Agentic Design Patterns (ADP) by Michaël Bettan

Definition: Agentic design patterns are reusable, vendor-neutral architectural blueprints that govern how an LLM-based system decomposes work, selects control flow, invokes tools, manages context memory, coordinates across specialized workers, and defers high-stakes decisions to deterministic code or human reviewers. They apply identically across Google ADK, the Claude Agent SDK, LangGraph, or direct model API loops.

What agentic design patterns solve for: A raw language model is a stateless, probabilistic token predictor—it can *propose* an action, but it cannot *execute* tools, enforce hard business invariants, persist session state, or halt a runaway loop. Architectural patterns define **who owns the control flow** (deterministic application code vs. dynamic model reasoning), **where authority sits** (code gates vs. model judgment vs. human sign-off), and **how context stays clean** as tasks scale.

> MENTAL MODEL
>
> The model is the **probabilistic reasoning engine**; the architecture is the **deterministic harness** (nervous system, brakes, and safety interlocks) that decides when the model thinks, when tools execute, and when a human must approve.

> RULE 0 (THE SIMPLICITY LADDER)
>
> **Start with the simplest architecture that passes your evaluation suite.** Climb from deterministic code → a single augmented LLM call → a predefined workflow → an autonomous agent **only when empirical evaluations prove the simpler tier fails.**

LLM vs. Workflow vs. Autonomous Agent vs. Harness

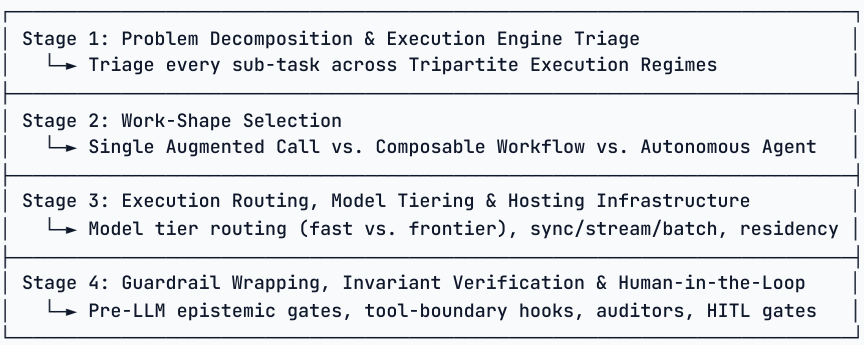
Layer	Control Owner	Role & Runtime Behavior	Memory Hook
Raw LLM	Caller (1 turn)	Stateless token predictor; reasons over prompt context and emits text or structured tool requests, but never executes code.	*"Brain in a jar"*
Augmented LLM	Caller (1 turn + tools)	Single model call equipped with **Retrieval** (RAG), **Tools** (MCP/functions), and **Memory** (read/write state). Handles most production features.	*"LLM with a backpack"*
Composable Workflow	**Application code**	Multi-step pipeline where LLMs and tools execute along **predefined code paths** with deterministic gates between steps.	*"Railway tracks"* — train cannot leave the rails
Autonomous Agent	**The model**	LLM dynamically directs its own multi-turn process, choosing which tools to call and when to stop based on environmental ground truth.	*"Taxi"* — driver picks the route dynamically
Agent Harness / Framework	Infrastructure	Runtime layer (ADK, Claude Agent SDK, LangGraph) supplying the event loop, state persistence, sandboxing, callbacks, and observability.	*"Chassis, brakes & dashboard"*

> CLASSIC ARCHITECTURAL TRAP
>
> Calling an LLM five times in a Python script does **not** make the system an "autonomous agent." If application code dictates the sequence, branching, and retry count, it is a **workflow**. It becomes an **agent** only when the model itself decides the next step and action trajectory at runtime.

01. Architectural Triage & Sequential Decision Framework

Before writing a single system prompt or selecting an orchestration framework, an architect must resolve four sequential decisions. Skipping step 1 or 2 is the primary cause of over-engineered, brittle, or unsafe agentic deployments.

Sequential Architectural Decision Framework



1. **Stage 1 — Problem Decomposition & Execution Engine Triage:** Separate the problem into deterministic rules, probabilistic reasoning, and human accountability using the **Tripartite Execution Regimes** below.
2. **Stage 2 — Work-Shape Selection:** Select the lowest rung on the complexity ladder (Single Call → Workflow → Autonomous Agent) using the **Architectural Evaluation Criteria** (§02).
3. **Stage 3 — Execution Routing, Model Tiering & Hosting Infrastructure:** Match each step to the right model tier and reasoning effort (e.g., high-capability planner producing a spec for a fast/balanced executor), transport mode (streaming vs. async batch), and regional data boundary.
4. **Stage 4 — Guardrail Wrapping, Invariant Verification & Human-in-the-Loop Controls:** Surround every probabilistic step with deterministic pre-checks, schema validators, tool-boundary hooks, independent critique, and Human-in-the-Loop (HITL) checkpoints.

Tripartite Execution Regimes: Deterministic Code, Probabilistic Inference & Human Oversight

Every enterprise workflow is a hybrid of three distinct execution regimes. Never assign Regime 1 or Regime 3 responsibilities to Regime 2:

Execution Regime	Execution Engine	What Belongs Here	Canonical Examples	Why Other Regimes Fail
Regime 1: Deterministic Code	Plain application code, SQL, regex, type schemas, policy hooks	Exact arithmetic, schema validation, RBAC & permission enforcement, idempotency keys, database writes, PII redaction, hard policy gates	Fee & tax calculations, date math, checking `amount ⩽ 500`, enforcing `user_id` tenant scoping, verifying cited section headers exist in source text	LLMs introduce **deterministic invariant degradation** (silent tail-risk arithmetic or rule violations); humans are too slow and inconsistent at scale
Regime 2: Probabilistic Model (LLM)	Augmented LLM, Workflow node, or Agent loop	Unstructured synthesis, semantic classification, ambiguous entity extraction, fuzzy matching, multi-step planning over open-ended inputs	Summarizing conflicting legal clauses, triaging free-text support tickets, diagnosing novel stack traces across a codebase, drafting code diffs	Rigid code/regex shatters on phrasing variations; humans cannot manually read 50,000 documents per hour
Regime 3: Human-in-the-Loop (HITL)	Human reviewer via durable approval queue / UI checkpoint	High-blast-radius irreversible actions, novel policy exceptions, regulated sign-offs, subjective value trade-offs	Approving a `$25,000` wire refund, executing a production database drop, granting an exception to legal policy, merging security-critical PRs	Code lacks contextual judgment for novel exceptions; models cannot bear legal, financial, or ethical accountability

Hard Invariant Enforcement in Code (Preventing Deterministic Drift)

Deterministic drift occurs when an engineering team delegates exact, rule-bound logic—such as calculating pro-rated subscription refunds, enforcing role-based access control (RBAC), checking date eligibility windows, or redacting Social Security Numbers—to a system prompt instead of application code.

- **Why it is dangerous:** Even when a frontier model follows a prompt rule on the vast majority of test runs, probabilistic token sampling guarantees a non-zero tail-risk failure rate—especially under **instruction crowding** (long prompts with 20+ competing rules) or adversarial prompt injection.
- **Architectural rule ("Deterministic Handcuffs"):** Let the LLM extract structured parameters or propose an action, but execute all math, eligibility comparisons, permission checks, and compliance gates in **deterministic code** (`PreToolUse` / `before_tool_callback` hooks, Pydantic validators, or SQL constraints).

02. Work-Shape Spectrum & Architectural Evaluation Criteria

Once Regime 2 (the probabilistic work) is isolated, choose between the three primary work shapes: **Single Augmented LLM Call**, **Composable Workflow**, and **Autonomous Agent**.

Architectural Evaluation Criteria: Single Call vs. Workflow vs. Autonomous Agent

Evaluation Criterion	Single Augmented LLM Call	Composable Workflow (DAG / Chain)	Autonomous Agent (Open Loop)
1. Task Path Predictability	**Fully known (1 step):** Input maps directly to output with retrieved context or a single tool lookup.	**Known decomposition (*K* steps):** Multi-step process where the sequence, branches, or fan-out categories are fixed in advance.	**Unknown / emergent (*N* steps):** Step $N+1$ depends entirely on what tool observation N revealed (e.g., debugging an unknown bug).
2. Latency Budget	**Sub-second to ~3 s:** Single model roundtrip; suitable for synchronous inline UI responses.	**~3 s to ~15 s:** Predictable bounded latency (*K* sequential stages or 1 parallel fan-out + join).	**Seconds to minutes+:** Open-ended $O(N)$ sequential turns; requires async streaming, progress UI, or background execution.
3. Error Blast Radius & Reversibility	**Low / read-only:** Output is displayed to a user or validated by a caller before any state mutation.	**Medium / gated:** Intermediate code gates catch errors between stages before downstream side-effects run.	**High if ungated:** Multi-turn tool mutations can compound; requires sandboxing, checkpoints, and HITL gates on writes.
4. Token Cost Ceiling	**~1× baseline:** Minimal prompt overhead; highest prompt-cache hit ratio.	**~3–5× baseline:** Bounded multiplier across *K* focused prompts; easy to route cheap tiers to simple nodes.	**~10–15×+ baseline:** Context accumulates across turns and tool outputs; requires strict token/turn caps and compaction.
5. Auditability & Compliance Rigor	**High:** Single input/output pair; trivial to log and reproduce.	**Highest for multi-step:** Deterministic transitions and explicit code gate pass/fail logs satisfy strict regulatory audits.	**Complex:** Nondeterministic trajectories require full event-ledger tracing (`tool_trajectory`) and replay harnesses.

Inherent Operating Characteristics of Probabilistic Systems

Every agentic architecture must explicitly defend against four inherent properties of language models:

1. **Non-determinism:** Identical prompts can yield different phrasings or tool trajectories across runs (and modern frontier APIs reject non-default sampling knobs like `temperature: 0` or still exhibit floating-point variance). *Mitigation:* Grade structured properties and behavioral rubrics, never exact string equality; enforce JSON schemas at the token level.
2. **Plausible-sounding errors (Hallucinations & Silent Corruption):** Models are trained to complete patterns helpfully. When fed garbled OCR text, an empty string `""` from a timed-out tool, or a missing policy clause, an unguarded model will invent confident facts or fabricated citations (`Section 99.1`). *Mitigation:* Pre-LLM epistemic quality gates, explicit `is_error: true` tool results, and deterministic citation verification against source text.
3. **Context sensitivity (Attention Dilution & Instruction Crowding):** Packing 30 rules, 15 irrelevant documents, and 40 tool schemas into one call degrades recall in the middle of the window and causes the model to silently drop late audit rules. *Mitigation:* Split generation from verification, keep tool catalogs ≤ 16 tools per agent, and isolate noisy exploration in subagents.
4. **Compounding multi-step error rates:** If a single autonomous step has a 95% reliability rate, a 10-step unverified chain succeeds only $0.95^{10} \approx 60\%$ of the time, because an early misread file or wrong assumption poisons every subsequent reasoning turn. *Mitigation:* Insert **ground-truth environmental feedback** (compilers, unit tests, schema validators, deterministic gates) after every step to reset the error curve.

03. The 5 Canonical Workflow Patterns

When a single call falls short because of instruction crowding, multi-stage dependencies, or latency/cost constraints—but the overall control structure is predictable—use one or more of the **five composable workflow patterns**. In all five, **application code owns the control flow** (or bounds the loop).

