AGENT SQUAD · CLASE PARA UN ESTUDIANTE DE INGENIERÍA

Agent Squad, paso a paso

De la oficina 3D al substrato: cómo un pedido en lenguaje natural se convierte en trabajo auditable, reconstruible y seguro

Los 5 documentos canónicos (el recorrido de onboarding real) que alimentan esta clase:

1. Inducción visual/narrada · agentsquad-induccion
2. `CONCEPTS.md` — el modelo mental
3. `ARCHITECTURE.md` — el mapa
4. `tutorials/first-day.md` — el lab (verlo en una corrida real)
5. `SELF-CHECK.md` — auto-evaluación

Complementada con `substrate-journey.html` — la documentación original del substrato (16-jun) — re-verificada claim por claim contra el código el 05-jul (315 claims: 48% vigente literal, lo desactualizado entra corregido y fechado).

Generado por Claude Code (Opus 4.8 + Fable 5) · archify (5 diagramas) + ai-solution-architect (pedagogía) · verificado contra el SELF-CHECK (answersAll7 ✓) · v2 · 2026-07-05

Contenido

Cada capítulo va en tres capas: ***Intuición*** *(la analogía primero) →* ***Mecanismo*** *(los nombres reales) →* ***A fondo*** *(el detalle y cómo se verifica). Al final: glosario y auto-evaluación.*

Antes de empezar

LA IMAGEN QUE HAY QUE RETENER

Antes de leer una sola línea de código, quedate con una imagen: **un estudio jurídico**. Cada pedido que entra abre un **expediente**. Adentro hay un plan de acción, la constancia de cada diligencia (quién la hizo, cuándo, qué salió), los documentos producidos y los hechos probados con su fuente. La regla de oro del estudio es una sola: **nada se hace por fuera del expediente**. Si pasó, quedó registrado; y si quedó registrado, se puede reconstruir mañana, dentro de un mes o en una auditoría.

Eso es el **substrato** de Agent Squad, y esa es toda la apuesta. Los agentes de IA son no-deterministas: improvisan, se equivocan, a veces alucinan. Un chatbot te devuelve texto y se olvida. Agent Squad, en cambio, convierte cada pedido en un **grafo trazable de punta a punta**. Al terminar esta clase vas a poder tomar un `trace_id` cualquiera y contar, paso a paso, qué se pidió, qué plan se compiló, qué agente ejecutó cada cosa, qué produjo, con qué información y dónde mirar si algo falló. Ese es el examen. Vamos por él.

00 De dónde viene: el origen, los forks y el primer smoke

Un control plane, no un Photoshop: el insight de Chamath, siete bifurcaciones decididas antes del código y un smoke test end-to-end de seis centavos.

INTUICIÓN

Todo estudio jurídico tiene un acta fundacional. La de Agent Squad es una transcripción: la charla de Chamath Palihapitiya en el Stanford AI Club («How to Win in the AI Era»). La palanca está entre los minutos 22:17 y 22:56: *«What is the value of a control plane? I would rather build MS-DOS and Windows, not Adobe Photoshop. For AI, we don't have that control plane… the ground source golden truth for all of these agents downstream will always be this hardware-independent, database-independent, language-independent, symbolic representation of what you want to do.»*

La lectura que fundó el proyecto: ese control plane no era un producto sino una **capa bajo los agentes** donde cada decisión, claim, intent y artifact queda trazado y disponible para futuras inferencias — eso es lo que hace que un equipo de agentes **mejore con el uso**. Las herramientas se desgastan; los substratos **compuestan**. La segunda cita (23:50–24:26) da la ambición de largo plazo: *«The N+1 company gets to leverage all the secrets of the N companies before it… a logarithmic expansion of abundance.»* Ese cross-empresa sigue diferido a propósito: `claims.workspace_id` es nullable, con el comment literal *'Fork 2, deferred'*.

Y el criterio con que se eligió cada pieza del stack — doce decisiones en una hora — cabe en una pregunta: «**¿qué cosa madura y barata cubre esto sin pintarnos en una esquina?**». Cada pieza por una razón operativa, ninguna por hype — *«si vas a robarte algo de este caso, robate el criterio»*. La base son marcas que cualquier estudiante reconoce (Postgres, nginx, Vercel, Cloudflare, Anthropic, Inngest); lo custom es una capa fina, tipada y testeada. La única apuesta original — la **memoria estructurada de decisiones humanas** — es justo la pieza que ningún framework vendía.

MECANISMO

De ese insight salió la **decisión fundacional**: el producto es el substrato, no la app. No construir un Photoshop: si construís el consumer sin substrato repetís el patrón de cada SaaS de IA — los usuarios producen pixels pero el sistema no aprende. Agent Squad consumer nació como la primera *ontology overlay* y la prueba pública de que el substrato sirve (cuando se diseñó, la app era un plan; hoy está en producción y es el único *ontology overlay* activo). «Substrato simbólico» quedó definido así: **memoria estructurada con primitives tipados donde cualquier afirmación es defendible, trazable y consultable por un sucesor**. Y todo salió de una sola sesión de trabajo: del insight a workflows productivos con LLM real.

Antes de tocar código se alinearon **siete forks** (bifurcaciones de diseño) en esa misma sesión. Tres para destacar:

- **Fork 03 — una sola ontology hasta saturar:** profundidad antes que ancho. Sigue vigente: el overlay único es `agent-squad-consumer`, y `ontology_overlays` permite relabels por vertical sobre los mismos primitives.
- **Fork 05 — plan compiler híbrido:** primero 6 meses de templates curados a mano; después, un compilador con vector lookup sobre los rankings de templates. Es la predicción que se cumplió transformada: hoy ese rol lo juega Nova con el desenlace MATCH.
- **Fork 06 — «Open spec. Closed engine.»:** el spec es publicable (JSON Schema + Zod, no un DSL propio); el runtime no. Los conteos originales quedaron viejos — hoy el catálogo tiene 26 Operations, 9 Evaluators y 7 Templates.
- **Fork 07 — Fase 0 en 4 semanas:** spec+infra → catálogo+template → hook web↔substrato → standup-digest end-to-end en producción. Todos los entregables existen y el cron sigue corriendo.

Y el fork del multi-tenant dejó su marca desde el día cero: `workspace_id` viaja en los datos desde la migración `0001` — el multi-tenancy que falta es capa de routing, no migración de datos («el identificador de tenant va en los datos antes que en el routing»).

El descarte de alternativas también fue explícito. El substrato combina **cuatro requisitos a la vez**: recuperación semántica (lo que da RAG), DAGs reusables (lo que prometía LangChain, estructurado), durabilidad + human-gate (lo que da un workflow engine) y el **grafo de claims tipados** — el cuarto es el moat que ninguna alternativa pública ofrece. Los aforismos de aquella sesión de diseño, para tener a mano cuando te objeten:

- «RAG te dice qué se parece; el substrato, qué fue afirmado, por quién y cuándo.»
- «LangChain es el pegamento entre prompts; el substrato es el motor durable + la memoria estructurada» — con estado en memoria, un crash es arrancar de cero, y no hay human-in-the-loop durable.
- «Los agent frameworks son improvisación; el substrato es partitura con margen de improvisación dentro de cada step.»
- «El workflow engine es el verbo; el substrato agrega el sustantivo (claims tipados). Inngest + Postgres + spec = substrato» — cada cambio de estado es una fila tipada, no una mutación en memoria.

Una pata más del mismo desarme, con fecha: al as-built de junio, 11 de las 17 operations de entonces eran handlers deterministas — el LLM solo entraba en las de composición de texto. Determinismo donde importa, modelo donde aporta.

La validación llegó rápida y barata: el **primer smoke test**, el 19-may-2026, con Sonnet 4.5 real. Los cuatro templates llegaron a `awaiting_human` por un costo total de **$0.064** (standup $0.0222, brief $0.0147, lead $0.0143, video $0.0124). Lo no trivial: Sonnet razonó sobre el propio backlog — `trace.query` le dio los traces reales y el narrative dijo textualmente *«5 digests en cola desde 16:32»* — validando la cadena retrieval→composer. Uno de los angles generados esa tarde: *«RAG te da respuestas parecidas. Un grafo de claims tipados te da respuestas defendibles. Cuando tu negocio depende de que la IA no invente, la arquitectura importa más que el prompt.»* El primer artifact aprobado (el brief «Substrato vs RAG») materializó 5 claims — `approvedBy=human:roberto`, comment *'Hook 2 gana'* — que siguen consultables hoy en la DB.

A FONDO

La categoría ya existía. El proyecto no la inventó: Karpathy plantea el LLM como kernel de un OS con memoria/tools/state como *substrate layer*; Anthropic lo implementa como MCP; Microsoft lo publicó como GraphRAG (un grafo de entidades y claims bajo el LLM); Letta/MemGPT (Berkeley) vende *«persistent agent memory substrate»*; Inngest — el motor de este proyecto — se posiciona como *«the durable execution layer for AI agents»*; y Temporal inventó el término *durable execution*. Lo hecho acá fue **materializar esa categoría emergente** con los primitives de Agent Squad. La comparación con los agentes personales tipo Hermes completa el mapa 2026: aquellos ponen la baranda en el contenedor (sandbox) y apuntan a autonomía personal; Agent Squad la pone en la acción (gates) y apunta a governance organizacional — SuperSkills y closed-learning-loop son la misma idea con distinto gatillo (acá promueve un humano; allá, auto).

Y la capa que se decidió NO agregar: cuando la junta de arquitectos evaluó sumar un context engine tipo Redis IRIS, la respuesta fue no. IRIS resuelve latencia de alto tráfico sincrónico espejando datos a RAM vía CDC; la carga de Agent Squad es la opuesta — pocos workflows profundos y asíncronos que tardan minutos *por diseño*. Una segunda copia vía CDC es un segundo sistema que mantener consistente («lo que falla un domingo a las 3am»); regla de Majors: «**no operes lo que no necesitás operar**». Test de Cherny para agregar una capa: ¿sobrevive al próximo modelo? Si algún día aparece un agente sincrónico de alto tráfico, el patrón cabe adentro de una operation. Herramienta correcta para la escala correcta.

control plane — La capa bajo los agentes que guarda la 'ground truth' simbólica de lo que querés hacer (Chamath, Stanford AI Club). En Agent Squad, ese rol lo cumple el substrato.

los siete forks — Las siete bifurcaciones de diseño alineadas en una sola sesión antes de escribir código: multi-tenant desde día cero, ontología única hasta saturar, open spec/closed engine y Fase 0 en 4 semanas, entre otras.

Open spec. Closed engine. — Fork 06: el spec del catálogo es publicable (JSON Schema + Zod, no un DSL propio); el runtime que lo ejecuta, no. Hoy: 26 Operations, 9 Evaluators, 7 Templates.

substrato simbólico — Memoria estructurada con primitives tipados donde cualquier afirmación es defendible, trazable y consultable por un sucesor.

smoke test fundacional — 19-may-2026: los 4 templates end-to-end con Sonnet 4.5 real llegaron a awaiting_human por $0.064 total; los 5 claims del primer artifact aprobado siguen consultables en la DB.

