Technical white paper

CrewWork Runtime and Domain Architecture

By Scott Weiss · CrewWork Engineering · Revision 2026-09-04

Executive summary

CrewWork is a self-hosted development platform that connects feature requests, errors, and quality findings to autonomous implementation and human review. Its central design rule is that a model's claim of success cannot satisfy the platform's completion-evidence requirement.

A shared Core runtime owns execution, budgets, recovery, and validation across product workflows. Domain services define the work; the workspace presents the plans, changes, and results people need to make decisions.

The application runs as a single-host Docker Compose stack. Generative model endpoints and a separate embedding model are configured independently, typically on local or LAN hardware. CrewWork is in active development; this paper describes implemented architecture, its controls, and the limits of what those controls establish.

Architecture at a glance

- One Core runtime contract for all non-chat execution, with a delegated inner runner for the model, tool, edit, validate, and repair loop (Section 2).
- Separate runtime transports for chat and non-chat flows.
- Nineteen independent domain packages, with cross-domain coupling measured rather than asserted (Section 3).
- Workbench surfaces keep platform operations and project-scoped work separate (Section 4).
- Two-stage structured extraction, so the same revision-pinned Qwen3.8 27B generative model serves planning, coding, review, extraction, and chat; embeddings use a separate model (Section 15).
- Event envelope and projection model for coherent frontend state (Section 11).
- Provider-neutral model transport with local-first routing across configured workload pools.
- Typed context graph with budget-aware prompt compilation and layered hash caching (Section 5).

• Artifact quality gates with automatic retry, and a same-reason breaker that stops repeated identical failures (Section 9).

• Review-first remediation: isolated delivery branches for project fixes, draft pull requests for platform self-repair.

1. System Model

The platform is organized around a clear separation: Core API as transport, domain services as planners and projectors, model transport as a provider-configured boundary, runtime as execution authority, and explicit data and observability layers.

The single-host Compose stack runs nginx, the React frontend, and the Core API; PostgreSQL, Redis, and Qdrant; the Events service; the Container Orchestrator, which holds the only Docker socket in the stack; three task-worker lanes (indexing, autonomous, and self-repair); and Prometheus, Grafana, Alertmanager, Loki, Promtail, and Tempo with node, Redis, and Postgres exporters.

Nginx owns every browser-reachable route: the frontend, the API and auth routes, the authenticated /ws WebSocket, the authorized preview proxy, and the monitoring proxy, which strips browser credentials before forwarding to Grafana, Prometheus, or Alertmanager. The Events service has no browser route; it is reached only at its service address, and all browser realtime traffic reaches Core through /ws. External MCP clients such as Claude Desktop and Cursor reach a stateless CrewMate endpoint that exposes six read-only, project-scoped tools (workspace listing and reading, code search, git status, and git diff) with a scoped API key.

