Member-only story

Production-Grade Agentic Coding Practices: Lessons from Claude Code, Codex and Gemini CLI

13 min read · Jul 7, 2025

Agent Native Follow

We recently dived deep into Claude Code, Codex CLI and Gemini CLI, and we reverse-engineered their **system prompts, guardrails, and memory tricks**, then battle-tested the findings in real use cases.

The result is **10 reusable patterns** that turn a weekend prototype into a fully-instrumented, policy-compliant *multi-agent assembly line,* whether you're parsing contracts, triaging lab results, or squeezing CTR out of ad spend.

Think JSON-first responses, phase-tagged telemetry, sandboxed tool calls, auto-extracted style schemas, the stuff frontier labs treat like Terraform for language models.

We packed it all into a one blog post you can copy-paste today and look like you invented tomorrow.

Grab a coffee and let's hot-rod your agents!

Why it really matters?

It's a daunting task to orchestrate an LLM-powered supply-chain of prompts, tools, memory, policies, and runtime sandboxes, especially in domain specific applications.

However if you look at the internals of state-of-the-art coding assistants today (Claude Code, Codex CLI and Gemini Code Assist), they all have converged on a core loop but diverge in how they hard-code guardrails, expose plans, and learn user taste.

These frontier labs treat system components like source-controlled infra-code, with peer review, A/B tests, and runtime metrics.

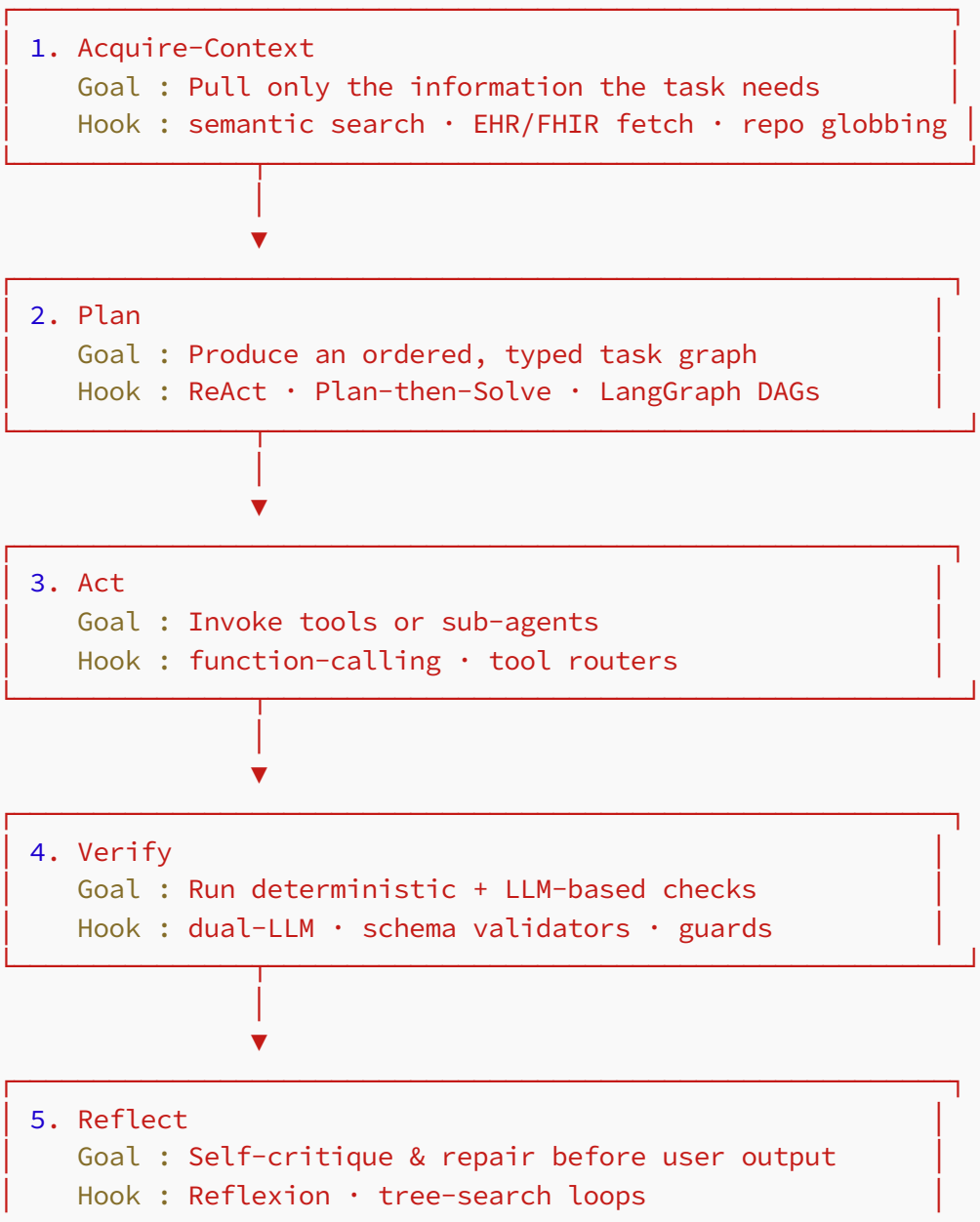
It might be confusing the take it all at once, that's why we want to break our findings down to 10 patterns that you can leverage today.