Pattern	Control Flow Shape	Primary Optimization	Canonical Use Case	Memory Hook
1. Prompt Chaining	`LLM 1 ⟶ [Code Gate] ⟶ LLM 2 ⟶ LLM 3`	Accuracy via focused sub-steps + early fail-fast	Drafting an outline → verifying structure in code → generating full localized report	*"Assembly line with quality inspectors"*
2. Routing	`Classifier ⟶ Specialist Handler A | B | C`	Separation of concerns & per-tier cost optimization	Directing billing, refund, and tech-support tickets to distinct prompts, tools, and model tiers	*"Hospital triage desk"*
3. Parallelization	`Fan-Out (N calls) ⟶ Programmatic Aggregator`	Wall-clock speed (*Sectioning*) or high confidence (*Voting*)	Running content safety screening in parallel with answer generation (*Sectioning*); multi-prompt code security audit (*Voting*)	*"Sectioning = slice the pizza; Voting = poll 5 judges"*
4. Orchestrator–Workers	`Orchestrator LLM ⟶ Dynamic Workers (1..K) ⟶ Synthesizer`	Runtime task decomposition when sub-tasks depend on input	Multi-file refactoring or multi-source research where the target files/sources aren't known upfront	*"Tech lead assigning tickets after reading the spec"*
5. Evaluator–Optimizer	`Generator ⟶ Critic/ Auditor ⟶ (Pass | Feedback Loop [max M])`	Iterative quality & compliance against a strict rubric	Legal clause extraction + adversarial citation auditor; literary translation refinement	*"Writer + strict editor"*

Pattern 1: Prompt Chaining (Sequential Pipeline with Code Gates)

Decomposes a complex task into a fixed sequence of narrower LLM calls, where each step consumes the validated output of the previous step.

Pipeline Stage	Component	Functionality	Invariant / Exit Condition
Step 1: Extraction	Extraction LLM	Extracts structured clauses and cited line anchors from source text.	Emits structured JSON matching schema.
Deterministic Code Gate	Application Code	Validates mandatory fields (`termination_clause`) and citations list.	If missing or invalid, halts execution immediately ($0 downstream spend).
Step 2: Risk Assessment	Assessment LLM	Analyzes validated clauses against risk rubrics.	Emits final structured risk assessment report.

- **The Gate (`gate_check`):** Always insert a deterministic code check (schema validation, regex citation lookup, length/completeness check) between steps so corrupted intermediate outputs fail fast for `$0` rather than poisoning downstream calls.
- **When NOT to use:** Do not chain steps that have no data dependency (use *Parallelization* instead to avoid serial latency) or tasks that a single well-structured prompt already solves cleanly.

Pattern 2: Routing (Intent & Complexity Dispatch)

Classifies an incoming request (via a fast small model, embedding similarity, or deterministic rules) and dispatches it to a specialized prompt, scoped toolset, or appropriate model tier.

- **Why routing beats a monolith prompt:** Optimizing a single mega-prompt to handle billing disputes, technical debugging, and account cancellations simultaneously causes instruction clash and forces every call to pay for the largest model tier and full tool catalog.
- **Model-tier routing:** Route high-volume, routine intents to a fast, low-cost tier (e.g., Haiku / Gemini Flash-Lite) with minimal reasoning effort, and reserve frontier tiers (Sonnet / Opus / Gemini Pro) for complex multi-step troubleshooting.

- **Preventing misrouting cascades:** Include an explicit `unclear / ambiguous` fallback category that asks a clarifying question or routes to a general handler, log confidence scores, and validate that the target handler rejects out-of-domain requests rather than guessing.

Pattern 3: Parallelization (Sectioning vs. Voting)

Executes multiple LLM calls concurrently (`asyncio.gather` / `ParallelAgent`) and merges their outputs programmatically or via a downstream synthesizer.

Variant	Architecture	Goal	Key Engineering Rule
Sectioning	Split a task into **distinct, independent subtasks** or personas run concurrently.	**Speed & focused attention:** Total latency equals the slowest branch rather than the sum of branches.	Keep each branch's output key isolated (`output_key="market"`, `output_key="pricing"`) so concurrent workers never overwrite shared mutable state. Running guardrails/PII screening in parallel with response generation avoids adding serial latency.
Voting	Run the **same task multiple times** (either identical prompt at $T > 0$ for *Self-Consistency CoT*, or diverse critique prompts) and aggregate via threshold/majority vote.	**High confidence & recall:** Reduces false negatives on high-stakes reviews.	Tune the vote threshold to match asymmetric risk: e.g., in a security vulnerability scan, flag the diff if **any 1 of 3** reviewers finds an exploit (high recall); in noisy classification, require **2 of 3** agreement (high precision).

Pattern 4: Orchestrator–Workers (Dynamic Task Decomposition)

A central **Orchestrator LLM** inspects the input, dynamically invents a list of subtasks (because the exact number and nature of subtasks cannot be hardcoded in advance), dispatches **Worker LLMs** with scoped context, and synthesizes their returned summaries.

- **Orchestrator–Workers vs. Parallelization:** Both fan out work to multiple calls, but in **Parallelization** the branches are **predefined in code** (e.g., always run Financial, Legal, and Technical checks), whereas in **Orchestrator–Workers** the branches are **determined dynamically at runtime by the orchestrator model** (e.g., inspecting a bug report and deciding which 4 out of 2,000 repository files need targeted worker edits).
- **Spec-Driven Planner → Executor split:** A high-capability model acts as the read-only orchestrator/planner to resolve ambiguities and emit a structured task specification, then dispatches mid-tier or fast-tier workers in fresh context windows to execute each bounded subtask.