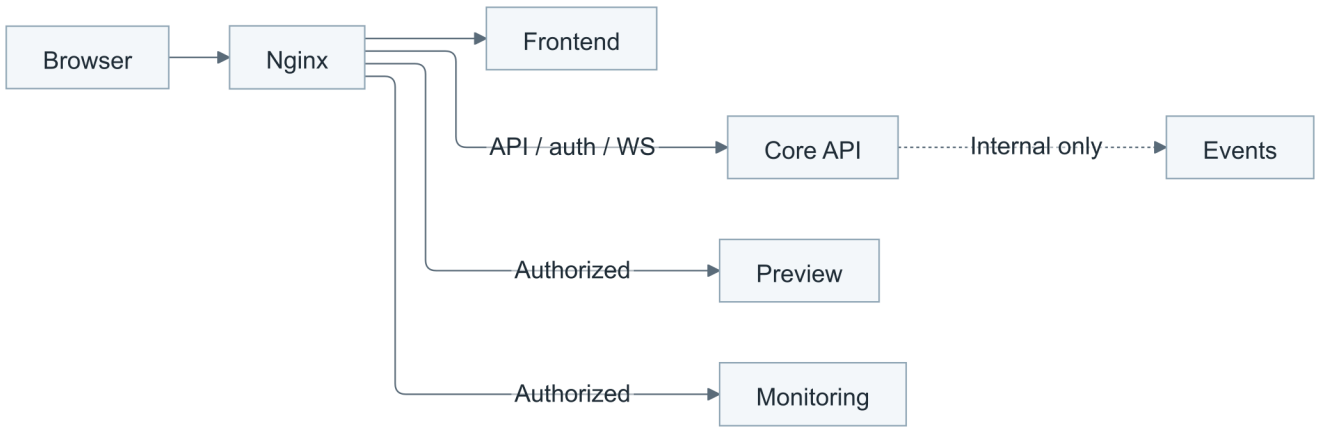

Figure 1. Ingress: every browser-reachable route passes through nginx; the Events service is internal only.

2. Runtime Contract

Every execution path submits canonical work items to the Core runtime. Runtime performs policy checks, applies workspace and lease control, executes through configured model transports, and publishes canonical results and events.

Execution has two harness layers. The outer delivery runtime owns project state, queues, leases, budgets, model lanes, source control, conversation history, events, evidence projection, and terminal decisions. The inner runner receives a typed work item and owns the model, tool, edit, validate, and repair attempt. Eligible delivery work defaults to a delegated inner-runner execution backend that runs inside the Container Orchestrator; an in-process agentic loop stays behind the same contract as the operator-controlled fallback. Admission, budget, turn, and validation failures fail closed rather than switching backends silently.

The lifecycle of one work item, in order:

1. A domain trigger fires and the domain planner shapes the request.
2. The domain submits a canonical work item; runtime policies are applied.
3. Runtime takes a lease and a workspace lock, then a budget and retry policy.
4. The inner runner (delegated backend by default, in-process loop as fallback) executes through the model transport to the configured host.
5. The repository diff is taken in the git worktree and validation and tests run against the changed workspace; a failure produces a diagnostics packet for a guided repair-and-retry loop.
6. The result and artifacts are persisted, execution events reach the Redis stream, domain projections update, and workbench surfaces receive them over WebSocket.

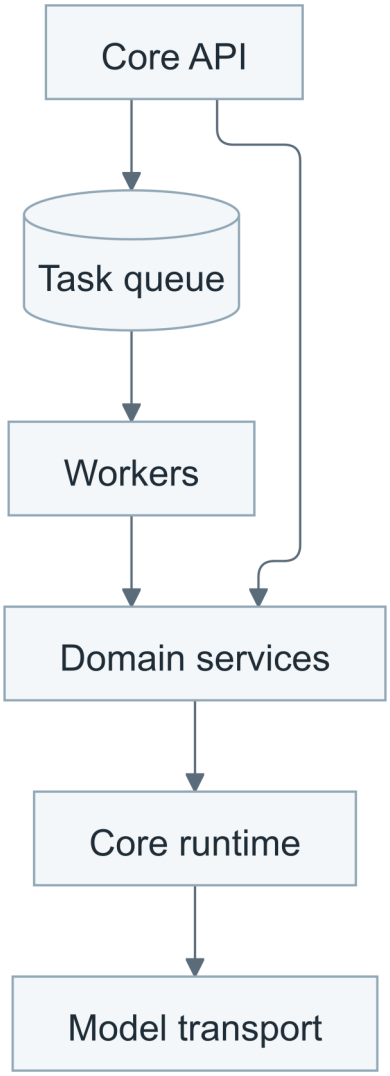

Figure 2. Core API, the Postgres task queue, worker pools, domain services, and the runtime behind them.

- Canonical result fields include status, reason code, files changed, diagnostics, and token usage.
- Attempt lifecycle is persisted and recoverable across worker restarts.
- In the in-process loop, a failed repair attempt persists a bounded conversation history and the next attempt resumes from it instead of restarting cold.
- Dry-run and policy constraints are enforced as first-class runtime outcomes.

