NLTlabs
NEW LOGIC TECH

Leveraging AgentForge

A practitioner's guide: extending the platform, using its tool surface, building on it, and operating a fleet

A technical white paper for engineers and architects — companion to "Building Agent Systems on AgentForge: How, and Why Not Just the SDK."

Version 1.2 (Parts I-V) — 2026-07-11

0. Executive summary

The companion paper argued why to adopt AgentForge and how to stand up your first agent and workflow. This one is for teams that have already adopted it and want to push further: extend the platform with your own compute and tools, use the built-in tool surface well, and build real capabilities (search, image, and 3D agents) on top of it.

One honest frame sits under everything here: **AgentForge is a thin orchestrator.** It routes prompts to agents, runs workflow DAGs, enforces policy/cost/audit, and hands artifacts between nodes. It is not where heavy or custom compute belongs. That compute lives at the **edges** — a host you own (a remote runner), a tool server you write (MCP), a job you poll (a script node), or simply your own repo called over HTTP. Internalize that and the extension model is simple; fight it and you'll look for a "container per agent" that was deliberately removed (§2).

The paper runs in five parts: **Part I: Extending AgentForge** (the honest map of adding your own compute and tools), **Part II: The tool surface you get** (search, memory, artifacts), **Part III: What you can build** (search, 2D, and 3D showcases), **Part IV: Automating and operating a fleet** (schedules, goals, multi-project, A2A, cost governance in practice), and **Part V: Integrating** (publish/invoke, credentials, personas, skills).

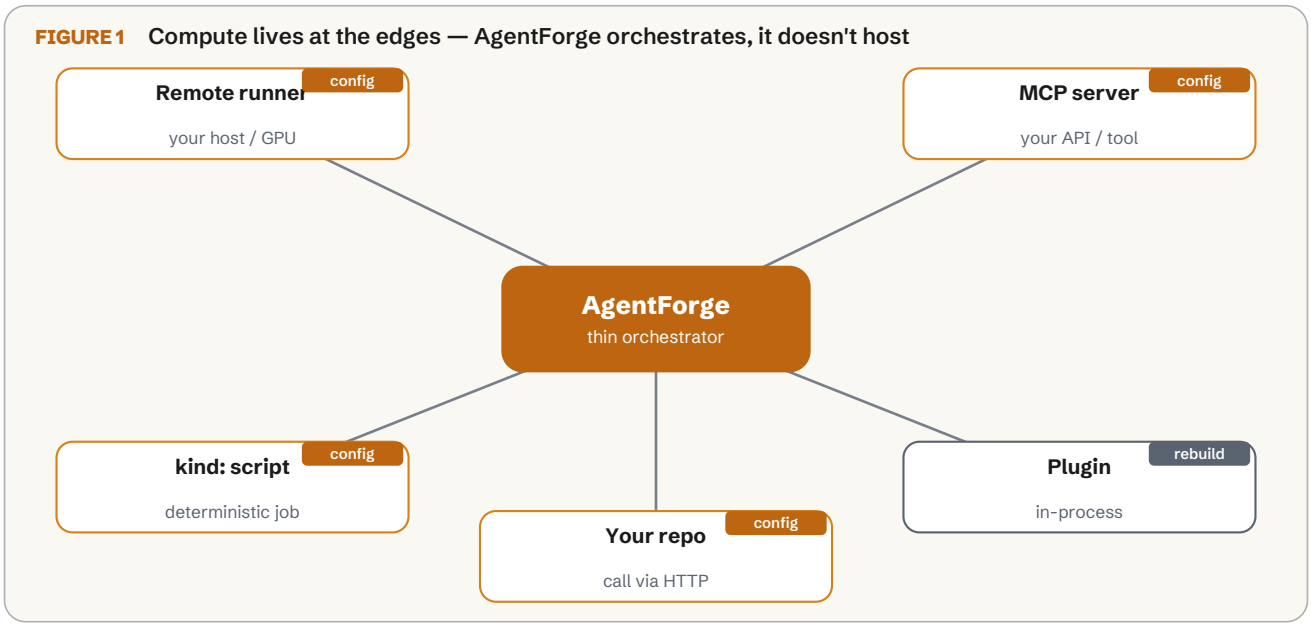
Naming note: the memory/RAG service is **NLT Memory** throughout. Where a capability is opt-in or config-gated, the paper says so.

Part I — Extending AgentForge (the honest map)

1. Mental model: compute lives at the edges

The execution unit is a **CliRunner** spawning a `claude -p` subprocess on the AgentForge host. Around that, the platform provides orchestration, policy, cost, audit, memory access, and artifact hand-off. What it does not provide is a sandbox for arbitrary consumer compute. So the first question when extending is always **"where does this run?"** — and the answer is rarely "inside AgentForge."

You want to…	It runs…	Mechanism
Run a model/tool on hardware you own (GPU, USB, local shares)	on **your host**	Remote runner (§3)
Call your own API, DB, or tool	in **your service**	MCP server (§4) or an `http` node
Poll a long external job to completion	in a **deterministic subprocess** on the AF host	`kind: script` (§5)
Add a bespoke in-process route or runner	**inside AF** (rebuild required)	Plugin (§6)
Run heavy deterministic compute	in **your repo**	Don't put it in AF — call it (§7)



The platform's job is to orchestrate these, not to host them. Keep that line and the rest of Part I is a menu, not a maze.