Orchestration Phase	Execution Engine	Mechanism	Guardrail
Phase 1: Dynamic Decomposition	Orchestrator LLM	Generates structured task plan based on runtime input inspection.	Strict Pydantic task schema (`subtasks: list[SubTask]`).
Phase 2: Scoped Worker Fan-Out	Parallel Worker LLMs	Workers execute bounded subtasks in fresh, isolated context windows.	Bounded concurrency waves (3–5 workers max) with private outputs.
Phase 3: Context Synthesis	Synthesizer LLM	Merges worker `HandoffContract` summaries into unified deliverable.	Artifact pointer pattern: large diffs stay on disk/storage.

Pattern 5: Evaluator–Optimizer (Bounded Generator–Critic Loop)

Pairs a **Generator** call with an independent **Evaluator / Critic** call that grades the draft against an explicit rubric and returns actionable revision feedback in a loop.

- **Two prerequisites for fit:** (1) Output quality measurably improves when a human provides structured feedback, and (2) an LLM (paired with deterministic checks) can reliably articulate that feedback.
- **Separation of Authority (Avoiding Self-Grading Bias):** Never ask the same prompt in the same turn to both draft an answer and grade its own compliance—it will rationalize its own mistakes. Use a separate critic call with a cold context (`include_contents='none'`), an adversarial rubric prompt, and deterministic verification tools (e.g., checking that every cited clause actually appears in the source document).
- **Mandatory loop hygiene:** Always enforce `max_iterations` (typically 2–3 rounds) and an explicit convergence/exit signal (`escalate=True` / `status="approved"`). If `max_iterations` trips without passing the rubric, do **not** silently ship the unverified draft—mark the state unapproved and escalate to HITL or a safe fallback.

Loop Phase	Agent / Mechanism	Inputs & Context	Exit / Escalation Criteria
Generation	Generator LLM	Task brief + prior critic feedback (if any).	Emits structured draft with cited section headers.
Pre-Critic Gate	Deterministic Code	Source text + draft citations.	Deterministically checks that cited headers exist in source text; fails fast if missing.
Evaluation	Cold-Context Critic LLM	Source text + draft + rubric (`include_contents='none'`).	Emits `approved: bool` and actionable critique. Loop terminates if `approved == True`.
Termination / Fallback	Deterministic Guard	Iteration counter (`attempt ⩽ max_iterations`).	If `attempt > max_iterations` without approval, fails closed to HITL escalation.

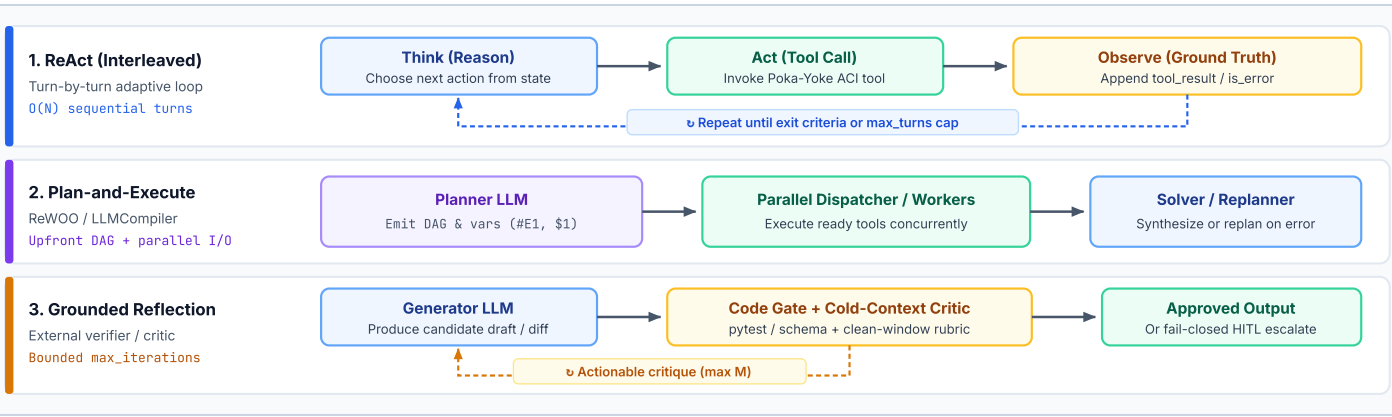
Cross-Vendor Pattern Vocabulary

Architectural Pattern	Anthropic / General Framing	Google Cloud / ADK Primitive	Graph / State Machine Framing
Single Building Block	Augmented LLM (Retrieval + Tools + Memory)	`LlmAgent` with `tools` + `MemoryService`	Single node with tool bindings
Pattern 1: Sequential	Prompt Chaining + Programmatic Gate	`SequentialAgent` / `Workflow` + `before_agent_callback`	Linear edges + conditional gate node
Pattern 2: Dispatch	Routing (Intent / Tier Classifier)	Coordinator (`sub_agents`) or custom router node	Conditional edge router from `START`
Pattern 3a: Split Work	Parallelization — Sectioning	`ParallelAgent` (distinct `output_key` s) + synthesizer	Fan-out branches → `JoinNode`
Pattern 3b: Consensus	Parallelization — Voting / Self-Consistency	`ParallelAgent` (*N*× runs) + vote aggregator	Parallel ensemble → majority/threshold reducer
Pattern 4: Dynamic	Orchestrator–Workers	`LlmAgent` invoking specialist `AgentTool` s	Dynamic map-reduce / send API
Pattern 5: Refinement	Evaluator–Optimizer	`LoopAgent(max_iterations=N)` + generator & critic	Cyclic edge bounded by iteration counter
Open-Ended Autonomy	Autonomous Agent (Tools in a Loop)	`LlmAgent` native ReAct loop (`Runner`)	Cyclic `agent` ⇄ `tools` graph
Human Checkpoint	Planned Checkpoint / HITL Approval	`FunctionTool(..., require_confirmation=True)`	`interrupt()` / `RequestInput` with checkpointer

04. Autonomous Agent Primitives & Reasoning Loops

When the task path is genuinely unpredictable—because step *N* + 1 depends on discovering what happens in step *N* (e.g., diagnosing a production outage, resolving an unfamiliar GitHub issue across a codebase, or navigating dynamic web workflows)—an **Autonomous Agent** takes control of the loop.