3. Domain Model

Domains are independent by default. Each of the 19 domain packages, from Infinite Coder delivery, hotfix, and platform self-modification to search, suggestions, diagnostics, projects, and releases, owns its planning and projection logic while delegating execution to the Core runtime. No domain owns its own retry, budget, or lease stack. Cross-domain coupling is measured, not asserted.

Measured on 2026-09-03 at platform commit be012db42f, 73 cross-package import edges connect the 19 packages. Projects is the hub, imported by 17 of the other 18; HotFix (9 dependents), Delivery (8), and Search (7) follow. Secret management has no cross-domain edges in either direction. Six domains submit work through the shared base seam (delivery runtime, diagnostics, hotfix, projects, search, and suggestions); delivery runtime drives the kernel directly, and HotFix pull-request review and Search reasoning also call the generation kernel. Suggestions has no Qdrant coupling; its similarity runs in memory over embeddings, a model-transport dependency.

CrewWork is a multi-tenant platform: every project is owned by an organization and a specific team, not by an individual user, and the creator is retained only as audit metadata. Authorization is layered on top of domain isolation: a platform role ceiling combines with organization and team membership roles to govern access to projects, source control, and other sensitive operations.

4. Workbench Scope Model

The workbench is the operator-facing shape of the architecture: a single shell that separates platform-wide work from project-scoped panels. The destination switcher carries Work, Projects, Organization, and the admin-gated CrewWork Control console, with personal Settings sitting apart from them. Work lands on an Inbox that separates Needs you, Active, and Done, alongside runs and cross-project action items. Projects includes cross-project Quality & Security and Releases posture views. Project-scoped Source Control and Preview panels operate against a selected repository, alongside read-only code tabs and the CrewMate assistant panel. Surface labels track the current workbench build and evolve with the product; the scope boundaries are the stable contract.

- The Work surface’s Runs tab presents runtime state, evidence, logs, diffs, and token usage from canonical projections, alongside cross-project Action Items, a Background Jobs queue, and remediation Campaigns.
- The Projects surface’s Code tab (the Project Intelligence panel) covers composition, hotspot spotlight, recommendations, cross-project search and reviewed refactoring, relationship browsing, and codebase Q&A. Security scanner output lives on the Quality & Security tab, and action items with linked external issues live on the Work tab, without mixing platform-only controls into project workflows.
- Source Control and Preview stay project-scoped because they act on repository state and preview containers.

- CrewWork Control remains global for service health, incidents, usage and per-model cost trends, governance, and a redacted platform audit log, restricted to platform administrators.

5. Context Graph and Prompt Compilation

The system builds a typed directed acyclic graph of about two dozen node types, including repo summary, service map, docker-compose overview, nginx config, test overview, API contracts, preview spec, run history, and run failures. It then compiles them into deterministic prompts with budget-controlled sections. When context is dropped due to token limits, the system automatically increases the budget for the next run.

Layered hash caching over repository, step, snapshot, and prompt-bundle state ensures graphs are only rebuilt when the codebase or task state changes. Planning-context and enrichment nodes, including semantic context, context freshness, and authority docs, are compiled from the canonical context graph into every task prompt. Budget optimization tracks token usage per section and reallocates it across runs to maximize context coverage within model limits. The result is a compiled, budget-aware, auto-tuning prompt rather than a file dump, which is what lets a smaller self-hosted model work from the evidence that matters.

6. Infinite Coder Workflow

Infinite Coder coordinates program intent, delivery planning, and completion review. The planner emits one coherent feature slice with a single implementation step for the whole unresolved charter, so the inner runner can complete it in one continuous session; a separate non-editing validation step is added only when the repository's own terminating validation requires it. Intermediate checkpoints use deterministic validation and direct inner-runner repair. One whole-program semantic review runs after implementation is exhausted, and an incomplete verdict produces one consolidated repair packet on the same anchor. Execution is Core-runtime-backed, so coding behavior stays consistent with hotfix and other autonomous workflows.

