything that touches a customer.

The platform is *agent-native*: designed from the first commit to have the AI agent as a first-class participant — a peer to the human staff member, with the same trust scaling everyone else uses. Three commitments shape every part of the architecture:

Commitment 1. Tools are typed, never magical

Every action the agent can take is a typed function with a validated input schema, a single service call underneath, and a declared trust tier. There is no "let the agent figure it out" layer. If the action isn't in `modules/*/tools/`, the agent cannot perform it. Period.

That's what makes the agent's behaviour boundable in production. We extend what the agent can do by adding a file. We cannot *fail* to know what it can do, because the catalogue is the codebase.

Commitment 2. The agent sees a filesystem of markdown

Identity, conversation transcripts, contact records, business policies, learned skills, and the agent's own memory all appear to the agent as plain-text markdown files at known paths in a virtual workspace. The agent navigates them with the same shell commands a software engineer would use: `cat`, `ls`, `grep`, `echo >>`. Customer memory lives at `/contacts/<id>/MEMORY.md`. Brand policies live at `/drive/BUSINESS.md`. The agent's job description lives at `/agents/<id>/AGENTS.md`. Staff context lives at `/staff/<id>/MEMORY.md`.

The point of this choice is human legibility. Staff can read every file the agent reads and edit every file the agent can edit, in a normal text editor. There is no embedding store to peer into, no opaque vector retrieval, no agent-internal scratchpad we have to dump and reconstitute. The agent's context is, quite literally, a set of markdown files on a virtual disk. Your team can see what it sees.

Commitment 3. Every mutation is one of three shapes

There are three ways the agent can change anything in the system: by appending prose to a memory file, by running a CLI verb that updates a structured field, or by calling a typed tool that produces a customer-visible side effect. There is no fourth shape. There is no back door. Every change the agent makes falls into one of these patterns, each of which is auditable and replayable. The ones that

touch customer-affecting state route through the propose-review-apply pipeline before anything is committed.

That's what answers the second-day question, *"What did the agent actually do?"*, by construction. There is no place to look that isn't one of the three.

4. Safety properties in practice

Architectures don't make systems safe by themselves. *Invariants* do. Properties the system maintains no matter what input it receives. We hold the platform to a small number of these. Each one is a property a production agentic system should be able to state in a single sentence.

The system prompt does not drift mid-conversation

The instructions that govern the agent are computed once when the agent wakes up, and held byte-identical across every turn of that wake. We compute a cryptographic hash of the prompt at the start and record it. If the same agent woke up on the same trigger tomorrow, the hash would match.

A drifting system prompt is the most common source of an agent that "starts strong and ends weirdly." It also defeats the prefix-caching most LLM providers offer, balloons cost, and breaks reproducibility. None of those are acceptable on a production system.

The agent's trust scales with where the input came from

The platform has three trust rungs: `contact`, `staff`, `admin`. Every conversation the agent runs is assigned a tier based on who initiated it. A wake started by a customer message inherits `contact`-tier trust; the agent can only see the actions any staff member would expose to a customer. A wake started by a staff member's internal note inherits `staff`-tier trust; the agent gains access to internal tools and richer context. A wake started by an operator running the platform CLI inherits `admin` trust.

This filtering happens at the *catalogue* level. The agent literally does not see tools or verbs it isn't trusted to use in the current context. It can't be talked into invoking them, because they aren't in its menu.

Every customer-affecting change goes through a review queue

When the agent decides to update a customer field, refine a memory note, send an outbound message, or queue a follow-up, the change is not applied directly. It enters a structured pipeline. The platform classifies the change by sensitivity, and the combination of *confidence* × *sensitivity* decides whether it auto-applies, queues for staff review, or is dropped as too low-confidence to be worth proposing.

Every applied change is recorded with a before-snapshot, an after-snapshot, a rationale, the decider, and a timestamp. Revert is a one-write operation. That's how we make the "what did the agent do?" question answerable in seconds.

Agents do not run continuously

A *wake* is one bounded execution of an agent. A single turn or set of turns, with a start, an end, a turn cap, and a cost ceiling. The agent is not a long-lived process consuming tokens between user messages. When the conversation ends, the wake ends. The agent stops. There is no always-on process to monitor for runaway behaviour, because there is no always-on process to begin with.

That also makes LLM cost predictable per interaction, not per hour. An average agent has a knowable run cost, and the maximum cost of a single wake is bounded because the turn cap and cost ceiling are set explicitly.

Agents do not talk over humans

Channels like WhatsApp have a quirk most teams discover the hard way. When a staff member replies from the WhatsApp Business app, the message echoes back to the platform as if it were a fresh inbound. A naive agent answers it. The customer ends up in a two-way conversation between the agent and the staffer, both confused.

Our channel adapters surface this behaviour to the agent as an explicit rule in its instructions: when the message has a `staff` role and an echo-source tag, stay silent. The agent does not need to be smart enough to figure that out on its own. The architecture tells it.

Agents do not wake themselves into loops

The platform's coordination primitive is the `@-mention`. When the agent flags a teammate inside an internal note, the platform wakes that teammate (human or agent) and ends the current turn cleanly. We hard-filter self-mentions, so an agent cannot mention itself into a recursive wake. The same filter applies to notes without any mention; they wake nobody.

5. Deployment scenarios

Architecture is easier to evaluate when it's concrete. Here are three representative scenarios. Each one is a single bounded wake, with a clean start, work, and exit.

Scenario A. A refund enquiry on WhatsApp

A customer messages on WhatsApp asking for a refund. The platform receives the webhook, resolves the contact, inserts the message into the conversation, and wakes a conversation-lane agent at `contact` trust.

The agent's first move is to read the refund playbook from its skills folder and the brand policies from its drive. Then it pulls up the order. The refund is inside the policy window and below the auto-approval ceiling, so the agent drafts a confirmation card with two buttons (*Confirm* and *Talk to a human*) and submits it for staff approval.

