VETS WHO CODE · HASHFLAG STACK · PHASE 3

The AI Engineering Playbook

A leveled, in-depth path for software engineers — from the first model call to spec-driven mastery. Eight rungs, four tiers, and the inversion at the top that turns a builder into an engineer.

RULE 01

Core before conditional. Build small.

RULE 02

Termination is the boss at every altitude.

RULE 03

Beginners climb up. Experts invert.

FIELD MANUAL · V4.0

THE COMPLETE CLIMB

HOW TO READ THIS

Architecture is a map. This is the path through it.

You already have the architecture — where each piece **sits**: spec, model, knowledge, harness, loop, and the agent-composition stack. This playbook is a different ordering: where each piece is **learned**. The two are not the same, and that difference is the point. The most notable reorder is deliberate — architecturally the knowledge layer sits before the harness, but here you build the loop first and add retrieval later, because retrieval bolted onto a loop you don't understand is how engineers ship RAG systems they can't debug. Dependency for *understanding* runs loop-first.

Rule 1 — Core before conditional.

Every rung is either **core** (present in almost every system) or **conditional** (only if your architecture calls for it — the knowledge layer is the clearest case; a pure tool-use or code-execution agent skips it entirely). This split is permission: **one agent with a few tools is the floor**, and a smaller first system is not a cut corner.

Rule 2 — Termination is the recurring boss.

"How do I call a tool" is easy. `How does it know it's done` is the actual job — and it returns at every altitude. The runtime loop, each subagent, the orchestrator, even the dev loop and the outer lifecycle loop all face the same problem raised one level. Where agents break is almost never the tool call; it's the stop condition.

Rule 3 — The through-line runs down to the token.

Tokenizer behavior and logprobs are not trivia. They become a **confidence signal**, which becomes a **loop-control decision**: low confidence → iterate; high confidence → exit. The deepest thread in the stack is `token entropy → confidence → termination`. Foundations are the control plane.

THE INVERSION

THE BEGINNER — L0 UPWARD

Climbs bottom-up.

Starts from a single model call and assembles primitives, because they don't yet know what's buildable. The ladder is how you learn what the pieces are.

THE EXPERT — L7 DOWNWARD

Starts at the spec.

Works top-down from intent because the primitives are internalized. Writes the test of "done" before building, and lets the spec govern everything below it.

CLIMB THE LADDER ONCE · THEN INVERT IT · THAT INVERSION IS WHAT "EXPERT" MEANS

THE WHOLE CLIMB, ONE PAGE

Map at a glance

Eight rungs in four tiers. Read the identity column as "what you can do when you're standing here." Tags mark whether the rung is core or conditional-by-architecture.

TIER I — FOUNDATIONS · CAN CALL A MODEL RELIABLY			
L0	**The Call**	Validated, structured data back from a single model call.	CORE
L1	**The Single Action**	The model can *do* one thing — one tool, one hop, no loop.	CORE
TIER II — THE AGENT · CAN SHIP A LOOP THAT TERMINATES			
L2	**The Loop**	Close the loop — your first real agent. Termination appears.	CORE
L3	**The Harness**	The nouns around the loop. It survives contact with reality.	CORE
TIER III — MAKING IT REAL · CAN GROUND IT AND PROVE IT			
L4	**Knowledge & Confidence**	Retrieval when needed; logprob gating as a loop-control signal.	COND+CORE
L5	**Proof**	The eval harness plus observability. "It works" becomes numbers.	CORE
TIER IV — SCALING JUDGMENT · KNOWS WHAT NOT TO BUILD			
L6	**Composition**	Skills, subagents, orchestrators — earned, never reflexive.	CORE
L7	**The Inversion**	Spec-driven development and the outer loop. You start at the top.	CORE

HOW TO USE THIS PLAYBOOK

- **Work it in order.** Each rung assumes the one below it. The ordering is learning-dependency, not architecture.
- **Do not skip L2.** Termination is where most self-taught engineers stall; every rung above it inherits the problem.
- **Conditional means optional-by-architecture,** not optional-by-effort. Skip retrieval only if your system genuinely doesn't need it.
- **Mastery of a rung** = every checkbox hit *and* you can teach its failure modes to someone else. Anything less is "passed through," not "cleared."

THE SHAPE OF THE CLIMB

The ladder, and the inversion

Eight rungs, bottom to top. You climb it once — assembling primitives from L0 upward. Then, at the top, you **invert**: the expert stops climbing and drives the whole stack downward from a spec. Learn upward; work downward.

BEGINNER CLIMBS UP FROM THE ATOM ↑

L7	The Inversion	SPEC-DRIVEN · OUTER LOOP
L6	Composition	SKILLS · SUBAGENTS · ORCHESTRATORS
L5	Proof	EVALS · OBSERVABILITY
L4	Knowledge & Confidence	RAG · LOGPROB GATING
L3	The Harness	CONTEXT · RECOVERY · STATE
L2	The Loop	TERMINATION APPEARS
L1	The Single Action	ONE TOOL · ONE HOP
L0	The Call	STRUCTURED OUTPUT

EXPERT DRIVES DOWN FROM THE SPEC ↓

Gold rung = the expert tier. **Red left edge** = conditional-by-architecture (L4 — skip if the system doesn't need retrieval). The two arrows are the two directions of travel.

TIER I

Foundations

The bar: you can get a model to return something your code can trust — and act on, once.

0

TIER I · FOUNDATIONS

The Call

CORE

You can get a model to return something your code can trust.

Before an agent, before tools, before any of it — the atom is a **single model call whose output your program can consume without a human in the loop**. Most reliability problems higher in the stack are actually unsolved L0 problems. If one call can't reliably return valid structured data, nothing you stack on top will be reliable either. `Get the atom right first.`

The atom. A single call is only trustworthy once it's validated and can repair a bad parse — the seed of every loop above it.