A generated delivery plan for a user-owned program now stops in a pending-approval state once it passes automated quality checks. Infinite Coder execution does not begin until the plan is explicitly approved or rejected. Service-owned platform self-repair programs remain auto-approved so the self-repair lane is not blocked on a human decision.

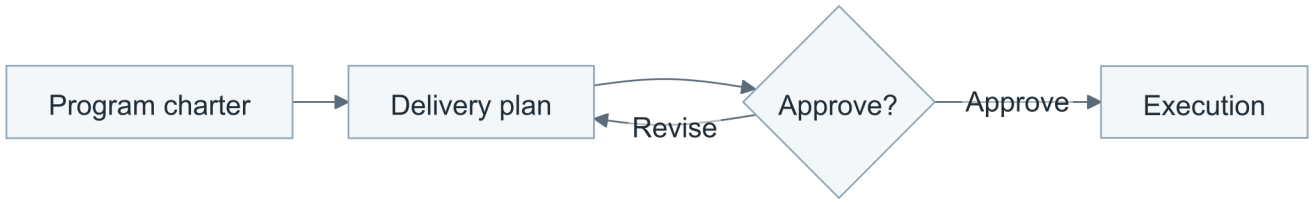

Figure 3a. A user-owned program moves from charter to plan approval.

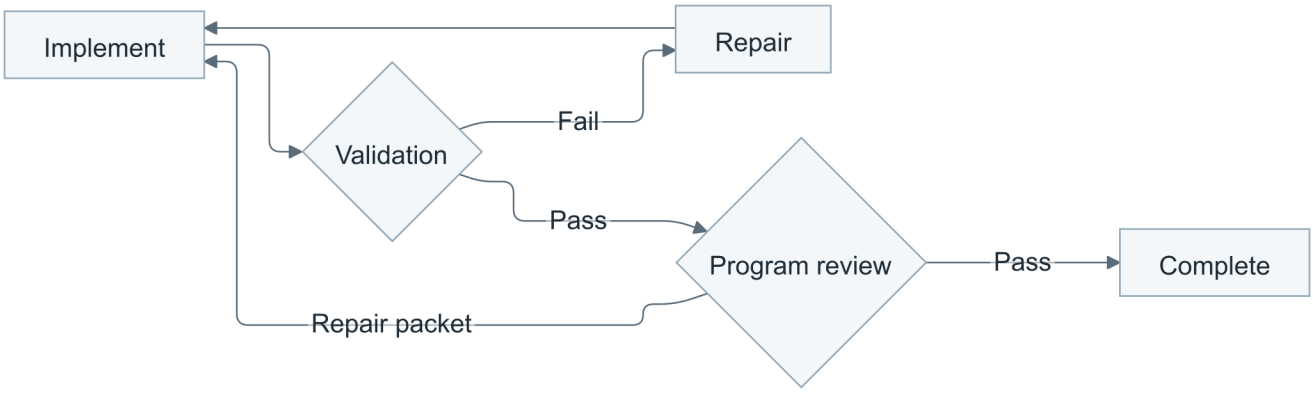

Figure 3b. Execution uses deterministic validation and bounded repair, followed by whole-program review.

7. Hotfix Remediation Workflow

Error ingestion from runtime, preview, and Sentry enters a scoped triage pipeline. Remediation work is executed through the Core runtime and returned with artifacts and policy outcomes.