2. What was retired, and why it matters

AgentForge once ran each agent in its own OCI container over the host Docker socket. **That path is gone.** The per-agent `ContainerRunner`, the docker-socket substrate, and the `runtime_deps.image` field were removed; declaring `image:` now **errors at agent load** with a message telling you to run deterministic compute in the consumer repo or dispatch to a remote runner instead.

Why it matters to you: **do not design around "each agent gets its own container."** It was removed for two reasons — the host-Docker-socket trust surface was a liability, and it duplicated, worse, what remote runners and consumer repos already do. The replacement isn't one feature; it's the honest menu in §1. The rest of Part I is that menu in detail.

3. Bring your own host — remote runners

This is the closest thing to "your own container." You run an AgentForge runner on a machine you control — with your GPU, your Docker, your local resources — and it registers back to the central instance. Agents that need that hardware declare an affinity, and the scheduler dispatches them there.

Operator setup (central AF): set `AF_RUNNER_SHARED_SECRET` (empty by default, which disables registration — closed by default). Runners then use three endpoints, all requiring an `X-Runner-Secret` header:

```
POST /runner/register    { "id": "workshop-1",
                           "url": "https://workshop-1.runners.internal",
                           "labels": {"node": "workshop", "gpu": "true"} }
POST /runner/heartbeat   { "id": "workshop-1" }      # keepalive; deadman TTL ~60s
# central AF POSTs work to the runner's own /runner/dispatch
```

Agent side (`AGENTS.md`): declare the labels a runner must advertise:

```
runtime-deps:
  host_affinity: { gpu: "true", node: workshop }
```

The selector is exact-match-all-labels; among matching, available runners the lowest `id` wins. No match → the task fails fast with a selection error, before dispatch. All of this is **config-only** — the agent change is hot-reloaded; no image rebuild.

Honest limits (read before you rely on it). AgentForge speaks plain HTTP to runners — **it does not terminate TLS**. You must put TLS in front (reverse proxy, WireGuard, Tailscale). The shared secret is all-or-nothing: no per-runner keys, no rotation beyond restarting with a new value, and a leaked secret is a full compromise. There's no mTLS, signed payloads, replay protection, or registration rate-limiting. The trust model assumes an **internal-LAN topology under one operator** — public exposure voids it. And it is not an autoscaler: you bring runners up and down, AF just dispatches to what's registered.

4. Bring your own tools — MCP servers

For your own APIs, databases, or tools, write an MCP server (any language, over stdio or HTTP), add it to the registry, and let agents declare it by name:

```
// mcp_servers.json  (or point AF_MCP_REGISTRY_PATH at your own)
{
  "myservice": {
    // stdio is the default transport — omit "type" for it
    "command": "python", "args": ["-m", "myservice.mcp"],
    "env": { "MYSERVICE_API_KEY": "${AF_MYSERVICE_KEY}" }
  }
}
```

```
# in an agent's AGENTS.md
mcp-servers: [myservice]
```

Env-var placeholders (any `${VAR}`) are substituted from operator config when the agent is invoked; the model never sees raw keys. Adding a server needs **no AF code rebuild** — agents reference it by name and AF wires it on demand.

When MCP vs. a CLI grant. This is a cost decision, not a style one (it's the same rule as the companion paper's §8). Every declared MCP server injects all its tool schemas into context on every run. So: for a mature binary the model already knows (`gh`, `git`, `aws`, `kubectl`), prefer a scoped CLI grant (`Bash(gh:*)` via `tool-targets`) — it injects nothing. Reserve MCP for integrations that are **auth-heavy, credential-bridged, stateful, or have no first-class CLI** (Linear, Google, NLT Memory, your internal service). Declare only the servers an agent actually uses; each one is paid for on every run.

5. Bring your own deterministic job — `kind: script`

An LLM driving a "POST the job, then poll until it's done" loop through Bash is unreliable — it bails early and fabricates a result. For that shape, use a `script` **node**: a registered subprocess, bounded by the node's `timeout_s` (not the CLI tool's ~10-minute ceiling), with JSON on stdin and stdout, where exit 0 means the node completed and its stdout is the node output. Zero LLM cost.

The platform ships a generic **external-job relay** you can use immediately — it POSTs an HMAC-signed body and polls with fast→slow backoff until a terminal state, with no consumer-specific identifiers baked in:

```
nodes:
  relay:
    kind: script
    script_id: external-job-relay
    timeout_s: 600
    inputs:
      base_url: https://builder.internal
      post_path: /build
      poll_path_template: /build/{job_id}
      body: { slug: $slug, correlation_id: $run_id }
      hmac_secret_env: MY_HMAC_SECRET     # env var name, never an inline secret
      terminal_success: [succeeded]
      terminal_failure: [failed, error]
```

To register **your own** script, call `register_script(script_id, path, …)` from a plugin's `register(app)` function. Two things to know: workflow YAML never names a filesystem path (only a pre-registered `script_id`, so path traversal is impossible), and because registration happens in code, adding a new script is a **plugin change** (§6) — it needs an image rebuild, unlike the shipped relay, which is available to any workflow out of the box.