The wake ends. A staff member sees the proposed card, taps approve, and the platform fires an `approval_resumed` trigger that wakes the same agent again with the conversation context. The agent dispatches the card, the customer taps confirm, and the agent writes a single line to that customer's memory: *"Prefers WhatsApp for support."* That note is now available to every future wake on that contact.

The agent didn't ship the refund itself. It staged the refund, paused for review, resumed once reviewed, and recorded what it learned for next time. The team got a draft they could ship in one tap. The customer got an answer in minutes.

Scenario B. A staff member asks the agent for context

A senior staff member picks up a conversation she didn't open. She drops an internal note: *"@triage-agent, can you summarise what this customer asked about last quarter? I'm taking this over."*

The mention fans out as a `staff_note` trigger. The agent wakes at `staff` trust, higher than the `contact` trust it would have had on an inbound message, which means a different set of internal tools and verbs is available. It greps the conversation history, scans the contact's memory, and posts a clean three-line summary as a follow-up note. No customer-visible message goes out. The wake ends.

The agent behaves as a teammate the staff can talk to inside the system, on the same trust scaling everyone else uses. It knows where to look because everything it knows about that contact lives in files at known paths.

Scenario C. A nightly inbox sweep

At 6pm local time, a recurring schedule fires a `heartbeat` trigger on the agent. It wakes in standalone mode, with no customer on the line, at `staff` trust.

The agent's first call is `summarize_inbox`, a read-only scan that the agent's instructions explicitly mark as the cheap first move. It surveys the active conversations, finds three that have been stale for more than 48 hours, and drafts a follow-up email to staff review for each. Nothing goes out yet. The agent writes one line to its own memory (*"3 stale conversations flagged on 2026-05-11"*) and ends the wake.

Agents do self-paced work without being always-on. The schedule is the trigger, the wake is the work, the cost is bounded, and nothing customer-visible ships without staff sign-off.

6. Evaluation criteria for agentic AI systems

A production agentic system should have a clear answer to each of the following. How the architecture in this document addresses each is noted in italics.

1. **What is the full list of actions the agent can take, and where does it live in the codebase?** *One folder per module under `modules/*/tools/`. The catalogue is the code.*
2. **Can the agent take an action that isn't in that list?** *No. Tool execution routes through a typed registry; anything unregistered cannot be called.*
3. **What stops the system prompt from drifting mid-conversation?** *We compute it once per wake and hash it. Any drift would change the hash, and we test for this in CI.*
4. **How does the agent's access scale with where the input came from?** *Three trust tiers, derived from the trigger, applied to the catalogue before the agent ever sees it.*
5. **When the agent updates customer-affecting state, who reviews it?** *Every such change is routed through a propose-review-apply pipeline. Sensitivity × confidence decides auto-apply vs. queue.*
6. **How do I see what the agent did to a specific record last Tuesday?** *One query against `change_history`. Before-state, after-state, rationale, decider, timestamp.*
7. **What happens with an agent left running continuously in the background?** *It doesn't. Agents don't run between user messages. Wakes are bounded, with a turn cap and a hard cost ceiling.*
8. **What happens when a human is also typing on the same channel?** *The agent recognises the echo as a `staff` role and goes silent. The rule lives in the channel adapter.*
9. **If we want to change the agent's behaviour for a recurring scenario, what do we edit?** *A markdown file under `/agents/<id>/skills/<name>/SKILL.md`. No retraining. The next wake picks it up.*
10. **If you change LLM providers, how much of our system breaks?** *None of it. The LLM is a single seam in the codebase. We treat the provider as a configuration choice.*

PART TWO

Technical Appendix

What follows is the architecture deep-dive an engineering team would cite during diligence. Every file path is real, and every invariant is enforced in code. The platform is referred to internally as *Vobase*.