Reasoning Loop Paradigms: ReAct vs. Plan-and-Execute vs. Reflection



Reasoning Paradigm	Execution Mechanics	Strengths	Failure Mode & When to Avoid
ReAct (Reason + Act)	Interleaves **Thought** → **Action** (tool call) → **Observation** (tool result) turn-by-turn.	Maximum adaptability; each action is grounded in the immediate observation from the previous step.	High sequential latency (*O*(*N*) LLM turns); context grows every turn; vulnerable to local rabbit holes without a high-level checklist.
Plan-and-Execute (incl. ReWOO & LLMCompiler)	**Planner** generates an upfront multi-step plan (or DAG of symbolic variables `#E1` , `$1`); **Executor / Dispatcher** runs steps (often in parallel); **Replanner / Solver** synthesizes or adjusts on failure.	Slashes latency and token re-ingestion (ReWOO uses 2 LLM calls + parallel I/O; LLMCompiler eagerly dispatches DAG nodes as dependencies resolve).	Brittle if early tool observations invalidate the entire upfront plan; requires dynamic replanning hook when the environment shifts.
Reflection / Self-Correction	Agent critiques its own draft or observes a deterministic failure (compiler error, test failure, schema error) and revises before emitting the final answer.	Catches syntax, schema, and logic defects automatically when grounded in external test/validator output.	**Ungrounded reflection drift:** Reflecting purely on internal prose without external ground truth (tests, linters, search) often degrades accuracy or reinforces hallucinations.

Stopping Conditions & Loop Hygiene

An autonomous loop without strict termination controls is a production outage waiting to happen. Enforce **five mandatory loop guardrails**:

1. **Explicit completion criteria:** Define objective ground-truth exit conditions (all unit tests pass, target schema validates, or explicit `finish_task` / `end_turn` reached).
2. **Hard turn & call ceilings (`max_turns` / `max_llm_calls`):** Cap total model iterations per run so a stuck agent exits cleanly (`error_max_turns`) rather than looping infinitely.
3. **Token & cost budget ceilings (`max_budget_usd`):** Track cumulative input/output token spend in session state and halt or escalate when the run approaches its budget ceiling.

4. **Duplicate tool-call detection:** Hash `(tool_name, canonical_args)` across recent turns. If the agent invokes the exact same failing or no-op tool call 2–3 times in a row, intercept the call in the harness and inject a loop-breaker error or terminate.

5. **Structured error feedback (`is_error: true`) & ACI Poka-Yoke:**
   - **Never swallow tool exceptions as empty strings `""`** (which tricks the model into thinking no records exist) and never let transient tool errors crash the whole process. Return a structured error result (`is_error: true` / `{"status": "error", "message": "..."}`) describing *what* failed and *how* to fix it so the model can self-correct or escalate.
   - **Craft the Agent-Computer Interface (ACI) with Poka-Yoke (mistake-proofing):** Write tool descriptions like docstrings for a junior engineer (what it does, when to use it, when *not* to use it), constrain inputs with enums/ `Literal` and strict JSON schemas, and require unambiguous formats (e.g., **absolute file paths** `/repo/src/app.py` instead of relative paths that break after working-directory changes).

Loop Guardrail Dimension	Detection Mechanism	Runtime Action on Trigger
Turn Budget	Counter tracking total completed model roundtrips.	Halts execution when `turn > max_turns`; returns `error_max_turns`.
Financial Cost Ceiling	Cumulative token spend tracker (`spent_usd ⩾ max_usd`).	Immediately terminates session; emits `budget_ceiling_exceeded`.
Duplicate Call Trap	Hash map tracking `(tool_name, hash(canonical_args))` across recent turns.	Intercepts 3rd duplicate invocation; returns `is_error: true` loop-breaker feedback.
Structured Tool Errors	Exception handler wrapping tool execution harness.	Emits `{"type": "tool_result", "is_error": true, "content": "..."}` with corrective advice.
Completion Signal	Inspects `stop_reason == "end_turn"` and output schema.	Returns completed payload with final answer and turn count.

ACI Design & Poka-Yoke Tool Engineering

Tool Design Dimension	Fragile Prototype Tool	Production Poka-Yoke Tool
Path Arguments	`path: str` (accepts relative `../config.yaml`; breaks after `cd`)	`absolute_path: str` (requires leading `/`; validated via `Path.resolve().is_relative_to(ROOT)`)
Constrained Choices	`status: str` (free text; model invents `"in_progress_now"`)	`status: Literal["open", "pending", "resolved"]` (enforced via `strict: true` JSON Schema)
Boundary Clarity	`search(query)` (ambiguous across 4 similar search tools)	`orders_search_by_customer_id(customer_id)` — docstring states *what* it queries, *limits*, and *when to use* `orders_get_by_id` *instead*
Output Payload Size	Dumps all 10,000 matching rows (`250 KB` JSON) into context	Paginates (`limit: int = 25, offset: int = 0`) and returns total count + compact summary fields
Error Semantics	Raises unhandled `KeyError` or returns `""` on timeout	Returns `is_error: true` with error category, retryability, and corrective hint

05. Memory Architecture & Context Budget Engineering

A 200K–1M token context window is **not** an excuse to dump every file, tool schema, and historical transcript into a single prompt. As context fills, latency and cost scale linearly while instruction adherence and mid-window recall degrade (**Context Rot**).

