NLTlabs
NEW LOGIC TECH

Building Agent Systems on AgentForge

How to build one, and why not just the SDK

A technical white paper for engineers and architects

Version 1.1 — 2026-07-11

0. Executive summary

The Claude Agent SDK is a good way to build a single agent. It ships a real agent loop, tool use, MCP client support, subagents, hooks, permission modes, prompt caching, and context compaction. If you are building one agent, or a handful that run in one process for one team, you probably do not need anything more, and this paper will tell you so plainly (§15).

The moment you cross into many agents, many tenants, long-running work, governed tool access, or accountable spend, you stop needing an agent runtime and start needing an agent **operating system** — and that gap is almost entirely undifferentiated platform code that every serious team ends up writing themselves. AgentForge is that platform: one Docker image, three configuration-driven operating modes, deployment-topology agnostic, with durability, multi-tenancy, policy enforcement, cost governance, audit, and an agent lifecycle pipeline as first-class primitives.

This paper is deliberately How **and** Why. Every "how" section names what the equivalent would cost you to build on the raw SDK, and every "why" section is grounded in a specific implemented mechanism, not in adjectives. It is written to be credible to someone who has actually shipped on the SDK.

The one-line decision rule: stay on the SDK while your agents fit in one process, one trust boundary, and one person's head; adopt AgentForge when durability, tenancy, policy, or cost accountability become someone's job.

Part I — Framing the choice

1. Two altitudes of "agent"

The word "agent" is overloaded, and most build-vs-adopt arguments fail because the two sides are comparing different things. This paper fixes the terms.

Altitude 1 — the agent runtime (what the SDK is). The Claude Agent SDK (formerly the Claude Code SDK; Python and TypeScript) wraps the Messages API in the loop that makes a model agentic: call the model, inspect `stop_reason`, execute any requested tool calls, feed results back, repeat until the model emits a final turn. On top of that loop it provides, out of the box:

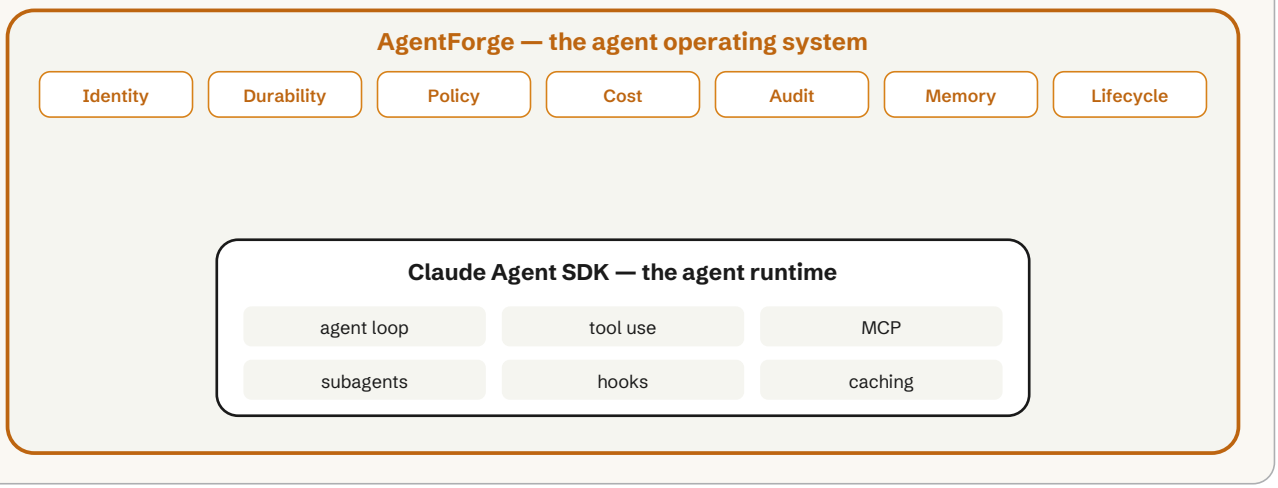
- **Tool use** — custom tools (JSON-schema + your executor), parallel tool calls, strict schema enforcement, and Anthropic-provided tools (`bash`, `text_editor`, `computer_use`, `memory_tool`, `web_search`, `web_fetch`, `code_execution`).
- **MCP client support** — connect to remote HTTP/SSE MCP servers with per-server and per-tool allow/deny control.
- **Subagents, hooks, and permission modes** — the SDK can spawn child agents, intercept tool calls with hooks, and gate actions behind permission prompts.
- **Streaming, prompt caching, context compaction, sessions** — incremental output, 0.1× cache reads on repeated prefixes, and automatic context management as history grows.

This is not a thin client. It is a capable single-process agent runtime, and for a large class of problems it is the right and sufficient tool. Do not let the rest of this paper talk you out of using it when it fits.

Altitude 2 — the agent operating system (what AgentForge is). An OS does not replace the runtime; it is the substrate that lets many runtimes run safely, durably, and accountably on shared infrastructure. It owns the things that are not any single agent's job: identity, isolation, scheduling, persistence, policy, audit, cost, and lifecycle. AgentForge runs Claude Agent runtimes as its execution units and adds that substrate around them.

The rest of Part I establishes where the boundary between these two altitudes actually sits — because that boundary, not a feature checklist, is the real decision.