6. In-process plugins (and when not to)

Some extensions genuinely must run inside the AF process — a bespoke runner, an internal HTTP route, a resource/event provider, or a `register_script` call. Those are **plugins**: a Python package with an `agentforge.plugins` entry point and a `plugin.json` manifest whose `register(app)` mounts routes and registers providers at startup.

```
[project.entry-points."agentforge.plugins"]
my-plugin = "mypackage.plugin:register"
```

Installing one is `uv sync` → **rebuild the image** → **restart the container**. There is no hot-deploy for plugin code, and this is deliberate: Python cannot safely reload live code (stale cached modules, no shutdown hooks for background tasks, arbitrary-code-execution risk), which is why every major agent framework registers extensions at startup too. The dev-only `POST /api/plugins/register` endpoint only activates a package already installed in the container's venv — it is not a way to introduce new code.

Guidance: reach for a plugin last. If a remote runner, an MCP server, or a `kind: script` relay can do the job — and usually one can — prefer it, because those are config-only and don't touch the platform's release cycle.

7. Decision guide: which extension point?

"I want to…"	Use	Rebuild?
run my model on a GPU/host I own	**remote runner** (§3)	no (config)
call my own API / tool / DB	**MCP server** (§4) or `http` node	no
poll a long external job to completion	`kind: script` + relay (§5)	no (shipped relay)
register my own deterministic script	`register_script` in a **plugin**	yes
add an internal route or bespoke runner	**plugin** (§6)	yes

"I want to..."	Use	Rebuild?
run heavy deterministic compute	**your repo** — call it via MCP/`http`	no

The default bias: **keep compute at the edge and let AgentForge orchestrate.** The platform is a thin orchestrator plus prompt-and-tool-call routing; the more of your logic that lives in your own services and repos, the less of it is coupled to AF's release cycle.

Part II — The tool surface you get

8. Web and search, native-first

An agent with an open tool grant runs with native `WebSearch` and `WebFetch` — no MCP server required — their cost folding into the invocation's normal token cost. (An agent that declares an explicit `allowed-tools` list must include them; the coherence checker flags the mismatch if it doesn't.) That is the intended default for most web work.

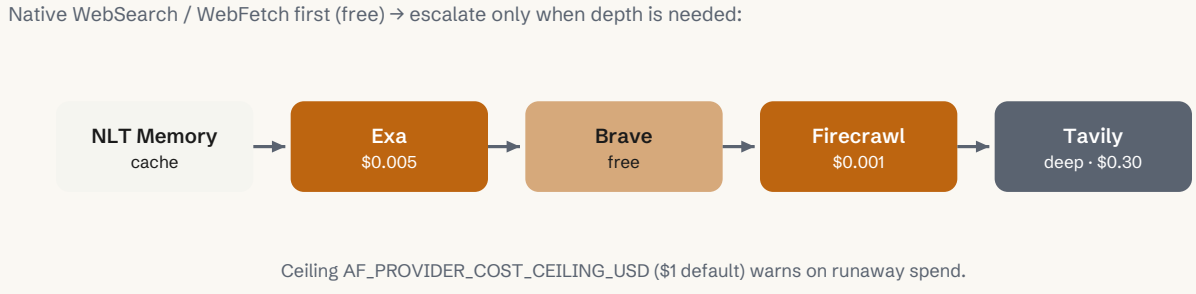
Workflow `http` nodes (and the in-process fetch substrate behind them) are hardened: a **block-list runs before the allowlist** (RFC1918, cloud metadata IPs, loopback — no crafted allowlist can reach them), the allowlist itself is **owned by agent/workflow config** (`runtime_deps.http_allowlist`) and can never be widened by the model, responses are size-capped, secret headers resolve from the agent's `secrets:` (never inline), and egress is rate-limited per workflow (`AF_HTTP_EGRESS_RATE_PER_MIN` / `AF_HTTP_EGRESS_BURST`). The design principle: the LLM can use egress but never expand it.

9. Paid search, when you need depth

Native-first doesn't mean native-only. When you need structured results, deep research, or bulk crawl, declare a paid provider in `mcp-servers` — Exa, Firecrawl, Tavily, or Brave — and reach for them in a deliberate order:

NLT Memory cache → Exa (fast) → Brave (free) → Firecrawl (crawl/scrape) → Tavily (deep research).

FIGURE 2 **The smart-research stack — cheapest source first, cost-capped**



Costs are real and uneven: roughly Exa $0.005/call, Firecrawl $0.001/call, Tavily $0.008/call, with **deep-research modes** priced far higher (Tavily's deep research and Exa's research modes carry conservative floors of ~$0.10–0.30 per call, because one careless deep call has burned tens of dollars). The platform guards this: `AF_PROVIDER_COST_CEILING_USD` (default **$1**) emits a structured warning at finalize when paid-search spend crosses it — catching a runaway search agent at the run, not on next month's vendor invoice — and cheap-triage agents are steered away from deep-research modes (a structured warning, not a hard block). Provider keys live in operator config (`AF_EXA_KEY`, `AF_FIRECRAWL_KEY`, `AF_TAVILY_KEY`, `AF_BRAVE_API_KEY`); the model never supplies them.