✓ **Checkpoint** — con este capítulo podés responder: *¿Por qué el producto es el substrato y no la app, y qué cuatro requisitos combina el substrato que RAG, LangChain o un workflow engine por separado no cubren?*

01 Las dos mitades y qué es el substrato

Agent Squad es un solo sistema partido en dos: la oficina que ve el usuario y el cerebro que ejecuta y archiva todo.

INTUICIÓN

Pensá en un estudio jurídico con **mostrador de atención** y **trastienda**. El cliente solo ve el mostrador: entra, hace su pedido, se va con un resultado. Pero el trabajo de verdad —y su archivo— pasa en la trastienda, donde se abre el expediente, se hace cada diligencia y se guarda todo.

En Agent Squad el mostrador es la **oficina 3D** (una oficina de voxels donde ves a tu squad trabajar). La trastienda es el **substrato**: el registro-grafo de todo lo que el squad piensa, hace y produce. El nombre viene de *capa de base* — el suelo sobre el que corren los agentes. Regla mental número uno: **esto no es un chatbot**, es un equipo con trazabilidad.

Corré el contrafactual: sin substrato, cada workflow arranca de cero, el usuario hace de memoria del sistema pegando contexto en cada prompt, y auditar es imposible. **Con substrato:** el workflow A deposita claims tipados y el grafo crece; el B lee los claims de A antes de empezar; cada decisión humana se vuelve un claim consultable; la trazabilidad es nativa; y hay **network effects** — cada workflow extra agrega contexto a los siguientes.

MECANISMO

Las dos mitades son dos programas que solo se hablan por HTTP; **no comparten ni base de datos ni memoria**:

- **La app** = `apps/web`, hecha en SvelteKit 5 / Threlte, desplegada en **Vercel** (managed, no corre en el servidor). Es la oficina 3D. Es intercambiable: es solo el frente.
- **El substrato** = `apps/api`, un runtime en **Bun** + **Hono** que corre en un servidor Hetzner como servicio systemd `agent-squad-api.service`. Es el **system-of-record**: la fuente de verdad.

La app llama al substrato por HTTPS con un **bearer token**. El corazón del substrato es el runtime; a su alrededor **usa** tres piezas de infraestructura: **Postgres** (donde vive el grafo), **Inngest** (motor de ejecución durable) y **Langfuse** (observabilidad de las llamadas al LLM).

> **Fail-soft al leer, ruidoso al escribir:** si el motor no responde en 2.5 segundos, la página carga igual con contenido de ejemplo etiquetado como tal — recuperar datos nunca bloquea usar el producto. Las escrituras son lo contrario: fallan ruidosamente. Es una regla transversal de diseño de la app.

A FONDO

La confusión clásica del practicante: creer que **el substrato es la base de datos**. No lo es. Postgres es *una pieza* que el substrato usa. El prefijo `substrate-` en la infra (`substrate-postgres`, `substrate-inngest`, `substrate-presidio`) marca lo que **pertenece** al substrato, no que cada pieza **sea** el substrato.

Cómo se verifica en un minuto: `curl -s 127.0.0.1:4000/health | jq` debe devolver `{"status":"ok"}` — el runtime está vivo. El runtime es **stateless**: el estado vive en la DB, no en el proceso (por eso una corrida sobrevive a un reinicio). Y una decisión de costo que revela la filosofía: el runtime genera texto con **Claude Code headless**, a **$0** vía la sesión Max, sin depender de una API paga.

substrato — El registro-grafo de todo lo que el squad piensa, hace y produce, con trazabilidad total. Vive en apps/api (Bun) y es el system-of-record.
system-of-record — La fuente única de verdad. El substrato lo es; la app web es solo el frente intercambiable.
agent-squad-api.service — El servicio systemd bajo el que corre el runtime del substrato en el servidor Hetzner.

✓ **Checkpoint** — con este capítulo podés responder: *Pregunta 1: ¿Qué es el substrato, en una frase? (y por qué NO es la base de datos)*

02 La cadena de datos: intent → plan → trace → step → artifact → claim

Todo pedido recorre siete conceptos encadenados; la distinción que más confunde es step (lo que hay que hacer) vs step_execution (la constancia de lo hecho).

> INTUICIÓN
>
> Mapeá cada eslabón al expediente jurídico y no lo olvidás más:
>
> - **Intent** = abrir un caso (el deseo declarado).
> - **Plan** = el plan de acción del caso.
> - **Step** = una diligencia *a realizar*.
> - **Trace** = el acta de *una* sesión de trabajo.
> - **Step Execution** = la *constancia* de una diligencia hecha.
> - **Artifact** = un documento del expediente.
> - **Claim** = un hecho probado, con su fuente.

MECANISMO

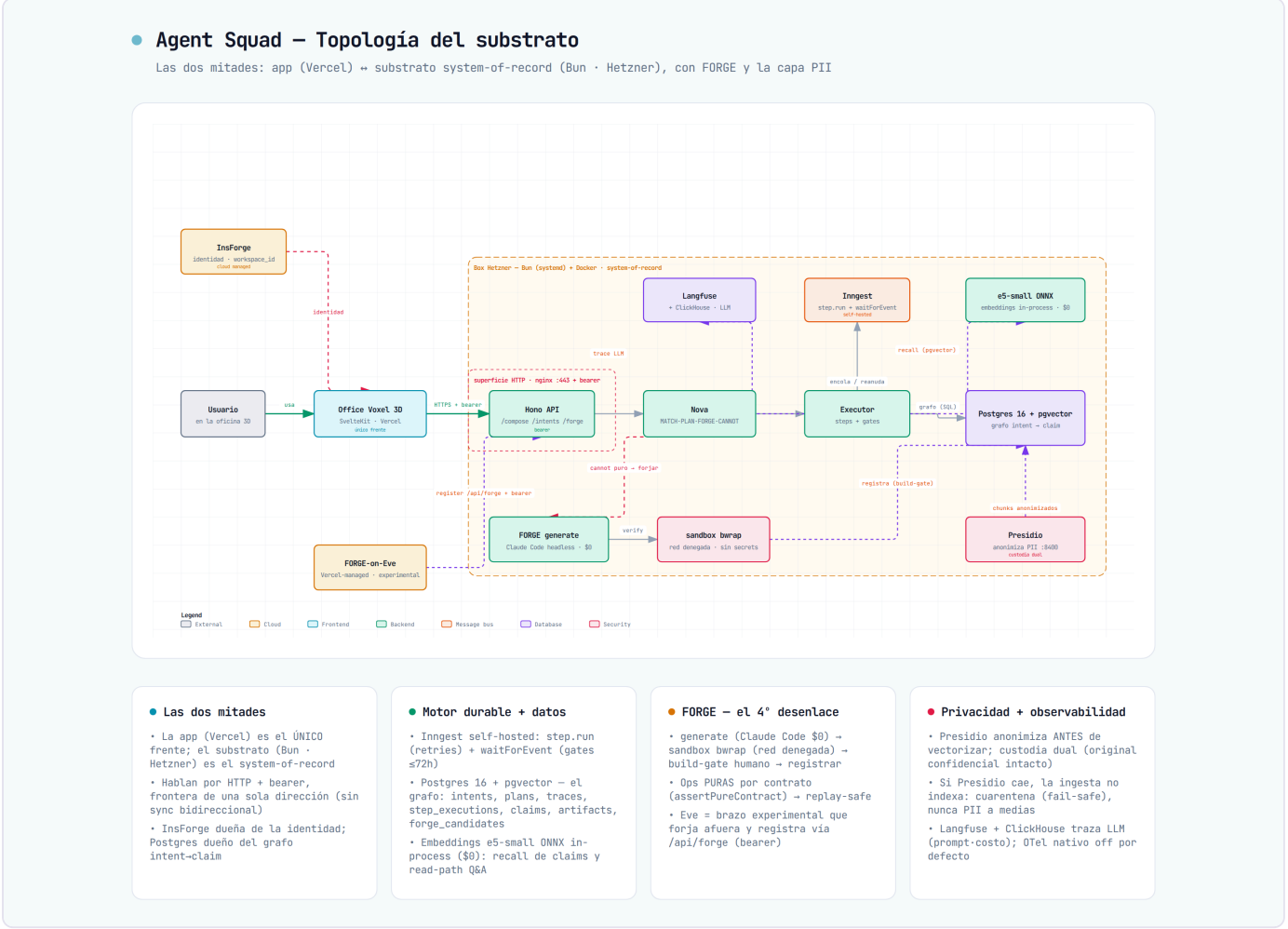
La cadena, con sus tablas reales de Postgres: **Intent** (tabla `intents`, el pedido declarado) → **Plan** (tabla `plans`, se **compila** desde una `PlanTemplate` como `standup-digest-v1`) → **Steps** (tabla `steps`, cada uno referencia **UNA** operación vía `operation_ref` + qué agente lo ejecuta + si necesita gate, ordenados por `ordinal`) → **Trace** (tabla `traces`, **UNA** corrida concreta del plan) → **Step Executions** (tabla `step_executions`, qué pasó al ejecutar cada step) → **Artifacts** (lo producido) y **Claims** (hechos aprendidos con su fuente).

La distinción clave: el **step** es la parte *estática* del plan (la diligencia a realizar). El **step_execution** es la *constancia* de esa diligencia **hecha dentro de una trace concreta**: status, timing, costo, output, error. Un mismo step puede tener muchas ejecuciones distintas — una por cada trace que corra ese plan.

Puente sobre el orden: el `step` vive en el `plan` (es la *definición*, estática); la `trace` y sus `step_executions` son la *corrida* (la ejecución). Por eso a veces se lista trace antes que step (el orden del expediente) y a veces step antes que trace (el orden de runtime): es la misma cadena vista desde definición vs. ejecución.

Claim vs log vs embedding — y por qué «simbólico»: tres formas de recordar lo mismo: un **log** guarda una oración suelta, difícil de razonar después; un **embedding** son 384 números — similares matemáticamente pero no auditables en su origen; un **claim** ya lo conocés de este capítulo: estructurado, leíble y trazable a quién lo puso. Decir «substrato simbólico» es declarar bando frente al enfoque connectionist: claims con estructura, no nubes de números.

¿Qué es un DAG? (y una honestidad sobre el paralelismo): Un DAG es un grafo dirigido sin ciclos — el mismo modelo de Git, Excel y Airflow. El motor le hace **topo-sort** (algoritmo de Kahn, con detección de ciclos) para arrancar cada step cuando sus inputs terminaron. Honestidad verificada: el executor actual recorre el orden topológico **secuencialmente** — el paralelismo de ramas es potencial del modelo, no comportamiento implementado.