FIGURE 1 The SDK gives you a runtime; AgentForge wraps it in a platform



2. Where the SDK's boundary actually is

Here is the honest version of the "why a platform" argument: the SDK gives you a runtime, and then an enterprise inevitably builds the same eight things on top of it. None of these eight are Claude-specific or differentiating — they are the undifferentiated heavy lifting that every multi-agent, multi-tenant deployment reinvents. Each is expanded, with the AgentForge mechanism, later in the paper.

#	What you end up building on raw SDK	Why the SDK can't give it to you	Covered in
1	**Durable / resumable execution** — if the process dies mid-task, work is lost. No checkpoint, no replay, no resume-from-node.	The SDK loop is in-memory and synchronous by design.	§7, §11
2	**Multi-tenant isolation** — one API key, one trust boundary. No per-tenant scoping of data or credentials.	The SDK is single-tenant; isolation is an application concern.	§12

#	What you end up building on raw SDK	Why the SDK can't give it to you	Covered in
3	**Centralized policy enforcement** — allow/deny lists exist, but no RBAC/ABAC, no server-enforced "agent X may call tool Y only under condition Z."	Policy is above the runtime's pay grade.	§7, §12
4	**Per-tenant cost accounting** — usage is in each response; aggregation, budgets, and wire-level enforcement are yours to write.	The SDK reports tokens; it does not hold a ledger or a budget gate.	§13
5	**Audit / observability** — usage numbers, but no OTel GenAI spans, no immutable per-tenant audit log.	Instrumentation is a deployment concern.	§12
6	**Agent lifecycle & versioning governance** — prompts and tool grants live in your code; no registry, no upgrade proposals, no conformance gates.	The SDK has no opinion on how agents are packaged or promoted.	§4, §9, §14
7	**Shared memory / RAG layer** — `memory_tool` is file-backed key-value; no vector store, no semantic recall, no cross-agent knowledge, no dedup.	It's a per-agent scratchpad, not a knowledge substrate.	§8
8	**Deployment topology management** — laptop vs. VM vs. K8s vs. multi-region is entirely your infrastructure.	The SDK is a library, not a deployable service.	§3, §15

The point is not that these are impossible on the SDK. The point is that they are the same code every team writes, they are hard to get right (durability and cost enforcement especially), and they are pure cost — none of them make your agents better, they just make a fleet of agents survivable in production.

3. AgentForge's answer in one diagram

AgentForge's identity is a **single Docker image, configuration-driven operating modes, deployment-topology agnostic**. The same image runs on a laptop, a VM, a Kubernetes cluster, or twenty instances on one host; the operator selects a posture with `AF_MODE` and optional `AF_CAPABILITY_*` overrides — no code changes, no image variants.

Three operating modes (a Kubernetes Pod Security Standards–style profile pattern):

AF_MODE	For	Identity	Policy	Supply chain
`permissive` (default)	Laptop, evaluation	logged	logged	off
`balanced`	Internal team / staging	strict	strict	warn
`strict`	Regulated production	strict	strict	strict

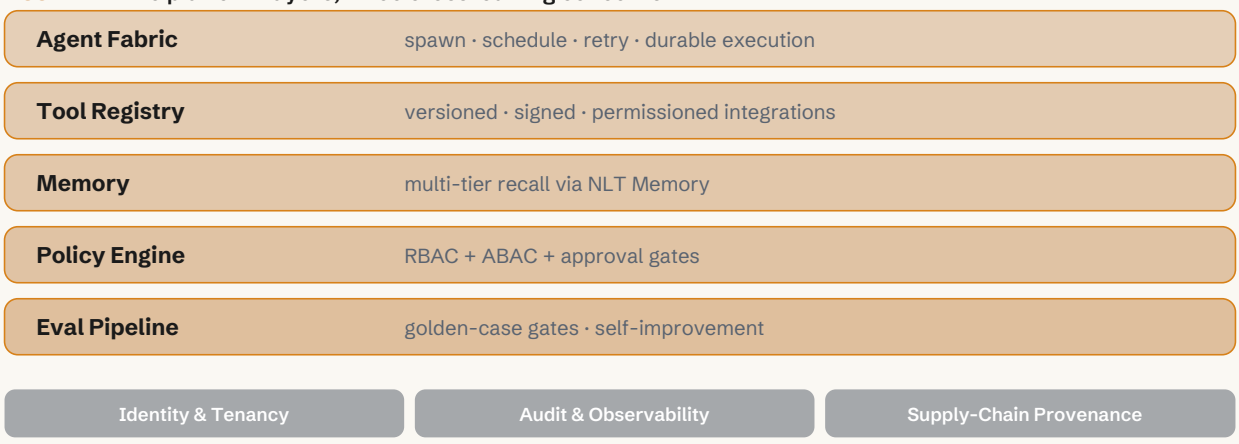
Five reference-architecture layers — the shape of the platform:

1. **Agent Fabric** — spawn, schedule, retry, terminate; durable workflow execution (`backend/orchestrator/`, `backend/executor/`, `backend/workflows/`).
2. **Tool Registry** — versioned, signed, permissioned integrations (`backend/supply_chain/`, Ed25519 trust store).
3. **Memory** — multi-tier working/short/long-term memory via NLT Memory over HTTP.
4. **Policy Engine** — RBAC + ABAC + approval policy, enforced at every tool action (`backend/policy/`).
5. **Eval Pipeline** — offline regression, golden-case promotion gates, canary, eval-gated self-improvement.

Three cross-cutting concerns — the things every layer needs:

- **Identity & Tenancy** — per-agent identity, per-task tokens, signed delegation.
- **Audit & Observability** — OTel GenAI semantic-convention spans, immutable per-tenant log.
- **Supply-Chain Provenance** — agent SBOM with signed attestations.