A. Architecture at a glance

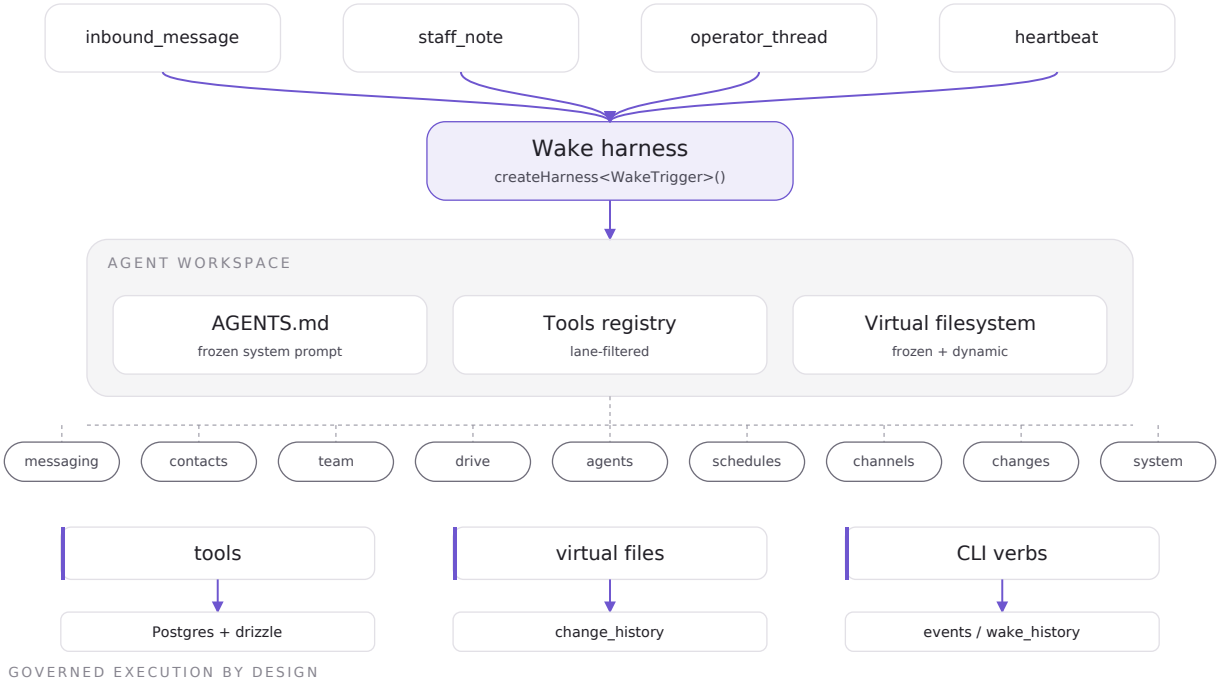

FIGURE 1. *Architecture at a glance.*

Voltade's agentic platform is a TypeScript-first monorepo organised by *module*. Each module owns one concern (`messaging`, `contacts`, `team`, `drive`, `agents`, `schedules`, `channels`, `changes`, `system`) and exposes a small, typed surface for the AI agent: tools, virtual files, CLI verbs, and AGENTS.md fragments. There is no central orchestration layer; the agent's capabilities are the union of what each module declares.

The single thesis behind every decision: **AI agents need codebases they understand.** One folder per feature, one pattern copy, one seam change. The same constraint makes the codebase legible to humans and to the AI. We optimise for both readers, in that order.

B. The wake: bounded agent execution

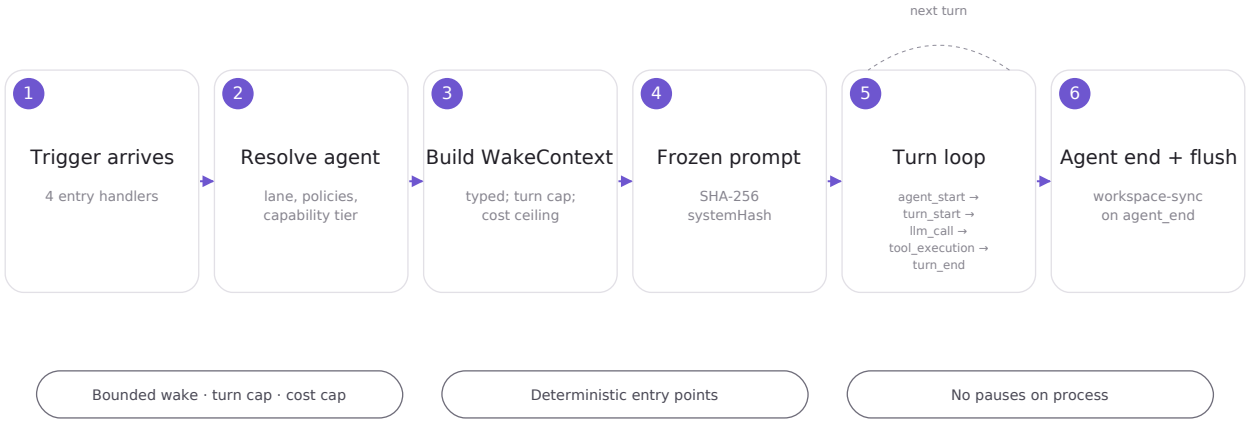

FIGURE 2. *The wake: bounded agent execution.*

A *wake* is one bounded execution of an agent. It has a single entry point, a typed `WakeContext`, a turn cap, a cost ceiling, and a defined exit.

Four entry handlers cover the trigger surface:

File	Trigger	Lane
`wake/inbound.ts`	Customer message arrived	conversation
`wake/staff-note.ts`	Staff `@-mentioned` an agent	conversation
`wake/operator-thread.ts`	Staff opened an operator DM with the agent	standalone
`wake/heartbeat.ts`	Cron tick from a registered schedule	standalone

Each handler resolves the agent definition, builds a typed `WakeContext`, calls the lane-appropriate config builder (`conversationWakeConfig` or `standaloneWakeConfig`), and hands off to `createHarness<WakeTrigger>()`. Inside the harness, the turn loop emits a strict event sequence: `agent_start → turn_start → llm_call → message_* → tool_execution_* → turn_end → agent_end`. Observers attach at well-defined points; nothing important happens between events.

Two lanes

- **Conversation lane** binds to a specific `(contactId, channelInstanceId, conversationId)`. The agent has access to customer messages, internal notes, the contact profile, and conversation-scoped tools (`reply`, `send_card`, `send_file`, `book_slot`).
- **Standalone lane** is for operator threads and heartbeats. No customer is on the line; conversation-only tools are filtered out at config-build time, before the LLM ever sees them.

Tool stdout budget

Tool results are size-bounded to keep prompts predictable. 4 KB inline fits in the next turn's side-load directly. 100 KB spill overflows the result to `/tmp/tool-<callId>.txt` for on-demand re-read. 200 KB turn ceiling is the hard cap per turn, with re-reads of spills exempt.

C. The frozen-prompt invariant

The system prompt is computed once at `agent_start` and held byte-identical across every turn of the wake. A `systemHash` (SHA-256 over the rendered prompt) is recorded on the `AgentStartEvent`. Mid-wake file writes do not change the system prompt; they appear in the *next* turn's side-load, not the system block.

Three reasons it matters:

1. **Provider prefix caches are byte-keyed.** Claude, OpenAI, Google, and the smaller hosted gateways all cache the prefix of a prompt and replay it without re-billing or re-evaluating. A drifting system prompt invalidates that cache on every turn and balloons cost and latency.

2. **Determinism.** Same trigger payload → same prompt bytes → same hash → easier to test, reproduce, audit, and explain. Retries don't quietly produce different agents.
3. **Race-safety.** The agent doesn't see its own mid-turn writes as system-prompt context, so it can't talk itself into a loop where each turn revises an instruction the next turn will revise again.