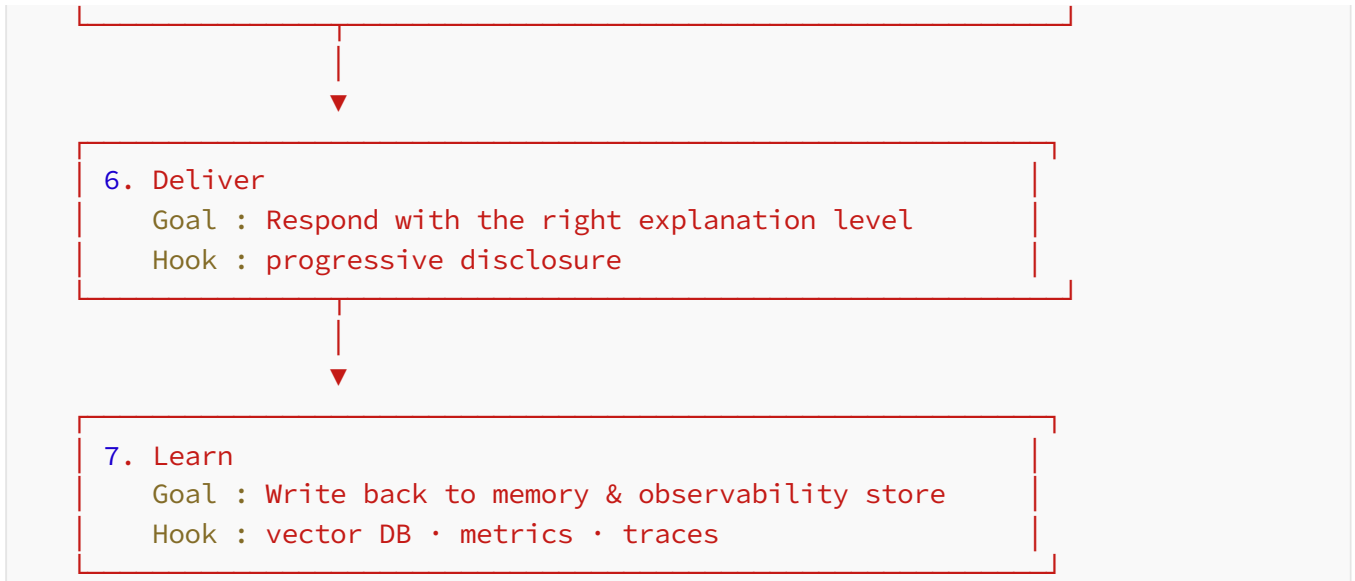
1. Canonical Control-Flow

Implementing the right control flow can be a challenge based on the domain, but in general they help you to achieve a few things:

- **Debuggability:** you can pinpoint which phase slowed or hallucinated.
- **Policy hooks:** each phase gates different permissions (read-only vs mutate).
- **Live UX:** the UI can stream progress because phases/firehose events are explicit.