FIGURE 2 Five platform layers, three cross-cutting concerns



Every capability is opt-in or config-gated. A single-tenant `permissive` deployment ignores most of it and behaves like a thin wrapper around the SDK runtime; a `strict` deployment turns the same image into a governed, audited, multi-tenant platform. That range from one image is the core of the "why."

Part II — How to build one

4. Anatomy of an agent

An AgentForge agent is a declarative package, not a block of imperative code. Its front matter lives in `AGENTS.md` (Linux Foundation `AGENTS.md` v1.0 spec; current platform schema `2.1`, validated by `backend/models/agent_config.py`). The load-bearing fields:

```
name: research-brief
description: Produces a sourced market brief from a topic.
model: sonnet          # haiku | sonnet | opus (canonical short names)
brain_rationale: "Why this agent exists + what prior runs taught us."  # required at schema 2.1
agent-type: expert     # expert | task | workflow
risk-level: low        # low | medium | high | trusted

allowed-tools: "Read, Grep, Bash(git:*), WebFetch(domain:api.example.com)"
tool-targets:          # scopes base Bash/Write/Edit — required unless trusted
  Bash: ["git", "uv"]
  Write: ["./out/**"]

mcp-servers: [nlt-memory, exa]      # registry keys, not raw plugin config
skills: [web-research, markdown-brief]

completion_criteria: "A brief with >=3 cited sources and a recommendation."
confidence_threshold: 0.0           # >0 enables an escalation gate
guardrails:
  - type: non-authoritative
output_schema: { type: object, properties: { brief: { type: string } } }
```

Two design choices are worth pausing on, because they are why the platform can govern agents the SDK cannot:

- **Tool grants are scoped and enforced.** `allowed-tools` plus `tool-targets` produce the exact `--allowedTools` set the runner passes to the Claude runtime. Base `Bash`/`Write`/`Edit` without targets is rejected at load time (unless `risk-level: trusted`). On the raw SDK, tool scoping is whatever you remember to enforce in each executor.
- **Agents are authored and gated, never hand-edited into the running system.** Installation runs a gate chain — `schema → policy → conformance → cohesion → interface` (`run_gates_on_config`; the final `interface` gate is warn-only) — before an agent can serve traffic. New schema fields ship with safe defaults and are populated for existing agents through the **upgrade / proposal pipeline** (`python -m backend.scripts.upgrade_agents`), not by editing files in place. This is the difference between a config file and a governed artifact.

Schema 2.1 also carries a **typed-interface contract** — `role` (one of `planner` / `producer` / `judge` / `gateway` / `aggregator` / `router`), `input_schema`, `failure_mode`, `capability`, and `memory_footprint` — which the `interface` gate reads to check that agents wired together actually compose.

5. Your first agent, end to end

Standing up the platform itself is one Docker image: `make up` brings the AgentForge container up (it serves the API and dashboard on `localhost:8001` by default), and it runs in `permissive` mode with no

auth gateway until you configure one. From there the developer loop uses the integration-kit skills:

```
af-init          # scaffold agentforge/projects/<slug>/, .mcp.json, starter AGENTS.md
# ... author agents/research-brief/AGENTS.md ...
af-publish       # PUT /projects/<slug>/agents/<name>  (versioned, content-hashed)
af-smoke-test    # health + auth + agent invoke + workflow dry_run
```

Then invoke it:

```
curl -sf http://localhost:8001/projects/<slug>/tasks/invoke \
  -X POST -H 'authorization: Bearer afp_...' -H 'content-type: application/json' \
  -d '{"prompt": "Brief me on the home-espresso market", "config_hint": "research-brief"}'
```

To see what the platform is doing for you, compare that to the minimal raw-SDK single agent — roughly 30 lines you write and own:

```
import anthropic
client = anthropic.Anthropic()
tools = [{ "name": "get_data", "description": "...", "input_schema": {...} }]
messages = [{"role": "user", "content": user_query}]
while True:
    resp = client.messages.create(model="claude-opus-4-8", max_tokens=1024,
                                  system=system, tools=tools, messages=messages)
    if resp.stop_reason == "end_turn":
        print(resp.content[0].text); break
    if resp.stop_reason == "tool_use":
        messages.append({"role": "assistant", "content": resp.content})
        results = [{"type": "tool_result", "tool_use_id": b.id,
                    "content": execute_tool(b.name, b.input)}
                   for b in resp.content if b.type == "tool_use"]
        messages.append({"role": "user", "content": results})
```

That code is correct and, for one agent, entirely sufficient. Everything AgentForge adds — versioned publish, the gate chain, tenant scoping on the invoke, an audit span, a spend ledger entry, a resumable run record — is invisible in this snippet because it is precisely the code you did not write. That is the trade in one screen: the SDK gives you the loop; the platform gives you everything around the loop that you would otherwise build and maintain.

Cost and latency (rough estimate): one Sonnet invocation with light web research is on the order of **$0.10–0.40** and **15–60 s**. You won't have to estimate in practice — AgentForge records the actual `cost_usd` and `duration_ms` for every node it runs (§13), so the real numbers arrive with each invocation.

