COMPLETE REFERENCE · ZERO INCREMENTAL STEPS · PLATFORM-AGNOSTIC & GOOGLE CLOUD

# The Assembled Hybrid Agentic Architecture

Both completed diagrams—platform-agnostic and mapped to Google Cloud—with a component-by-component walkthrough of how an event moves through the system from start to finish.

By Eva Khmelinskaya · Companion reference to A Mental Model for Designing Agentic Systems

This page presents the finished architecture directly as one assembled system, skipping the six incremental design stages of Episode 02. It governs asynchronous, event-driven workflows woken up by external signals that must interpret messy, potentially untrusted inputs, investigate context across external stores and tools, verify a proposed fix in a clean environment, and apply an external change safely under traffic bursts, duplicate deliveries, and worker crashes.

> **Reference Scenario:** A self-healing code repository system handling `10,000` deployments a day across `20` to `30` repositories ( `1%` failure rate = `100` failures/day). When a build, canary rollout, or error-rate alarm fires, the system contains live impact immediately, diagnoses the root cause, tests a candidate patch in an isolated sandbox, and opens a verified draft pull request for human review.

*Figure 1* shows the complete platform-agnostic architecture. Language models sit only at the two purple nodes where unstructured interpretation (**Quarantined LLM**) and open-ended search (**Privileged LLM**) are needed; deterministic plain code owns fast containment, edge authentication, queueing, locks and leases, verification, external write credentials, and post-merge memory.

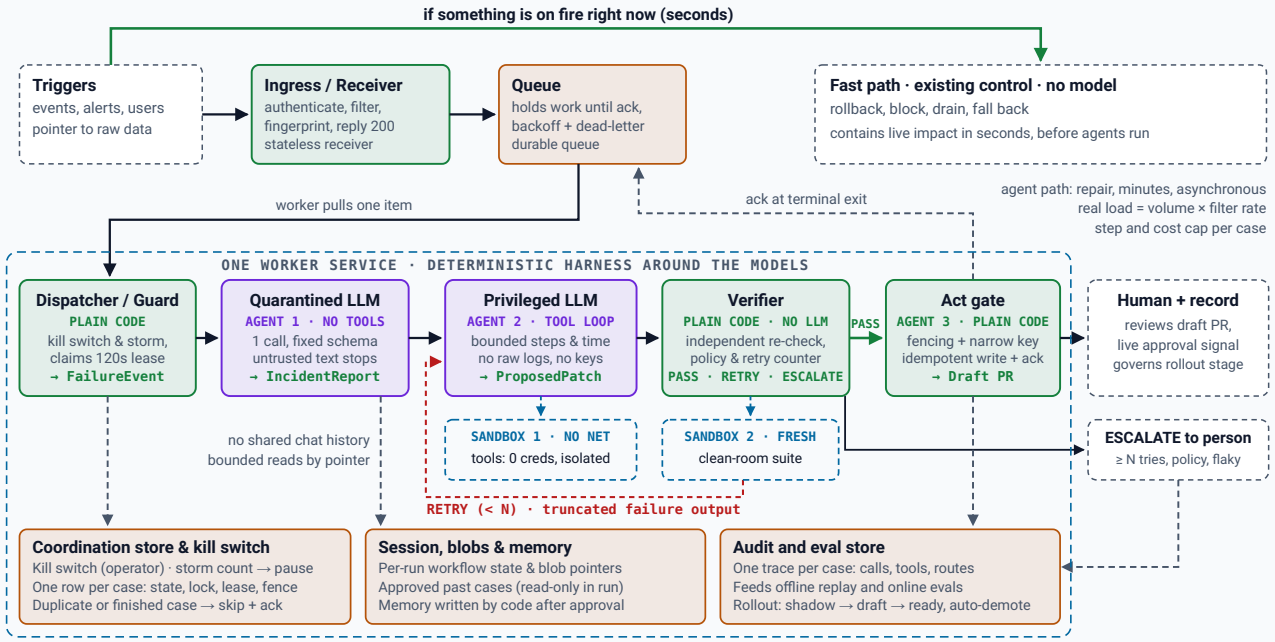



Figure 1 — Complete platform-agnostic architecture: urgent fast path, stateless intake, coordination store, Dual-LLM harness, isolated sandboxes, human gate, and audit/eval store.