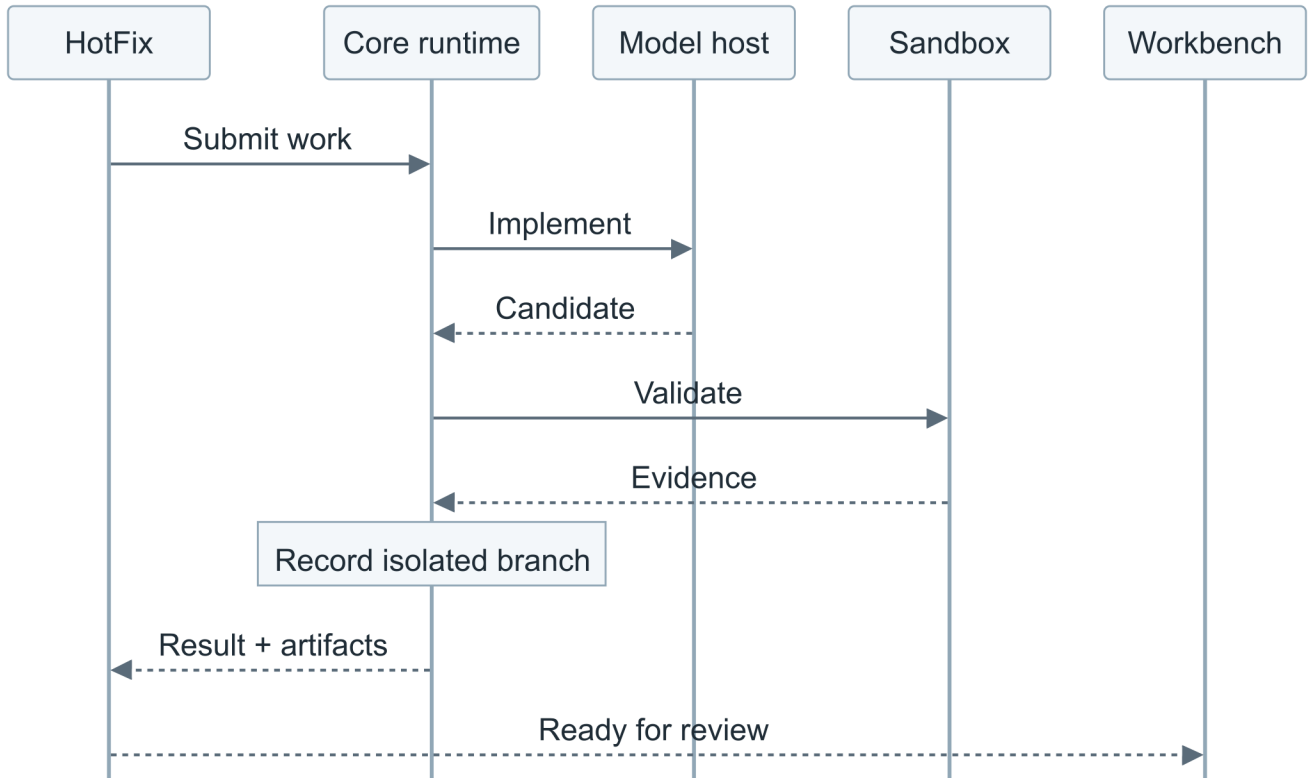

Figure 4. After Core API routes an error to HotFix, the runtime coordinates implementation and isolated validation. Project fixes stay on delivery branches; platform self-repair prepares draft pull requests.

8. Search and Indexing

Code intelligence uses parser-backed symbol extraction, relationship graphing, and vector indexing to support semantic search and guided implementation actions.

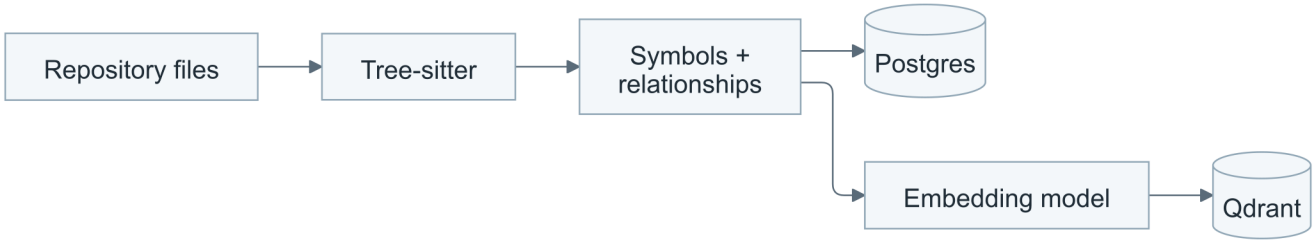

Figure 5a. Indexing stores code structure in Postgres and model-generated embeddings in Qdrant.