6. From one agent to a workflow

A workflow is a declarative DAG of nodes. Spec YAML normally lives in the consumer's own repo and is mounted via `AF_EXTERNAL_WORKFLOWS_DIR` (the platform-owned `backend/workflows/specs/` set is intentionally near-empty). Six node kinds cover the useful cases:

`kind`	Runs	Requires	LLM?
`task` (default)	An agent (Claude runtime)	`agent:`	yes
`gate`	A human-in-the-loop pause	`depends_on` or `payload_from`	no
`python`	An in-process function	`func: module:callable`	no

kind	Runs	Requires	LLM?
`http`	A guarded HTTP fetch	`url:` + `http_allowlist`	no
`script`	A registered subprocess	`script_id:`	no
`run_ref`	A cross-workflow handoff	upstream run reference	no

Nodes wire together with `depends_on`, and pass artifacts through `$ref` interpolation:

```
name: market-brief
inputs: [topic]
nodes:
  research:
    agent: research-brief
    inputs: { topic: $inputs.topic }
    output_schema: { type: object, properties: { findings: {type: array} } }
    output_repair_retries: 1        # re-invoke with the schema error on mismatch
  writeup:
    agent: brief-writer
    depends_on: [research]
    inputs: { findings: $research.findings }   # scalar/array pulled by dotted ref
output: writeup
```

The mechanically important detail: a dotted ref (`$research.findings`) pulls a typed field out of an upstream node's validated `output_schema` output. Without a schema, `$research` is the raw model text and `$research.findings` is empty (under `spec_version: 2`, an unresolved dotted ref instead errors rather than silently emptying) — so schemas are how you get reliable structured hand-off between agents, not just validation. (For work whose shape you don't know up front, an agent can set `orchestrate: true` and let a planner compile a DAG dynamically — see §10.1 — but a hand-written spec is the default and the easiest to reason about.)

Cost and latency (rough estimate): two agent invocations in series, so roughly **$0.20–0.80** and **40–120 s** end to end — about double the single agent, because `writeup` waits on `research`.

7. The hard parts workflows handle for you

These four are where "I could just write it in SDK glue" stops being true.

- **Gates (human-in-the-loop).** A `gate` node pauses the run (`awaiting_gate`), emits a `gate.pending` SSE event, and persists the pause to Postgres. A human resolves it with `POST /workflows/runs/{run_id}/gates/{node_id}`; descendants run on approval, are marked skipped on rejection. Paused runs auto-cancel after `gate_timeout_seconds`. On the SDK this is a durable state machine you build and host yourself.
- **Loops with convergence.** A loop re-runs its member nodes up to `max_iterations` with an early-exit predicate. Four convergence kinds: `truthy` (a member emits its own done-signal), `equality` (a field is stable across iterations), `judge` (a dedicated judge owns the stop decision), and `expression` (a safe predicate over member fields) — the primitives behind debate, voting, and refine-until-good patterns, with an optional `convergence_verify` ground-truth gate so a loop can't declare success it didn't achieve. Cost is bounded by `max_iterations` — each pass costs at most its members' spend, itself capped by their per-node `timeout_s`; cycles between loop members are exempted from the DAG acyclicity check, everything else stays acyclic.

- **Deterministic external-job dispatch (`kind: script`).** An LLM driving a "POST then poll until done" loop through Bash is unreliable — it bails early and fabricates a result. A `script` node runs a registered, deterministic subprocess (e.g. `external-job-relay`: HMAC-signed POST, a real `while` poll loop) bounded by the node's `timeout_s`, not the runtime's ~10-minute ceiling. The script's stdout envelope is the node output.
- **Artifact verification.** A node can declare `artifact_outputs`; the platform captures the file from the node's working directory, PUTs it to the artifact store, and verifies content-type, minimum size, and SHA-256 at node close. The agent writes the file; it does not write storage or verification code.

Each of these is durable across process death (§11), which is what separates them from an in-memory `asyncio` orchestration you'd hand-roll.

8. Memory and tools, done right

Memory. The SDK's `memory_tool` is a file-backed key-value scratchpad — fine for one agent's working state, not a knowledge layer. AgentForge agents talk to **NLT Memory** over HTTP (`POST /v1/recall`, `/v1/remember`, `/v1/reinforce`, plus a knowledge-graph surface): hybrid RAG with semantic recall, automatic dedup and decay, confidence reinforcement on cited memories, and scoped sharing (`private` / `domain` / `hive` / `group:<name>`) enforced server-side. The canonical pattern for external data is cache-through NLT Memory: check it first, on a miss fetch (Exa/Firecrawl), safety-gate, then store — so the second agent to need the same page pays nothing. On the SDK you would stand up a vector store, an ingestion pipeline, and a dedup strategy yourself.

Tool transport. AgentForge treats the CLI-grant-vs-MCP-server choice as a token-cost lever, not a style preference. Every declared MCP server injects all its tool schemas into context on every run (measured as `mcp.schema_tokens`). So the rule is: prefer a scoped **CLI grant** (`Bash(gh:*)`, `Bash(git:*)`) for any mature binary the model already knows, and reserve **MCP servers** for auth-heavy, credential-bridged, or stateful integrations (Linear, Google, NLT Memory). The Docker `github` MCP server (~93 tools, ~55k schema tokens on every run) was retired in favor of the `gh` CLI for exactly this reason. The SDK gives you both transports but no telemetry or policy to choose between them.

9. The wiring you don't write

Everything so far assumed you told the platform which agent to run and how the nodes connect. AgentForge's more distinctive claim is that, for a large class of work, **it wires the agents for you**. Four mechanisms do this, and none exist in a raw SDK app — there you write the router, the config, the orchestration graph, and the reload path by hand.