WHAT YOU'RE ACTUALLY LEARNING

The **request/response contract** — messages and roles, and how the system prompt sets durable behavior while the user turn carries the task. Then **sampling and determinism**: temperature and top-p, why the same prompt yields different outputs, and when to drive temperature to zero for extraction and classification versus keeping it up for generation.

The core skill of the whole rung is **structured output** — the difference between prose you read and data you parse. That runs from JSON mode, to using a tool/function schema as the output shape, to the deeper move of **constrained (grammar-guided) decoding** that makes malformed output structurally impossible rather than merely discouraged. Paired with it is **schema enforcement and repair**: validate against a typed model, and when it fails, feed the validation error back and retry within a bounded budget.

Underneath sits **token economics**: input versus output tokens, why prompt size is a cost and latency lever, and the context window as a hard budget you are already spending on turn one.

CONCEPTS TO MASTER

- **Roles** — system vs. user vs. assistant, and what belongs in each.
- **Determinism controls** — temperature / top-p; when zero, when not.
- **Structured output** — JSON mode vs. tool-schema output vs. `constrained/grammar decoding`.
- **Validation + repair** — typed-schema (Pydantic-style) validation and the parse-error retry loop.
- **Idempotent, retryable calls** — a single call you can safely re-issue.
- **Token accounting** — input/output cost and latency; counting before you call.
- **Prompt-as-contract** — one call's prompt is a spec (the seed of L7).
- **Output failure modes** — truncation, schema drift, and detecting them.
- **Format priming** — few-shot examples and system prompts that lock shape.
- **Prompt caching** — reusing a stable system prompt to cut cost and latency.

WHAT YOU BUILD

A CLI that takes arbitrary text and returns a **validated object** — e.g. `{summary, topics[], sentiment, confidence}` — against a typed schema, with a repair path that catches a malformed response, feeds the validation error back, and retries within a bounded number of attempts before failing loudly.

CLEAR IT WHEN

- [ ] 20+ runs on varied input return 20+ valid objects, **zero manual fixes**.
- [ ] You can force a schema and programmatically recover from a bad parse.
- [ ] You can explain why your extraction runs at temperature zero.
- [ ] You can state the token cost of a call **before** you make it.

WHERE PEOPLE GET STUCK

- × **String-parsing model output** instead of validating structured data — the original sin; every flaky agent traces back here.
- × **Trusting the first response** with no repair loop.
- × **Leaving temperature high** on a task that needs determinism.
- × **Treating the context window as infinite.**

CONNECTS *Bottom of the model/prompt foundation — no loop yet. The* **repair loop** *here is the seed of the runtime loop at L2;* **prompt-as-contract** *is the seed of spec-driven development at L7.*

1

TIER I · FOUNDATIONS

CORE

The Single Action

The model can *do* one thing — not just say things.

Tool use is where a language model stops being a text generator and becomes something that can **affect the world**. But the first version is deliberately **not a loop** — it's a single round trip. Getting that round trip clean, with the **model** (not your code) deciding whether to act, is the whole lesson. This is the hinge between "chatbot" and "agent." `One hop, done right.`

One hop, not a loop. The model decides *whether* and *which* — your code never routes. Master this before adding iteration.

WHAT YOU'RE ACTUALLY LEARNING

The **tool-calling round trip**, in four steps and one hop: you expose tools with schemas; the model emits a structured tool-call request; your code executes and returns a tool-result message; the model produces its final answer using that result. Not iteration — a single exchange.

Tool schemas are contracts — name, description, typed parameters — and the description is prompt engineering, because the model routes on it. Vague descriptions cause bad routing. The competence to build is **model-decides, not code-decides**: the model chooses *whether* and *which* tool to call. If your code hardcodes the call, you built an if-statement in a costume.

Result formatting matters as much as the call — the model has to consume what you return, and errors should come back as tool *results*, not thrown exceptions, so the model can react to them. And you learn the **boundary**: why this is explicitly one hop, so you can isolate "can it call correctly" before adding "can it decide when to stop."

CONCEPTS TO MASTER

- **The four-step round trip** — expose → request → execute → answer.
- **Schema design** — description-as-routing-signal; typed parameters.
- **Argument validation** — check args before execution.
- **Result shaping** — return model-consumable results, including errors.
- **Parallel tool calls** — multiple calls in one turn, and when that's safe.
- **Hop vs. loop** — knowing this is one exchange, not iteration.

- **Result grounding** — answer from the result; don't reason around it.
- **Tool vs. inline reasoning** — when a call earns its cost over thinking.
- **Sequential vs. parallel** — dependency-aware call ordering.

WHAT YOU BUILD

A one-tool (then two-tool) assistant. Give the model a calculator and a lookup function; it must decide whether the query needs a tool at all, pick the right one, call it with well-typed arguments, and answer from the result — with **no hardcoded routing**.

CLEAR IT WHEN

- ☐ The model correctly **declines** a tool when the answer doesn't need one, and calls the right one when it does.
- ☐ Malformed arguments are caught before execution; tool errors return as recoverable results.
- ☐ You can add a second tool **without rewriting your control flow**.

WHERE PEOPLE GET STUCK

- × **Hardcoding "always call the tool"** — code deciding, not the model.
- × **Vague tool descriptions** that cause mis-routing.
- × **Throwing on tool failure** instead of returning an error result.
- × **Jumping to a loop** before the single hop is solid.

CONNECTS *Top of the model foundation / the tool-dispatch primitive at the floor of the harness. The next rung turns this single hop into a cycle — and that's where the hard part lives.*

TIER II

The Agent

The bar: you can ship a working loop that finishes real tasks and never runs away.