10. NLT Memory as a cache and knowledge layer

NLT Memory is a hybrid-RAG service AgentForge consumes over HTTP, and every agent touches it without writing memory code: the orchestrator **recalls relevant memories once per invoke** and injects them into the system prompt, governed by the agent's `memory-profile` (`full` = recall + save + reinforce, `readonly`, or `none` to skip the round-trip). Writes are scoped by `share_scope`:

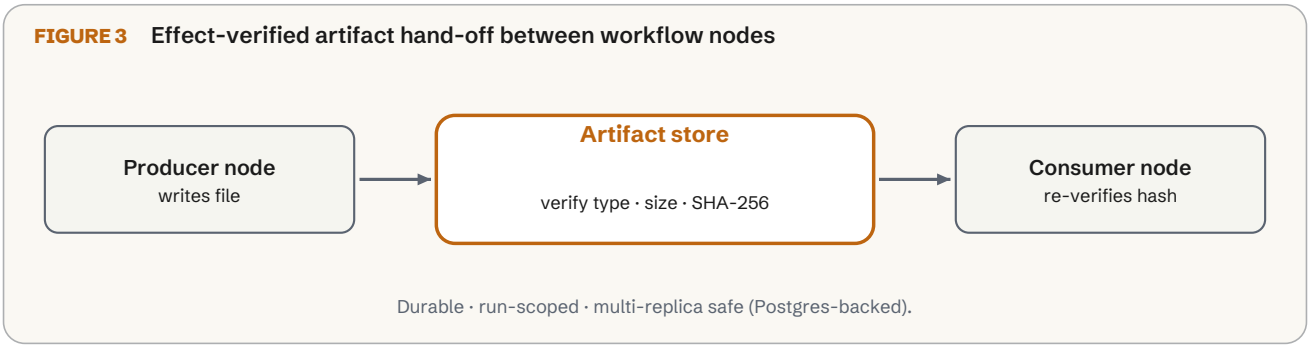
Scope	Visible to
`private` (default)	only the writing agent
`domain`	all agents on the same memory profile
`hive`	all agents in the configured hive
`group:<name>`	named group members

The pattern that pays off is **cache-through**: an agent reads an expensive web page, saves it at `hive` scope, and every downstream agent auto-recalls it on future invokes — so the fetch is paid once. Reinforcement is **citation-aware**: only memories the agent actually used in its output get a confidence boost, so useful knowledge rises and noise decays. Agents that want live memory tools (explicit recall/search/consolidate) additionally declare `mcp-servers: [nlt-memory]`; most agents don't need to, because recall is automatic.

11. The run-scoped artifact store

Workflows hand binary artifacts — PDFs, images, 3D models — between nodes through a **run-scoped, Postgres-backed store** that is durable and multi-replica safe. A node declares what it produces:

```
artifact_outputs:
  - name: brief.pdf
    content_type: application/pdf
    min_bytes: 2048
    source_path: out/brief-$run_id.pdf     # relative to working_directory
```



The platform captures the file from `source_path`, stores it, and **verifies it at node close** — content-type, minimum size, and a SHA-256 re-hash. This is effect-verified, not self-reported: a node that claims success but produced no (or a stale) file **fails** with a structured diff, which is how "shipped the old render as if it were new" bugs are eliminated. Downstream nodes fetch by `run_id` + `node_id` + `name` and re-verify the hash; because Postgres is the transport, producer and consumer can run in the same container or on different replicas. Size is bounded by `AF_ARTIFACT_MAX_BYTES` (default 25 MB per artifact) and a per-run total (default 100 MB); writes are rejected once a run is terminal.

Part III — What you can build

12. Framing: platform primitives vs. the agent layer

Be precise about the boundary, because it's easy to overclaim. AgentForge does **not** ship "2D rendering" or "3D generation" as platform features. What it ships are **generic primitives** that any external-API agent leans on:

- **Credential preflight** — an agent declares `credentials: [{key, scope, required}]`; the platform resolves them before spawn and **fails the node with a** `missing_credential` **envelope** if a required one is absent, so you never burn a run to discover a missing key.
- **Artifact verification** (§11) — declare `artifact_outputs`; the platform proves the file exists, is the right type/size, and hashes correctly.
- `source_path` **capture** — the agent just writes a file where it likes; the platform captures it. No storage-API code in the agent.
- **Spend ledger** — per-invocation cost broken out by source (`llm_cost_usd`, `provider_cost_usd`, `meshy_cost_usd`, `flux_cost_usd`, `other_cost_usd`), so non-token spend on image/3D APIs is tracked, not invisible. The same ledger also records context telemetry — `mcp_schema_tokens`, `cache_hits` / `cache_saved_usd`, `brain_injection_tokens` — so prompt-context cost is visible too.

None of those are image- or 3D-specific. The **agent** supplies the rest: the model/API call, the prompt, and the output contract. The showcases below are therefore consumer agents — things you build — resting on platform primitives.

13. Three showcases

All three run the same way: as a `task` node whose agent is a `claude -p` subprocess on the AF host, calling an external REST API over `httpx`. **There is no separate service or container per showcase** — the agent is the engine, and the platform captures its artifacts and cost.