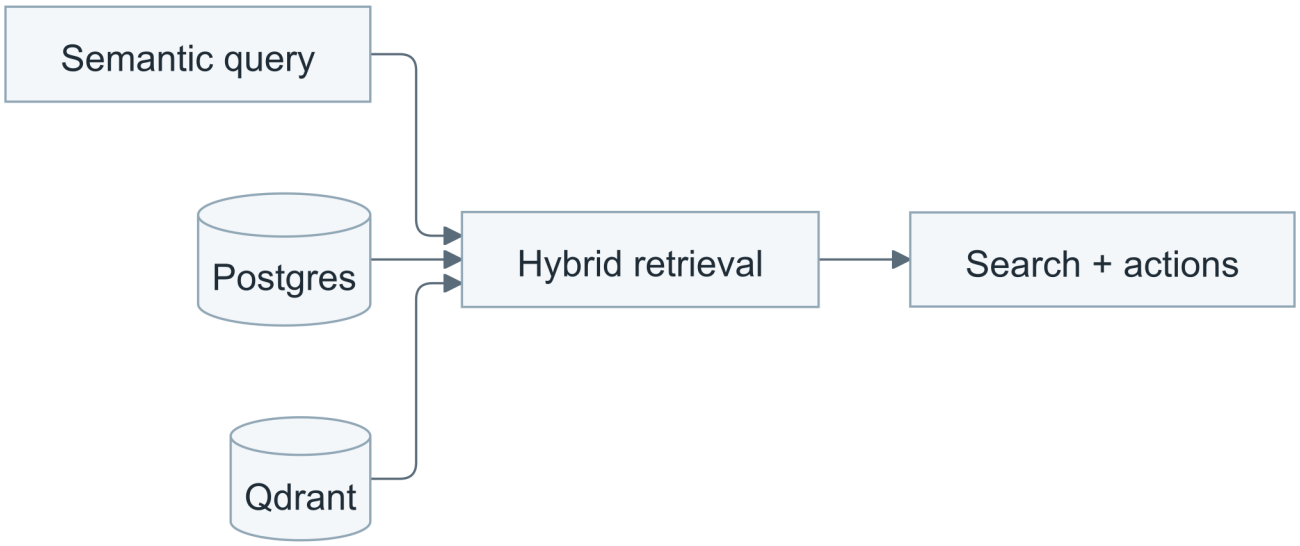

Figure 5b. Hybrid retrieval combines stored code context and vector search for a query.

9. Validation and Policy Enforcement

Validation is integrated as a runtime outcome, not a disconnected post-process, and it is not one monolithic pipeline. Plan quality, execution quality, completion evidence, and artifact quality evaluate different parts of the work. The mandatory completion floor requires admissible platform-produced evidence. A model's success claim cannot bypass it. Additional enforced gates can stop a step; failures return a diagnostics packet that feeds a guided repair-and-retry loop. The provisioned test-run engine executes across 10 language stacks on allowlisted, digest-pinned images. Dependencies install in a separate network-enabled phase against an allowlisted package index; the container is then disconnected from every network and the isolation is verified before any test command runs. Additional quality enforcement is mode-aware: shadow or matrix validation modes and per-gate flags determine which findings are advisory and which block progress. These settings do not make model-reported success admissible completion evidence. Platform self-repair promotion validates at

a shadow, baseline, or full tier chosen from the changed files: shadow only for docs and Markdown, full for high-risk paths, baseline otherwise.

Separate from the per-attempt gates, a five-part aggregate quality gate governs publication: it evaluates tests, coverage, security findings, code-health analysis, and live-preview quality together as one canonical decision and posts as its own GitHub commit status. Default thresholds are 80 percent minimum line coverage, a minimum health score of 70, and zero tolerance for new critical or high severity findings.