2

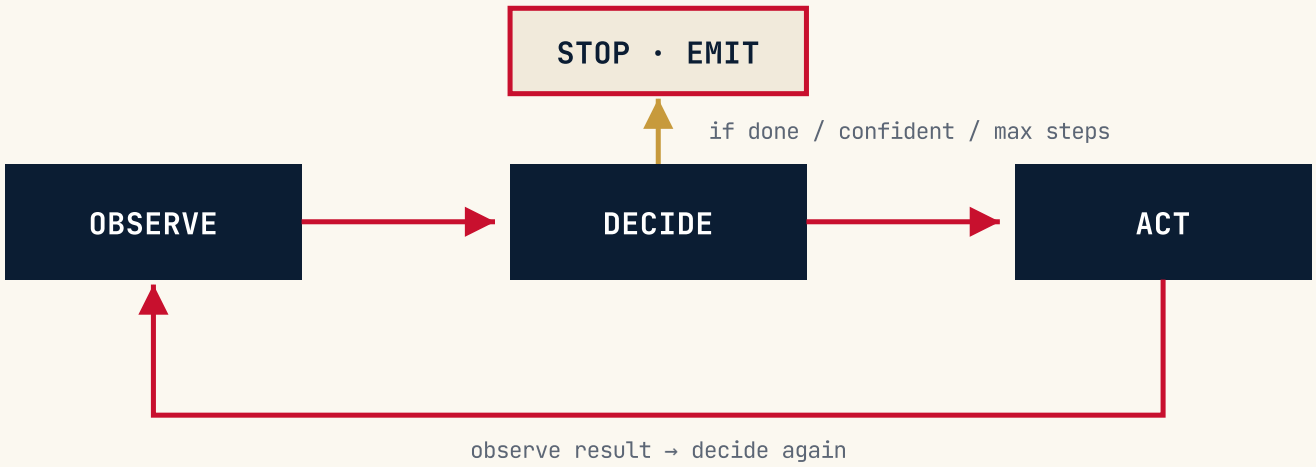
TIER II · THE AGENT

The Loop

CORE

Your first real agent — because you closed the loop.

The exact moment you feed a tool result back and let the model decide **again** — call another tool, or stop — you have an agent. Everything before was a pipeline; this is the first thing that **pursues a goal**. And with the loop comes the single hardest problem in agent engineering, which recurs at every rung above this one: **termination**. Not "how do I call a tool" — that was L1 and it's easy — but `how does it know it's done, and how do I stop it when it doesn't.`



The agent loop. The cycle is the verbs; the gold branch is termination — the gate every rung above L2 inherits.

SINGLE-PASS

`call → answer`

One tool, one hop, no iteration. This is L1 — the baseline before the loop.

ReAct

`think ⇄ act ⇄ observe ↻`

Interleave reasoning and acting each turn until done. The default agent loop.

PLAN-EXECUTE

`plan → [ s1 · s2 · s3 ]`

Commit a plan up front, then run the steps. Best when the path is knowable.

REFLEXION

`act → critique → retry`

Self-evaluate the attempt, revise, try again. Costly; use when quality justifies it.

WHAT YOU'RE ACTUALLY LEARNING

The **agent loop** itself: observe → decide → act → observe result → decide again → … → terminate. The minimal cycle that defines agency. Then **loop topologies**, and choosing between them *is* the design work — not a detail: single-pass tool use; **ReAct** (interleaved reasoning and acting); **plan-then-execute** (plan up front, then run the steps); and **reflexion / self-critique** (act, evaluate your own output, revise).

Termination conditions, in depth, and real agents use several at once: a **max-step ceiling** (the safety net), an **explicit done-signal** (the model declares completion, often via a "finish" tool), a **goal-satisfaction check** (external verification the objective is met), and a **confidence threshold** (completed at L4). Alongside them, the **failure taxonomy**: runaway loops (never stops), oscillation (A→B→A→B), premature termination (declares done too early), and thrash (busy but not progressing) — each with a different fix.

Finally, **progress guarantees** — ensuring each iteration moves toward the goal via step budgets, state deltas, and no-progress detection — and **loop-carried state**: what the cycle carries forward, and why naive accumulation blows the context budget (which L3 solves).

CONCEPTS TO MASTER

- **The cycle** — observe / decide / act / terminate.
- **Topologies** — `ReAct vs. plan-execute vs. reflexion`; tradeoffs and fit.
- **Multiple stop conditions** — ceiling + done-signal + goal check together.
- **The finish-tool pattern** — an explicit, model-emitted done-signal.
- **Failure detection** — oscillation, thrash, no-progress.
- **Progress guarantees** — step budgets and state deltas.
- **Loop-carried state** — and its context cost.
- **Human-in-the-loop** — a person as a stop, branch, or approval gate.
- **Budget-aware loops** — token and cost ceilings as hard stops.
- **Deterministic replay** — re-running a trace to reproduce a decision.

WHAT YOU BUILD

A multi-step agent that chains 2–4 tool calls to complete a real task — look something up, transform it, verify the result — and **stops itself** correctly. Implement it once as `ReAct` and once as `plan-execute` so you feel the difference, with a hard step ceiling and an explicit done-signal on both.

CLEAR IT WHEN

- [ ] It finishes genuine multi-step tasks **and** never runs away — you can point to the exact condition that stops it.
- [ ] You can trigger, detect, and recover from oscillation.
- [ ] You can explain why you chose your topology over the alternatives.
- [ ] Given a task it **can't** complete, it terminates cleanly instead of looping forever.

WHERE PEOPLE GET STUCK

- × **No stopping theory at all** — the #1 reason first agents fail.
- × **One stop condition where several are needed** — e.g. only a ceiling, so it "succeeds" by hitting the cap.
- × **Unbounded state accumulation** that overflows the window mid-run.
- × **Reflexive topology choice** — ReAct for tasks that want a plan, or vice versa.

CONNECTS *This is the behavioral core of the harness — **loop engineering**, the verbs. L3 builds the nouns around it. Termination introduced here goes **fractal at L6**; the confidence-threshold stop condition is completed at **L4** by the token-level work.*

3

TIER II · THE AGENT

CORE

The Harness

Your loop survives contact with reality.