A. A search / research agent. Pure tool-surface composition: native `WebSearch`/`WebFetch` first, a paid provider (Exa/Firecrawl) only when depth is needed, and NLT Memory cache-through so repeat topics are nearly free. Output is schema-constrained (`{findings: [...]}`) so downstream nodes can pull typed fields. Cost profile: mostly tokens, plus optional paid search bounded by the provider ceiling (§9).

B. A 2D render agent. A Claude agent that calls an image API (e.g. Gemini image generation) via `httpx`, declares `credentials: [GEMINI_API_KEY]` (so a missing key blocks at preflight), writes its PNGs, and declares them in `artifact_outputs`. The agent emits a `spend_usd`; the platform ledger records the image cost under `other_cost_usd`. Everything platform-specific is a primitive; the "2D" is your agent.

C. A 3D image-to-3D agent. Calls a 3D API (e.g. Meshy multi-image-to-3D) with `credentials: [MESHY_API_KEY]` at project scope, required — so preflight blocks a keyless run. It makes **one billable call** with watchdog-safe settings, produces a `.glb` plus a preview, hands them off as verified artifacts, and can run a small vision **fidelity judge** (compare the model thumbnail to the source render, threshold-gated). It's wired **best-effort**: a 3D failure doesn't fail the larger workflow. The ledger tracks the call under `meshy_cost_usd` (a conservative floor, since the vendor doesn't expose per-call cost on the wire).

The lesson across all three: you write a small agent that calls an API; the platform handles identity (credential preflight), hand-off (verified artifacts), and accountability (the spend ledger).

14. Build-your-own template

Any external-API agent is the same shape. The `AGENTS.md`:

```
---
name: my-image-agent
schema_version: "2.1"
model: opus
brain_rationale: "What this agent is for + what prior runs taught us."  # required at 2.1
credentials:
  - { key: EXTERNAL_API_KEY, scope: global, required: true }
secrets: [EXTERNAL_API_KEY]
artifact_outputs:
  - { name: out.png, content_type: image/png, min_bytes: 1024,
      source_path: output/out.png }
---
## Inputs
You receive `subject` and `style`.
## Output
Return JSON: { "status": "complete|blocked|skipped", "confidence": 0.0-1.0,
               "image_file": "output/out.png", "spend_usd": 0.0,
               "summary": "..." }
```

The entrypoint (Python, or anything that reads stdin and writes stdout):

```
import sys, json, os, httpx
from pathlib import Path

inputs = json.loads(sys.stdin.read())
```

```
api_key = os.environ["EXTERNAL_API_KEY"]          # resolved by AF vault → env

resp = httpx.post("https://api.example.com/generate",
                  headers={"Authorization": f"Bearer {api_key}"},
                  json={"subject": inputs["subject"], "style": inputs["style"]})
out = Path("output/out.png"); out.parent.mkdir(parents=True, exist_ok=True)
out.write_bytes(resp.content)

print(json.dumps({"status": "complete", "confidence": 0.95,
                  "image_file": str(out), "spend_usd": 0.05,
                  "summary": "Generated via EXTERNAL_API."}))
```

And the workflow wiring:

```
nodes:
  my_image:
    agent: my-image-agent
    inputs: { subject: $subject, style: $style }
    artifact_outputs:
      - { name: out.png, content_type: image/png, min_bytes: 1024 }
```

Checklist: `schema_version: "2.1"`; add `brain_rationale` (required at 2.1); declare `credentials` (with `required`) and `secrets`; call the external API over `httpx`/`requests` (not an AF-owned MCP); write the file and declare it in `artifact_outputs`; emit `spend_usd`; set a `max_budget_usd` on the run. That's the whole contract — the platform does the rest.

Prove it with a golden case. Don't ship an agent on vibes — attach a `golden_cases` block and the platform gates promotion on it:

```
golden_cases:
  - id: renders-four-angles
    prompt: "Render the home-espresso machine."
    assertions:
      - { kind: tool_sequence, expected_tools: ["Bash"], match: subsequence }
      - { kind: output_schema_valid }
      - { kind: 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. Once live, a held-out rolling score (`GET /agents/{name}/quality`) surfaces drift, and `auto_promote_on_pass: true` promotes a candidate only when it beats the baseline with zero regressions — so "it works" is a gate, not an opinion.

Part IV — Automating and operating a fleet

15. Scheduled and event-triggered workflows

Workflows don't have to be kicked off by hand. A `triggers:` block fires them on a clock or on an event.

On a clock. Declare exactly one of `cron:` or `interval_seconds:`:

```
triggers:
  - type: schedule
```

```
    interval_seconds: 900        # or:  cron: "0 */6 * * *"
    inputs: { topic: "weekly market scan" }
```

A scheduler fires due workflows on time; the agent-job dispatch behind it is governed by a priority queue, a concurrency cap, bounded retries, and a backpressure limit that keeps a flood of due jobs from overrunning the box. One floor to know: a workflow containing `http` nodes must schedule no tighter than `interval_seconds: 60` (rejected at load otherwise) — a guard against hammering an upstream API.

On an event. Repo events can trigger a workflow: a trigger maps an `event_type` + `action` to a `workflow_name` with an `input_mapping` (JSONPath from the event payload), and the webhook endpoint (`POST /repos/{repo_id}/events/github`) is HMAC-verified. That's how "PR opened → run the review workflow" is wired without polling.

Watchdogs. Long or stuck runs are bounded by two clocks: `AF_IDLE_TIMEOUT_SECONDS` (a no-output watchdog on the subprocess, default 150s) and a separate `AF_HEARTBEAT_TIMEOUT_SECONDS` liveness clock for genuinely long nodes. Combined with async kickoff (`?kickoff=async` → `202` + `sse_url`), you start a scheduled run and watch it over SSE rather than blocking.

16. Goal-first agents and autonomous loops

Two mechanisms turn "do this" into "keep going until it's actually done."

A verifiable goal per invoke. Pass `context.goal` (a sentence, ≤ 4000 chars) and the platform injects it as a `## Goal` section, then — after the agent runs — has a held-out verifier judge whether the goal was met. The response carries `goal_status` (`met` / `unmet` / `not_set`), `goal_turns`, and a short `goal_verifier_reason`. The verifier is **fail-closed**: on a transport hiccup it does not silently pass — the run retries toward the goal until `max_retries` is spent. The `af-goal` skill wraps this.

Loops with convergence. A loop re-runs its member nodes until a predicate holds or `max_iterations` is hit:

```
loops:
  refine:
    members: [draft]
    max_iterations: 5
    convergence_kind: judge        # truthy | equality | judge | expression
    convergence: "draft scores >= 8/10"
```

The four kinds cover the useful patterns: `truthy` (a member emits its own done-flag), `equality` (a field stabilizes across passes), `judge` (a held-out grader owns the stop decision), and `expression` (a safe predicate like `judge.status == "converged" and score >= 0.9`). A separate `convergence_verify` block adds a **ground-truth gate** — `file_exists`, `predicate`, or `judge` — so a loop that claims convergence but leaves no real artifact keeps going. This is the machinery behind debate, critique, and refine-until-good, all bounded.

17. A fleet of projects in one instance

One AgentForge instance hosts many projects. Each has a slug (`^[a-z][a-z0-9-]{1,62}$`), an owner, and a memory profile, and there are two ways to populate it.

HTTP self-registration (recommended for production):