The 4-Tier Agent Memory Model (CoALA Taxonomy)

Memory Tier	Cognitive Role	Storage & Lifetime	Production Engineering Rules
1. Working Memory	Active scratchpad for the current task (system instructions, recent turns, live tool observations, `temp:` state)	In-window `messages[]` + ephemeral invocation state (`temp:*`); cleared after turn/session	Keep lean; place byte-identical static instructions and tool schemas **first** to maximize prompt-cache hits; externalize live checklists (`TodoWrite` / state keys) so goals survive long runs.
2. Episodic Memory	Record of specific past sessions, interaction trajectories, and checkpoints (`session.events`)	Persistent session store (`DatabaseSessionService` , JSONL transcripts, checkpointer) or consolidated session summaries	Used for session resume (`--resume`), audit trails, trajectory evaluation, and few-shot recall of how similar tasks were solved previously.
3. Semantic Memory	Verified domain knowledge, entity facts, and consolidated user preferences (`user:*`)	Vector/hybrid RAG index or managed Memory Bank (`Vertex AiMemoryBankService`) across sessions	**Consolidate, don't just append:** Use background LLM consolidation to update superseded user facts rather than retrieving contradictory raw chat logs. Keep enterprise policy stores **read-only** during inference.
4. Procedural Memory	How the agent operates: system prompt heuristics, tool definitions, and reusable playbooks (**Skills**)	Code repository (`CLAUDE.md` , `.claude/rules/` , `SKILL.md` bundles, MCP prompt templates)	Use **3-level progressive disclosure** for skills: load only compact YAML `name` + `description` (~60–100 tokens) at startup, load the full `SKILL.md` body only when triggered, and execute bundled `scripts/` outside context so only stdout enters the window.

Context Rot Taxonomy & The 4 Context Engineering Strategies

Context Rot Failure Mode	Root Mechanism	Correct Remediation
Context Poisoning	An early hallucination, misread file path, or rejected refusal turn enters `messages[]` and gets repeatedly trusted or reinforced on subsequent turns.	**Do NOT run summarization/compaction** (which bakes the poisoned premise into the summary!). **Rewind / truncate** history to the checkpoint prior to the bad turn, or **clear/reset** into a fresh context window with only verified facts.
Context Confusion	Raw 50 KB JSON tool dumps, full file reads, and 60 unused MCP tool schemas crowd the attention window, causing the model to lose track of core constraints.	**Prune & Isolate:** Truncate/paginate tool outputs (`offset` / `limit`), strip bulky historical tool payloads after extraction, defer unused tool schemas behind on-demand search (`ToolSearch` / `tool_filter`), and delegate noisy searches to isolated subagents.
Context Clash	Contradictory rules accumulate across global instructions, subdirectory rules, and multi-turn user corrections.	Audit instruction hierarchy, scope path-specific rules (`paths:` frontmatter) so they load only when relevant files are touched, and resolve conflicts explicitly in a structured spec before execution.

The 4 Context Strategies (`Write` , `Select` , `Compress` , `Isolate`) & Compaction Mechanics:

1. **Write (Externalize State):** Persist plans, intermediate artifacts, and extracted schemas outside the fragile message history (into typed session state keys, scratchpad files on disk, or `ArtifactService`).
2. **Select (Just-in-Time Retrieval & Progressive Disclosure):** Pull in only the exact file ranges (`Grep` → paginated `Read`), scoped rules, and active skill bodies needed for the current step.
3. **Compress (Sliding Window, Compaction & Tool-Output Pruning):**
   - **Tool-output pruning:** Replace multi-kilobyte raw tool outputs from 10 turns ago with a 2-line digest once the relevant fields have been extracted into structured state.
   - **Summarization checkpoints (Compaction):** When a healthy long-running session approaches a token threshold (e.g., 70–80% of the context budget), trigger a compaction pass that summarizes older turns into a structured checkpoint block (preserving architectural decisions, modified file paths, verified facts, and open checklist items) while keeping the most recent *K* turns verbatim (**sliding window overlap**).
   - **Prompt-cache prefix discipline:** Never inject per-request dynamic timestamps, random UUIDs, or changing user metadata at the top of the system prompt—keep the prefix byte-identical so cached system instructions, tool schemas, and prior turns hit cache at ~75–90% lower input cost.
4. **Isolate (Subagent Sandboxing):** Run exploratory search, multi-file analysis, or independent verification inside a separate subagent with its own clean `messages[]` array, returning only a compact structured summary to the coordinator.

06. Multi-Agent Topologies & Handoff Contracts

Multi-agent systems partition complex work across specialized agents—not to engage in anthropomorphic role-play, but to **enforce context isolation, separate conflicting toolsets, parallelize independent I/O, and separate generation from verification.**

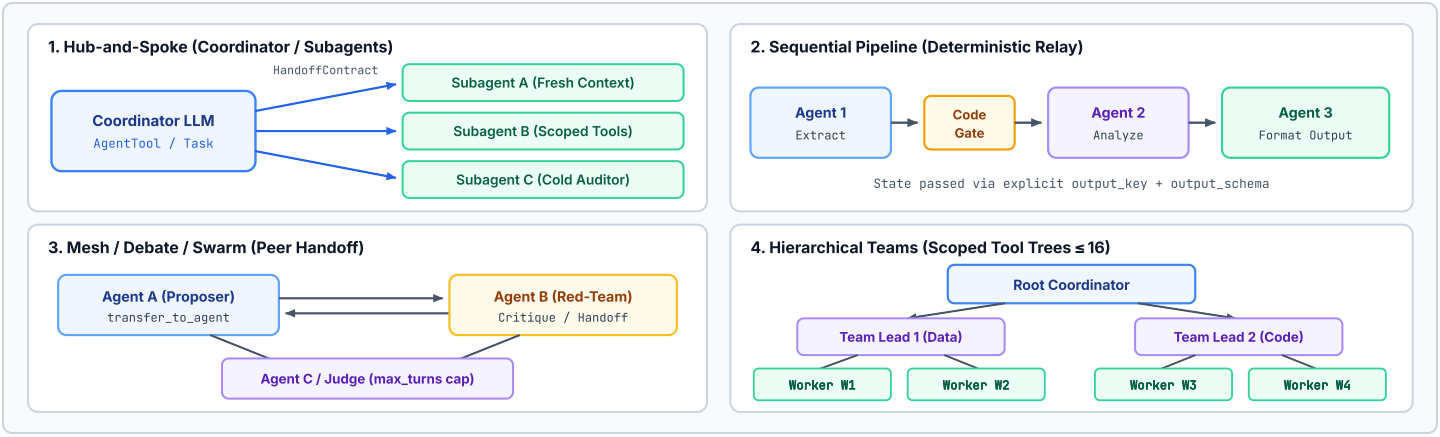
The "Agent Theater" Litmus Test

Before splitting a system into multiple agents, apply the **Multi-Agent Justification Rule**:

> **Multi-Agent is justified** ⟺ Subagents have distinct tools/permissions, isolated noisy context, parallel I/O, or adversarial verification.