An L2 loop works in a demo and dies in production, because production is tool timeouts, malformed output, rate limits, and context windows that overflow on turn nine. The harness is everything you build **around** the loop so it degrades gracefully instead of crashing — or worse, failing silently. If L2 is the verbs, L3 is **the nouns**: the static machinery the loop cycles through.

The context budget. Every turn competes for one fixed window. When it fills, you truncate by recency or relevance, or compact old turns — the highest-leverage harness skill.

WHAT YOU'RE ACTUALLY LEARNING

Context assembly and truncation is the highest-leverage harness skill. Each turn you decide what enters the window — system prompt, tool definitions, relevant history, retrieved context, current state — and what gets dropped when you exceed budget: truncation by recency or relevance, and **summarization / compaction** of old turns. "Just send everything" fails, and this is where most quality-under-load problems actually live.

Tool dispatch hardening: timeouts, retries with backoff, idempotency, concurrency limits, failures returned as results not exceptions. **Error recovery**: distinguishing recoverable errors (retry, reformulate) from terminal ones (stop, escalate), and wrapping the loop so a single bad turn doesn't kill the run. **State and memory**: short-term working transcript versus persistent cross-session memory — and recognizing that memory is a **context-management problem in disguise**.

Streaming for UX and the engineering it forces — partial parsing, tool calls mid-stream, cancellation. And **output parsing under load**: the structured-output enforcement from L0 applied at *every* turn, with repair, because at scale the model will occasionally return something malformed.

CONCEPTS TO MASTER

- **Context budgeting** — `truncation & compaction` strategies.
- **Rolling memory** — summarization to fit long runs.
- **Dispatch hardening** — timeouts, backoff, idempotency, concurrency limits.
- **Error classification** — recoverable vs. terminal.
- **State tiers** — short-term vs. persistent; memory as context management.
- **Streaming** — partial parsing and cancellation.
- **Per-turn enforcement** — structured output + repair at every step.
- **Prompt/response caching** — avoid recomputing stable context.
- **Rate-limit & quota handling** — backpressure without dropping work.
- **Idempotency keys** — safe retries for side-effecting tools.
- **Observability hooks** — tracing baked into the loop, not bolted on.

WHAT YOU BUILD

> Take your L2 agent to production shape by **injecting faults deliberately**: a tool that throws, a tool that hangs (add timeouts), a model that returns garbage (add repair), a run long enough to overflow the window (add truncation/compaction). Add streaming and persistent state. It must handle every fault **without crashing and without silently doing the wrong thing**.

CLEAR IT WHEN

- [ ] Kill a tool, feed it junk, overflow the window — it recovers or fails **cleanly and loudly**, never silently.
- [ ] Context stays under budget across a long run via a strategy you designed on purpose.
- [ ] It streams and can be cancelled mid-turn without corrupting state.
- [ ] You can trace any run and see exactly what was in the window at each turn.

WHERE PEOPLE GET STUCK

× **A loop that only works on the happy path.**

- × **"Send the whole history every turn"** until it overflows — no truncation strategy.
- × **Silent failures** — swallowing errors so it looks like it worked but didn't.
- × **Memory as storage** rather than as a context-budget decision.

CONNECTS *This completes the harness — the orchestration layer. The* **context-assembly** *skill here is exactly what makes subagents worth building at L6 (they exist largely to keep the parent's context clean). L5 is what lets you* **prove** *the harness is robust rather than assume it.*

TIER III

Making It Real

The bar: you can ground the agent in real knowledge and prove it works with numbers.

4

TIER III · MAKING IT REAL

COND + CORE

Knowledge & Confidence

The agent knows things it wasn't trained on — and knows when it doesn't.

Two distinct capabilities share this rung because both are about **grounding**: one grounds the agent in external knowledge (**retrieval** — conditional; skip it entirely for pure tool/code agents), the other grounds the loop's decisions in the model's own certainty (**confidence** — core; the completion of the termination work from L2 using the token-level signal).

Retrieval pipeline (conditional). Chunking is the dominant quality lever; reranking is the precision multiplier.

Confidence gate (core). Token certainty becomes a loop-control decision — the stop condition promised at L2.

Retrieval CONDITIONAL HALF

The **pipeline**: ingest → chunk → embed → store → retrieve → rerank → inject, each stage with its own failure modes. The dominant lever is that **chunking dominates quality** — size, overlap, and structure-aware splitting matter more than model choice, and most bad retrieval is bad chunking. **Embeddings and vector stores** give you similarity search and its limits; **reranking** is the quality multiplier — retrieve broadly, rerank precisely, trading recall for precision.

Context injection connects back to L3's budget — retrieved chunks cost window — and you learn the distinction between retrieval-augmented *generation* and retrieval as a **tool the loop calls**. Crucially, **when not to use RAG**: if the knowledge fits in context, or the task is tool/computation-bound, retrieval is pure overhead.

Confidence CORE HALF

Logprobs as a signal, not a curiosity — token-level log-probabilities expose how certain the model is, per token — and **entropy / perplexity** over the distribution as an aggregate measure. The payoff is **confidence as loop control**: low confidence → iterate (gather more, reformulate, ask a human); high confidence → exit and emit. `This is the confidence-threshold stop condition promised at L2.`

With it comes the **calibration caveat** — confidence is not correctness; you learn where token confidence misleads and how to combine it with external verification — and **constrained decoding revisited**: enforcing structure at the token level as a reliability mechanism the harness owns.