Enforcement: trigger renderers are pure functions (no DB reads, no clock, no RNG, no environment access); session context is resolved once at wake start; the workspace-sync observer flushes file changes only on `agent_end`. The static instruction block carries the rule explicitly so the agent itself reasons about it: *"files in your frozen prompt were snapshotted at the start of this wake. Mid-wake writes persist, but only show up in the NEXT turn's side-load, not in the system prompt."*

D. Audience tiers: the trust model

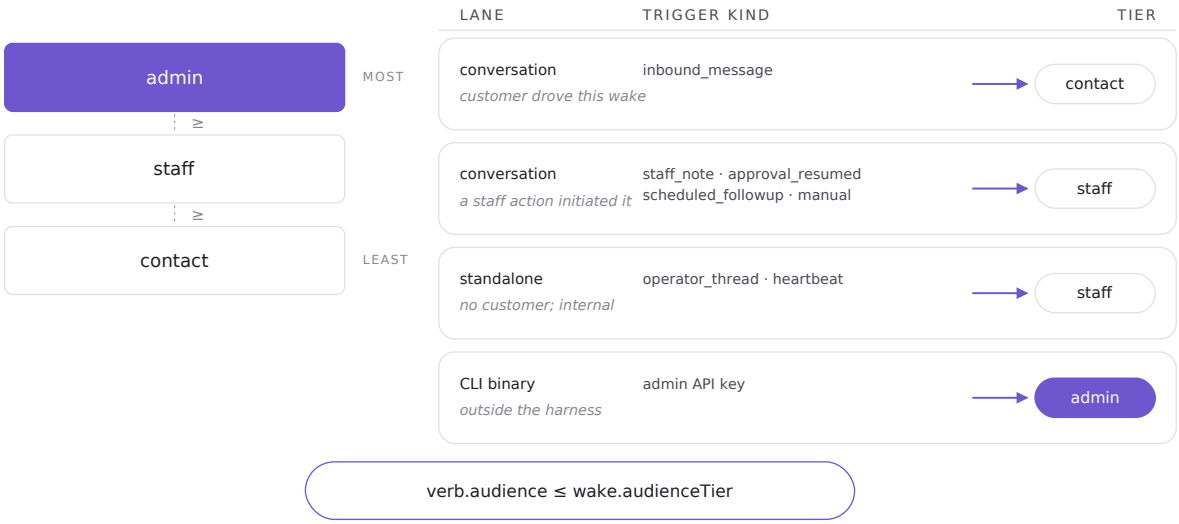

FIGURE 3. *Audience tiers: the trust model.*

`AudienceTier` is a three-rung monotonic trust scale: `'contact' | 'staff' | 'admin'` (least → most trusted). Every wake is assigned a tier at config-build time:

`(lane, triggerKind)`	`audienceTier`	Why
`conversation` + `inbound_message`	`contact`	Customer drove this wake; treat input as untrusted.
`conversation` + `staff_note` / `approval_resumed` / `scheduled_followup` / `manual`	`staff`	A staff action initiated it.
`standalone` + `operator_thread` / `heartbeat`	`staff`	No customer; internal context.
CLI binary with admin API key	`admin`	Operator-driven, outside the harness.

Every CLI verb declares its own `audience` tier. The visibility rule is `verb.audience ≤ wake.audienceTier`. The same filter applies to tools by `lane`. The agent never sees a tool or verb that would be rejected at runtime, so it cannot be cajoled into calling something it can't see.

Filtering happens in two places: the AGENTS.md `## Commands` block (which lists the verbs the agent knows about) and the in-bash `vobase --help` output. Identical predicate, two surfaces.

E. The agent's virtual filesystem

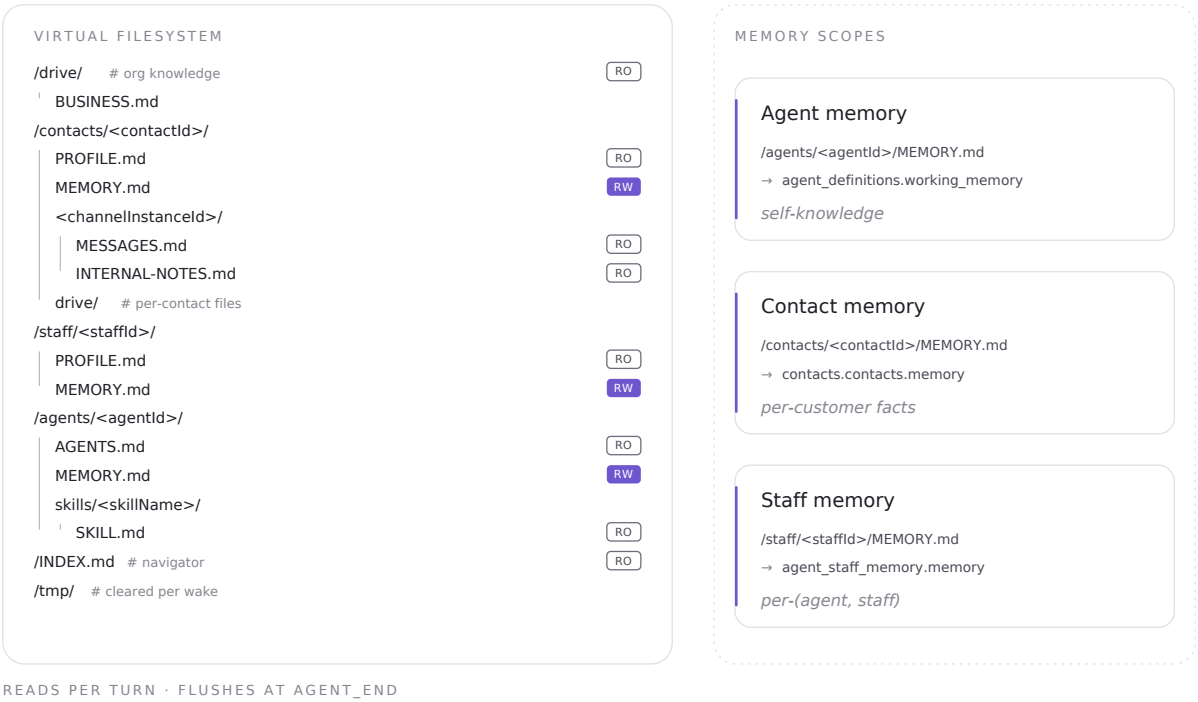

FIGURE 4. *The agent's virtual filesystem.*

The agent operates inside a bash sandbox with a virtual filesystem materialised at wake start. It runs `cat`, `ls`, `grep`, `echo >>`, and `vobase` CLI verbs. No SQL, no HTTP, no internal IDs to memorise, no opaque APIs to learn.