All three agents implement a *ReAct-style* finite-state machine:





This flow is somewhat identical in Claude Code's CLI, Gemini ADK flows and Codex-derived copilots, only the wrappers differ.

The question is, how do we convert such ReAct flow into typed and observable finite state machine.

Here's a simple reference:

```python
# fsm.py
from enum import Enum, auto
from typing import Callable, Awaitable
import asyncio, logging

log = logging.getLogger(__name__)

class Phase(Enum):
    ACQUIRE=auto(); PLAN=auto(); ACT=auto()
    VERIFY=auto(); REFLECT=auto(); DELIVER=auto(); LEARN=auto()

Handler = Callable[[], Awaitable[None]]

class AgentFSM:
    def __init__(self):
        self.handlers: dict[Phase, Handler] = {}
        self.phase = Phase.ACQUIRE

    def on(self, phase: Phase):
        def register(fn: Handler): self.handlers[phase] = fn; return fn
        return register

    async def run(self):
```

```
        while self.phase:
            log.info("➡ %s", self.phase.name)
            await self.handlers[self.phase]()
            self.phase = Phase(self.phase.value + 1) \
                         if self.phase != Phase.LEARN else None
```

We will later see how each handler wraps in an OpenTelemetry span.

2. Response Ontology which go JSON-first

Brevity is good, but **structured brevity** is better.

```
# schema.py
from typing import Literal, Union
from pydantic import BaseModel, Field

class RAnswer(BaseModel):
    kind: Literal["answer"] = "answer"
    value: str              = Field(..., max_length=1_000)

class RCommand(BaseModel):
    kind: Literal["command"] = "command"
    shell: str

class RPatch(BaseModel):
    kind: Literal["patch"] = "patch"
    diff: str              = Field(..., max_length=20_000)

class RYesNo(BaseModel):
    kind: Literal["yesno"] = "yesno"
    value: bool

class RLoc(BaseModel):
    kind: Literal["loc"] = "loc"
    file: str
    line: int

AgentResponse = Union[RAnswer, RCommand, RPatch, RYesNo, RLoc]
```

This is useful in many ways:

- Enables IDE adapters to render rich UIs (diff viewers, terminal panes).
- Compresses tokens, every enum value is ~1 token instead of 5–15 words.

```
import orjson, tiktoken, asyncio, websockets

enc = tiktoken.encoding_for_model("gpt-4o-mini")

async def stream_response(ws_uri:str, resp:AgentResponse):
    raw = orjson.dumps(resp.model_dump())
    # optional compression pass
    tokens = enc.encode(resp.model_dump_json())
    if len(tokens) > 600:                  # arbitrary soft cap
        resp.value = summarise(resp.value)  # your compressor
        raw = orjson.dumps(resp.model_dump())
    async with websockets.connect(ws_uri) as ws:
        await ws.send(raw)
```

- Makes post-hoc analytics trivial (count kinds per session, latency, success rate).
- Ensure each vertical to ship templates tuned to local signal-to-noise:

```
@dataclass
class ResponseRules:
    max_char: int
    required: list[str]
    style: Literal["bullet", "table", "json"]

RULES = {
    "healthcare": ResponseRules(280, ["finding","next_step"], "bullet"),
    "legal":      ResponseRules(320, ["issue","citation","action"], "bullet"),
    "marketing":  ResponseRules(420, ["insight","metric","cta"], "table"),
}

def enforce_rules(resp:RAnswer, vertical:str):
    rule = RULES[vertical]
    assert len(resp.value) <= rule.max_char
    for slot in rule.required:
        assert slot in resp.value.lower()
```

As a tip, you should call `enforce_rules` in **Verify** phase.

If it fails, bounce to **Reflect** for auto-shrink or slot repair.

3. Phase ⇄ Task spectrum

Claude exposes **low-level todos**; *Gemini* exposes **high-level phases** (Understand → Plan → Implement → Verify) and asks for user approval before entering *Act* mode.

This is important because busy users want **one glance**: "what phase are we in?", and engineers want **fine-grained telemetry**: which micro-step failed?