The artifact-quality gate inspects every modified file and rejects placeholder implementations, scratch/debug files, no-op change sets, and non-code-only changes for code tasks, with per-gate enforcement flags controlling whether findings block completion. Failed checks trigger automatic retry with targeted feedback. A durable same-reason breaker permanently stops a step or plan after a default limit of three consecutive identical-reason failures. That count persists across worker restarts and lineage replacement, so a crash cannot reset the streak. This keeps a stuck repair from looping indefinitely on the same failure.

10. Sentry Integration Model

CrewWork supports platform-level and project-level Sentry integration with explicit connect state, webhook verification, threshold controls, and health visibility in domain surfaces.

- A local standalone Sentry ships as an opt-in compose overlay for local stacks.
- Sentry.io project integration remains supported for external project workflows.
- Health state is surfaced as operational status, not hidden inside logs.
- A successful HotFix remediation automatically marks the originating Sentry issue resolved, closing the loop end-to-end rather than only updating CrewWork’s internal triage record.
- A HotFix that is verified on a branch but not yet published carries a STAGED status, separate from resolved, so operators can see work that is finished but intentionally held back from release.

11. Realtime State and Projections

Frontend state is projection-driven. Event envelopes emitted from runtime and domain services feed websocket and API projections so UI status remains coherent with backend truth.

- Attempt and task status are emitted as explicit events with scope metadata.
- Runs timelines and token usage read from canonical projections.
- Transport endpoints avoid reconstructing lifecycle from loose text payloads.

12. Operations and Observability

CrewWork exposes service health, runtime diagnostics, event state, and remediation health through platform diagnostics surfaces. A full observability stack ships in the base single-host stack for correlated metrics, logs, and traces: Prometheus, Grafana, Alertmanager, Loki, Promtail, and Tempo, with node, Redis, and Postgres exporters.

- OpenTelemetry traces across API, runtime, and service integrations.
- Every model invocation is wrapped in an OpenTelemetry span correlated to durable token-usage records; versioned per-provider, per-model pricing policies surface USD cost per call and in aggregate, without ever persisting prompt or output content.
- Sentry-backed issue ingestion and remediation loop visibility.