Las dos mitades

- La app (Vercel) es el ÚNICO frente; el substrato (Bun · Hetzner) es el system-of-record
- Hablan por HTTP + bearer, frontera de una sola dirección (sin sync bidireccional)
- InsForge dueña de la identidad; Postgres dueño del grafo intent→claim

Motor durable + datos

- Inngest self-hosted: step.run (retries) + waitForEvent (gates ≤72h)
- Postgres 16 + pgvector — el grafo: intents, plans, traces, step_executions, claims, artifacts, forge_candidates
- Embeddings e5-small ONNX in-process ($0): recall de claims y read-path Q&A

FORGE — el 4° desenlace

- generate (Claude Code $0) → sandbox bwrap (red denegada) → build-gate humano → registrar
- Ops PURAS por contrato (assertPureContract) → replay-safe
- Eve = brazo experimental que forja afuera y registra vía /api/forge (bearer)

Privacidad + observabilidad

- Presidio anonimiza ANTES de vectorizar; custodia dual (original confidencial intacto)
- Si Presidio cae, la ingesta no indexa: cuarentena (fail-safe), nunca PII a medias
- Langfuse + ClickHouse traza LLM (prompt·costo); OTel nativo off por defecto

A FONDO

Detalles finos que verificás en el JSON de la vista de traza: un `artifact` tiene `content_addr` (un **sha256**) y su linaje en `produced_by.{trace_id, step_id}`. Un `claim` es una afirmación **sujeto-predicado-objeto** con `provenance = trace_id + step_id` (FK al step exacto): esa es la **trazabilidad cerrada** — de un resultado vas hasta su origen.

`step_executions` es la **capa 1 de observabilidad**: siempre existe, aun sin Langfuse, con una fila por cada step. Y una señal que vas a usar mucho: en el JSON, un step puede tener `exec: null` si **todavía no se ejecutó** — típicamente un `human_gate.approve` mientras la trace está `awaiting_human`. El `status` del plan es vestigial (siempre `queued`); el que importa es `trace.status`.

Costo por paso desde el día uno: cada step registra su costo notional en dólares aunque el CLI cueste $0 marginal. Por eso la economía unitaria del switch a API se conoce hoy — $0.013–0.022 por digest — y «prender la API es configuración, no re-arquitectura».

PlanTemplate — La forma reusable de un plan (un DAG de steps). El plan de una corrida se COMPILA desde una plantilla, ej. standup-digest-v1.
step_execution — La fila que registra qué pasó al ejecutar un step dentro de una trace: status, timing, costo, output, error. Una por step por corrida.
provenance / lineage — Los enlaces trace_id+step_id que atan cada claim y cada artifact al paso exacto que lo produjo.
content_addr — El hash sha256 que identifica el contenido de un artifact.

✓ **Checkpoint** — con este capítulo podés responder: *Pregunta 2: Explicá la cadena intent → plan → trace → step → artifact → claim y la diferencia step vs step_execution.*

03 Nova y los 4 desenlaces

Nova es la recepcionista que lee cada pedido y produce uno de cuatro resultados: MATCH, PLAN, FORGE o CANNOT — pero nunca ejecuta steps.

INTUICIÓN

Nova es la **recepcionista** del squad: recibe el pedido y decide qué se hace con él. O, si preferís la cocina: es el **mesero** que ante un pedido decide entre cuatro caminos — *ya lo tengo en carta* (MATCH), *lo cocino con una receta conocida* (PLAN), *invento un plato nuevo y lo pruebo con el chef antes de servir* (FORGE), o *te digo honestamente que eso no lo hacemos* (CANNOT).

Lo importante: Nova **compone/planifica pero NO ejecuta**. Reparte, no cocina.

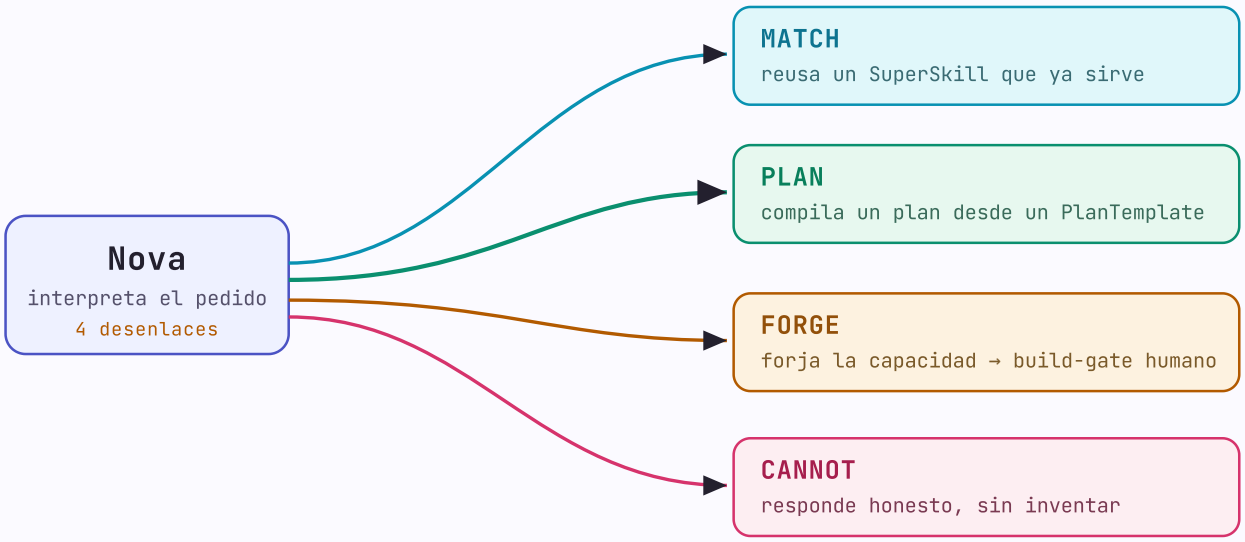
MECANISMO

El pedido en lenguaje natural entra por `POST /api/workspaces/:id/compose`. Nova interpreta el texto y produce un **desenlace** (el resultado de cómo resolvió el pedido):

- **MATCH**: ya existe un **SuperSkill** del workspace que sirve; lo reusa (matcheando el pedido contra los SuperSkills guardados). Es el camino más barato: reuso, no compilación.
- **PLAN**: arma un plan nuevo combinando operaciones del **catálogo cerrado** (un DAG de steps).
- **FORGE**: la capacidad no existe; la construye en tiempo real (capítulo 5).
- **CANNOT**: no se puede honestamente — requiere poderes que el sistema no se auto-otorga (ej. efectos externos sin credenciales).

Hay una **segunda puerta de entrada**: `POST /api/intents`, para lo programático (crons, integraciones, el proxy de un MATCH). Declara un intent estructurado (`subject_label` + `kind`) directo, **sin la interpretación de Nova**, y compila el plan desde su PlanTemplate.

A los planes que Nova compone no se les cree: pasan una **validación server-side de 4 capas** antes de ejecutarse: (1) cada operación existe en el catálogo — la misma verificación de `resolveOperation` que vas a ver en el capítulo 8, (2) sin ciclos — Kahn, (3) tope de 10 steps, (4) el gate humano final **lo agrega el sistema, no el modelo**. Todo lanzamiento re-valida contra el catálogo, y el costo estimado lo calcula el servidor desde el catálogo — el modelo no puede mentir el costo. El propósito de fondo: que «el modelo decidió» nunca sea la explicación de una ejecución. Al modelo no se le confía nada que se pueda verificar. Y cada pedido a Nova — incluso los CANNOT — queda registrado en `plan_drafts` con su desenlace: los «no pude» son telemetría de demanda que prioriza el roadmap.



A FONDO

Regla mental para las dos puertas: `compose` = *"decile a Nova qué querés"* (lenguaje natural, ella juzga); `intents` = *"ya sé exactamente qué workflow correr"* (declaración estructurada, determinista). El cron del `standup-digest` usa `/api/intents`, no compose.

Bajo el capó: la lógica de compilación vive en `handle-intent-declared.ts`. El schema de `/api/intents` (verificado en `routes/intents.ts`) exige `workspace_id`, `kind` ∈ {produce_artifact, answer_question, monitor_event, execute_action, analyze_data}, `subject_label` y un `acceptance_criteria_ref` **obligatorio** que debe matchear `^[a-z][a-z0-9_.]*@\d+...` (una ref versionada tipo `answer.accuracy@1`). El **MATCH** compara contra los SuperSkills **de ese workspace** (aislado por `workspace_id`) usando búsqueda vectorial con pgvector: no es un catálogo global. Y el **CANNOT** es la encarnación de la tesis *"nunca inventa"*: prefiere decir que no antes que alucinar.

desenlace — El resultado de cómo Nova resolvió un pedido: MATCH, PLAN, FORGE o CANNOT.
compose — POST /api/workspaces/:id/compose — entrada en lenguaje natural que Nova interpreta.
intents — POST /api/intents — entrada estructurada (kind + subject_label + acceptance_criteria_ref) que compila el plan sin interpretación de Nova.
CANNOT — El desenlace honesto: el sistema no puede porque requiere poderes/credenciales que no se auto-otorga. No alucina una solución.

✓ **Checkpoint** — con este capítulo podés responder: *Pregunta 3: ¿Quién es Nova y cuáles son sus 4 desenlaces? (+ bonus: compose vs intents, y contra qué matchea el MATCH)*

04 El ciclo de vida de una corrida y los gates

Una trace pasa de queued a succeeded, y en el medio un run-gate puede suspenderla de forma durable esperando la firma de un humano — que no es lo mismo que el build-gate de FORGE.

> INTUICIÓN
>
> Dos aprobaciones humanas que **no** hay que confundir:
>
> - **run-gate** (o `human_gate`) = *firmar el entregable antes de que salga*. Aprobás un **resultado** dentro de una corrida ("aprobá el digest antes de publicarlo").
> - **build-gate** = *aprobar una herramienta nueva antes de meterla a la caja*. Aprobás una **capacidad** nueva (es de FORGE, capítulo 5).
>
> Principio que recorre todo el sistema: "**nada está listo porque alguien lo diga**". Vos tenés la última palabra sobre el entregable.

MECANISMO

El ciclo de una trace: `queued` → `running` → (opcional) `awaiting_human` → `succeeded`. El **Executor** resuelve los steps y los encola en **Inngest**. Cada step corre `operación + agente` y luego un step de calidad, `evaluator.run` (actor `system:evaluator`), que valida el artifact contra los criterios de aceptación y deja un `verdict` — normalmente **después** de producir el artifact y **antes** del gate humano.

Si el plan tiene un **run-gate**, la corrida se **suspende de forma durable hasta 72h** con `step.waitForEvent()` de Inngest, **sin consumir recursos**, esperando la decisión. El humano decide con `POST /api/approvals` (`workspace_id`, `artifact_id`, `decision`) contra el artifact en `pending_review`; eso **reanuda** la corrida y la lleva a `succeeded`. Orden típico en un standup-digest: … `compose (s5)` → `evaluator.run (s6)` → `publish (s7)` → `human_gate (s8)`.