CONCEPTS TO MASTER

- **The retrieval pipeline** — ingest/chunk/embed/store/retrieve/rerank.
- **Chunking strategy** — size, overlap, structure-aware; the dominant lever.
- **Embeddings + vector search** — recall vs. precision.
- **Reranking** — the precision multiplier.
- **RAG-as-tool vs. augmentation** — and when to skip RAG entirely.
- **Logprobs, entropy, perplexity** — confidence as a measurable signal.
- **Confidence-gated control** — `retry / escalate / exit`.
- **Calibration** — confidence vs. correctness.
- **Hybrid search** — dense vectors + sparse (BM25) for recall.
- **Metadata filtering & routing** — narrow the space before retrieval.
- **Chunk provenance** — carry citations through to the answer.
- **Self-consistency** — sampling multiple traces to estimate certainty.

WHAT YOU BUILD

> Add a **retrieval-backed answer path with reranking** to your harness, and — separately — wire an **entropy/confidence gate** that changes the loop's behavior: on a low-confidence turn it retries, gathers more, or escalates rather than emitting. **Measure both against a no-retrieval / no-gate baseline.**

CLEAR IT WHEN

- [ ] Retrieval measurably beats no-retrieval on your task — you have the numbers.
- [ ] Your chunking strategy is a **deliberate choice you can defend**, not a default.
- [ ] The confidence gate demonstrably catches low-quality turns before they reach the user.
- [ ] You can say where token confidence is trustworthy for your task and where it isn't.

WHERE PEOPLE GET STUCK

- × **Bolting RAG onto a loop they don't understand** — why you did L2/L3 first.
- × **Blaming the model** for what is actually a chunking problem.
- × **Logprobs as a dashboard number** instead of a decision input.
- × **Assuming high confidence means correct.**

CONNECTS *The knowledge layer folds back in* **after** *the loop (the deliberate reorder). The confidence half closes the loop opened at L2 —* **token entropy → confidence → termination**, *the deepest thread in the stack. L5 generates the numbers that prove any of this works.*

5

TIER III · MAKING IT REAL

Proof

CORE

You can say "it works" and back it with numbers.

Everything up to here can be **demoed**. None of it can be **trusted** until it's measured. This rung is the line between someone who made an impressive demo and someone who **operates a system** — and it's where many engineers stall, because evaluation is less fun than building and more necessary than anything else. Two halves: the **eval harness** (offline proof) and **observability** (online proof).

Agent harness

Runs the agent in production — the loop, tools, context, streaming.

Eval harness

Runs benchmarks *against* the agent — datasets, judges, regression gates.

Same word, different rig. One executes the agent; the other measures it. Offline evals gate the build; observability watches production.

Evaluation THE OTHER HARNESS

Eval datasets tied to the spec — your test set encodes what "done" means, drawn from the same source of truth as the build (foreshadowing L7). **Offline eval types**: task-completion, exact and semantic accuracy, and critically **regression** testing, so a change that fixes one thing doesn't silently break three. **LLM-as-judge** for grading at scale, along with its failure modes — length bias, position bias, self-preference — and how to constrain and validate the judge.

Adversarial / red-team evals probe how it breaks under hostile or edge input, and **safety evals** are dedicated tests for the behaviors your system must never exhibit. Keep the **eval-harness-vs-agent-harness** distinction sharp: same word, different rig — one runs the agent, the other runs benchmarks against it.

Observability OPERATING IT

Tracing makes every run inspectable end-to-end — prompts, tool calls, retrievals, decisions. **Cost and latency monitoring** per run and per component is the operational reality of shipping. **Structured logging** and **online / production evals** grade real traffic, not just the test set, and **feedback capture** closes the loop with real user signal — the input to L7's revision.

CONCEPTS TO MASTER

- **Spec-derived datasets** — the test set encodes "done."
- **Eval types** — task-completion, accuracy, `regression`.
- **LLM-as-judge** — its biases and mitigations.
- **Adversarial + safety evals** — hostile input and must-not behaviors.
- **Eval harness vs. agent harness** — same word, different rig.
- **Tracing** — end-to-end run inspection.
- **Cost/latency + structured logs** — the operations layer.
- **Online evals + feedback capture** — grading real traffic.
- **Golden datasets** — curated, versioned sets that encode "good."
- **Component vs. end-to-end** — grade sub-agents and the whole system.
- **Judge calibration** — align the LLM judge against human labels.
- **Cost/latency as criteria** — budgets are pass/fail, not afterthoughts.

WHAT YOU BUILD

An **eval suite** that runs your agent against a fixed dataset and reports pass/fail plus **regression deltas between versions**, and **tracing** on every run so any production failure is reconstructable with its cost and latency attached.

CLEAR IT WHEN

- ☐ A change to a prompt or the harness produces a **measured delta, not a vibe**.
- ☐ You catch a regression **before** it ships.
- ☐ You can explain any production failure **from traces alone**.
- ☐ Your evals derive from your spec, not from what was convenient to test.

WHERE PEOPLE GET STUCK

- × **"It looked good in the demo"** — anecdote as evidence.
- × **No regression suite** — every fix risks a silent break.
- × **An unvalidated LLM judge** whose biases you inherit.
- × **Evals disconnected from the spec** — passing them proves little.

CONNECTS *The evaluation and observability layers — the* **gate between build and ship**, *and the operating layer after. Eval-first thinking here is the on-ramp to L7's inversion: at expert level you write the eval* **before** *the build.*

TIER IV

Scaling Judgment

The bar: you compose agents deliberately, and you start from the spec instead of the code.

6 Composition

TIER IV · SCALING JUDGMENT

CORE

You compose agents — deliberately, not reflexively.

There is really **one primitive** — the agent — and three ways it elaborates: a thing **below** it (skills), a **role** it plays (subagent), and a **specialization** of it (orchestrator). They stack at different altitudes, and the discipline of this rung is **restraint**: composing only what the task earns. The trap of the current moment is multi-agent architecture as `premature optimization`.