Let's think of a data model for a second that will help us expose such information:

```
# state.py
from __future__ import annotations
from enum import Enum, auto
from collections import deque
from dataclasses import dataclass

class Phase(Enum):
    UNDERSTAND = auto(); PLAN = auto()
    IMPLEMENT = auto(); VERIFY = auto()

@dataclass
class Task:
    id:  int
    txt: str
    status: str  # todo | doing | done

@dataclass
class AgentState:
    phase : Phase
    tasks : deque[Task]

    def as_dict(self):                # for front-end render
        return {"phase": self.phase.name.lower(),
                "tasks": [t.__dict__ for t in self.tasks]}
```

We can now render `AgentState` in the sidebar, let the user tick tasks or reorder them to get free RLHF signals on the plan quality.

```
# websocket_feed.py
import json, websockets, asyncio
from agent.state import AgentState

async def push_state(ws_uri:str, state_q:asyncio.Queue[AgentState]):
    async with websockets.connect(ws_uri) as ws:
```

```
            while (st := await state_q.get()):
                await ws.send(json.dumps(st.as_dict()))
```

Your front-end (React / Vue) simply re-renders on each payload, and dragging a card emits `PATCH /agent/task/{id}` which you funnel back into the queue → **RLHF reward = Δordering.**

This way, you can also emit a **typed todo list** that maps 1-to-1 to graph nodes:

```
[ ] gather_lab_results          ← Acquire-Context
[~] compare_to_guidelines       ← Act   (runtime)
[ ] cross-check drug doses      ← Verify
[ ] draft SOAP note             ← Deliver
```

Each DAG node **owns** exactly one `Task.id`:

```
graph node                sidebar line
──────────────            ──────────────────────────
ctx.fetch_lab  ──────►    [ ] gather_lab_results
act.compare    ──────►    [~] compare_to_guidelines
```

So when the executor marks a node *done*, the same event ticks the UI **in real time,** zero extra plumbing.

4. Convention & Dependency Heuristics

Reviewers down-vote PRs that break their lint rules.

Courts throw out briefs written in the wrong citation format.

Hospitals reject notes that violate EHR templates.

That why ALL assistants stress “follow the repo’s style”, for example:

```
def infer_conventions(root:Path)->Conventions:
    deps = parse_deps(root)            # req*.txt, package*.json
```

```
    exts = Counter(f.suffix for f in root.rglob('*') if f.is_file())
    lints = detect_linters(root)       # eslint, flake8, go vet …
    fmt   = detect_formatters(root)    # prettier, black, clang-format
    return Conventions(deps, exts, lints, fmt)
```

They call this **before** planning patches.

To apply this to domain specific contexts, you should treat “style” as structured data. Here’s what the pipeline will look like:

1. **Bootstrap** with a hundred real artefacts (contracts, discharge notes, ad briefs).
2. Extract latent patterns → `spacy`, `textstat`, embedding clustering.
3. Materialise as **JSON Schemas** the model must fill.
4. Feed those schemas to the function-calling layer.

```
# conventions.py
import re, json, subprocess, pathlib
from collections import Counter
from typing import NamedTuple
from gensim.models import Phrases
import spacy, textstat, jsonschema, json, pathlib

class Conventions(NamedTuple):
    deps: list[str]
    exts: dict[str, int]
    linters: list[str]
    formatters: list[str]

def parse_deps(root:pathlib.Path)->list[str]:
    dep_files = list(root.glob("**/requirements*.txt")) + \
                list(root.glob("**/package*.json"))
    pkgs = []
    for f in dep_files:
        pkgs += re.findall(r"^[A-Za-z0-9_\-]+", f.read_text(), re.M)
    return sorted(set(pkgs))

def detect_tools(root:pathlib.Path, patterns:list[str]):
    return [p for p in patterns if (root / p).exists()]

def infer_conventions(root:pathlib.Path)->Conventions:
    return Conventions(
        deps=parse_deps(root),
        exts=Counter(p.suffix for p in root.rglob("*") if p.is_file()),
```

```
        linters=detect_tools(root, ["flake8", ".eslintrc", ".golangci.yml"]),
        formatters=detect_tools(root, [".prettierrc", "pyproject.toml",
                                       ".clang-format"])
)


def schema_from_corpus(files:list[pathlib.Path])->dict:
    docs = [f.read_text() for f in files]
    # 1. terminology via n-gram mining
    bigram = Phrases([d.split() for d in docs]).export_phrases([[]])[1]
    # 2. length/statistics constraints
    avg_len = int(sum(map(len, docs))/len(docs) * 1.2)
    # 3. build schema
    return {
        "type":"object",
        "properties":{
            "body":{"type":"string",
                    "maxLength":avg_len,
                    "pattern":"|".join(bigram)}
        },
        "required":["body"]
    }

# usage
func_spec = {"name":"draft_brief","schema":schema_from_corpus(corpus)}
resp = llm.chat(model="gpt-4o-mini", functions=[func_spec])
jsonschema.validate(resp["arguments"], func_spec["schema"])   # hard fail if of
```