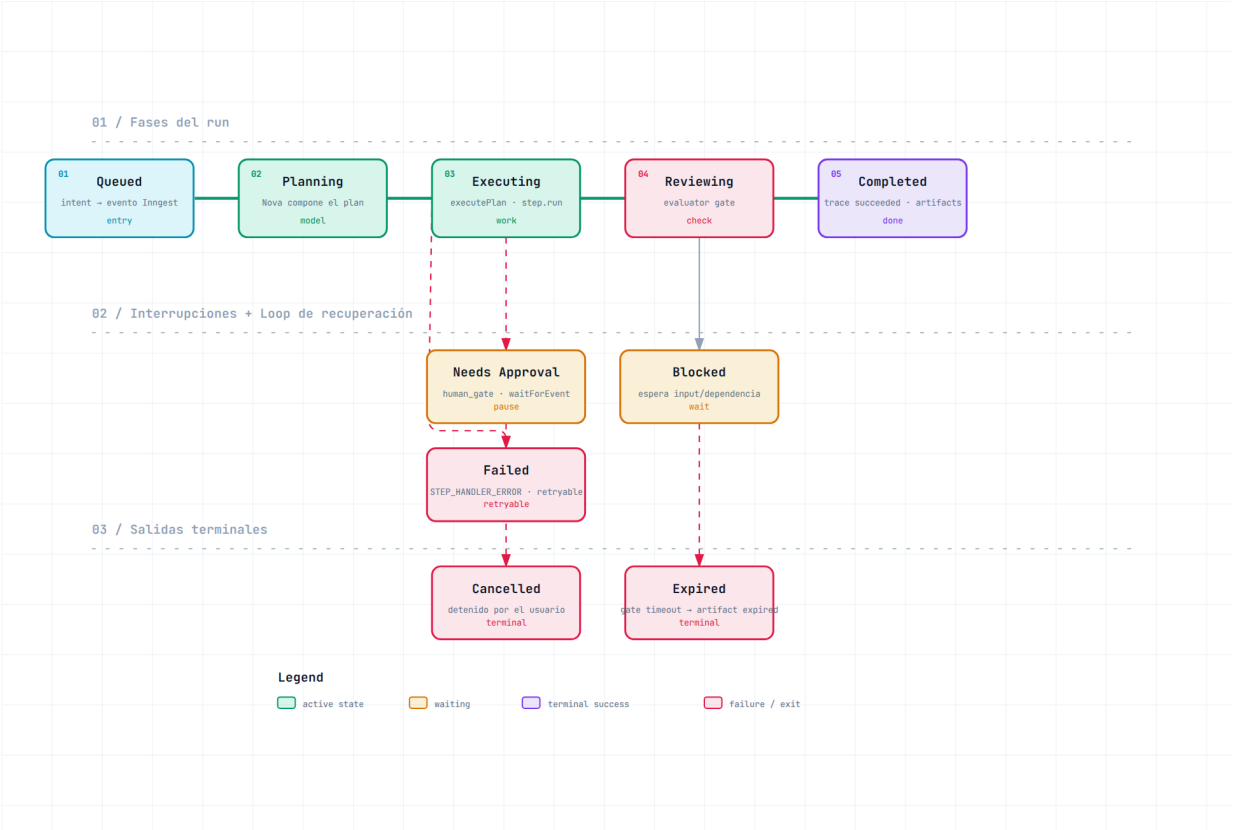
El run-gate no es una convención: es **estructural**. Todo lanzamiento — plan de fábrica, compuesto por Nova o SuperSkill custom — pasa por el mismo helper: `assertHumanGatePresent` exige al menos un gate humano terminal con fallback exactamente `'fail'` antes de lanzar; sin gate no hay launch («el servidor no puede confiar en el upstream; lo fuerza acá»). Las garantías no dependen de quién creó el plan. Y el formulario de relanzado se deriva mecánicamente de las variables `{{intent.constraints.*}}` que el plan declara.

No todo workflow lleva gate: pricing-watch corre solo: Ojo con sobre-generalizar: `pricing-watch` está en producción como template `monitor_event` + una función Inngest con cron diario (04:00 Bogotá) que declara intents como `system:scheduler` — el primer workflow que corre sin humano que lo dispare. Es read-only y **sin human gate** (igual que `document-query`). El gate estructural aplica a lo que se lanza desde la app y a lo que tiene efectos con firma; los read-only autónomos no lo llevan. El primer side-effect externo (email-triage) todavía no existe, y entrará con tres condiciones: idempotencia ante retries, observabilidad del envío y gate antes de actuar hacia afuera.

Cicatriz: async.data vs event.data: en `step.waitForEvent` la condición de match debe escribirse sobre `async.data` (el evento que va a llegar). Escribirla sobre `event.data` evalúa al momento de la suspensión, matchea `null` — y todos los gates quedan colgados indefinidamente. Bug crítico aprendido en el camino.

Agent Squad — Lifecycle de una corrida (Intent → Trace)

Máquina de estados del executor durable: planning, ejecución de steps, human gate, reintentos y salidas terminales



Camino feliz

- Cinco fases ordenadas: queued → planning → executing → reviewing → completed
- El lifecycle vive en un solo riel horizontal (el Trace)
- Completed es una fase, no una caja suelta: trace 'succeeded' + artifacts publicados

Gates: humano + input

- Needs Approval pausa sin terminar el run (human_gate, waitForEvent durable)
- Blocked espera input o una dependencia faltante
- Los wait states reanudan de vuelta a planning/executing

Terminal + recuperación

- Failed reintenta mientras quede presupuesto (retry_policy en runWithRetry)
- Cancelled y Expired son salidas del lifecycle
- Las salidas terminales NO vuelven a ejecución activa

A FONDO

Cómo detectar un run-gate activo de forma **confiable**: la trace está en `awaiting_human` y el step tiene `operation_ref = human_gate.approve` con `exec: null`. **Trampa del practicante:** el campo booleano `human_gate` del step suele venir `false` en data de seed aunque ESE step sea el gate — no te guíes por él, guiate por el `operation_ref` + `exec: null`.

La durabilidad es lo que hace todo esto posible: `step.waitForEvent()` sobrevive reinicios del servicio, así que una espera durable no se pierde si el box se reinicia. Si nadie firma en el plazo, la corrida muere con un error de familia "proceso": `HUMAN_GATE_TIMEOUT`. El plazo real es **72h** (`GATE_TIMEOUT_MS = 259200000` ms en nova-compose.ts y en todos los templates; el 24h de una versión anterior es historia) — y HUMAN_GATE_TIMEOUT no es una config: es el código de error que emite el executor al expirar (no es que el sistema falló — faltó la firma). Diferencia dura para memorizar: **run-gate aprueba resultados; build-gate aprueba capacidades.**

¿Dónde vive la espera durable? **Fuera del proceso**: Redis con AOF + snapshot cada 60s, y SQLite en volumen. Se verificó con un test de restart a mitad de un gate suspendido. La durabilidad cubre el dominio *frecuente* (deploys, crashes); contra lo *catastrófico* está el backup nightly de las 03:00 (pg_dump gzip, retención 30 días, push a R2, RPO ≤24h) — luego reforzado con pgBackRest (PITR) y un restore-drill semanal. Saber qué mecanismo cubre qué dominio de falla — y decirlo — vale más que prometer multi-región.

run-gate / human_gate — Un step que suspende la corrida (awaiting_human, plazo 72h — GATE_TIMEOUT_MS; al expirar emite el error HUMAN_GATE_TIMEOUT) esperando que un humano apruebe un RESULTADO. Se resuelve por POST /api/approvals.
step.waitForEvent() — La primitiva de Inngest que suspende una corrida de forma durable —sobrevive reinicios— hasta que llega el evento de aprobación.
evaluator.run — El step de control de calidad (actor system:evaluator) que valida el artifact contra el acceptance_criteria y deja un verdict, antes del gate humano.
pending_review — El estado en que queda un artifact producido hasta que el run-gate se aprueba.

✓ **Checkpoint** — con este capítulo podés responder: *Pregunta 6: ¿Cuál es la diferencia entre el run-gate (human_gate) y el build-gate?*

05 FORGE: el 4º desenlace (y por qué solo operaciones puras)

Cuando falta una capacidad y es pura, el sistema la construye solo en runtime, la verifica en un sandbox sin red ni secrets, y la registra recién tras la firma de un humano.

INTUICIÓN

FORGE es la **fragua**: el lugar donde se forjan herramientas nuevas cuando la caja de herramientas (el catálogo) no tiene la que hace falta. No es una sigla — es la metáfora de forjar.

Es la **tesis viva** del sistema: la misma petición que hoy es un `cannot` puede, tras pasar por la fragua y una firma humana, convertirse mañana en un `plan` normal. El sistema amplía su propio catálogo — pero de forma auditable y segura.

MECANISMO

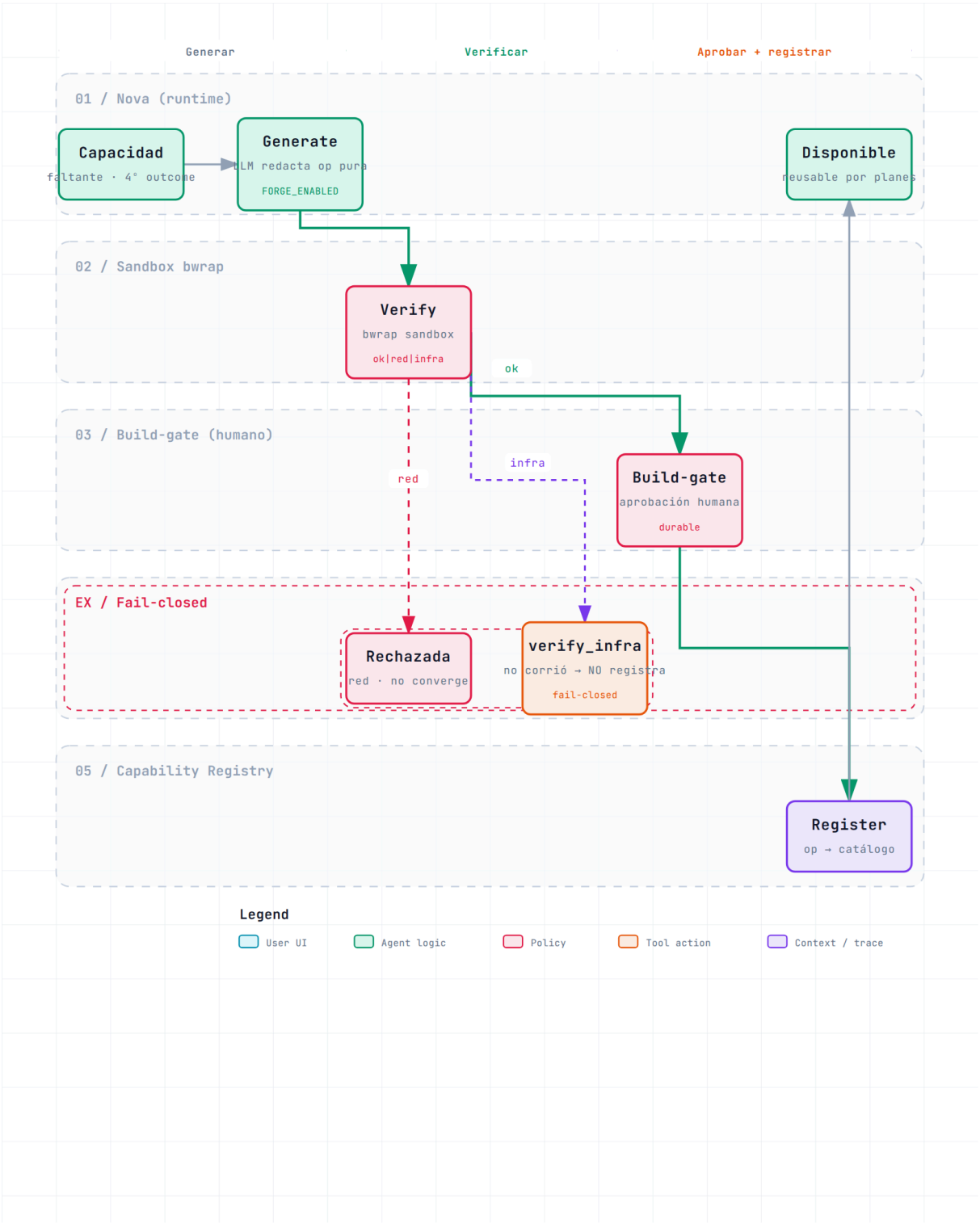
FORGE se dispara cuando un pedido es un **CANNOT genuino Y** la capacidad faltante es **PURA** (sin efectos externos). El loop:

1. **Generate**: produce spec + tests + código (con Claude Code headless, $0).
2. **Verify**: lo ejecuta en un **sandbox aislado** (`bwrap` , red denegada, sin secrets).
3. **Build-gate**: pasa por aprobación humana (`forge.approval` , gate durable de Inngest, 72h).
4. **Registry**: si se aprueba, se registra en el catálogo tras `assertPureContract` + `staticSafetyCheck` — y queda callable.

En la práctica: pedís algo fuera de catálogo por `/compose` ; Nova responde `status:"forging"` y construye en background (~20-90s). El candidato aparece en `GET /api/forge/pending` (el build-gate); lo aprobás con `POST /api/forge/<CANDIDATE_ID>/decision` . Después, el mismo pedido ya se resuelve con la op forjada (aparece el agente `Forge` en un step).

Agent Squad — Pipeline FORGE (construir una capacidad en runtime)

generate → verify (sandbox bwrap) → build-gate humano → register, con taxonomía fail-closed



Camino feliz

- Nova detecta una capacidad faltante (el 4° outcome)
- El código generado se verifica ANTES de tocar el sistema

Aislamiento por proceso

- verify corre en bwrap: --unshare-all, FS read-only, --clearenv
- Las ops son PURAS por contrato

Taxonomía fail-closed + observabilidad

- ok = verificó y pasó · red = verificó y NO pasa (no converge)
- infra = no se pudo verificar →

• Solo tras aprobación humana la op entra al catálogo

(sin red/DB) → replay-safe

• En prod, si el sandbox no está contenido, falla-duro (kind: infra)

NUNCA se registra

• Cada paso emite un span forge.verify a Langfuse (fuera del hot path)

A FONDO

El **porqué de "solo puras"**: el código generado corre aislado **sin red ni secrets** tanto en el verify como en el execute (`executeInSandbox` también corre en bwrap, no in-process). Una capacidad con efectos externos no podría verificarse de forma segura ni correr sin credenciales que el sandbox deniega. El `staticSafetyCheck` por regex es **defensa en profundidad**, no el muro: el muro real es el **aislamiento de kernel de bwrap**. Requiere `FORGE_ENABLED=true`.

Detalles que caza el practicante avanzado: el handler forjado trae marcas `// ponytail:` que auto-anotan cada atajo y su techo (ej. *'regex simple, no RFC 5322'*) para revisar el gate en segundos. Si se dispara el **mismo** cannot 2× antes de aprobar, el segundo corta con `status:'duplicate'` (dedup in-flight). Existe un **brazo experimental** sobre Eve de Vercel (`experiments/forge-eve/`): mismo contrato de seguridad y mismo build-gate, durabilidad managed, pero **nunca escribe la DB directo** — registra vía `POST /api/forge/register` con bearer + token verify-pass HMAC. El substrato sigue siendo siempre el system-of-record.

FORGE — El 4º desenlace de Nova: construir en runtime una capacidad que falta, cuando es pura. Metáfora de la fragua.

bwrap (bubblewrap) — El sandbox de aislamiento de kernel con red denegada y sin secrets donde FORGE verifica y ejecuta el código forjado.

assertPureContract / staticSafetyCheck — El contrato + chequeo estático que exige que la operación forjada sea pura antes de registrarla. Defensa en profundidad sobre el muro de bwrap.

build-gate — La aprobación humana (forge.approval) de una CAPACIDAD nueva antes de que entre al catálogo. Distinto del run-gate.

✓ **Checkpoint** — con este capítulo podés responder: *Pregunta 4: ¿Qué es FORGE, cuándo se dispara y por qué solo construye ops puras?*

06 El read-path de Q&A: document.query

El mismo motor de embeddings entra tres veces; para responder preguntas combina búsqueda vectorial y léxica, las fusiona con RRF y deja que un reranker externo elija el top-5.

INTUICIÓN

Imaginá un **bibliotecario** (el modelo `e5-small`) que hace tres trabajos distintos: (1) te dice *qué sección* de la biblioteca resuelve tu pregunta —eso es el ruteo de Nova (MATCH)—; (2) te trae *los libros que ya leíste* sobre el tema —el recall de contexto—; y (3) para una consulta puntual *cruza dos catálogos* (por tema y por palabra exacta) y deja que un **experto** (Cohere) reordene los mejores antes de dártelos.

MECANISMO

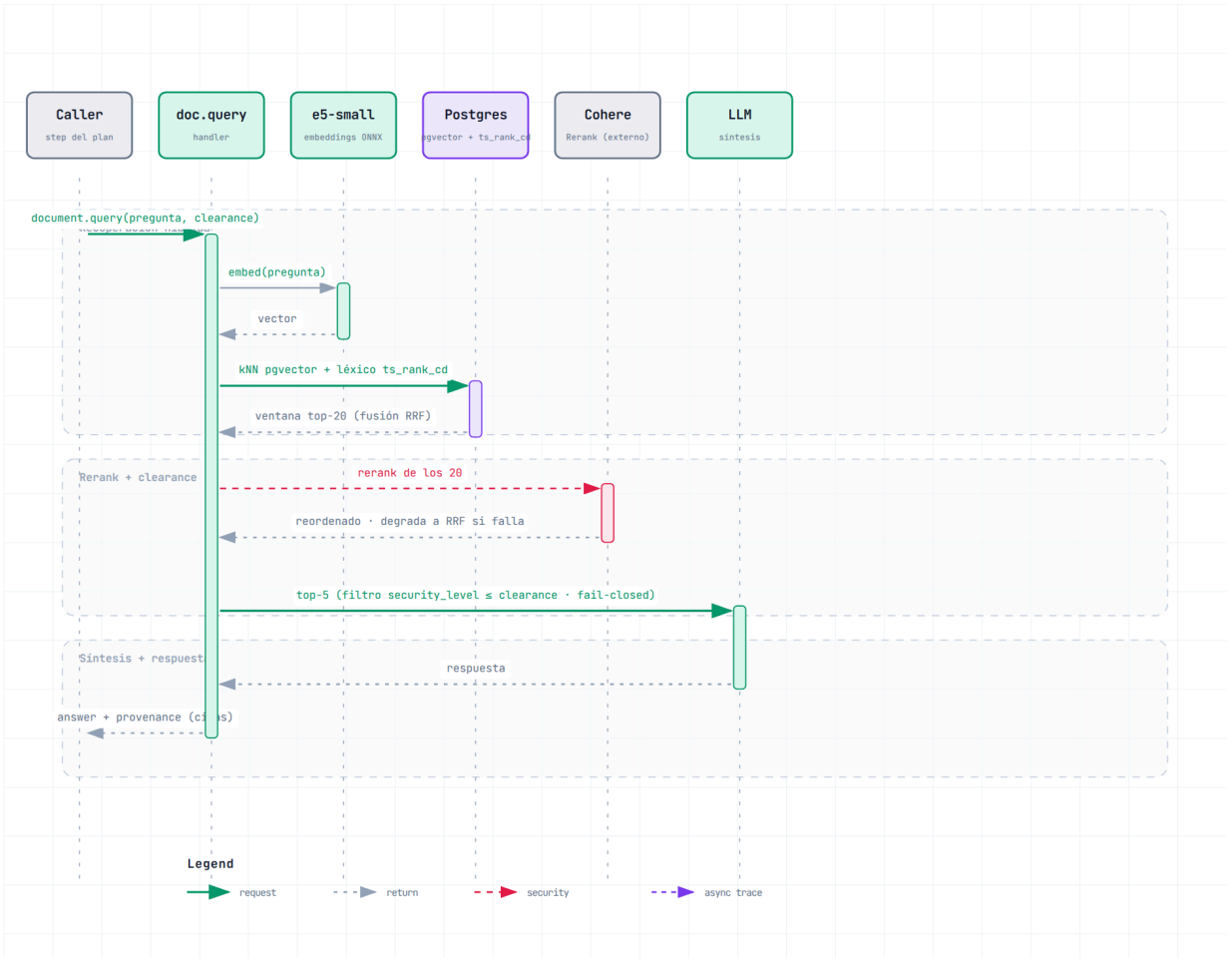
Una precisión que la ingeniería inversa dejó en evidencia (05-jul): el ruteo del **MATCH no es vectorial** — Nova recibe los SuperSkills del workspace (hasta 20, cargados fail-soft) **inline en el system prompt**, y una sola llamada LLM decide el desenlace con validación server-side de cada rama (`compose.ts`). El retrieval vectorial entra en el ciclo normal en el **recall de contexto**: durante la ejecución el executor recupera claims previos del grafo (`claim.recall_*`) para no rehacer trabajo — un RAG sobre el **conocimiento ya establecido**.

Aparte está el **read-path de Q&A**, la operación `document.query`, que hace **recuperación híbrida**: rama vectorial (`e5`) + rama léxica (`ts_rank_cd` de Postgres), fusionadas por **RRF** (Reciprocal Rank Fusion). Luego un **reranker activo** (**Cohere Rerank 4 Pro**) reordena una ventana de 20 candidatos y manda el **top-5** al LLM. Si el proveedor externo falla, **degrada limpio a RRF** — no rompe.