9.1 Routing — the right agent without naming it

You can pin a specific agent with `config_hint`, but you don't have to. Send a bare prompt and the **matcher** selects the agent: a hybrid of semantic embeddings (384-dim) and BM25 lexical scoring, fused by reciprocal-rank fusion and scored against each agent's description, keywords, and example utterances. Confident matches route in ~15 ms; ambiguous ones fall back to an LLM classifier. Thresholds are explicit (exact ≥ 0.6, close ≥ 0.3; below that, no match — which can itself trigger a proposal, §9.2).

Why it matters: on the SDK, "which agent handles this request" is a routing layer you design and maintain. Here it is a platform primitive, and the first gate every invoke passes through.

9.2 Authoring — agents the system proposes and repairs

Agents need not be hand-written. A **proposal engine** (itself a system agent) turns a request — "create an agent that does X" — into a schema-valid `AGENTS.md` with model tier, tool grants, completion criteria, and output schema filled in. A second flow runs the other direction: given failure telemetry for an existing agent (failure rate, retry rate, recurring blocker strings), it proposes revisions to three low-risk fields — `guardrails`, `completion_criteria`, `risk-level` — as evidence-backed edits.

Proposals are not trusted blindly: every one flows through the same gate chain (§4) and the quality pipeline (§14) before it can serve traffic. The system drafts; the gates decide.

Why it matters: on the SDK, every agent config is yours to write and every fix is a manual edit from a hunch. Here, authoring and repair are a governed, evidence-driven loop.

9.3 Orchestration — complex work that plans itself

The mechanics of a compiled dynamic plan appear in §10.1; the decision in front of it is automatic. A conservative dispatcher inspects each incoming task and decides whether it needs the multi-agent planner at all — an explicit `orchestrate: true`, a long prompt, or an architecture-classification with multi-deliverable cues ("build X **and** Y **and** a Z"). Simple tasks stay on the fast single-agent path; only genuinely multi-step work is escalated. You don't choose the graph, and you don't pay planning cost on work that doesn't need it.

Why it matters: the SDK makes you decide, up front and in code, whether a request is single- or multi-agent. Here that decision is made per task, at runtime, by heuristics tuned to keep the common case cheap.

9.4 Registration — agents go live without a redeploy

New or edited agents wire themselves into the running system. A filesystem watcher on the external-agents root reloads on any `AGENTS.md` add/change/delete and rebuilds the matcher's indexes within a ~300 ms debounce — no restart, no redeploy. (Production deployments can instead self-register over HTTP with a versioned, content-hashed, rollback-safe `PUT`.) Either way, publishing an agent is a data operation, not a deploy.

Why it matters: on the SDK, changing an agent means shipping code. Here it means writing a file or making one API call, and the platform re-wires routing around it live.

Taken together, these four are the "auto agent wiring" that most separates an agent OS from an agent runtime: the platform decides which agent runs, how it was authored, whether it needs help from others, and when it goes live — leaving you to write agents, not the machinery that runs them.

10. A worked example: growing into an orchestrated system

Now carry the same example — the `research-brief` agent from §5 — all the way up. Each rung adds exactly one capability, and nothing below it changes.

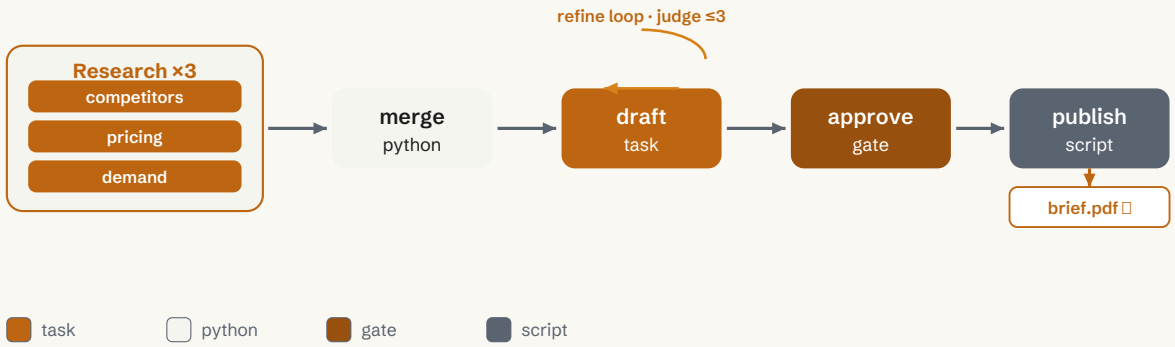
- **Rung 1 — one agent (§5).** `research-brief` takes a topic, returns a brief.
- **Rung 2 — a linear workflow (§6).** `research → writeup`, two nodes.
- **Rung 3 — an advanced workflow.** Make it production-shaped: research three angles in parallel, merge them deterministically, refine the draft until a judge is satisfied, get a human sign-off, then deliver a verified PDF.