If three "persona" agents share the exact same tools, read the same short prompt, and run sequentially without independent verification or context isolation, you have built **Agent Theater**—multiplying token spend and latency while introducing cross-agent drift for something a single structured prompt handles better.

Four Canonical Multi-Agent Topologies



Topology	Control & Routing Mechanics	Strengths	Key Risks & Required Safeguards
1. Hub-and-Spoke (Coordinator / Subagents)	Central coordinator decomposes the request, invokes specialist subagents as tools (`AgentTool` / `Task`), collects compact results, and synthesizes the final response.	Central visibility, strict context isolation per spoke, easy to enforce hub-level guardrails and wave limits (3–5 concurrent workers).	Coordinator becomes a bottleneck if spokes return raw 50 KB dumps instead of structured summaries. Block subagents from spawning recursive subagents unless explicitly architected.
2. Sequential Pipeline	Fixed relay (`Agent 1` → `Gate` → `Agent 2` → `Agent 3`) passing state via explicit keys (`output_key` + `output_schema`).	Zero routing overhead; deterministic transitions; ideal for Staged Extract → Redact → Analyze → Format pipelines.	Rigid; if step 1 misclassifies an input and no gate catches it, steps 2 and 3 compound the error.
3. Mesh / Debate / Swarm	Peer specialists transfer control directly to one another (`transfer_to_agent`) or critique each other's proposals without a central hub.	Useful for adversarial red-teaming, multi-perspective design critique, or seamless specialist conversation handoffs.	Highest risk of infinite ping-pong loops, state corruption, and diffusion of responsibility. Requires strict `max_turns` , explicit exit tools, and a deterministic tie-breaker/judge.
4. Hierarchical Teams	Multi-tier tree where a root orchestrator delegates to domain leads, which in turn manage narrow execution workers.	Scales to massive enterprise workflows with dozens of tools by keeping each node's tool catalog ≤ 16.	High cumulative latency and token cost; trace debugging requires correlation IDs across every parent-child boundary.

Vertical (MCP) vs. Horizontal (Subagents / A2A) Integration

Integration Axis	Protocol / Mechanism	Architectural Boundary	Trust & Context Contract
Vertical (`Agent → Tool/Data`)	**MCP (Model Context Protocol)** or native `FunctionTool`	Connects an agent to deterministic APIs, databases, filesystems, and prompt/resource templates.	Agent sees tool schema and passes typed arguments; scrub PII and untrusted content at the tool boundary (`after_tool_callback`).
Horizontal In-Process (`Parent → Subagent`)	**`AgentTool` / `Task` / `sub_agents`**	Delegates a cognitive subtask to a specialist agent running inside the same application runtime.	Subagent runs in an isolated context window with scoped tools and returns a typed `HandoffContract` (plus optional state/artifact deltas).
Horizontal Cross-Boundary (`Agent → Remote Agent`)	**A2A (Agent2Agent Protocol)**	Delegates long-running, stateful tasks across team, cloud, or organizational boundaries via an `AgentCard` .	Remote agent is **opaque** (never shares internal prompts, memory, or tools); caller authenticates via card `security Schemes` and validates all returned artifacts as untrusted input.

The Context Isolation Rule & Handoff Contracts

The single most important rule in multi-agent engineering is **Context Isolation**:

- **Fresh, task-scoped context by default:** When a coordinator spawns a subagent to research a codebase, audit a diff, or query a database, the subagent must start with a **fresh context window** containing *only* its specialist system instructions and the explicit task payload—**not** the coordinator's entire 80-turn noisy conversation history. (Reserve full-history context forking strictly for deliberate cache-sharing continuations where the child needs the exact prior transcript.)
- **No shared mutable scratchpads across parallel workers:** When running parallel subagents in waves (e.g., 3–5 concurrent workers), assign each worker a distinct `output_key` or isolated file path / git worktree so concurrent writes never race or clobber one another.
- **Explicit `HandoffContract` schema:** Because a subagent's intermediate tool calls stay inside its private window, **only its final response crosses back to the parent**. Enforce a typed `Handoff Contract` (via Pydantic `output_schema` / structured JSON) across every agent boundary:

```python
class HandoffContract(BaseModel):
    task_id: str
    status: Literal["completed", "partial", "blocked", "escalation_required"]
    confidence_band: Literal["high", "medium", "low"]
    summary: str = Field(max_length=1200, description="Compact synthesis")
    artifact_uri: str | None = Field(default=None, description="Artifact path")
    verified_citations: list[dict] = Field(default_factory=list)
```

ARTIFACT POINTER PATTERN

If a worker generates a 2,000-line code diff or a 30-page extracted table, do **not** stuff the raw payload into the coordinator's message history. Have the worker write the full payload to disk or an `ArtifactService` (`artifact_uri`) and return only the compact `HandoffContract` summary and URI to the coordinator.