Top-level layout:

```
/drive/                          # org-wide knowledge base (read-only)
  BUSINESS.md                    # brand, policies, FAQ
  ...                            # uploaded docs

/contacts/<contactId>/
  PROFILE.md                     # RO; identity, frontmatter
  MEMORY.md                      # writable; per-customer facts
  <channelInstanceId>/
    MESSAGES.md                  # RO transcript
    INTERNAL-NOTES.md            # RO staff notes
  drive/                         # per-contact files

/staff/<staffId>/
  PROFILE.md                     # RO; name, title, expertise
  MEMORY.md                      # writable; per-(agent, staff) memory

/agents/<agentId>/
  AGENTS.md                      # RO; the system prompt itself
  MEMORY.md                      # writable; agent's own working memory
  skills/<skillName>/
    SKILL.md                     # playbook for a recurring scenario

/INDEX.md                        # top-level navigator
/tmp/                            # scratch; cleared per wake
```

Materialisers

Each module declares a list of `WorkspaceMaterializerFactory<WakeContext>` functions on its `agent.ts`. At wake start the harness invokes every factory with the live `WakeContext` and collects the resulting `WorkspaceMaterializer[]`.

Each materialiser declares a `path`, a `phase` (`frozen` = rendered once, `dynamic` = re-rendered per turn), and an async `materialize()` returning the file body.

Drive overlays

Some paths are not files in storage. `/contacts/<id>/MEMORY.md`, for instance, is backed by the `memory` column on `contacts.contacts`. The drive subsystem exposes a *provider* interface (`registerDriveOverlay`); each module registers a provider that synthesises virtual rows for `listFolder` and resolves bodies for `read` and `write`. Providers always scope by `organizationId`. Every overlay read or write is type-safe and audit-logged. The file is the agent-facing projection of a service call.

F. The memory model: three scopes

The agent has three durable memory scopes. All three are written by direct file appends inside the wake (`echo "- ... " >> /…/MEMORY.md`); a workspace-sync observer flushes the diffs to backing columns at `agent_end`.

Scope	Virtual path	Backing column	Meaning
agent	`/agents/<id>/MEMORY.md`	`agent_definitions.working_memory`	Self-knowledge. *"Always confirm refund window before quoting an amount."*
contact	`/contacts/<id>/MEMORY.md`	`contacts.contacts.memory`	Per-customer facts. *"Prefers SMS over email. Account on Pro tier since 2025-03."*
staff	`/staff/<staffId>/MEMORY.md`	`agent_staff_memory.memory`	Per-(agent, staff) memory. *"@billing-lead handles refund overrides above $100."*

A budget header is prepended at materialisation time (`<!-- memory-budget scope=agent ... -->`) showing soft cap usage; it's stripped before storage so re-renders don't stack headers. Default soft cap: 8000 UTF-16 code units per scope.

Capture heuristics, surfaced in AGENTS.md, decide which scope a memory write lands in. A line that contains a contact name routes to the contact's memory file. A line that starts with *always*, *never*, or *from now on* routes to the agent's working memory. A line that names a specific staff member's pattern routes to staff memory. Anything else stays in the agent's own working memory by default.

Staff memory has no `/drive/**` mirror; outside a wake, the only way to read or edit it is via Drizzle Studio. That's intentional. Staff memory is internal context for the agent, separate from customer-visible data.

G. AGENTS.md: the system prompt, assembled

`AGENTS.md` is the agent's read-only view of its own job description. It is the bulk of the system prompt. It's rebuilt for every wake, lane-aware, tier-filtered, and assembled from contributions across modules.

Nine sections, in order:

1. **Preamble:** active agent / contact / channel IDs.
2. **System IDs block:** explicit IDs for org, channel instance, contact, agent.
3. **Session context:** channel kind, contact name, assignee, conversation status, customer-since date.
4. **Platform hints:** channel-specific authoring guidance.
5. **`/agents/<id>/MEMORY.md`:** agent's lesson capture.
6. **`/agents/<id>/AGENTS.md`** body: instructions, tools table, CLI verbs table, module-contributed fragments.
7. **`/drive/BUSINESS.md`:** org policies, brand, FAQ.
8. **Skills list:** files discovered under `/agents/<id>/skills/` with descriptions.
9. **Static instructions:** bash sandbox rules, frozen-zone rule, verb-vs-tool guidance.

The rendered string is SHA-256'd into `systemHash` and held constant for the wake.

Module fragments

Modules contribute fragments via their `agent.ts` (the `agentsMd` slot). Each fragment is an `IndexContributor` with a priority and an optional gating predicate over `(lane, triggerKind)`. A fragment whose predicate doesn't match the current wake simply isn't emitted, so the agent never reads a section that doesn't apply.