This auto-learns thousand-line firm-specific clause banks or hospital abbreviations without human tagging.

The result is the model literally *cannot* produce off-brand copy. No manual review loops.

> *We publish "how-to" guides and thought pieces for building agents.*
>
> *We pour our **passion, expertise, and countless hours into creating content** that we believe can make a difference in your journey.*
>
> *But only 1% of our readers follow or engage with us on Medium.*
>
> ***If you ever found value in our content**, it would mean a lot if you could **follow Agent Native on Medium**, give this article a clap, and drop a hello in the comments!*

> *It's a small gesture but it tremendously helps us deliver much better content and guides for you!*
>
> *Thank you for taking your time to be here, we* really appreciate it.

5. Hard Safety Boundaries

Gemini requires an explicit allow/deny for any mutating tool; read-only ops are auto-approved.

Codex is executed in a directory-confined sandbox with networking disabled.

Mirror this with a **kernel-level policy layer:**

```
bwrap --ro-bind /project /workspace \
      --tmpfs /tmp           \
      --unshare-net          \
      --proc  /proc          \
      --dev   /dev
```

They probably expose a `sandox_info()` syscall so the model can adapt commands (e.g., avoid `sudo`, write to `/tmp`).

To apply this to domain-specific context, you should stop relying on "don't jailbreak" strings, and apply the **Sandboxed-Mind patterns** instead.

- **Action-Selector is a** narrow toolset (e.g., EHR read-only) which is safest but least flexible
- **Plan-Then-Execute** is for complex pipelines with extra latency
- **Dual-LLM** is untrusted input (email, web) but doubles cost
- **Code-Then-Execute** is for financial / infra ops which requires runtime sandbox

For example, a typical `policy.yaml` for healthcare agent would look like:

```
allowed_files:
  - /workspace/patient/*.json
```

```
blocked_patterns:
  - "SELECT .* FROM transactions"
```

The verifier reads this file and kills any command that violates the regex.

6. Verification Pipeline

Gemini's docs recommend running linters & tests **every loop** but community tips insist on "add unit tests and then call them".

For example:

```
async def verify():
    await run("npm run lint --silent")
    await run("pytest -q")
    await run("mypy . || true")  # soft type-check
```

If any step fails, you can summarize the diff / traceback and re-enter **Plan** with the failure context.

In healthcare context, you can similarly add domain guards before DELIVER phase:

```
async def medical_guard(plan:dict) -> None:
    schema.validate(plan)                          # JSONSchema
    if await async_gpt_check("off-label-use", plan):
        raise ValueError("Off-label recommendation.")
    await guidelines_lut.assert_compliant(plan)
    if plan["risk_score"] > 0.7:
        raise HumanReviewNeeded()
```

Codex-derived agents embed similar guards for package-manager writes and Gemini 2.0's doc-extraction recipe externalises rule tables so analysts edit Excel, **not** prompts.