ORCHESTRATOR	An agent whose action space is **other agents**. Loop: decompose → dispatch → collect → synthesize → done.
SUBAGENT	The agent in the **child position** — own window, own loop, returns a distilled result. A tool from the outside.
AGENT	**The base unit** — model + harness + loop. The smallest thing that can pursue a goal on its own.
SKILLS / TOOLS	Capability packages loaded on demand — **no loop of their own**. Below the agent, like libraries to a program.

One primitive (the agent). Below it: skills. A role it plays: subagent. A specialization: orchestrator. Earn each layer.

WHAT YOU'RE ACTUALLY LEARNING

Skills — capability packaging, below the agent: instructions plus optional scripts and resources, bundled and loaded on demand, with **no loop of their own** — inert until invoked. Skills are to agents what libraries are to programs, and a skill definition is itself a **spec at the capability grain** (SDD, one level down). Progressive disclosure — load the capability only when relevant — protects context.

Subagents — the agent in the child position, a role rather than a new type: its own context window, its own loop, returning a **distilled** result to its caller. From the parent it looks like a tool; inside it's a full agent. The real reason it exists is usually **context isolation** — delegate so the parent's window stays clean and gets back a compressed answer — which is why so much "multi-agent architecture" is really **context architecture**, a direct callback to L3. The second reason is **parallelism**: fan out independent subtasks.

Orchestrators — an agent whose action space is other agents, a specialization whose loop is decompose → dispatch → collect → synthesize → decide-done, sitting at the top of the coordination layer. And **termination goes fractal**: every composed agent (except skills, which don't loop) must know when *it's* done, and the orchestrator faces the harder nested version — **when is the collective done?** That's L2's problem raised a level, and it's where multi-agent systems actually break.

CONCEPTS TO MASTER

- **Skills** — loaded-on-demand capability packages with no loop.
- **Progressive disclosure** — load capability only when relevant.
- **Subagent** — agent in child position; own window/loop; `distilled return`.
- **Multi-agent as context architecture** — isolation + parallelism.
- **Orchestrator** — decompose / dispatch / collect / synthesize.
- **Fractal termination** — per-agent *and* collective done-ness.
- **Earned-complexity discipline** — justify every layer in one sentence.
- **Result contracts** — the schema a subagent returns to its caller.
- **Failure isolation** — a dead subagent must not kill the run.
- **Shared vs. isolated memory** — what state crosses agent boundaries.
- **Cost of coordination** — every hop adds latency, tokens, failure surface.

WHAT YOU BUILD

Take a system that's grown unwieldy and: **(1)** extract one bloated capability into a reusable **skill**; **(2)** isolate a context-heavy subtask into a **subagent** that returns a compressed result; **(3)** add an **orchestrator** only if the task genuinely decomposes into independent parts — and if it doesn't, deliberately don't. Then **remove at least one layer** that isn't earning its place.

CLEAR IT WHEN

- [ ] You can justify every added layer in one sentence — packaging, isolation/parallelism, or genuine decomposition.
- [ ] You've **removed** a layer that wasn't pulling its weight.
- [ ] Your orchestrator has a real answer to "**when is the collective done?**"
- [ ] You can explain why a given task got a single loop instead of a swarm.

WHERE PEOPLE GET STUCK

- × **A subagent swarm under an orchestrator** where one loop would have shipped.
- × **Multi-agent as a capability story** when it's really a context story.
- × **No collective-termination logic** — the orchestrator that never cleanly finishes.
- × **Skills, subagents, and tools conflated** instead of placed at their altitudes.

CONNECTS *This spans the full height of the harness — skills at its floor, orchestrator at the top of coordination. It's the last thing before the inversion: once you can build* **and compose** *one solid, measured agent, you're ready to stop climbing and work from the spec.*

---

7 The Inversion

TIER IV · SCALING JUDGMENT

EXPERT

Expert. You don't climb the ladder anymore — you start at the top and drive down.

Every rung so far was **bottom-up**: assemble primitives because you don't yet know what's buildable. Expertise is the **inversion** — you start from intent, from a spec, and drive downward, because the primitives are now internalized. The whole playbook exists to be climbed once and then **run in reverse**. Spec-driven development is the governance that makes it possible; `evals-first` is how you think.

The outer loop. The spec (gold) is the single source of truth feeding BUILD and EVAL alike; production signal flows back to revise it.

WHAT YOU'RE ACTUALLY LEARNING

Spec-driven development as governance: the spec is a single durable artifact — behavior contracts, acceptance criteria, success metrics, constraint budget — that drives **both** the implementation and the evals. The same document your build follows is the document your eval suite derives from, and that shared source of truth is what makes the system verifiable.

The outer lifecycle loop: spec → build → eval → ship → observe → revise spec → … — the slow loop that wraps everything. Inside it sits the fast **inner dev loop** (prompt/harness iteration, dozens of times a day), and inside *that*, at runtime, the **agent loop from L2**. Three loops at three timescales; SDD governs the outer one. **Evals-first thinking** is the L5 skill promoted to a default stance — you specify the test of "done" before you build.

And the **inversion itself**: why beginners must go bottom-up (they don't yet know the shape of the buildable) and why experts go top-down (intent → governed build). Being able to hand a spec to another engineer — or to an agent — and get the intended system back is the mark of the level. Finally, **revising the spec from production signal**: the observability from L5 feeds spec revision, closing the outer loop without breaking it.

CONCEPTS TO MASTER

- **Spec as single source of truth** — for build *and* eval.
- **Contracts + criteria + budgets** — the anatomy of a spec.
- **Three nested loops** — `runtime / dev / lifecycle` and their timescales.
- **Evals-first** — specify "done" before building.
- **Spec-driven delegation** — to humans and to agents.
- **Closing the outer loop** — revise from production signal.
- **Executable acceptance criteria** — criteria that *are* evals.
- **Constraint budget** — latency, cost, and safety as first-class spec terms.
- **Versioned artifacts** — prompt, model, and data pinned like code.
- **Spec change log** — every revision traceable to a signal.