Plain code LLM step Queue / store Isolated sandbox

EXECUTION WALKTHROUGH · COMPONENT BY COMPONENT

# How an Event Flows Through Every Component

Tracing a single failure on commit `a3f91c2` from initial signal to verified draft pull request, human review, and audit.

## PHASE 1 · UPSTREAM SIGNAL & SUB-SECOND CONTAINMENT (BEFORE ANY WORKER RUNS)

### 01 Triggers (Normalized Pointer Event)

EXTERNAL EVENT SOURCE

**Problem solved:** Failure signals arrive from separate systems in incompatible schemas, often attached to tens of megabytes of build or runtime logs that choke event buses and model context windows.

CI runners, canary controllers, and monitoring alarms keep raw logs in object or log storage and emit only a small normalized pointer payload carrying the repository name, exact `commit_sha`, error metadata, and `log_pointer`.

### 02 Fast Path · Existing Control (Zero Model Calls)

PLAIN DETERMINISTIC AUTOMATION

**Problem solved:** Live users hit by a bad canary rollout require relief in seconds, whereas reading source files, synthesizing a patch, and running test suites takes minutes.

Whenever an alert indicates live production impact, existing deployment automation acts immediately along the top green path in *Figure 1*—rolling back the canary, draining traffic, or blocking bad requests in seconds—while the repair workflow runs asynchronously in the background on a code branch.

## PHASE 2 · STATELESS EDGE INTAKE, DURABLE QUEUEING & COORDINATION GUARD

### 03 Ingress / Receiver

PLAIN CODE · STANDALONE SERVICE #1

**Problem solved:** Webhook senders time out after a few seconds, public HTTP endpoints attract spoofed requests, and 99% of deployment events represent healthy builds.

A lightweight stateless HTTP service verifies the caller's HMAC or OIDC signature, filters incoming webhooks down to real failures, computes a deterministic `failure_fingerprint` hash from error type, location, and cleaned message, publishes a typed `FailureEvent` to the queue, and replies `200 OK` in under 100 milliseconds.

### 04 Durable Queue & Dead-Letter Path

DURABLE MESSAGE BROKER

**Problem solved:** Outage bursts overwhelm synchronous workers, and a worker crashing mid-run loses any event acknowledged on receipt.

Buffers `FailureEvent` messages durably and delivers one item at a time to available workers. A message remains unacknowledged while the run executes and is deleted only when the worker records a terminal state (DONE, PARKED, SKIP, or ESCALATED) and sends `ack`. Repeated crashes back off exponentially and route to a dead-letter queue after 5 failed deliveries.

### 05 Dispatcher / Guard + Coordination Store & Kill Switch

PLAIN CODE + TRANSACTIONAL DB

**Problem solved:** At-least-once queues redeliver duplicates, shared infrastructure outages break twenty healthy repos at once, bad prompt/model releases need an instant manual stop, and crashed workers leave stale locks behind.

As the entry node of the single worker process, the Guard executes three transactional checks against the **Coordination store** before any model runs: (1) reads the operator **kill switch** flag—if tripped for that repo or globally, marks the case PARKED and sends `ack` so work pauses cleanly and resumes when cleared; (2) counts distinct repositories failing over a **5-minute sliding window**—if 5 or more fail together, trips the **storm breaker**, pauses automated PRs for 15 minutes, and pages on-call once; (3) atomically claims the `(repo, failure_fingerprint)` row as `IN_PROGRESS` with a **120-second expiring lease** (renewed every 30 seconds by worker heartbeats) and increments a monotonic **fencing generation (epoch)**. Duplicate or already-finished rows skip immediately and send `ack`.

## PHASE 3 · INSIDE THE SINGLE-WORKER HARNESS: DUAL-LLM SPLIT, SANDBOXES & VERIFIER

Once the Guard claims a lease, the deterministic worker harness drives every internal step in-process, passing small code-validated artifacts with shared chat history disabled (`include_contents="none"`):

RECEIVER → WORKER

**FailureEvent**

```
{
  "repo":
"acme/payments",
  "commit_sha":
"a3f91c2",
  "failure_fingerprint":
"fp_8d4e",
  "log_pointer":
"dep_4471"
}
```

A pointer to the logs. The 50 MB payload stays in the logging system.