7. Memory Model

Memory and context management in agentic applications is very important due to several factors

- **Latency vs. recall:** Flat vector stores grow unbounded and slow.
- **Regulation:** GDPR/AI Act force *purpose limitation* & *storage minimisation*.
- **Prompt-token tax:** Every extra byte re-ingested costs cash and latency

Which makes “Pyramid Memory” suitable as a design pattern:

```
┌──────────────┐
│  L1 Cache    │  Scratchpad   (ephemeral CoT, cleared ↻ every step)
├──────────────┤
│  L2 Store    │  Session dict (Redis, 30 min TTL)
├──────────────┤
│  L3 RAG DB   │  Vector DB    (docs, past tickets) - per-repo / per-tenant
├──────────────┤
│  L4 Lake     │  Long-term    (Parquet/S3, analytics only, opt-in retention)
└──────────────┘
```

We basically persist upward, search downward.

Here’s a very rough implementation:

```python
# memory.py
from typing import Iterable, Any
from chromadb import Client as Chroma
import redis, json, time, uuid

class PyramidMemory:
    def __init__(self, redis_url:str, chroma_url:str, ttl:int=1800):
        self.l2 = redis.StrictRedis.from_url(redis_url)
        self.l3 = Chroma(path=chroma_url).get_or_create_collection("rag")
        self.ttl = ttl                          # seconds

    # ---------- public API ----------
    def recall(self, q:str, k:int=3) -> list[str]:
        if hit := self.l2.get(q):
            return json.loads(hit)
        docs, _ = self.l3.query(q, n_results=k)
        self.l2.set(q, json.dumps(docs), ex=self.ttl)
        return docs

    def persist(self, items:Iterable[str]):
        # Assume items already PII-scrubbed.
```

```
            ids = [str(uuid.uuid4()) for _ in items]
            self.l3.add(ids=ids, documents=list(items))
```

Here you can additional implement compliance hook to run `gdpr_purge()` nightly to evict expired L3 vectors:

```
def gdpr_purge(days:int=30):
    cutoff = time.time() - days * 86_400
    ids = [id_ for id_, meta in self.l3.get(include=["metadatas"]).items()
           if meta["timestamp"] < cutoff and not meta["retention_ok"]]
    self.l3.delete(ids)
```

8. Parallel & Speculative Tool Calls

Wall-clock time is dominated by *longest* branch of your task DAG (“critical path”), and most tool invocations are I/O-bound, that are perfect for `asyncio.gather`.

In such cases, speculative execution squeezes idle tokens, call *candidate* tools in parallel, and drop losers.

Let’s have a look at a simple code snippet to understand this better.

This is what a simple broker look like:

```
# broker.py
import asyncio, heapq
from typing import Awaitable

class Task:
    def __init__(self, name:str, coro:Awaitable, deps:set[str]):
        self.name, self.coro, self.deps = name, coro, deps

async def dispatch(tasks:list[Task]):
    graph = {t.name: t for t in tasks}
    ready = [t for t in tasks if not t.deps]
    running, done = set(), set()

    async def runner(t:Task):
        running.add(t.name)
        try:
            return await t.coro
        finally:
```

```
                running.remove(t.name)
                done.add(t.name)

        while ready or running:
            # Kick off all ready tasks
            awaitables = [runner(t) for t in ready]
            ready.clear()
            if awaitables:
                await asyncio.gather(*awaitables, return_exceptions=True)
            # Re-evaluate readiness
            for t in tasks:
                if t.name not in done and t.deps.issubset(done):
                    ready.append(t)
```

And here's speculative variant:

```
from contextlib import suppress

async def speculative(first:Awaitable, *maybes:Awaitable):
    # race tasks, cancel losers
    first_done, pending = await asyncio.wait(
        [first, *maybes], return_when=asyncio.FIRST_COMPLETED
    )
    for p in pending:
        p.cancel()
        with suppress(asyncio.CancelledError):
            await p
    return list(first_done)[0].result()
```

As a tip, you should log both **critical-path** and **wall-time**, auto-split a big task if CP ≪ wall-time.

9. Git-Aware Ops

Git-aware ops provide you a few advantages over vanilla setup:

- **Reproducibility**: you tie every change back to a hash.
- **Style consistency**: happier reviewers, lower diff noise.
- **Pre-commit**: last guard before dangerous code ships.