Representative fragments:

- `agents.self-state` (priority 20): describes MEMORY.md, the skills folder, the `/tmp` scratch space.
- `agents.memory-capture-triggers` (priority 25): the *always / never / from now on* keyword list.
- `drive.organization-knowledge` (priority 30): `/drive/*` as the read-only knowledge base.
- `contacts.contact-context` (priority 40): PROFILE.md, MEMORY.md, per-contact `/drive/`.
- `messaging.conversation-surface` (priority 50): conversation-lane tools and `vobase conv reassign`.
- `messaging.staff-note` (priority 60, gated to conversation + staff_note): three-step playbook for staff `@-mentions`.
- `messaging.standalone-no-customer` (priority 60, gated to standalone): *"No customer is on the line."*
- `channels.whatsapp-echoes` (priority 55, gated to conversation): coexistence rules for WhatsApp Business App echoes.
- `team.staff-roster` (priority 60): staff PROFILE/MEMORY conventions.

H. Tools catalogue

Every tool is a typed function with a Zod input schema, a declared lane, and a single service call.

Messaging (`modules/messaging/tools/`)

Tool	Lane	Effect
`reply`	conversation	Append text message; dispatch via channel adapter.
`send_card`	conversation	Append rich card (text/image/divider/actions/fields/link children). Approval-gated if `agent.cardApprovalRequired`.
`send_file`	conversation	Send drive file; scope-checks file against conversation contact.
`book_slot`	conversation	Stub for calendar integration. Approval-gated.
`add_note`	both	Append internal note. Each `@-mention` enqueues a staff-note wake.
`summarize_inbox`	standalone	Read-only scan of conversations, safe to call repeatedly.
`draft_email_to_review`	standalone	Queue email draft for staff review; nothing ships until approved.

Contacts (`modules/contacts/tools/`)

Tool	Lane	Effect
`update_contact`	standalone	Mutate contact (displayName, phone, email, segments).
`propose_outreach`	standalone	Queue outbound message for staff review.

Agents (`modules/agents/tools/`)

Tool	Lane	Effect
`remember`	both	Commit a lesson to durable memory through the change-proposals pipeline; sensitivity × confidence routes the change to `auto_written` / `pending` / `dropped`.
`dismiss_candidate`	both	Mark a learning candidate as dismissed.

Schedules (`modules/schedules/tools/`)

Tool	Lane	Effect
`create_schedule`	standalone	Insert recurring heartbeat.
`pause_schedule`	standalone	Disable a schedule.

Drive (`modules/drive/tools/`)

Tool	Lane	Effect
`request_caption`	conversation	Annotate a file with extracted-text caption.

Per-tool prompts carry strong opinions. `reply` is reserved for pure acknowledgements, free-form questions, single-sentence facts with no CTA, and short clarifying asks. `send_card` is the preferred reply format whenever the customer has options to choose, confirm, compare, or act on. `summarize_inbox` is marked as the cheap first move when triaging: read-only, safe to call repeatedly within a wake.

I. CLI verbs: the bash sandbox

Inside a wake the agent runs `vobase ...` commands as if it were a staff member at a terminal. Each verb registers in a shared `CliVerbRegistry`; the same body runs both in-process during a wake and over HTTP-RPC when the actual `vobase` binary is invoked by an operator. Verbs declare `audience` so the in-bash `--help` filters by wake tier.

Canonical set:

Verb	Module	Audience	Purpose
`agents list`	agents	admin	Inventory agent definitions.
`agents show <id>`	agents	staff	Full definition.
`agents inspect <id>`	agents	admin	Dump instructions + memory tail + allowed tools + model.
`team list`	team	contact	Staff roster.
`team get <id>`	team	contact	Staff profile.
`drive propose --path --body --rationale?`	drive	contact	Raise a change proposal against a drive file.
`drive upload`	drive	contact	Upload file to drive.
`drive search <query>`	drive	contact	Keyword search.
`conv reassign --to=user:<id>|agent:<id>|unassigned --reason?`	messaging	contact	Reassign conversation. Handoff playbook requires a customer-facing acknowledgment first.
`channels list`	channels	contact	Channel instances.
`messaging close`	messaging	staff	Close a conversation.
`messaging show`	messaging	staff	Show conversation detail.
`install`	system	admin	Tenant bootstrap.

Argv flags coerce to JSON-Schema types before validation, so verb schemas use strict `z.number()` / `z.boolean()`. No `z.coerce.*` allowed.

J. Skills: user-editable playbooks

Skills are markdown files materialised under `/agents/<id>/skills/<name>/SKILL.md`. Each has YAML frontmatter (`name`, `description`) and a body containing the playbook. They're discovered at wake start and listed by name in the AGENTS.md `## Skills` section, so the agent knows when to `cat` them.

Default skills ship under `modules/<m>/skills/`. Two representative examples:

- **`refund-flow`:** a five-step playbook. Confirm order id via `vobase contacts show`. Check `/drive/BUSINESS.md#Policies` for eligibility window. Within window and amount ≤ $100, draft a confirmation card. Over $100 or outside window, mark `requires_approval=true` and mention `@billing-lead`. Never confirm verbally before the card is acknowledged.
- **`template-selection`:** when to use `send_card` vs `reply`. *"When in doubt, send a card. Cards are tap-to-reply; prose forces the customer to type."*

Skills can be edited by staff (through change proposals) or added at install time. They are the primary way operators encode local policy without rewriting the agent's instructions. Change the `.md`, and the next wake picks it up.