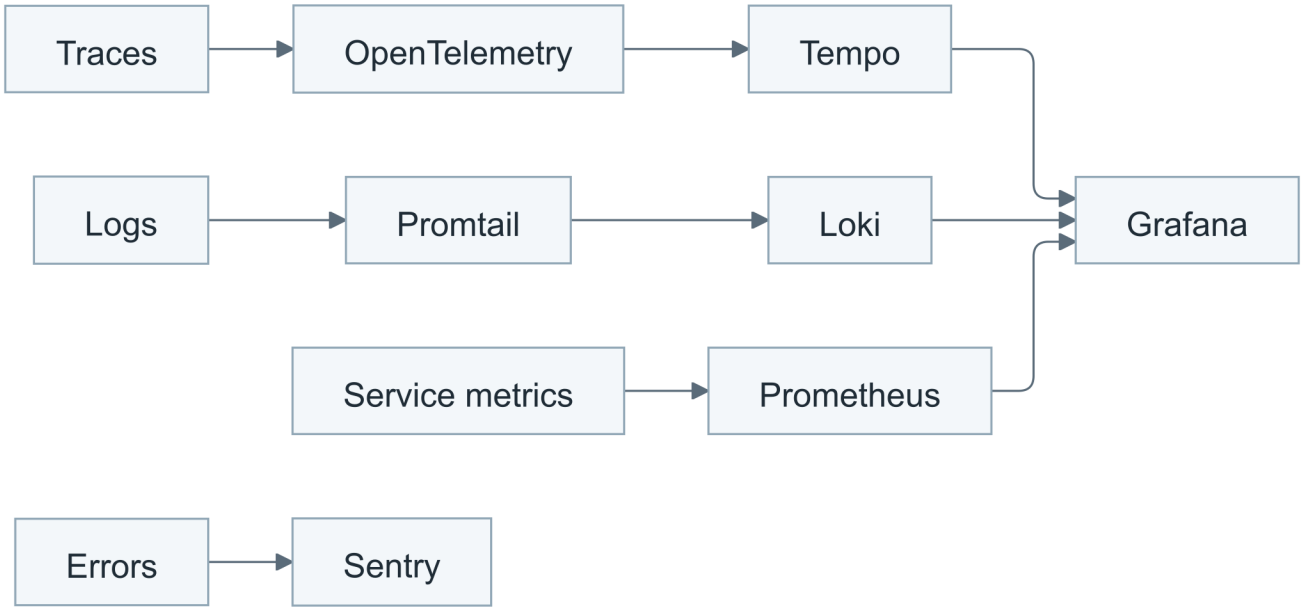

Figure 6. Telemetry paths: traces to Tempo, logs to Loki, metrics scraped by Prometheus, all correlated in Grafana.

13. Security and Governance

CrewWork's governance model combines role-based access control, workspace and path containment, audit trails, prompt trust boundaries, and policy-driven remediation constraints. It is underpinned by gVisor (runsc) kernel-boundary sandboxing for every untrusted or candidate code execution path, and by a scored, manifest-recorded self-modification assurance model for platform self-repair. See the Security page for the full protections, approval gates, and compliance-evidence detail, and Runtime Isolation for the sandboxing and resource-cap model.

14. Deployment Model

CrewWork is documented and positioned as a single-host Docker Compose stack for local or LAN deployments today. Application hosting is separate from model serving: generation endpoints can run on other LAN hosts, and semantic search uses a separate embedding model. Public model endpoints require explicit opt-in. The optional LAN release path builds project artifacts once and deploys them to other hosts on their own network through an outbound-only mutual-TLS release agent, with no inbound connectivity to the target. See the Compare page for the full deployment model, requirements, and honest tradeoffs.

15. Scaling Strategy

Scaling starts operationally rather than architecturally: increase worker concurrency, route model traffic across configured model hosts, and partition high-throughput services while keeping the same runtime and domain contracts.

The model routing layer resolves one typed routing snapshot across execution, extraction, embeddings, and review pools, with task classification, host capacity scoring, and bounded saturation steering across hosts; when every capable route is saturated the request receives a finite, retryable refusal rather than a silent fallback. Provider-compatible transport keeps the model boundary configurable without tying execution semantics to one host implementation. The structured extraction pipeline uses a two-stage think-then-extract pattern that generates natural-language analysis with mandatory sections before typed data extraction.

Because execution semantics are centralized in the Core runtime, horizontal scaling does not require duplicating workflow logic in each product surface.

16. Evidence and Limitations

Completion evidence records what the platform actually checked. Passing those checks does not establish that every defect has been found, that the plan fully captured the user’s intent, or that a change is suitable for production. A completed run does not imply that every available test or scanner ran.

The mandatory completion floor accepts platform-produced evidence, not the model’s own success report. Additional quality checks can be advisory or enforced according to their configuration; a shadow-mode finding is not equivalent to a passed enforced gate. Review the recorded contract and results for each run.

No representative task-success rate or comparative model benchmark is claimed here. The package/import measurements in Section 3 are tied to their stated commit and date. They describe code structure, not product effectiveness. Scaling mechanisms describe the architecture’s configuration options, not a measured capacity guarantee.

The development model configuration uses two generative hosts running the same Qwen3.8 27B artifact, with embeddings alongside one host. That is a reference setup, not a minimum hardware specification. Capacity depends on model quantization, context length, concurrency, repository size, and configured checks. Human review remains the final decision about what merges.

Further reading

The architecture overview is the concise component map; Truth & Trust Engineering covers how completion is verified.

Back to architecture overview

Truth & Trust Engineering