Prefer to start small? The two-node `research → writeup` workflow from §6 is already complete and useful. Read the advanced spec below as a menu — each construct (fan-out, merge, refine loop, gate, delivery) is independently optional. Add one when you need it; nothing forces you to adopt all of it at once.

```
name: market-brief
inputs: [topic]
on_error: partial
nodes:
  # 1. Fan out — three angles run concurrently (no depends_on between them)
  competitors:
    agent: research-brief
    inputs: { topic: $inputs.topic, angle: "competitive landscape" }
    output_schema: { type: object, properties: { findings: {type: array} } }
  pricing:
    agent: research-brief
    inputs: { topic: $inputs.topic, angle: "pricing and willingness to pay" }
    output_schema: { type: object, properties: { findings: {type: array} } }
  demand:
    agent: research-brief
    inputs: { topic: $inputs.topic, angle: "demand signals" }
    output_schema: { type: object, properties: { findings: {type: array} } }

  # 2. Deterministic merge + dedup — no LLM, no token cost, typed output
  merge:
    kind: python
    func: briefs.merge:dedup_findings
    depends_on: [competitors, pricing, demand]
    inputs: { a: $competitors.findings, b: $pricing.findings, c: $demand.findings }
    output_schema: { type: object, properties: { findings: {type: array} } }

  # 3. Draft — re-run under the refine loop until a judge is satisfied
  draft:
    agent: brief-writer
    depends_on: [merge]
    inputs: { findings: $merge.findings }

  # 4. Human sign-off before anything leaves the building
  approve:
    kind: gate
    depends_on: [draft]
    payload_from: { draft: $draft.brief }
    gate_timeout_seconds: 86400          # park up to a day, persisted

  # 5. Deliver — deterministic POST, bounded, artifact hash-verified
  publish:
    kind: script
    script_id: external-job-relay
    depends_on: [approve]
    inputs: { body: $draft.brief }
```

```
    timeout_s: 600
    artifact_outputs:
      - name: brief.pdf
        content_type: application/pdf
        min_bytes: 2048
        source_path: out/brief-$run_id.pdf

loops:
  refine:
    members: [draft]
    max_iterations: 3
    convergence_kind: judge
    convergence: "brief scores >= 8/10 on sourcing and clarity"

goal:
  description: "An approved, sourced market brief delivered as a PDF."
output: publish
```

FIGURE 3 The market-brief workflow — fan-out, merge, refine loop, gate, deliver



What each construct buys, and why it isn't free on the raw SDK:

- **Fan-out** — `competitors`, `pricing`, `demand` share no `depends_on`, so the platform runs them concurrently, leases each independently, and checkpoints each. On the SDK you'd manage the concurrency, partial-failure, and per-branch retry yourself.
- `python` **merge** — deterministic dedup at zero token cost, with a typed output the next node can pull fields from.
- **Refine loop (`judge`)** — the draft is re-run up to three times until a judge scores it ≥ 8/10. Cost is bounded by the three iterations — no runaway.
- **Gate** — the run parks for up to a day awaiting a human; the pause is persisted, so a restart mid-wait loses nothing.
- `script` **deliver + artifact** — a deterministic POST bounded by `timeout_s`, and the PDF is content-type/size/SHA-256 verified before the run is called complete.

What the platform guaranteed for that run. Kill the box right after `merge` and restart: recovery skips the three research nodes and the merge, and resumes at `draft` (see §11). The whole run executed under one budget; the loop could not overspend; the gate could not be bypassed; every node emitted an audit

span and a spend-ledger line. On the raw SDK, each of those guarantees is code you write, test, and keep working.

Cost and latency (rough estimate). Order-of-magnitude figures for Sonnet with light web research — the platform reports the real `cost_usd` and `duration_ms` per node (§13), so you never have to trust the estimate:

Node(s)	Kind	Est. cost	Est. time
`competitors` / `pricing` / `demand`	task × 3, parallel	~$0.15–0.50 each	~30–90 s (concurrent)
`merge`	python	$0	< 1 s
`draft` (× 1–3 via refine loop)	task + judge	~$0.12–0.50 per pass	~20–60 s per pass
`approve`	gate	$0	human — excluded
`publish`	script	$0	seconds–minutes
Whole run		**~$0.60–3.00**	**~2–5 min compute**

Two things worth calling out. First, the three research nodes run concurrently, so wall-clock tracks the **slowest** one, not their sum — the fan-out is nearly free in time. Second, elapsed time and compute time are different: this run is a few minutes of compute, but the human `gate` can hold it for up to a day (`gate_timeout_seconds: 86400`) at **zero token cost** while it waits for a sign-off. The `python`, `gate`, and `script` nodes spend nothing — only the agent `task` nodes and the judge cost money.

10.1 When the shape isn't known: orchestration

A DAG assumes you know the steps up front. Sometimes you don't — "handle whatever this inbound request needs," or "research this until you can answer, however many steps that takes." Two options, both running on the same durable substrate:

- **Dynamic orchestration.** An agent sets `orchestrate: true`; a planner reads the goal and emits a plan, a deterministic compiler lowers that plan into the same workflow-spec format shown above, and the same executor runs it. You get the durability, cost, and audit guarantees of a static DAG on a graph that was generated at runtime.
- **Supervisor loop.** A supervisor agent plans, dispatches its member sub-agents (which communicate through the loop transcript), judges their results, and loops until a convergence predicate holds — the debate / critique / delegate pattern, but bounded.

Static vs. dynamic — how to choose. Prefer the static DAG whenever the steps are known: it is cheaper, predictable, trivially auditable, and easy to reason about. Reach for orchestration only when the plan genuinely depends on what the agents discover — dynamic flexibility costs tokens and predictability, so buy it deliberately.

Open-ended does not mean unbounded. The guardrails that make the static workflow safe apply unchanged to orchestration:

- Agent-to-agent call depth is capped (`AF_CAPABILITY_INVOKE_DEPTH`, default 3) and circular call-chains are detected — an orchestrator cannot recurse forever.
- Every loop is bounded by `max_iterations` (and each member's per-node `timeout_s`).

- A per-invocation cost ceiling (`AF_PROVIDER_COST_CEILING_USD`) plus the run's wire-enforced budget (§13) cap spend no matter how the plan unfolds.