K. Auditable changes: propose, decide, apply, history

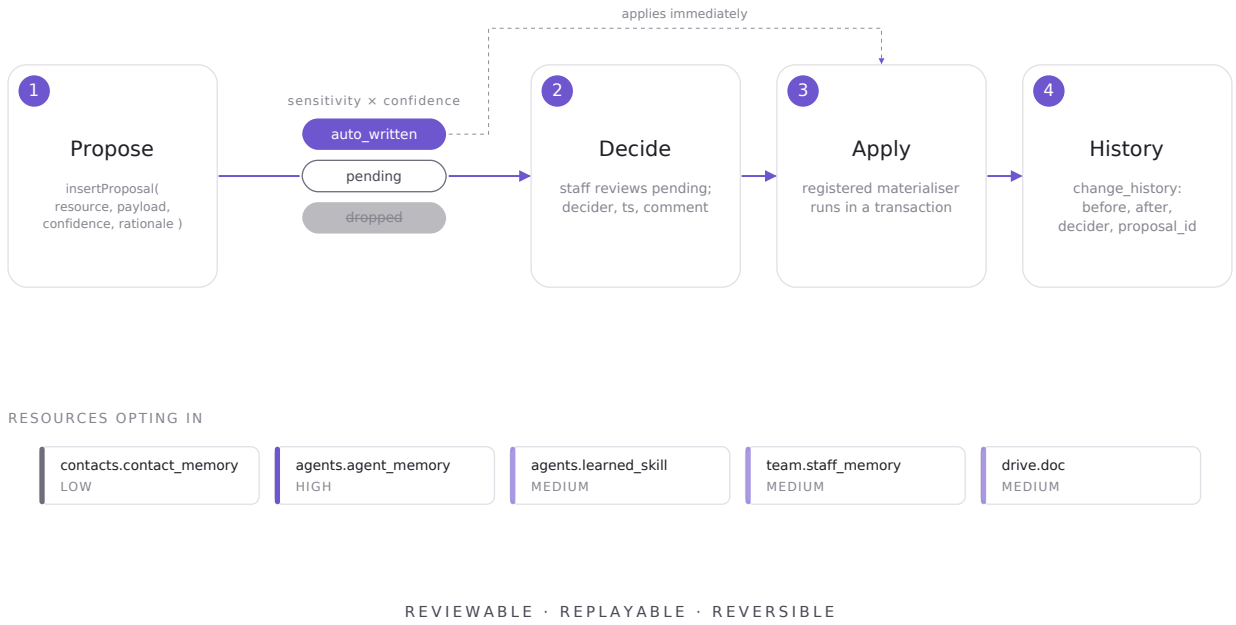

FIGURE 5. *Auditable changes: propose, decide, apply, history.*

Every non-trivial mutation an agent makes against shared state flows through `modules/changes/`. This includes memory writes above a confidence threshold, edits to learned skills, structured updates to contact fields, and outbound message proposals.

The flow

1. **Propose.** `insertProposal({ resourceModule, resourceType, resourceId, payload, confidence, rationale, expectedOutcome, conversationId })`. Routes by `effectiveSensitivity(resource, payload) × confidence`:
   - High confidence + low sensitivity → `auto_written` (applies immediately).
   - Lower combinations → `pending` (staff review queue).
   - Below floor → `dropped`.
2. **Decide.** Staff reviews `pending` proposals at `/api/changes/proposals/{id}/decide`. Decisions are recorded with the deciding user, timestamp, and any free-text comment.
3. **Apply.** Accepted proposals run their registered materialiser in a transaction.
4. **History.** `change_history` records before-snapshot, after-snapshot, decider, timestamp, applied proposal id. Queryable per resource.

Materialiser registry

Each resource module registers a materialiser at boot:

```
{
  resourceModule, resourceType,
  sensitivity,                // 'HIGH' | 'MEDIUM' | 'LOW'
  sensitivityForFields?,      // per-field overrides
  promptHint,                 // short description used in triage
  materialize(proposal, tx)   // applies the change
}
```

Resources opting in (canonical example: `contacts`):

- `contacts.contact_memory`: LOW; agent prose to customer memory.
- `agents.agent_memory`: HIGH; large agent self-edits.
- `agents.learned_skill`: MEDIUM; skill body changes.

- `team.staff_memory`: MEDIUM.
- `drive.doc`: MEDIUM.

What you get out of the pipeline

The pipeline makes every customer-affecting state change *reviewable*. It either auto-applies with a recorded rationale or queues for staff with a diff that shows exactly what would change. It makes the outcome *replayable*. A `change_decided` trigger can wake the original agent to follow up with the customer (*"Approved; here's your refund timeline."*). And it makes the change *reversible*. History stores the before-state, so revert is a write of the prior payload.

L. Channel adapters

Channels are pluggable adapters under `modules/channels/adapters/<name>/`. Each adapter knows how to receive inbound messages from its medium and dispatch outbound ones. The agent doesn't talk to channels directly. It calls `reply` / `send_card` / `send_file`, and the messaging service routes to the right adapter.

Inbound flow

```
Message arrives on adapter (web socket, WhatsApp webhook, ...)
  → contact resolved or created (upsertByExternalKey)
  → row inserted into conversation_events
  → messaging fan-out fires inbound_message trigger
  → wake/inbound.ts builds conversation-lane wake
  → agent reads MESSAGES.md, calls tools
  → outbound tools route to adapter
  → adapter translates and dispatches
```

Adapter-specific rules as AGENTS.md fragments

- **web:** markdown rendered, near-instant reply expected.
- **whatsapp:** plain text only (markdown renders literally), 24-hour session window for free-form replies (outside the window: only pre-approved templates), `send_card` renders as buttons or list pickers. The adapter also contributes a coexistence rule: *"When you see `role='staff'` with `metadata.echoSource='business_app'` (human staffer answering from the WhatsApp Business app): stay silent. Do not reply, send card, drop a note, or reassign. End turn cleanly."*

M. Coordination patterns

Staff-agent collaboration via `@-mentions`

`add_note` accepts a `mentions` array. Each mentioned staff member receives a notification. Each mentioned *agent* is woken (`staff_note` trigger). This produces a clean two-way channel:

- An agent that hits a question outside its competence calls `add_note(body: "...", mentions: ["@billing-lead"])` and ends its turn.
- The staff member replies in the UI or in chat.
- The reply fires a `staff_note` wake on the original agent with the staff's answer in context.
- The agent relays the answer to the customer and writes a one-line lesson to memory.

Two guard rails: an agent's own internal notes never fan out to itself (hard filter in `notes.ts`), and notes without any `@-mention` wake nobody.