El efecto compuesto del recall — y su gap real: Antes de componer, el motor ejecuta `claim.recall_decisions` y le pasa al modelo las decisiones previas («hace 1 día, sobre un brief similar, Roberto aprobó Hook 2») — el LLM no adivina ni recuerda: **hereda la decisión como input**. Ese circuito tuvo un gap hasta el 05-jul-2026: `recall_decisions` filtra por los predicates `consolidatedDecision`, `workspace.decision` y `decision`, y el `approvalComment` quedaba excluido. Hoy está **cerrado**: al aprobar/rechazar con comment, el executor emite además un claim `decision` (con embedding por tema) que el recall sí hereda — y los 33 comments históricos se backfillearon al grafo.

Agent Squad — Read-path Q&A (document.query)

Recuperación híbrida (vector + léxico) → fusión RRF → rerank Cohere → filtro por clearance → LLM



Recuperación híbrida

- Dos búsquedas en paralelo: semántica (embeddings e5-small sobre pgvector) y léxica (ts_rank_cd)
- Se fusionan por RRF (Reciprocal Rank Fusion) en una ventana de 20 candidatos
- El embedding es local, in-process y $0 — el mismo motor que usa Nova para el MATCH

Seguridad del read-path

- El reranker Cohere reordena los 20 y manda el top-5; si el proveedor externo falla, degrada a RRF
- Filtro por security_level según el clearance del request — fail-closed a 'público'
- Solo texto anonimizado sale del box hacia Cohere y el LLM (ver capa PII)

Respuesta auditable

- El LLM recibe solo el top-5 ya filtrado y reordenado
- La respuesta lleva provenance: de qué chunks salió cada afirmación
- Es un RAG sobre documentos — distinto del RAG sobre el catálogo de capacidades (MATCH)

A FONDO

El motor de embeddings es **multilingual-e5-small** en **ONNX**, corriendo **in-process** dentro del runtime: **$0**, sin red, sin dependencia externa paga (decisión registrada en `ADR-0001`). La primera vez se carga (~4s) y queda cacheado. Esto no es un detalle menor: eliminó una clase entera de fallos por cuota/crédito que antes reventaban corridas (los viejos `STEP_HANDLER_ERROR` de sistema).

Lo que hace este read-path **seguro** además de preciso: filtra por `security_level` según el `clearance` del request en **las tres ramas** de la búsqueda (vectorial, léxica y el guardrail anti-cherry-picking), con **fail-closed a** `publico`. Eso conecta directo con la capa de privacidad del capítulo siguiente.

document.query — La operación de Q&A que hace recuperación híbrida (vector + léxico), fusiona por RRF, rerankea con Cohere y filtra por clearance.
RRF (Reciprocal Rank Fusion) — El método que fusiona el ranking de la rama vectorial y la léxica en una sola lista; también es el fallback si el reranker externo cae.
ts_rank_cd — La función de ranking léxico full-text de Postgres — la rama de 'palabra exacta' del retrieval híbrido.
multilingual-e5-small — El modelo de embeddings local (ONNX, in-process, $0) que alimenta el ruteo, el recall de claims y la rama vectorial de document.query. ADR-0001.
Cohere Rerank 4 Pro — El reranker externo activo que reordena la ventana de 20 candidatos y deja el top-5 para el LLM.

✓ **Checkpoint** — con este capítulo podés responder: *Amplía la Pregunta 7 (el read-path como parte de lo observable) y sostiene la tesis de confiabilidad: recuperar conocimiento sin filtrar ni degradar.*

07 La capa de privacidad: PII fuera de los vectores + acceso por nivel

Ningún dato personal crudo entra a los vectores ni sale del box: se anonimiza antes de indexar (fail-safe) y se filtra por clearance al recuperar (fail-closed).

INTUICIÓN

Imaginá una oficina que, antes de archivar cualquier documento en la **sala de consulta pública**, saca una fotocopia **tachando los nombres** y guarda el original bajo llave. Además, cada estante tiene un **nivel de acceso** y solo te dejan ver hasta el de tu credencial. Y si la tachadora se rompe, **no archivan nada** — mejor no procesar que filtrar a medias.

Eso es la privacidad en capas de Agent Squad: **custodia dual** (original bajo llave, copia tachada en circulación) + **control de acceso por nivel**.

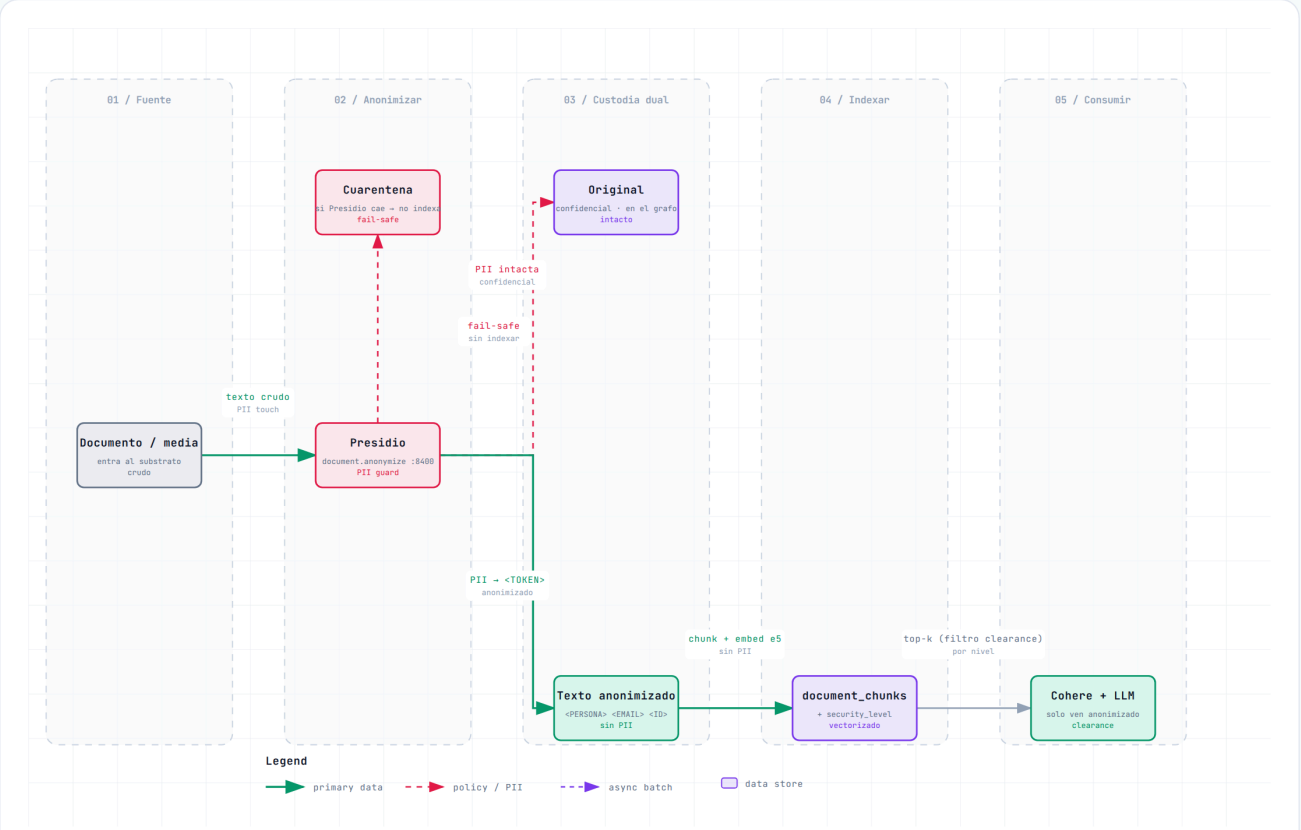
MECANISMO

Antes de que cualquier documento o transcripción entre a la tabla de vectores, pasa por **anonimización PII** con **Microsoft Presidio** (microservicio Docker local `substrate-presidio` en `127.0.0.1:8400`). La regla es **custodia dual**: el artefacto original con PII se marca `confidencial` y queda **intacto** en el grafo; a `document_chunks`, al reranker (Cohere) y al LLM solo entra **texto anonimizado** con placeholders (`<PERSONA>`, `<EMAIL>`, `<ID_TRIBUTARIO>`…). Cubre nombres, emails, IDs tributarios LATAM y empresas.

Fail-safe: si Presidio cae, la ingesta **NO indexa nada** y encola el original en **cuarentena** — nunca filtra PII a medias. En la **recuperación**, cada chunk lleva un `security_level` jerárquico (`publico` < `interno` < `confidencial`; se eleva a confidencial si el chunk tuvo PII), y `document.query` filtra por el `clearance_level` del request, **fail-closed a** `publico`: sin clearance explícito, solo ves chunks públicos.

Agent Squad — Capa de privacidad PII (custodia dual)

Anonimización con Presidio antes de vectorizar · original confidencial intacto · fail-safe a cuarentena



Custodia dual

- El original con PII se marca 'confidencial' y queda INTACTO en el grafo
- A document_chunks, al reranker y al LLM SOLO entra texto anonimizado
- Lo que sale del box ya no tiene PII cruda — por eso la tesis de cumplimiento es honesta

Fail-safe

- Si Presidio (:8400) cae, la ingesta NO indexa nada
- El original se encola en cuarentena — nunca filtra PII a medias
- Presidio corre como microservicio Docker local (127.0.0.1:8400)

Acceso por nivel

- Cada chunk lleva un security_level (público < interno < confidencial)
- document.query filtra por el clearance del request — fail-closed a 'público'
- Es control de acceso a nivel de aplicación, no criptográfico

A FONDO

Presidio es **stateless** (los modelos spaCy vienen en la imagen) y corre con `mem_limit 1g`. Punto conceptual importante: es **control de acceso a nivel de aplicación** (confía en que el caller declare su clearance), **no criptográfico** — el valor del `fail-closed` es que un error, por defecto, **niega** el acceso en vez de exponerlo.

Esto es lo que hace **honesta** la tesis de cumplimiento aun usando un reranker externo: lo que sale del box hacia Cohere ya no tiene PII cruda. Y el filtro por nivel aplica en las tres ramas de la búsqueda y en el guardrail anti-cherry-picking, así que no hay una rama por la que se escape un chunk de más nivel del permitido. Es la **cuarta capa** de la tesis de confiabilidad, sobre la que vamos ahora.

Presidio (substrate-presidio) — El microservicio Docker local (Microsoft Presidio + spaCy, en 127.0.0.1:8400) que detecta y reemplaza PII por placeholders antes de chunkear.
custodia dual — El original con PII se marca confidencial y queda intacto en el grafo; a los vectores/reranker/LLM solo entra el texto anonimizado.
security_level — Nivel jerárquico de un chunk: publico < interno < confidencial. Sube a confidencial si el chunk tuvo PII.
clearance (fail-closed) — El nivel de acceso declarado en el request. La recuperación solo devuelve chunks con security_level ≤ clearance; sin declaración explícita, se filtra como publico.