AGENT 1 → AGENT 2

**IncidentReport**

```
{
  "category":
"test_regression",
  "suspected_files":
["payments/tax.py"],
  "failing_test":
"tests/test_tax.py",
  "confidence": 0.86
}
```

**Raw logs stop here.** Agent 2 never sees log text or Agent 1's prompt.

AGENT 2 → VERIFIER → AGENT 3

**ProposedPatch**

```
{
  "target_files":
["payments/tax.py"],
  "unified_diff": "---
a/… +++ b/…",
  "explanation": "Default
missing vat_rate"
}
```

Agent 3 receives it only after the verifier's own test run passes.

## 06 Quarantined LLM (Agent 1 · Detect — 1 Call, Zero Tools)

LLM STEP · QUARANTINED

**Problem solved:** Build and runtime logs mix unstructured stack traces with untrusted strings (commit messages, test inputs, dependency output) capable of hijacking a tool-calling agent via indirect prompt injection.

Deterministic code fetches at most 100 screened failure lines from `log_pointer` and invokes the **Quarantined LLM** in a single turn with zero tools, `include_contents="none"`, and a closed output schema (`IncidentReport`). Plain code verifies that `category` belongs to the allowed enum (`CODE_BUG`, `INFRA`, `FLAKY`, `UNKNOWN`) and that `suspect_files` and `target_test` exist at `commit_sha`. Raw untrusted log text stops permanently at this boundary; `INFRA` and `FLAKY` cases route straight to escalation or skip.

### 07 Privileged LLM (Agent 2 · Suggest Fix) + Sandbox 1 + Read-Only Memory

`LLM TOOL LOOP + ISOLATED SANDBOX`

**Problem solved:** Synthesizing a code fix requires iterative search and test runs where each lookup depends on the previous result, yet running repository code inside the worker container would expose cloud identity and write tokens.

The harness passes only the validated `IncidentReport`, read-only matches of past human-merged fixes from **Long-term memory**, and (on retries) tail-sliced test failure output into the **Privileged LLM** under hard step, time, and token caps. Model API calls happen from the worker, while every file search, edit, and test command executes across an isolation boundary inside **Sandbox 1**—an ephemeral container checked out at `commit_sha` with zero network egress and zero credentials—emitting a structured `ProposedPatch` before `Sandbox 1` is destroyed.

### 08 Plain-Code Verifier + Sandbox 2 (Clean-Room Suite, Bounded Retry & Escalate)

`PLAIN CODE + FRESH SANDBOX`

**Problem solved:** Models grading their own diffs overlook broken edge cases or weakened assertions, and `Sandbox 1` may carry leftover state from exploratory edits.

Plain code first enforces path and diff policies on `ProposedPatch` (blocking edits to CI workflows, secrets, auth rules, or oversized diffs). Next, it boots a brand-new, networkless **Sandbox 2** at `commit_sha`, confirms that `target_test` fails *before* the patch (proving the bug was reproduced), applies `git_diff`, and runs the full test suite and linter. If verification passes (PASS), control advances to the Act gate. If verification fails and `fix_attempts < 3` in **Session state** (RETRY), the harness increments `fix_attempts`, attaches `stderr[-2000:]`, and loops back to the Privileged LLM. Once `fix_attempts` reaches 3 (ESCALATE), the harness marks the case ESCALATED, pages/opens a diagnostic ticket for a human engineer, and sends terminal `ack`.

## PHASE 4 · GATED EXTERNAL WRITE, HUMAN REVIEW, MEMORY & EVALUATION

## 09 Act Gate (Agent 3 · Open Draft PR)

PLAIN CODE · NARROW WRITE SCOPE

**Problem solved:** Opening a pull request follows a fixed API contract that needs zero model reasoning, and a worker stalled past its 120-second lease could wake up late and double-write after another worker took over.

Implemented strictly as plain deterministic code holding a repository credential restricted to `autofix/*` branches. Right before calling the repository API, it performs a second transactional check on the **Coordination store**—verifying that the manual **kill switch** remains clear and that its **fencing generation (epoch)** still owns the row—then pushes the branch idempotently, opens a **draft pull request**, marks the row DONE, and sends terminal `ack` to the queue.

## 10 Human + System of Record & Post-Merge Memory Write

HUMAN GATE + CURATED MEMORY

**Problem solved:** Production merges require human accountability, and allowing agents to write directly to long-term memory during a run lets unverified or prompt-injected notes poison future runs.