Handoff to a human

`vobase conv reassign --to=user:<staffId> --reason='customer asked for human'`. Convention: acknowledge the customer first (*"Connecting you with a teammate now..."*), drop an `add_note` with context for the receiving staff member, then reassign. The conversation status flips and the next message routes to the human inbox.

Self-paced work (heartbeats)

An agent can call `create_schedule(slug, cron, timezone?)` to install a recurring heartbeat. At each tick, `wake/heartbeat.ts` fires a `heartbeat` trigger with the schedule context. Use cases include nightly inbox sweeps, weekly summary email drafts, periodic memory consolidation, and recurring data-quality checks. Harness errors on a heartbeat are swallowed and logged so one broken schedule doesn't starve siblings.

Approvals

Tools tagged for approval (`send_card` when `agent.cardApprovalRequired`, `send_file`, `book_slot`, `draft_email_to_review`, `propose_outreach`) insert into `pending_approvals` instead of dispatching. When a staff member decides, an `approval_resumed` wake fires on the original agent so it can finish the turn, whether that's confirming to the customer, logging to memory, correcting course, or escalating to a teammate.

N. Safety invariants and enforcement

Invariant	Mechanism
System prompt is byte-stable across a wake.	Pure trigger renderers, frozen session context, observer flushes deferred to `agent_end`.
Agent cannot see tools above its trust tier.	`lane` filter + `audience ≤ wake.audienceTier` filter applied to both AGENTS.md and the tool catalogue.
Every mutation has one write path.	`check:shape` script (CI) restricts writes to `messages`, `conversation_events`, `change_proposals`, `change_history` to specific service files.
No cross-tenant data leak through overlays.	Drive overlay providers must scope by `organizationId`; any provider returning cross-org data is a Sev-1.
Customer-visible writes are auditable.	Approval-gated tools insert into `pending_approvals`; all state-changing proposals route through `modules/changes/`.
Agent cannot talk over a human on the same channel.	Channel-specific echo rules surface as AGENTS.md fragments (e.g., WhatsApp Business App echo silence).
Agent cannot recursively wake itself.	`add_note` fan-out hard-filters self-mentions in `notes.ts`.
LLM provider lock-in is contained.	Single seam at `wake/llm.ts`: Bifrost gateway when `BIFROST_*` env vars are set, direct OpenAI / Anthropic / Google otherwise.

O. Distinguishing properties of the architecture

A short index of properties that fall out of the design. Each one is a *consequence* of how the system is built, with no feature flag toggled on:

1. **Tools are typed and bounded.** No free-form function-calling against arbitrary internal APIs. The agent can do exactly what `modules/*/tools/` enumerates.
2. **The agent's view of the world is reviewable code.** Open `modules/messaging/agent.ts` and read every conversation-lane tool, every AGENTS.md fragment, every file it can read, in two screens of source.
3. **Memory is prose, addressable by path.** Storage is markdown. Staff read and edit it in any text editor. A new agent on a known contact reads exactly what the previous agent wrote.
4. **Every wake is bounded.** Single execution unit, frozen system prompt, turn cap, cost ceiling. No always-on agent process.
5. **Multi-agent coordination is a first-class primitive.** `@-mentions` wake either humans or agents; a triage agent can hand to billing, which can hand to `@billing-lead`, which can hand back.
6. **Channel quirks are agent guidance.** WhatsApp echoes, 24-hour session windows, web vs WhatsApp markdown all surface as AGENTS.md fragments contributed by the channel adapter. The agent reads them as natural-language rules.
7. **The CLI is the same for agents and operators.** Staff typing `vobase drive propose ...` and an agent running `vobase drive propose ...` inside its sandbox execute the same code path with the same authz; only the audience tier differs.
8. **Determinism over cleverness.** Pure trigger renderers, SHA-256'd prompts, budgeted stdout, deferred observers. Same input → same wake.
9. **Skills as user-editable artifacts.** Local policy lives in skill markdown files staff can rewrite. No retraining, no fine-tuning, no model surgery.
10. **Provider-agnostic.** Bifrost or direct OpenAI / Anthropic / Google is a configuration choice. Model upgrades are an env var.

P. Glossary

- **Wake:** one bounded execution of an agent.
- **Lane:** `conversation` (bound to a customer) or `standalone` (no customer present).
- **Trigger:** what woke the agent (`inbound_message`, `staff_note`, `heartbeat`, etc.). Mapped to a lane and a wake-reason cue.
- **`AudienceTier`:** `'contact' | 'staff' | 'admin'`. Derived from `(lane, triggerKind)`. Filters verbs and tools.
- **`WakeContext`:** per-wake fact bag handed to every materialiser factory. Identity, handles, lane-filtered slices.
- **`AgentContributions<WakeContext>`:** boot-time aggregate of all modules' agent slots (tools, materialisers, agentsMd fragments, roHints, listeners, sideLoad).
- **Materialiser:** a function that produces a file in the agent's virtual workspace. Frozen materialisers run once; dynamic ones re-run per turn.
- **AGENTS.md:** the agent's read-only system-prompt body. Assembled per wake from module fragments.
- **MEMORY.md:** writable prose memory at three scopes (agent / contact / staff).
- **Skill:** a playbook markdown file at `/agents/<id>/skills/<name>/SKILL.md`.
- **Drive overlay:** a virtual-path provider that synthesises filesystem rows backed by a service.
- **Change proposal:** a routed mutation. Auto-applied or queued for staff review based on confidence × sensitivity.
- **`systemHash`:** SHA-256 of the rendered system prompt. Held constant across a wake. Powers prefix-cache reuse.
- **Steer queue:** between-turn customer messages append here; drained after `tool_execution_end`. Staff notes and approval-resumed events hard-abort and re-wake instead.

Reference implementation

The architecture described here can be verified against the platform source. File paths, contracts, and invariants are all present in code at github.com/vobase/vobase.