WHAT YOU BUILD

> Run a real feature **spec-first**. Write the behavior contract, acceptance criteria, and eval targets **before** any implementation. Let the spec drive both the build and the eval suite. Ship it, observe it in production, and **revise the spec from the signal** — one full turn of the outer loop, deliberately.

CLEAR IT WHEN

- [ ] You start from a **spec, not a prompt**.
- [ ] Your evals **derive from** your spec.
- [ ] You can hand the spec to another engineer or an agent and get the intended system back.
- [ ] You revise the spec from production signal without breaking the loop.

WHERE PEOPLE GET STUCK

- × **Staying a builder** — starting from code forever, never inverting.
- × **A spec that governs the build but not the evals** (or vice versa) — the two drift.
- × **The spec as a one-time document** instead of the thing you revise each cycle.
- × **Skipping the spec entirely** — the one thing that makes the loop closeable.

CONNECTS *Phase 0, wrapping the entire outer loop — the discipline that governs every rung below it. This is the destination the whole ladder was built to reach: the point where you no longer think in rungs, only in* **specs and loops**.

— FUNDAMENTALS, TRANSLATED

This is still software engineering

AI engineering isn't a break from software engineering — it's software engineering with new materials. Two classic disciplines map onto this stack: **SOLID** (the fit is analogical, but the discipline transfers) and the **SDLC** (which your outer loop already is).

SOLID, translated to agent architecture

PRINCIPLE	IN AGENT SYSTEMS	RUNG
Single Responsibility	One skill, subagent, or tool does **one job**. Keep loop-control logic separate from tool logic.	L1·L6
Open / Closed	The harness is **open to new tools and skills, closed to modification** of the core loop — add capability without rewriting control flow.	L3·L6
Liskov Substitution	Any subagent behind the tool interface is **swappable**; a model or provider can be substituted without breaking callers.	L6
Interface Segregation	**Narrow, focused tool schemas** over god-tools — an agent shouldn't depend on a bloated interface it mostly ignores.	L1
Dependency Inversion	Depend on **abstractions** — a model / tool / retrieval interface — not a specific vendor implementation.	L1·L3

The honest caveat: SOLID is a set of class-level OOP principles; its mapping onto agent architecture is **analogical, not literal**. But the underlying discipline — single responsibility, substitutability behind interfaces, depending on abstractions — is exactly what keeps a growing agent system from ossifying. Treat it as a design compass, not a rulebook.

The SDLC, made AI-native

CLASSIC PHASE	AI-NATIVE EQUIVALENT	RUNG
Requirements	The **spec** — behavior contract, acceptance criteria, constraint budget.	L7
Design	Architecture — model/prompt foundation, harness, loop topology, composition.	L1–L3·L6
Implementation	Building the loop and harness, plus the knowledge layer if the system needs it.	L2–L4
Testing	**Evaluation** — offline evals, adversarial, safety. Probabilistic outputs make testing **distributional**, not pass/fail unit tests.	L5
Deployment	Serving and ship — the body that exposes the harness over the wire.	L5
Maintenance	**Observability** + revise-spec — continuous, and a far tighter loop than classic ops.	L5·L7

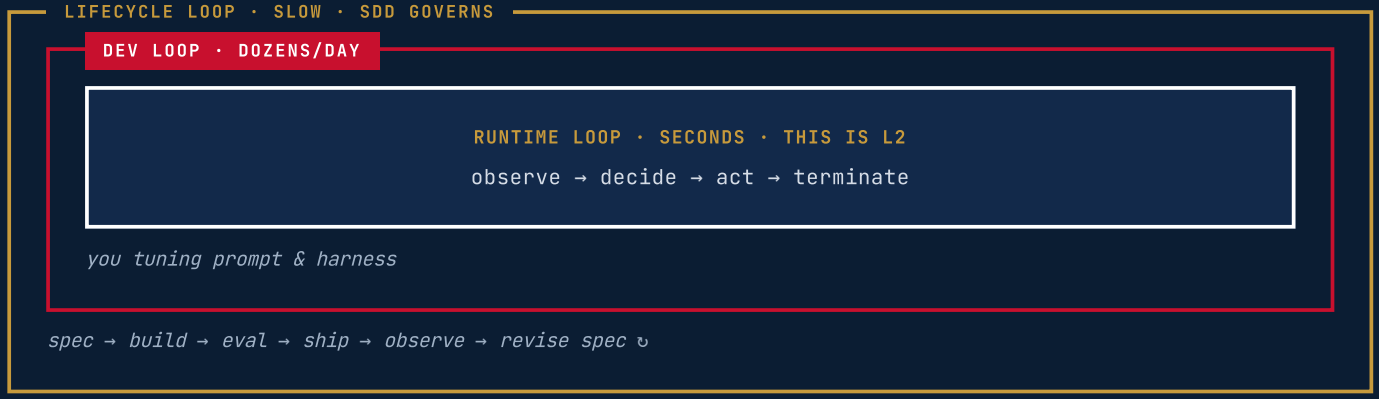
The punchline: the outer lifecycle loop — `spec → build → eval → ship → observe → revise` — **is the SDLC**, compressed and made continuous. The load-bearing difference is evaluation: classic testing assumes deterministic behavior, so it can't certify a probabilistic system. Evals are how the SDLC survives non-determinism.

THE CLIMB, INVERTED

Bottom-up was how you learned. Top-down is how you work.

SPEC → BUILD → EVAL → SHIP → OBSERVE → REVISE SPEC ↻

The whole ladder exists to be internalized and then run in reverse. Once the primitives are muscle memory, you stop assembling from the bottom and start **governing from the spec**. That inversion — from builder to spec-driven operator — is the actual definition of expert in this stack. Everything below L7 is the apprenticeship for it.