A human code owner reviews the draft pull request alongside its reproduction and test summary. Only after a human merges the pull request does deterministic post-merge automation write the verified incident-to-patch pair into **Long-term memory**, keeping memory read-only during active runs.

## 11 Audit & Eval Store (Golden Replay & Staged Rollout)

TELEMETRY + OFFLINE/LIVE EVALS

**Problem solved:** Prompt, model, or tool updates silently degrade fix quality unless every trajectory is recorded and gated against real historical incidents and human acceptance rates.

Every model input/output, tool invocation, token cost, verifier exit code, and downstream PR outcome (*merged without edits*, *merged with edits*, or *closed*) lands in the **Audit and eval store**. Prompt or model changes replay offline against a golden set of historical failures before promotion, and live rollout advances from shadow mode to draft PRs with automatic demotion if reviewer rejection spikes.

CLOUD REFERENCE · GOOGLE CLOUD & GEMINI ENTERPRISE AGENT PLATFORM

# The Full Architecture Mapped to Google Cloud

Every box and wire from *Figure 1* mapped 1-to-1 onto managed Google Cloud services—deployed as two services plus ephemeral gVisor sandboxes.

*Figure 2* maps the platform-agnostic architecture directly onto Google Cloud and the **Gemini Enterprise Agent Platform** (shortened to **Agent Platform**; model calls stay family-neutral). At ~100 failures a day, one **Spanner** database handles the operator kill switch, the 5-minute distinct-repo storm query, and the per-incident lock, 120-second lease, and fencing generation check.

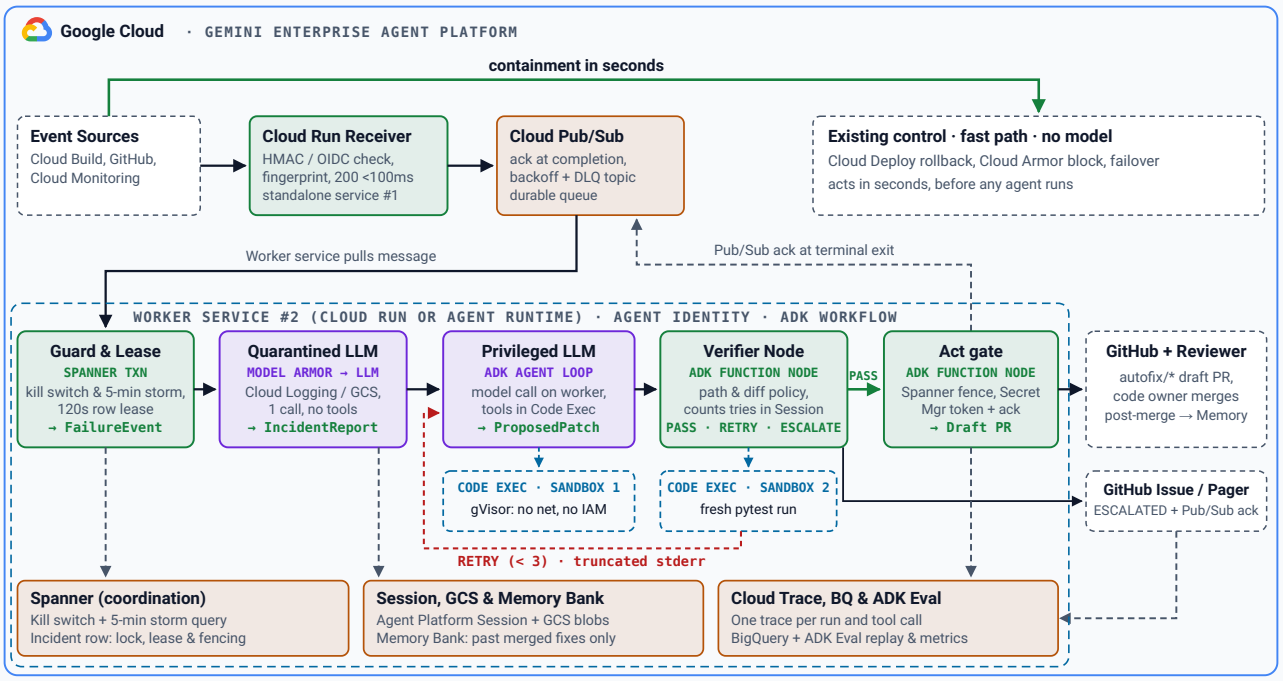



Figure 2 — Complete architecture mapped 1-to-1 onto Google Cloud and Gemini Enterprise Agent Platform.