✓ **Checkpoint** — con este capítulo podés responder: *Refuerza la Pregunta 4 (seguridad del sistema): la 4ª capa de la tesis de confiabilidad, con fail-safe en ingesta y fail-closed en recuperación.*

08 La taxonomía y la tesis de confiabilidad

Operación/plan/workflow/agente se ordenan con ladrillo/pared/casa/quién; y lo que hace confiable a un sistema no-determinista es 'catálogo cerrado · nada sin firma · nunca inventa' + privacidad.

INTUICIÓN

La regla mental que ordena los cuatro nombres que más se confunden: **operación** = **ladrillo** · **plan** = **pared** · **workflow/superskill** = **casa** · **agente** = **quién pone los ladrillos**.

Y la frase que resume por qué esto es seguro en producción: "**catálogo cerrado · nada sin firma · nunca inventa**" — más una cuarta capa, la privacidad del capítulo anterior. Son las **barandas** que contienen tanto al modelo como al humano.

MECANISMO

La jerarquía, con ejemplos reales:

- **Operation** = la pieza atómica del catálogo, pura y tipada. Ej.: `text.compose_narrative`, `artifact.publish`. Los planes se arman con éstas.
- **PlanTemplate** = un DAG de steps reusable. Ej.: `standup-digest-v1`.
- **Workflow** = el nombre **de cara al usuario** de un flujo ejecutable. Ej.: "Digest diario".
- **SuperSkill** = un flujo **propio del workspace**, promovido desde un plan que funcionó (la receta guardada del usuario). Es contra esto que matchea Nova en MATCH.
- **Skill** (`SKILL.md`) = ojo, es distinto: el **formato de archivo** (Agent Skills) en que se describen workflows instalables. Ej.: `skills/thalx/SKILL.md`.
- **Agente** = una persona del squad que ejecuta steps (el `actor` de un step).

Los tres pilares de la tesis: **catálogo cerrado** (los agentes solo usan operaciones registradas, no improvisan) · **nada sin firma** (toda capacidad nueva la aprueba un humano una vez, el build-gate) · **nunca inventa** (Nova responde `cannot` en vez de alucinar).

La **inversión catálogo ↔ personas**, el concepto detrás del roster: las personas del squad son etiquetas de identidad, no procesos con capacidad. La capacidad vive en las operations del catálogo, y Nova asigna el actor desde ahí — **nunca lee el squad**. El «3» de los primeros squads era un default de presentación: hoy una conversación inicial de setup (multi-turno) descubre qué trabajo necesita el usuario, y `deriveSquad` deriva el equipo **determinísticamente** de los skills elegidos («el LLM jamás decide composición de equipo»). El equipo nace del trabajo, no al revés — la inversión que ningún flujo de nodos tiene.

Dos reglas inviolables del catálogo: Regla A — el WorkspaceManifest es primitive: el agente se rehúsa a ejecutar sin manifest; se pre-computa nightly y el hot-path no hace queries inestables. **Regla B** — todo acceso vectorial se declara (`knowledge_access.requires_vector`): el runtime rechaza un vector lookup no declarado — ~70% menos queries a pgvector y auditabilidad de qué operations son caras de leer. Ambas siguen enforced hoy en el catálogo del spec.

A FONDO

Cómo se verifica el **catálogo cerrado** en el código: `resolveOperation` **tira** si un `operation_ref` no existe — un plan no puede referenciar una operación que no está registrada. El error clásico del practicante es mezclar capas: confundir **SuperSkill** (receta guardada del workspace) con **Skill** (formato de archivo) o con **Operation** (ladrillo del catálogo). Son tres cosas distintas.

Sobre el roster: de cara al marketing la narrativa es *"16 agentes, 3 equipos"*, pero los `actor` reales de los steps en el substrato son concretos: **Nova** (compone, no ejecuta), **Karina** (PMO), **Alexa** (Sales/ICP), **Sofia** (Contenido), **Marcus** y **Mae** (Media/video), **Forge** (`agent:forge`, ejecuta capacidades forjadas), **Control de calidad** (`system:evaluator`, no es persona) y **Vos** (`human:owner`). El complemento de FORGE es `learn-failures.ts`: un cron diario que agrupa fallos recurrentes por `(operation_ref, code)`, separa proceso/sistema y alerta a un humano — **nada se auto-arregla** (FORGE construye lo que falta; esto caza bugs de lo que ya existe).

Operation — La capacidad atómica, pura y tipada del catálogo cerrado. El ladrillo con el que se arman los planes. Ej.: artifact.publish.
SuperSkill — Un flujo propio del workspace, promovido desde un plan que funcionó tras validación humana. La receta reutilizable del usuario; lo que matchea el MATCH.
Skill (SKILL.md) — El FORMATO DE ARCHIVO en que se describen workflows instalables. No confundir con SuperSkill ni con Operation.
catálogo cerrado — Los agentes solo pueden usar operaciones registradas; resolveOperation tira si el ref no existe. Primer pilar de la tesis de confiabilidad.
learn-failures.ts — El cron diario que agrupa fallos recurrentes (operation_ref, code), separa proceso/sistema y alerta a un humano. Nada se auto-arregla.

✓ **Checkpoint** — con este capítulo podés responder: *Pregunta 5: Diferenciá operación · workflow · superskill · agente (regla ladrillo/pared/casa) — y la tesis de confiabilidad que atraviesa todo.*

09 Cómo trazar una corrida: los 4 lugares

Con solo un trace_id reconstruís la corrida entera por un endpoint, y para profundizar tenés cuatro capas: DB, Inngest, Langfuse y journald.

INTUICIÓN

La prueba final del expediente: te dan un **número de caso** (un `trace_id` + `workspace_id`) y tenés que **reabrir el expediente** y encontrar el acta completa — con la constancia de cada diligencia y su fuente. Si el sistema es de verdad auditable, con ese número solo alcanza para contar toda la historia.

Y si el acta no te basta, hay **cuatro archivos** distintos donde vive la evidencia, cada uno con más detalle que el anterior.

MECANISMO

La vista completa en un solo lugar: `GET /api/workspaces/:id/traces/:trace_id` con bearer. Lo ensambla la función `buildTraceDetail` (head + steps + artifacts + claims) — es el mismo endpoint que usa la oficina/demo. En el JSON identificás: el **intent** (qué se pidió), el **plan** (de qué PlanTemplate salió), los **steps** con su `exec` (status/timing de cada ejecución), los **artifacts** y **claims** con su provenance, y el **summary**.

Los **4 lugares** para profundizar:

1. **DB (Postgres)** — `step_executions`, la capa 1: **siempre** está, una fila por step, con status/cost/error.
2. **Inngest** — el executor durable (`execute-plan.ts`); se correlaciona por `traces.inngest_run_id`. Dashboard en `127.0.0.1:8288`.
3. **Langfuse** — un span por step con id **determinista** `${trace_id}-${stepId}` + la generation LLM anidada (prompt/tokens/costo). En `127.0.0.1:3030`.
4. **journald** — los logs del systemd `agent-squad-api.service`.

La cicatriz del idleTimeout y los tres anillos de verificación: Una cicatriz instructiva: el default de Bun corta toda conexión muda de más de 10s, y el modelo razona en silencio (los thinking deltas se filtran del stream) — las preguntas cortas pasaban, las «de trabajo real» morían y degradaban al camino clásico. Fix: `idleTimeout: 120`. El bug pasó todos los tests unit y E2E: lo cazó el **anillo exterior** — un manual de 34 pasos que un navegador re-ejecuta contra producción con usuario efímero y un `verification.json` por paso («si un paso se rompe, su sello deja de decir VERIFICADO»). Tres anillos: unit (app+motor), E2E Playwright y el manual vivo. «Si tu sistema le habla a usuarios, algo tiene que recorrerlo como un usuario.» Precisión: la suite E2E de CI corre contra un **mock determinista** del motor (puerto 4998), por push — el manual vivo es el único que toca producción; no hay robot nocturno. Otra cicatriz del mismo cuaderno: un runaway de ClickHouse clavó 7.7 cores durante 19 horas — hoy corre capado a 2 CPUs.

A FONDO

Lo que hace esto poderoso: el span de Langfuse tiene un id **determinista** (`${trace_id}-${stepId}`), así que lo cruzás con la fila de la DB **sin adivinar**. Con solo el `trace_id` respondés: qué steps corrieron, en qué orden, con qué inputs/outputs, cuánto costó cada uno, qué `verdict` dio el evaluador, dónde falló y —para los LLM— el prompt exacto y los tokens.

Para **debuggear un fallo**, clasificás en dos familias leyendo el `error` estructurado (`code` / `message` / `retryable`): **proceso** (`HUMAN_GATE_TIMEOUT` — nadie firmó; el sistema funcionó, faltó la firma) vs **sistema** (`STEP_HANDLER_ERROR` — un backend reventó). Distinguirlas es media batalla en un incidente: una se resuelve con una firma, la otra arreglando un backend. Y una capa extra: **OpenTelemetry nativo** (issue #26) exporta OTLP portable a Honeycomb/Jaeger en paralelo a Langfuse, **off por defecto** (no-op sin endpoint). Cuidado: un span OTel manual mal ubicado difiere los spans del executor y rompe la durabilidad de Inngest — eso solo lo caza el smoke e2e `verify-demo-e2e.ts`, no la suite unit.

El 4º lugar — journald: los logs crudos del proceso systemd (stack traces, arranque, errores no capturados en la trace). Se leen con `journalctl -u agent-squad-api.service -f`. Es la red de última instancia cuando algo revienta tan temprano que ni llegó a escribirse en la trace.

buildTraceDetail — La función que ensambla la vista completa de una trace (head + steps + artifacts + claims) para GET /api/workspaces/:id/traces/:trace_id.
inngest_run_id — El campo de traces que correlaciona una corrida del substrato con su ejecución durable en Inngest (execute-plan.ts).
span determinista — El id ${trace_id}-${stepId} del span de Langfuse, que permite cruzar la observabilidad con la fila de step_executions sin adivinar.
familias de fallo — Proceso (HUMAN_GATE_TIMEOUT — faltó la firma) vs sistema (STEP_HANDLER_ERROR — reventó un backend). El campo retryable decide si reintentar sirve.

✓ **Checkpoint** — con este capítulo podés responder: *Pregunta 7: Te dan un trace_id + workspace_id — ¿cómo ves la corrida completa y qué 4 lugares mirás (DB / Inngest / Langfuse / journald)?*

Glosario

Cada término en llano y en preciso.

Agente

Quién pone los ladrillos: la persona del squad que ejecuta.

El actor de un step. Roster real: Nova (compone), Karina, Alexa, Sofia, Marcus, Mae, Forge (agent:forge), Control de calidad (system:evaluator), Vos (human:owner).

---

anillos de verificación

Las tres capas de pruebas: tests unitarios, E2E de navegador, y un manual vivo que recorre producción como un usuario real.

Unit (app+motor) · E2E Playwright en CI contra un mock determinista del motor (puerto 4998) · manual de 34 pasos re-ejecutado contra producción con verification.json por paso. El manual es el único anillo que toca producción; no hay robot nocturno.

Artifact

Un documento del expediente.