THREE LOOPS, THREE TIMESCALES

Runtime loop — the agent cycling at inference. Seconds per turn. `This is L2.`

Dev loop — you tuning prompt and harness. Dozens of times a day.

Lifecycle loop — spec → build → eval → ship → observe → revise. `SDD governs this one.`

TWO THREADS RUN THE WHOLE CLIMB

Termination is fractal. The stop-condition problem from L2 recurs at every altitude — each subagent, the orchestrator, every loop.

Confidence runs to the token. `Token entropy → confidence → termination.` Foundations are the control plane, not trivia.

THE EXPERT'S DAY — ONE TURN OF THE OUTER LOOP

01 **Start at the spec.** Behavior contract, acceptance criteria, eval targets — before a line of implementation.

02 **Build the smallest thing that satisfies it.** Core before conditional. One loop before a swarm.

03 **Measure against the spec.** The eval suite derives from the same document the build does.

04 **Ship, then watch production.** Traces, cost, latency, online evals, real feedback.

05 **Revise the spec from signal** — and close the loop without breaking it.

SHARED LANGUAGE

Glossary

The working vocabulary of the stack, in one place — so a cohort reads every rung the same way.

Agent
Model + harness + loop; the smallest unit that pursues a goal on its own.

Agent loop
The observe → decide → act → terminate cycle at runtime.

Augmented LLM
A model equipped with retrieval, tools, and memory.

Chunking
Splitting source text into retrievable units; the dominant RAG quality lever.

Compaction
Summarizing old turns to fit the context budget.

Confidence gating
Using token certainty to decide whether the loop iterates or exits.

Constrained decoding
Forcing output to match a grammar or schema at the token level.

Context window
The fixed token budget a model can attend to per call.

Entropy / perplexity
Aggregate measures of a model's uncertainty over its output.

Eval harness
The rig that runs benchmarks against an agent — distinct from the agent harness.

Guardrails
Safety and policy checks around a model's inputs and outputs.

Harness
The machinery around the loop: tools, context, recovery, state, streaming.

LLM-as-judge
Using a model to grade outputs at scale; prone to position, verbosity, and self bias.

Logprobs
Per-token log-probabilities; the raw signal behind confidence.

Orchestrator
An agent whose action space is other agents.

pgvector
A Postgres extension for vector similarity search.

Plan-execute
A topology that commits a plan up front, then runs the steps.

ReAct
Interleaving reasoning traces with actions each turn.

Reflexion
Self-critique stored in memory to improve the next attempt.

Reranking
Reordering retrieved candidates for precision after broad retrieval.

RAG
Retrieval-augmented generation: grounding output in retrieved passages.

Skill
A loaded-on-demand capability package with no loop of its own.

Spec-driven development
Governing both the build and the evals from one durable specification.

Structured output
Model output shaped as data your code can parse and validate.

Subagent
An agent in the child position; own window and loop, returns a distilled result.

Termination
The stop-condition problem — knowing when the loop is done.

Tool calling
A model emitting a structured request your code executes.

Trace / observability
End-to-end inspection of a run: prompts, calls, cost, latency.

Truncation
Dropping context by recency or relevance to fit the window.

GO DEEPER

Field references

Primary sources behind the rungs — the papers and practitioner guides worth reading in full, grouped by where they land on the ladder.

L1–L2 · TOOL USE & THE LOOP

- **ReAct: Synergizing Reasoning and Acting in Language Models.** Yao et al., ICLR 2023 · arXiv:2210.03629. Interleaves reasoning traces with actions — the founding pattern for the agent loop.
- **Reflexion: Language Agents with Verbal Reinforcement Learning.** Shinn et al., NeurIPS 2023 · arXiv:2303.11366. Self-critique held in an episodic memory buffer — the reflexion topology, no weight updates.
- **Toolformer: Language Models Can Teach Themselves to Use Tools.** Schick et al., NeurIPS 2023 · arXiv:2302.04761. Foundational treatment of models deciding when and how to call a tool.

L2 & L6 · AGENT & WORKFLOW ARCHITECTURE

- **Building Effective Agents.** Anthropic, 2024 · anthropic.com/research/building-effective-agents. The workflows-vs-agents distinction, the augmented LLM, and five composable patterns including orchestrator-workers. The best single primer on agent architecture.

L3–L4 · CONTEXT & KNOWLEDGE

- **Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.** Lewis et al., NeurIPS 2020. The original RAG formulation — parametric generation plus non-parametric retrieval, motivated by provenance and updatability.
- **Lost in the Middle: How Language Models Use Long Contexts.** Liu et al., 2023 · arXiv:2307.03172. Why position in the window matters — direct input to context-assembly and chunk-injection decisions.

L5 · EVALUATION

- **Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena.** Zheng et al., NeurIPS 2023 · arXiv:2306.05685. Establishes LLM-as-judge and names its biases — position, verbosity, self-enhancement.
- **G-Eval: NLG Evaluation using GPT-4 with Better Human Alignment.** Liu et al., EMNLP 2023 · arXiv:2303.16634. A widely used pattern for model-graded evaluation with chain-of-thought scoring.
- **Evaluation Best Practices.** OpenAI, developer docs. Practical eval-driven development — evaluate early and often, wire evals into CI as a merge gate.

L7 · SPEC- & EVAL-DRIVEN DEVELOPMENT

- **Eval-Driven Development.** DeepEval & practitioner guides, 2025–26. The top-down inversion — define the standards (evals) first, then build toward them.
- **LLM Evals FAQ.** Husain & Shankar, 2026. The counter-view: for open-ended systems, prefer error-analysis-first over writing every evaluator up front. Read alongside the above.

> **One debate worth teaching, not resolving:** "evals-first" is a genuine stance, but its strong form — write every evaluator before any code — is contested. The defensible middle: let the **spec** fix your acceptance criteria up front, and let the **eval set grow** from failures you actually observe. Specify what "done" means; discover how it breaks.