Plain code  LLM step  Queue / store  Sandbox

## How the Flow Runs on Google Cloud

- **Edge & Fast Containment (Steps 01–04):** Failure alerts from Cloud Build, GitHub, or Cloud Monitoring trigger immediate **Cloud Deploy** canary rollback when live users are affected, while posting webhooks to the standalone **Cloud Run Receiver** (Service #1). The Receiver verifies HMAC/OIDC signatures, computes `failure_fingerprint`, enqueues `FailureEvent` into **Cloud Pub/Sub** (configured with exponential backoff and a dead-letter topic), and returns `200` in under 100 ms.
- **Single Worker Service & Spanner Guard (Step 05): Worker Service #2** (deployed on **Cloud Run** or **Agent Runtime** with keyless **Agent Identity**) runs the entire **ADK 2.x** `Workflow` graph in-process so sequential steps avoid network hops. Its entry `Guard & Lease` function node runs a **Spanner** transaction checking the kill switch, running the 5-minute storm query, and claiming a 120-second row lease with a fencing epoch.
- **Quarantined & Privileged LLM Nodes + gVisor Sandboxes (Steps 06–08):** The `Quarantined LLM` node reads ≤100 log lines from **Cloud Logging / Cloud Storage** through **Model Armor** inline prompt-injection screening with zero tools to produce `IncidentReport`. The `Privileged LLM` ADK agent loop reads past merged fixes from **Memory Bank** and dispatches repo search/edit/test tools into **Agent Platform Code Execution Sandbox 1** (ephemeral gVisor sandbox with no network and no IAM credentials). The plain-code `Verifier Node` tracks `fix_attempts` in **Agent Platform Session** and runs the clean-room test suite in a fresh **Code Execution Sandbox 2**.
- **Act Gate, Post-Merge Memory & ADK Eval (Steps 09–11):** On `PASS`, the plain-code `Act gate` function node re-verifies the Spanner kill switch and fencing epoch, fetches the narrow `autofix/*` GitHub token from **Secret Manager**, opens the draft PR, and sends the terminal **Pub/Sub** `ack`. Post-merge automation writes human-approved fixes into **Memory Bank**, while **Cloud Trace**, **BigQuery**, and **ADK Eval** capture every trajectory and run offline golden replays.

### COMPONENT-TO-GOOGLE CLOUD SERVICE MAPPING

| GENERIC COMPONENT | GOOGLE CLOUD SERVICE | IN THE SELF-HEALING REPO |
|---|---|---|
| **Ingress / Receiver** | Cloud Run service | Verifies HMAC/OIDC, hashes the fingerprint, publishes `FailureEvent` . |
| **Queue** | Pub/Sub with a dead-letter topic | Holds the event until the terminal `ack` ; dead-letters after 5 failed deliveries. |
| **Fast path** | Cloud Deploy rollback, as the existing control | Rolls back the failing canary in seconds. |
| **Worker and harness** | Cloud Run or Agent Runtime, running an ADK 2.x `Workflow` | All workflow nodes run in-process; one message drives one run. |
| **Coordination store and kill switch** | One Spanner database | A control row for the kill switch, the 5-minute storm query, and an `incidents` table with lock, 120s lease and fencing. An in-memory cache such as Memorystore becomes useful at thousands of events per second. |
| **Quarantined LLM input** | Cloud Logging and Model Armor | At most 100 log lines, screened before the single model call. |
| **Session and blobs** | Agent Platform Session and Cloud Storage | `fix_attempts` and typed outputs in the session; logs and diffs by pointer. |
| **Long-term memory** | Memory Bank | Human-merged fixes, written after the merge and read-only during a run. |
| **Sandboxes 1 and 2** | Agent Platform Code Execution | Tool calls and the verifier's test run, with no network and no credentials. |
| **Act gate credential** | Secret Manager | The repository token, limited to `autofix/*` branches. |
| **Audit and eval store** | Cloud Trace, BigQuery and ADK Eval | One trace per case and the golden replay set. |

← Back to Episode 02 (Progressive 6-Stage Build & 13 Questions) · All About Agents Home