Algo producido por la corrida (kind, content_addr sha256, produced_by.{trace_id,step_id}). Queda en pending_review hasta que se aprueba el gate.

---

assertHumanGatePresent

El guardia de la puerta de salida: ningún plan se lanza sin su gate humano final, sin importar quién armó el plan.

Helper server-side que exige ≥1 human gate terminal con fallback exactamente 'fail' antes del launch — «el servidor no puede confiar en el upstream».

---

assertPureContract / staticSafetyCheck

Los chequeos que exigen que la op forjada sea pura.

El contrato + chequeo estático (forge/safety.ts) que verifican pureza antes de registrar una capacidad forjada.

---

build-gate

Aprobar una herramienta nueva antes de meterla a la caja.

Aprobación humana (forge.approval) de una CAPACIDAD forjada antes de registrarla en el catálogo. Pertenece al loop de FORGE, no al plan del usuario.

---

bwrap (sandbox)

La caja aislada donde se prueba el código nuevo sin que toque nada.

Aislamiento de kernel (bubblewrap) con red denegada y sin secrets; se usa en verify y en execute (executeInSandbox). El muro real; el staticSafetyCheck es defensa en profundidad.

CANNOT

Eso no lo hacemos — dicho honestamente.

Desenlace honesto cuando faltan poderes/credenciales para efectos externos. Encarna la tesis 'nunca inventa'.

Claim

Un hecho probado, con su fuente.

Afirmación sujeto-predicado-objeto con provenance (trace_id+step_id, FK al step exacto). La trazabilidad cerrada: de un resultado vas a su origen.

---

claim.recall_decisions

La operación que le recuerda al composer lo que ya decidiste antes — el modelo hereda tu decisión como input, no la adivina.

Recupera decision-claims del grafo (predicates consolidatedDecision / workspace.decision / decision) y las inyecta al composer. El gap histórico (approvalComment excluido del filtro) se cerró el 05-jul-2026: el gate emite claims decision cuando hay comment, y los 33 históricos se backfillearon.

---

compose vs intents

compose = decile a Nova qué querés; intents = ya sé qué workflow correr.

POST /compose (lenguaje natural, Nova interpreta) vs POST /api/intents (estructurado: kind+subject_label+acceptance_criteria_ref, compila desde PlanTemplate sin interpretación). El cron del standup usa /api/intents.

CPUWeight / MemoryMax

Carril prioritario y límite de combustible en un servidor compartido.

Drop-in systemd (resources.conf): CPUWeight=800 (gana el scheduler bajo contención) y MemoryMax=4G (techo de memoria) para que otros jobs del box no starven al substrato.

Custodia dual

El original bajo llave; solo circula la copia tachada.

El artefacto con PII se marca confidencial e intacto en el grafo; a document_chunks/reranker/LLM solo entra texto anonimizado (<PERSONA>, <EMAIL>, <ID_TRIBUTARIO>…).

DAG

Un grafo de pasos sin ciclos — como las dependencias de celdas en Excel: cada paso arranca cuando sus insumos están listos.

Grafo dirigido acíclico; el motor lo ordena con topo-sort (Kahn, con detección de ciclos). El executor actual recorre ese orden secuencialmente — el paralelismo de ramas es potencial del modelo, no comportamiento implementado.

---

deriveSquad

La función que arma el equipo a partir del trabajo elegido — nunca al revés, y nunca lo decide el modelo.

Deriva la composición del squad determinísticamente de los skills seleccionados en la conversación de setup (multi-turno); comment literal en el código: «El LLM JAMÁS decide composición de equipo».

---

document.query

El buscador que responde preguntas cruzando dos catálogos y filtrando por acceso.

Read-path híbrido: rama vectorial (e5) + léxica (ts_rank_cd) fusionadas por RRF, reranker Cohere (ventana 20 → top-5), filtro por security_level/clearance en las 3 ramas.

---

evaluator.run

El control de calidad del plan.

Step (actor system:evaluator) que valida el artifact contra el acceptance_criteria y deja un verdict, entre la producción del artifact y el run-gate. El humano puede overridear el verdict.

Fail-safe vs fail-closed

Si la tachadora se rompe no archiva nada; si no sabés tu nivel, te tratan como público.

Fail-safe (ingesta): si Presidio cae, no indexa y encola en cuarentena. Fail-closed (recuperación): sin clearance explícito, se filtra como publico.

FORGE

La fragua: el sistema construye la herramienta que le falta.

4º desenlace de Nova. Ante un cannot genuino con capacidad PURA: genera spec+tests+código → verify en sandbox bwrap (sin red/secrets) → build-gate → registry. Off por defecto (FORGE_ENABLED).

Inngest

La cinta transportadora que no pierde ninguna tarea aunque se corte la luz.

Motor de ejecución durable self-hosted (Docker): step.run() con retries + step.waitForEvent() para gates. Dashboard en 127.0.0.1:8288.

Intent

Abrir un caso: el pedido ya declarado.

El deseo estructurado (tabla intents; kind + subject_label + acceptance_criteria_ref). Entrada del substrato que se compila a un plan.

Langfuse

El registro de cada llamada al LLM.

Observabilidad LLM (v3 + ClickHouse, Docker): un span por step con id determinista ${trace_id}-${stepId} + la generation (prompt/tokens/costo). En 127.0.0.1:3030.

learn-failures.ts

El cron que caza patrones de fallo — pero no arregla solo.

Cron diario que agrupa fallos por (operation_ref, code), separa proceso/sistema y alerta a un humano. Complemento de FORGE; nada se auto-arregla.

MATCH

Ya lo tengo en carta: reuso algo hecho.

Desenlace en que Nova reusa un SuperSkill del workspace (los SuperSkills del workspace — hasta 20 — entran inline al prompt de Nova y la llamada LLM decide el match). El camino más barato.

multilingual-e5-small

El bibliotecario local que entiende de qué va cada texto, gratis.

Modelo de embeddings ONNX in-process ($0, sin red; ADR-0001). Alimenta el ruteo (MATCH), el recall de claims (claim.recall_*) y la rama vectorial de document.query.

Nova

La recepcionista: recibe el pedido y decide qué se hace.

El agente que compone planes (no ejecuta steps). Ante un pedido produce un desenlace: MATCH, PLAN, FORGE o CANNOT.

OpenTelemetry (issue #26)

Export de trazas portable a otras herramientas, apagado por defecto.

Export OTLP nativo a Honeycomb/Jaeger en paralelo a Langfuse, off por defecto (no-op sin endpoint). Un span manual mal ubicado rompe la durabilidad de Inngest (lo caza verify-demo-e2e.ts).

Operation

El ladrillo: la pieza mínima reutilizable.

Capacidad atómica, pura y tipada del catálogo cerrado (ej. text.compose_narrative, artifact.publish). resolveOperation tira si el ref no existe.

Plan

El plan de acción del caso.

Secuencia de steps (tabla plans) compilada desde una PlanTemplate. Su status es vestigial; el que importa es trace.status.

PlanTemplate

La receta reusable de un plan.

Un DAG de steps reusable (ej. standup-digest-v1) desde el que se compila el plan concreto de cada corrida.

Presidio (substrate-presidio)

La máquina que tacha los datos personales antes de archivar.

Microservicio Docker local (Microsoft Presidio + spaCy, 127.0.0.1:8400, mem_limit 1g, stateless) que detecta y reemplaza PII por placeholders antes de chunkear.

Provenance / Lineage

El resultado conoce su origen.

Los enlaces que van de un artifact/claim hasta el step exacto que lo produjo (produced_by.step_id / provenance.step_id).

RRF

La forma de combinar dos rankings en uno.

Reciprocal Rank Fusion: fusiona el ranking vectorial y el léxico; también es el fallback si el reranker externo (Cohere) falla.

---

run-gate / human_gate

Firmar el entregable antes de que salga.

Step que suspende la corrida (awaiting_human, ≤72h, step.waitForEvent) esperando aprobación humana de un RESULTADO. Señal: operation_ref human_gate.approve + exec:null. Se resuelve por POST /api/approvals.

---

security_level / clearance

Cada estante tiene un nivel; solo ves hasta tu credencial.

security_level jerárquico (publico<interno<confidencial, sube a confidencial si hubo PII). document.query filtra por el clearance del request, fail-closed a publico. Control de acceso de aplicación, no criptográfico.

Skill (SKILL.md)

El formato de archivo de un workflow instalable.

El formato Agent Skills en que se describen workflows instalables (ej. skills/thalx/SKILL.md). Distinto de SuperSkill y de Operation.

Step

Una diligencia a realizar.

La definición estática de un paso (tabla steps): referencia UNA operación (operation_ref) + el agente (actor) + si necesita gate; ordenado por ordinal.

Step Execution

La constancia de una diligencia hecha.

El registro de ejecutar un step dentro de una trace (tabla step_executions): status/timing/cost/output/error. Una fila por step por corrida; capa 1 de observabilidad, siempre existe.

Substrato

La trastienda donde el squad de verdad hace y archiva el trabajo — el expediente de todo.

El registro-grafo del sistema (apps/api, Bun, systemd agent-squad-api.service), system-of-record que usa Postgres, Inngest y Langfuse. No es la base de datos.

SuperSkill

La receta propia que guardaste porque funcionó.

Flujo propio del workspace promovido desde un plan exitoso (con validación humana). Aislado por workspace_id; es contra esto que matchea el MATCH.

Tesis de confiabilidad

Las barandas que hacen seguro un sistema de agentes.

'Catálogo cerrado · nada sin firma · nunca inventa' + privacidad en capas (fail-safe + fail-closed). El contrato de seguridad del substrato.

Trace

El acta de una sesión de trabajo — una corrida concreta.

UNA ejecución del plan (tabla traces) con estado queued → running → awaiting_human → succeeded; se correlaciona con Inngest por inngest_run_id.

Workflow

La casa: el flujo con nombre para el usuario.

El nombre de cara al usuario de un flujo ejecutable respaldado por un PlanTemplate (ej. 'Digest diario').

Auto-evaluación

Respondételas solo, sin mirar. Si podés las 7 con confianza, tenés el modelo mental.

1. ¿Qué es el "substrato", en una frase? (y por qué NO es la base de datos) → Cap. 1

2. Explicá la cadena intent → plan → trace → step → artifact → claim, y la diferencia step (definición) vs step_execution (lo que pasó). → Cap. 2

3. ¿Quién es Nova y cuáles son sus 4 desenlaces? Una frase por cada uno. → Cap. 3

4. ¿Qué es FORGE, cuándo se dispara, y por qué solo construye ops "puras"? → Cap. 5

5. Diferenciá: operación · workflow · superskill · agente (la regla ladrillo/pared/casa). → Cap. 8

6. ¿Cuál es la diferencia entre el run-gate (human_gate) y el build-gate? → Cap. 4

7. Te dan un trace_id + workspace_id: ¿cómo ves la corrida completa, y qué 4 lugares mirás (DB / Inngest / Langfuse / journald)? → Cap. 9