To achieve these three, you can implement the following workflow:

1. **Dry-run** (`git status --porcelain`) – bail if dirty working tree.
2. **Scope file list** → `.agent/scope.txt`.
3. Run static scans / policy checks on that list.
4. Let LLM generate commit message **template**, keepter edits it.
5. Auto-amend if LLM mutated files (stopgap till large refactor).

Here's a sample pre-commit hook:

```
#!/usr/bin/env python3
# .git/hooks/pre-commit
import subprocess, sys, json, re, pathlib

root = pathlib.Path(__file__).resolve().parents[2]

def shell(*cmd): return subprocess.check_output(cmd).decode()

files = shell("git diff --cached --name-only").splitlines()
(root/".agent").mkdir(exist_ok=True)
(root/".agent/scope.txt").write_text("\n".join(files))

# --- compliance checks ---
from agent.policy import scan_files
violations = scan_files(files)
if violations:
    print("[ABORT] policy violations:")
    for v in violations: print(" •", v)
    sys.exit(1)

# --- optional LLM amend ---
if "--fix-style" in sys.argv:
    from agent.llm import style_fix
    style_fix(files)
    shell("git add -u")      # re-stage
```

and a auto-commit message generator:

```
def commit_message(diff:str) -> str:
    prompt = f"Write an imperative commit title (≤50 chars) " \
             f"and bullet body for the diff:\n{diff}\n"
    msg = llm_chat(prompt, model="gpt-4o-mini")
```

```
    # enforce style
    title, *body = msg.strip().splitlines()
    title = re.sub(r"[.]$", "", title).capitalize()
    return title + "\n\n" + "\n".join(body)
```

10. Observability Hooks

As you have noticed, we have many layers to instrument

Good news is we can easily instrument all of these layers with OpenTelemetry:

```
# observability.py
from opentelemetry import trace
from functools import wraps
tracer = trace.get_tracer("agent")

def span(name:str, **attrs):
    def deco(fn):
        @wraps(fn)
        async def wrapper(*a, **kw):
            with tracer.start_as_current_span(name) as sp:
                for k, v in attrs.items(): sp.set_attribute(k, v)
                sp.set_attribute("args.len", len(a)+len(kw))
                return await fn(*a, **kw)
        return wrapper
    return deco


# usage
```

```
@span("agent.tool_call", tool="ehr.fetch")
async def fetch_ehr(id_:str): ...
```

And a few dashboarding tips will be handy here:

- **Gantt by phase** chart (span duration stacked) will catch slow tools.
- **Guard-trip heatmap** over time will show drift/regressions.
- Pipe traces to **Grafana Cloud** or **Honeycomb**, alerts on p95 latency ↑20 %.

Reference Implementation Skeleton

Cool, if you have come this far, congratulations!

Lastly, let's have a look a simple skeleton that you can use as a reference:

```
# agent/__init__.py
from __future__ import annotations
from pathlib import Path
from collections import deque
from enum   import Enum, auto
from typing import Any

class Phase(Enum): OBSERVE=auto(); PLAN=auto(); ACT=auto()
                   VERIFY=auto(); REFLECT=auto(); RESPOND=auto()

class State:
    def __init__(self): self.phase=Phase.OBSERVE; self.tasks=deque()

class Agent:
    def __init__(self, repo:Path):
        self.repo   = repo
        self.state  = State()
        self.ctx    = infer_conventions(repo)
        self.memory = PyramidMemory(
            redis_url="redis://localhost:6379/0",
            chroma_url=str(repo/".agent/chroma")
        )

    # ----------- top-level step -----------
    async def step(self, user:str) -> str:
        obs   = await Observer(self).run(user)
        plan  = await Planner(self).run(obs)
        acts  = await Actor(self).run(plan)
        ok    = await Verifier(self).run(acts)
        await Reflector(self).run(ok, acts)
```

```
            return await Responder(self).run()

    # ------ example plugin: Planner ----------
    class Planner:
        def __init__(self, agent:Agent): self.agent=agent

        async def run(self, obs:Any) -> Plan:
            prompt = make_plan_prompt(obs, self.agent.ctx)
            raw    = await llm_chat(prompt, model="gpt-4o-mini")
            plan   = parse_plan(raw)                # to DAG
            self.agent.state.tasks = deque(topo_sort(plan))
            return plan
```

Having said all of that, there is always a fine balance between shipping and optimizing and that's where you need to decide the trade-off and pull the trigger based on user requirements and go-live timeline.