The progression, then, is one ladder: a single agent when one call will do; a workflow when the steps are known and fixed; orchestration when the plan itself must be discovered — and the platform's durability, cost, and audit guarantees hold at every rung without you re-earning them.

Part III — Why it holds up in production

11. Durability and scale

This is the single biggest gap between a runtime and an OS. On the raw SDK, if the process dies during a tool call or an LLM turn, the work is gone — there is no checkpoint and no resume.

AgentForge runs on a **lease-based resumable runtime** over Postgres:

- Runs and nodes are checkpointed to Postgres (`workflow_runs`, `workflow_nodes`); recovery resumes from the last completed node. (A checkpoint/snapshot model, not Temporal-style deterministic replay — deterministic replay is ill-defined when turns are non-deterministic LLM calls.)
- A run is owned via `owner_id` + `lease_expires_at` + an `owner_generation` fencing token. Workers claim with `SELECT … FOR UPDATE SKIP LOCKED` over pending-or-lease-expired runs and heartbeat every `lease_ttl/3`. Expired leases are reclaimed and resumed, not failed — this is what makes multiple replicas safe and zero-downtime deploys possible.
- **No stuck runs.** A crashed or orphaned run keeps an expired lease and is reclaimed and resumed from its last completed node by any free worker — crashed work is never silently failed. A poison-pill reaper parks a run only after it has been reclaimed too many times (a genuine crash loop), and it is lease-aware, so it never touches a run a live sibling replica still owns. A hard `SIGKILL` loses at most one in-flight node.

Building this yourself means a lease schema, a claim loop, heartbeats, fencing, a drain protocol, and idempotency keys for side effects — and getting it subtly wrong means either double-execution or silently stuck work.

12. Governance and trust

- **The Agent Gateway pattern (verdict envelope).** Server-enforced decisions return a machine-readable verdict — `{ok, checks: [{code, severity, locus, reason}]}` on the diagnose path — rather than prose. A hard rejection from the conformance gates (`run_gates_on_config`) is an HTTP 422 RFC-9457 `problem+json` body (`{type, title, status, detail, gate, failures:[{locus, reason}]}`) a client can route on, instead of re-deriving the rules client-side.

- **Multi-tenant isolation.** Every data-bearing table carries `project_id` / `portfolio_id`, and queries are tenant-scoped at the application layer; NLT Memory additionally enforces private reads with Postgres row-level security. Per-tenant schemas were deliberately deferred — row-level scoping is sufficient at the operator-controlled topologies AgentForge targets, and schema-per-tenant adds migration cost without demand.
- **Audit and provenance.** OTel GenAI semantic-convention spans carry the tenant as an `agent.project` attribute; an immutable per-tenant event log records what happened; agent SBOMs carry signed attestations (`signature` / `publisher` / `publisher_key_id` / `sbom_uri`). The SDK emits usage numbers; the audit trail is yours to build and retain.

13. Cost control you can trust

The failure mode this closes is specific and real: a model emits a plausible cost number without doing the work, or a deep-search loop runs away, and you find out on the vendor dashboard a month later. Self-reported token math cannot catch that, because the thing reporting the cost is the thing you don't trust.

A **per-tenant cost egress gateway** observes spend at a boundary the agent does not control:

- A self-hosted LiteLLM proxy is the egress chokepoint and holds the provider keys; AgentForge holds only the proxy URL and master key.
- Each run gets a virtual key with a hard `max_budget`. The proxy refuses over-budget calls on the wire (returns a 4xx) — the cap is enforced where the money actually leaves, not in volatile per-process memory.
- Spend is read back from the proxy post-run and persisted to a ledger, broken out by source (LLM, provider, image, other). Wire-observed spend reconciles against — and is preferred over — self-report.

Wire-observed enforcement applies to the platform-API lane (the OAuth lane is type-locked and can't carry a virtual key). Building this yourself is a proxy sidecar, virtual-key minting, wire-level budget enforcement, and a reconciliation ledger.

14. Quality as a pipeline

Agents drift; a platform treats quality as a gated pipeline rather than a vibe:

- **Golden-case promotion gates.** An agent declares `golden_cases` (prompt + assertions like `tool_sequence`, `output_schema_valid`, `guardrails_clean`). The suite runs as a smoke gate at staging→promote and hard-blocks (HTTP 422) on failure. The agent sees only the prompt, never the assertions.
- **Held-out rolling quality score** (`GET /agents/{name}/quality`) surfaces drift over a rolling window; self-reported confidence is recorded as a diagnostic only, never fed into the score.
- **Eval-gated self-improvement.** With `auto_promote_on_pass`, a candidate is promoted only when it beats the baseline with zero regressions.
- **The upgrade / proposal pipeline** keeps every agent on the current schema with safe defaults and human-or-auto-approved verdict trails.

Part IV — Deciding

15. Decision framework

Be honest with yourself. Reach for AgentForge when you can say **yes** to several of these; stay on the SDK if you're mostly saying **no**.

Signals you should adopt AgentForge:

- More than one tenant, customer, or trust boundary shares the deployment.
- Runs are long-lived and must survive restarts, deploys, and crashes.
- Someone is accountable for per-tenant or per-run **spend limits**.
- You need an **audit trail** for compliance (SOC 2, EU AI Act) or forensics.
- Tool access needs **policy** — RBAC/ABAC, approval gates — not just allow/deny.
- You operate a **fleet** of agents that needs versioning, promotion, and drift detection.
- Agents need a **shared knowledge layer**, not per-agent scratchpads.