```
POST /projects                                  # create
PUT  /projects/{slug}/agents/{name}             # publish a version (content-hashed)
GET  /projects/{slug}/agents/{name}/versions    # history
POST /projects/{slug}/agents/{name}/activate    # atomic version switch
```

Publishing is idempotent — identical content is a no-op — and versioned, so rollback is just an `activate` of a prior version. Per-project bearer keys (`afp_*`) scope access; `AF_PROJECT_AUTH_REQUIRED=true` enforces them (open by default for local dev).

Filesystem mount (good for local multi-project): point `AF_EXTERNAL_AGENTS_ROOT` at `<root>/<slug>/<agent>/AGENTS.md`; a watcher auto-discovers and hot-reloads on change (~300ms debounce). Skills overlay the same way under `AF_EXTERNAL_SKILLS_ROOT`.

Isolation. Projects are row-scoped, not schema-per-tenant: agent keys are namespaced (`{slug}-{name}`), runs are stored as `__project__/<slug>/<name>`, memory is bound to the project's profile, and `GET /projects/{slug}/invocations` is scoped. For **cross-customer** isolation — not just cross-project — the boundary is the container: run separate instances per customer.

18. Agent-to-agent and federation

Agents can call agents. The native paths are `POST /tasks/invoke` (the matcher entry point) and `POST /invoke/{ns}/{group}/{name}` (a direct, namespaced call for agent-to-agent re-entry); the two cross-link via an RFC 8288 `Link` header. For external callers, a single `POST /a2a` endpoint speaks the Google A2A JSON-RPC wire spec, and `GET /.well-known/agent-card.json` publishes one AgentForge card whose `skills[]` export **only safe fields** (`id/name/description/tags`) — never guardrails, prompts, or MCP wiring.

The safety rails are the point. A delegation chain is tracked in a ContextVar with a hard depth cap (`AF_CAPABILITY_INVOKE_DEPTH`, default 3, range 1–5), circular chains are detected, and each hop records a provenance header (`X-AF-Caller-Tool`). So an agent graph can't recurse forever or loop back on itself, and shared knowledge flows through the memory scopes from §10 (`private` / `domain` / `hive` / `group:<name>`) rather than ad-hoc messaging.

19. Cost governance and observability in practice

Three budget layers stack, soft to hard:

- **Per-run budget** (`AF_DEFAULT_MAX_BUDGET_USD`, default $1) caps a single run.
- **Monthly portfolio cap** (`AF_MONTHLY_BUDGET_USD`) warns at 80% and, at 100%, flips the portfolio into **manual mode** — events queue but don't dispatch until an operator resets it. Honest limit: this counter is per-process and in-memory, so it resets on restart; treat it as a guardrail, not an accountant.
- **Per-run wire enforcement** (opt-in): with the LLM gateway configured, each run gets a virtual key with a hard budget and the proxy returns a 4xx **on the wire** once it's crossed — the cap enforced where the money leaves, on the platform-API lane. Plus the paid-search ceiling from §9.

Observability is first-class. OTel GenAI spans export to `OTEL_EXPORTER_OTLP_ENDPOINT`; each node emits `node.start` / `node.close` events with model, status, and token usage (cost rides the OTel span attribute

`af.usage.cost_usd`). You can query the invocation log (`GET /stats/invocations`, or the project-scoped variant), pull a timestamp-interleaved run trace, and retrieve raw node logs (`GET /workflows/runs/{run_id}/nodes/{node_id}/logs?stream=stderr&tail=N`, retained `AF_NODE_LOGS_RETENTION_DAYS`, default 7). A failed node can be re-run in place (`POST …/nodes/{node_id}/retry` with a required `Idempotency-Key`, reusing upstream outputs). And cost survives crashes: an authoritative `cost_usd` when captured, a vendored-pricing `cost_usd_estimate` when the process died mid-node, and a `cost_source` telling you which you're reading.

Debugging a failed run. The platform is built to fail loud, so diagnosis follows the envelope. A node that failed on a missing key returns a `missing_credential` envelope (fix the vault entry, §21); one that failed artifact verification returns a structured diff of declared-vs-stored (the file wasn't produced, or was the wrong type/size). For anything else, pull the node's stderr (`GET /workflows/runs/{run_id}/nodes/{node_id}/logs?stream=stderr`), read the timestamp-interleaved run trace to see where it diverged, then re-run just that node in place (`POST …/nodes/{node_id}/retry` with an `Idempotency-Key`) once the cause is fixed — upstream outputs are reused, so you don't re-pay for the whole run. The `af-troubleshoot` skill wraps this sequence.

Part V — Integrating

20. Consumer repo → AgentForge

The integration kit's `af-*` skills cover the loop end to end: `af-init` scaffolds the project directory and `.mcp.json`, `af-connect` checks health and port, `af-publish` validates then `PUT`s each agent and activates it (and publishes workflows), `af-smoke-test` runs a health + auth + invoke + workflow-dry-run probe, and `af-troubleshoot` is the diagnostic runbook.

Invoking is sync or async. Sync `POST /tasks/invoke` blocks and returns the result. Async `POST /projects/{slug}/tasks/invoke?kickoff=async` returns `202` with a `run_id` and `sse_url`; you then stream `GET /workflows/runs/{run_id}/stream` for `node.start` / `node.complete` / `gate.pending` / `workflow.complete` events instead of polling. Publishing is content-hash idempotent, so re-running `af-publish` on unchanged agents is a no-op.

Upgrading the **platform itself** — a new AgentForge image and the integration kit — is a separate, deliberate step: the `af-upgrade` / `af-kit-upgrade` skills bump the version and migrate agent configs, distinct from the per-agent proposal/upgrade pipeline that keeps individual agents on the current schema.

21. Credentials and secrets

Two custody models, by tenancy.

Per-project vault (multi-tenant path). `POST /secrets` stores an encrypted-at-rest secret at `scope: project:<slug>`, `agent:<id>`, or `global`; the agent declares only the **key name** in its `credentials:` list. Resolution happens **at invocation, not at prompt assembly** — the model never sees the value — and

walks a tier order (vault → Docker `/run/secrets` → host env var). A non-valued audit log (`GET /secrets/audit`) records access. Injection shapes cover `bearer`, `basic`, `header`, `query`, and `env`.