That's it, hope you enjoyed the post.

Make sure you drop your own patterns in the comments too!

Bonus Content : Building with AI

And don't forget to have a look at some practitioner resources that we published recently:

Reverse Engineering Anthropic's Agent Blueprint to Outperform Claude Opus 4 By 90%

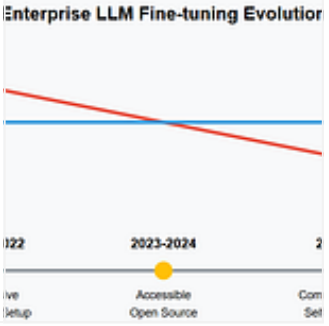
End-to-end implementation of Anthropic's blueprint

agentissue.medium.com

LLM Fine-Tuning Strategy: 4 Open Source Toolkits That You Should Know

With everything at our fingertips today, fine-tuning large language models (LLMs) can get overwhelming fast.

agentissue.medium.com



Thank you for stopping by again, see you around.

Claude Code | Gemini Cli | Agents | Agentic Ai | Agentic Workflow

Follow

Written by Agent Native

3.2K followers · 0 following

Your front-row seat to the future of Agents.

No responses yet

Bgerby

What are your thoughts?

More from Agent Native

Agent Native

LangChain and LangGraph v1.0: Beyond Release Notes, Into Real ROI

Today, I'm not here to recap release notes. I'll share what actually works across industries, and how it maps to the new features in v1.0.

4d ago 6

Agent Native

Anthropic Teams' 100x Claude Workflows: What They Didn't Tell You, Until Now

The following is a battle-tested, developer-to-developer walkthrough on turning Claude into a genuine teammate rather than "just another...

Jun 8 388 12

Agent Native

DeepSeek-OCR: Context Compressor for Enterprise Agents

I went into this release expecting “another OCR,” then had one of those brain-tilt moments where your mental model updates in real time.

2d ago 1

Agent Native

Reverse Engineering Anthropic’s Agent Blueprint to Outperform Claude Opus 4 By 90%

End-to-end implementation of Anthropic’s blueprint

Jun 21 311 1

See all from Agent Native

Recommended from Medium

In Towards AI by Teja Kusireddy

We Spent $47,000 Running AI Agents in Production. Here’s What Nobody Tells You About A2A and MCP.

Multi-agent systems are the future. Agent-to-Agent (A2A) communication and Anthropic’s Model Context Protocol (MCP) are revolutionary. But...

Oct 16 1.6K 37

In Coding Nexus by Code Coup

Claude Desktop Might Be the Most Useful Free Tool You’ll Install This Year

I didn't expect much when I first saw the announcement for Claude Desktop. Another AI wrapper, I thought. Maybe with a shiny UI.

Oct 23 297 12

In Dare To Be Better by Max Petrusenko

Unleash Your Inner Wizard: Claude Skills Are Automating Enterprise for Pennies

How a simple folder is replacing $50K consultants and saving companies literal days of work

Oct 17 452 6

In AI Software Engineer by Joe Njenga

Cursor 2.0 Has Arrived — And Agentic AI Coding Just Got Wild

Cursor has released version 2.0 , bringing the most powerful agentic AI we have seen yet, more autonomous than ever before,here's what's...

2d ago 222 5

Agen.cy

20+ Genius Ways Power Users Are Using Claude Code Right Now

Here are 10+ ways power users are using Claude Code🧵

Oct 23 33 1

Manojkumar Vadivel

The .claude Folder: A 10-Minute Setup That Makes AI Code Smarter

If you’re new to Claude Code, it’s a powerful AI coding agent that helps you write, refactor, and understand code faster. This article...

Sep 15 72 2