Signals you should stay on the SDK (for now):

- One agent, or a few, in one process for one team.
- Runs are short; "just retry it" is an acceptable failure mode.
- One trust boundary, one API key, one budget you eyeball.
- No compliance-grade audit requirement.
- You value keeping the whole system in one person's head over operational guarantees.

There is no shame in the second column. AgentForge's own `permissive` mode exists precisely so the on-ramp from column two is gradual, not a rewrite.

16. Migration path

Because an AgentForge `task` node is a Claude runtime invocation, wrapping an existing SDK agent is mostly declarative, not a rewrite:

1. Move the agent's system prompt, tool list, and completion criteria into an `AGENTS.md` package; keep any genuinely bespoke tool logic behind an MCP server or a `python` node.
2. `af-publish` it and validate with `af-smoke-test`; the gate chain will tell you exactly what's non-conforming.
3. Run in `permissive` mode first — behaves like your SDK app plus a durable run record, an audit span, and a spend ledger entry.
4. Ramp the posture: `balanced` when a second team or tenant shows up, `strict` when you go to regulated production. Same image, config only.

The migration is incremental precisely because the platform is configuration-driven: you adopt durability, then tenancy, then policy, then cost governance as each becomes someone's problem —

never all at once.

Appendix A — Operating modes and capability flags

`AF_MODE` selects a profile (`permissive` / `balanced` / `strict`); individual `AF_CAPABILITY_*` env vars override single flags:

- `AF_CAPABILITY_IDENTITY_ENFORCEMENT` — `permissive` | `strict`
- `AF_CAPABILITY_POLICY_ENFORCEMENT` — `permissive` | `strict`
- `AF_CAPABILITY_SUPPLY_CHAIN_VERIFICATION` — `off` | `warn` | `strict`
- `AF_CAPABILITY_INVOKE_DEPTH` — `1`–`5` (default `3`; A2A recursion bound)

Related but outside the capability-profile system: `AF_ENV` (dev/prod marking + approval posture), `AF_TRACE_SINK` (audit sink), `AF_AGENT_RUNTIME`, `AF_ROLE` (`api` | `worker` | `all` — same image, different entry point). A separate family of governance-gate posture flags (`AF_COHESION_MODE`, `AF_INTERFACE_GATE_MODE`, `AF_EVAL_REQUIRED_MODE`; each `warn` | `block`) tunes the non-capability gates independently of `AF_MODE`.

Appendix B — Capability matrix

Capability	SDK built-in	You build on SDK	AgentForge provides
Agent loop, tool use, MCP client	✓	—	✓ (uses the SDK runtime)
Subagents, hooks, permission modes	✓	—	✓ + policy layer
Prompt caching, context compaction	✓	—	✓
Durable / resumable execution	⊗	⊖ hard	✓ lease-based runtime
Multi-tenant isolation + RLS	⊗	⊖	✓ row-level scoping
Centralized RBAC/ABAC policy	⊗	⊖	✓ Policy Engine
Per-run cost budget on the wire	⊗	⊖ hard	✓ egress gateway
OTel GenAI audit + per-tenant log	⊗	⊖	✓
Agent registry, versioning, upgrade	⊗	⊖	✓ proposal/upgrade pipeline
Auto-routing prompt → best agent	⊗	⊖ hard	✓ hybrid matcher
Auto-authoring / repair of agents	⊗	⊖	✓ proposal engine
Auto-decide single vs. multi-agent	⊗	⊖	✓ dispatch heuristics

Capability	SDK built-in	You build on SDK	AgentForge provides
Hot-reload agents (no redeploy)	⊗	⊖	✓ watcher + self-register
Golden-case + rolling quality gates	⊗	⊖	✓ Eval Pipeline
Shared RAG / knowledge memory	⊖ file KV	⊖	✓ NLT Memory
One-image, config-driven deploy modes	⊗	⊖	✓

Legend: a filled check = provided; an amber dash = possible but you build and own it; a hollow ring = not available.

Appendix C — Key design decisions

The mechanisms in this paper rest on a small set of deliberate design choices. Each is stated here in plain terms, independent of any internal document:

- **One image, configuration-driven modes.** A single deployable artifact with three operating postures (`permissive` / `balanced` / `strict`) rather than separate builds per environment — the platform is topology-agnostic.
- **Postgres snapshot durability, not deterministic replay.** Runs resume from the last completed node. Replay is intentionally avoided because LLM turns are non-deterministic and cannot be faithfully re-executed.
- **Lease-based ownership for multi-replica safety.** Runs are owned by a lease with a fencing token; expired leases are reclaimed and resumed, enabling zero-downtime deploys without double-execution.
- **Row-level tenancy, not schema-per-tenant.** Isolation is `project_id` / `portfolio_id` scoping plus row-level security — sufficient for the operator-controlled topologies AgentForge targets, without per-tenant schema migration cost.
- **CLI grants over MCP servers by default.** Tool transport is chosen for token cost: scoped CLI grants for mature binaries, MCP only for auth-heavy or stateful integrations.
- **Wire-observed cost, not self-report.** Spend is enforced at an egress proxy the agent cannot bypass, with a hard per-run budget.
- **Plan → compile → execute for dynamic work.** Orchestration lowers a runtime-generated plan into the same workflow format a human would hand-write, so dynamic graphs inherit the same durability and audit guarantees.

Prepared from the AgentForge codebase. Claims are grounded in implemented mechanisms; where a capability is opt-in or configuration-gated, the paper says so.