Host-env passthrough (single-tenant path). For a single-operator deployment, an agent's `secrets:` list names host environment variables to forward at spawn (validated names, reserved prefixes blocked). Simpler, but no per-tenant isolation — use the vault when more than one party shares the instance.

22. Personas and skills

Personas shape how an agent reasons about stakeholders without hard-coding it into the prompt. A deployment-level doctrine plus per-run `context.personas` compose into a reserved `## Personas` section (doctrine → personas → goal). Where a persona names an observable success criterion for a non-verbal user, a function gate can verify it against the actual output instead of trusting the agent's self-report.

Skills are reusable capability bundles an agent pulls in by name (`skills: [web-research, markdown-brief]`). At runner construction the platform resolves each skill to its tool grants and a prompt fragment, merges the grants into the agent's `--allowedTools`, and injects the fragment — progressive disclosure, so the agent gains the capability without you re-authoring it. Resolution is per-project isolated (an `X-Project-Id` context scopes it), while built-in and entry-point skills stay globally visible.

Appendix A — Extension decision matrix (rebuild vs. hot-reload)

Extension	Mechanism	Hot-reload?
Agent persona / prompt	`AGENTS.md`	✅ yes
Tool grant / targets	`allowed-tools` / `tool-targets`	✅ yes
Host affinity (remote runner)	`runtime-deps.host_affinity`	✅ yes
Runtime tool dependency	`runtime-deps.tools` (must be on PATH)	⛔ conditional
MCP server (already installed)	`mcp_servers.json` + agent declares	✅ yes
Remote runner	`POST /runner/register` + heartbeat	✅ yes (operator)
External agents (mounted)	`AF_EXTERNAL_AGENTS_ROOT` + watcher	✅ yes
Workflow specs (mounted)	`AF_EXTERNAL_WORKFLOWS_DIR`	⊗ startup only
Registered script	`register_script` in a plugin	⊗ rebuild
In-process route / runner / provider	`agentforge.plugins` entry point	⊗ rebuild + restart

Appendix B — Environment-variable reference

Extension & tool surface (Parts I–III):

Var	Purpose	Default
AF_RUNNER_SHARED_SECRET	Enable/authenticate remote runners	empty = disabled
AF_RUNNER_HEARTBEAT_TTL_SECONDS	Runner deadman timer	60
AF_MCP_REGISTRY_PATH	Override MCP server registry	built-in
AF_<PROVIDER>_KEY (Exa/Firecrawl/Tavily/Brave)	Paid-search credentials	none
AF_PROVIDER_COST_CEILING_USD	Per-invocation paid-search warning	1.0
AF_PROVIDER_PRICING	Override per-provider unit prices (JSON)	2026 mid-tier
AF_ARTIFACT_MAX_BYTES	Per-artifact size cap	25 MB
AF_ARTIFACT_RUN_TOTAL_MAX_BYTES	Per-run artifact total cap	100 MB
AF_HTTP_EGRESS_RATE_PER_MIN / AF_HTTP_EGRESS_BURST	http-node egress rate limit	60 / 10
TAPPS_BRAIN_HTTP_URL / TAPPS_BRAIN_AUTH_TOKEN	NLT Memory HTTP bridge (the TAPPS_BRAIN_* prefix is the historical env-var name)	required if enabled

Fleet & operations (Parts IV–V):

Var	Purpose	Default
AF_IDLE_TIMEOUT_SECONDS	Subprocess no-output watchdog (0 = off)	150
AF_HEARTBEAT_TIMEOUT_SECONDS	Liveness clock for long nodes (0 = off)	0
AF_DEFAULT_MAX_BUDGET_USD	Default per-run budget	1.0
AF_MONTHLY_BUDGET_USD	Per-portfolio monthly cap (0 = off)	0
AF_CAPABILITY_INVOKE_DEPTH	Agent-to-agent delegation depth ceiling (1–5)	3
AF_PROJECT_AUTH_REQUIRED	Enforce per-project bearer keys	false
AF_EXTERNAL_AGENTS_ROOT / AF_EXTERNAL_WORKFLOWS_ROOT / AF_EXTERNAL_SKILLS_ROOT	Multi-project mount roots (single-project workflows use AF_EXTERNAL_WORKFLOWS_DIR)	empty
AF_NODE_LOGS_RETENTION_DAYS	Node-log disk retention	7
OTEL_EXPORTER_OTLP_ENDPOINT	OTel span export target	none
AF_LLM_GATEWAY_URL / AF_LLM_GATEWAY_MASTER_KEY	LLM cost egress gateway (opt-in)	none
AF_GATEWAY_VAULT_KEYS	Provider keys materialized into the gateway from AF's vault	MESHY_API_KEY
AF_CONTEXT_TOKEN_BUDGET	Per-run context-token budget (warn on overflow)	none

Appendix C — Key design decisions

- **Compute lives at the edges.** The per-agent container/docker-socket runner was retired; you extend via a host you own, a tool server you write, a script you register, or your own repo — not a sandbox inside AF.
- **Egress is config-owned, never LLM-widened.** Allowlists live in agent/workflow config; the model can use egress but cannot expand it.
- **Native-first search.** Free native web tools are the default; paid providers are a justified, cost-guarded exception.
- **Effect-verified artifacts.** Nodes prove they produced the file (type, size, hash) — no self-reported success.
- **Primitives, not features.** The platform ships credential preflight, artifact verification, and a spend ledger; you ship the agent that calls the API. "2D/3D/search" are things you build on AgentForge, not features of it.

Appendix D — Security posture at a glance

For readers evaluating against a compliance bar (SOC 2, EU AI Act — see the companion paper's §15), the trust-boundary claims scattered through this guide, collected in one place:

- **No host-Docker-socket surface.** The per-agent container runner that interposed on the host Docker socket was removed (§2); agents run as subprocesses, and custom compute runs on a remote runner or in your repo.
- **Egress is config-owned.** Allowlists live in agent/workflow config and can't be widened by the model; a block-list (RFC1918, cloud metadata, loopback) runs before the allowlist (§8).
- **Remote-runner trust is explicit.** Shared-secret auth, closed by default, internal-LAN only, with TLS termination your responsibility (§3).
- **Secrets never reach the model.** Per-project vault, encrypted at rest, resolved at invocation rather than prompt assembly, with a non-valued audit log (§21).
- **Tenancy is row-scoped.** `project_id` / `portfolio_id` on every table; the container is the cross-customer boundary (§17).
- **Federation is bounded.** Agent-to-agent calls are depth-capped and cycle-detected; agent cards export only safe fields, never prompts, guardrails, or tool wiring (§18).
- **Spend is bounded.** Per-run budgets and, with the gateway, 4xx-on-wire enforcement; paid-search ceilings warn on runaway cost (§9, §19).

Complete (Parts I–V). Claims are grounded in implemented mechanisms; where a capability is opt-in or config-gated, the paper says so.