07. Anti-Patterns, Failure Modes & Architectural Checklist

Production Trust Architecture: Why Single-Prompt "Happy-Path" Prototypes Fail

While a single multimodal LLM call with a JSON schema can easily summarize a clean 1-page demo PDF in 2 seconds, unguarded single-prompt pipelines fail at enterprise scale for five structural reasons:

1. **Token burn & hallucinations on corrupted inputs (Missing Epistemic Gate):** Sending a 120-page unreadable scan or wrong document type directly to a frontier model wastes time and tokens, and the model often guesses to be helpful. *Fix:* Enforce a **$0 deterministic Pre-LLM Epistemic Gate** checking file metadata and a **composite extraction quality score** (OCR token confidence, lexical dictionary ratio, character entropy, structural page completeness); if the score falls below threshold, halt and escalate without calling the LLM.
2. **Instruction crowding & self-grading bias:** Asking one prompt to extract 30 fields, format Markdown, and audit every citation causes the model to prioritize fluent generation and silently skip strict verification rules. Even strong prompts remain probabilistic, whereas a deterministic Python post-check paired with an independent cold-context critic enforces hard invariants reliably.
3. **Unverified citations:** Checking only that the output matches the regex `(Source: Section X)` misses fabricated references (`Section 99.1` on a 4-page contract). *Fix:* **Grounded citation verification** checks deterministically in code that every cited section header or quote exists in the extracted source text.
4. **Cloud PII exposure:** Sending raw binary documents directly to external model APIs leaks unredacted PII before any prompt rule runs. *Fix:* Parse locally (`Docling` / local layout parser) and run deterministic PII scrubbing at the pre-LLM and `after_tool_callback` boundaries.
5. **Rule drift across repositories:** Copy-pasting compliance prompts across 10 microservices guarantees policy drift. *Fix:* Package shared domain validators, prompts, and policy thresholds into a central **MCP server** (*Rules as Infrastructure*).

Symptom → Root Cause → Architectural Fix

Production Symptom	Root Architectural Cause	Deterministic Architectural Fix
Agent miscalculates fees, discounts, or date windows on edge cases	**Deterministic drift:** Delegating Bucket 1 arithmetic or policy thresholds to probabilistic prose instructions.	Move calculations and threshold checks into deterministic Python/SQL tools or `PreToolUse` / `before_tool_callback` hooks.
Simple 3-step pipeline takes 45 seconds and costs $0.80 per run	**Over-agentification:** Using an open-ended autonomous ReAct agent (or Agent Theater) where the task path is known in advance.	Replace the autonomous loop with a **Prompt Chain** or **Parallel Sectioning Workflow** using fast-tier models for routine steps.
Coordinator hits `model_context_window_exceeded` after calling 4 research workers	**Missing context isolation:** Workers return raw 40 KB tool dumps into the parent window, or inherit full parent history.	Enforce the **Context Isolation Rule**: spawn workers in fresh windows, write bulky artifacts to disk/storage, and return a bounded `Handoff Contract` summary.
Parallel workers intermittently lose data or overwrite each other's findings	**Shared mutable state race:** Concurrent branches write to the same session state key (`state["result"]`) or edit the same file simultaneously.	Give each parallel branch a unique `output_key` (`result_a` , `result_b`) merged by a downstream `JoinNode` /synthesizer, or isolate file edits in separate worktrees.
Generator–Critic loop burns 15 turns and still ships a hallucinated citation	**Unbounded loop + ungrounded reflection:** No `max_iterations` cap, and the critic grades prose by "vibes" without checking source text in code.	Cap the loop at `max_iterations=3` , verify citations deterministically against source text (regex/substring match), and escalate to HITL if the cap trips without approval.
Agent tells the user "You have no active orders" during a database timeout	**Swallowed tool error:** Tool wrapper caught the timeout exception and returned `""` or `{}` with HTTP 200 success semantics.	Return a structured error block (`is_error: true` or `{"status": "error", "retryable": true, "message": "DB timeout"}`) so the model knows the lookup failed.
After compacting a confused session, the agent keeps repeating the same wrong file path	**Compacting poisoned context:** Summarization compressed the earlier hallucinated file path into the compacted summary block.	Use **rewind / hard context reset (`/clear`)** to discard poisoned turns entirely instead of summarizing them.
Agent selects the wrong tool when given 45 enterprise MCP endpoints	**Tool catalog saturation:** Model attention degrades when forced to discriminate among > 16 overlapping tool schemas in one prompt.	Group tools by domain across specialist subagents (≤ 16 tools each) or enable on-demand **progressive tool disclosure / Tool Search** plus `tool_filter` allowlists.

Production Symptom	Root Architectural Cause	Deterministic Architectural Fix
Summarizer agent reading an external email is tricked into calling `send_email`	**Indirect prompt injection + excessive agency:** Untrusted data entered the same context that held high-privilege mutation tools.	Enforce **least privilege** (strip `send_email` from the reader agent), screen tool outputs in `after_tool_callback`, and gate external sends with a **HITL checkpoint**.

8-Point Pre-Production Architectural Checklist

□ **1. 3-Bucket Triage Verified:** Are all math, schema validation, RBAC, and hard compliance rules executed in deterministic code (Bucket 1), and all high-blast-radius irreversible actions gated by HITL (Bucket 3)?

□ **2. Complexity Ladder Justified:** Have you proven with an evaluation baseline that a single Augmented LLM call or deterministic Workflow cannot meet the bar before deploying an Autonomous Agent?

□ **3. Pre-LLM Epistemic Gating Active:** Does deterministic code validate file types, input completeness, and OCR/extraction quality *before* spending model tokens, with `escalation_required = True` as the safe default?

□ **4. Loop Hygiene Enforced:** Does every agent and evaluator-optimizer loop enforce `max_turns` / `max_iterations`, token budget ceilings, duplicate tool-call detection, and explicit escalation when the cap trips?

□ **5. ACI & Poka-Yoke Hardened:** Are tool catalogs ≤ 16 per agent, arguments constrained via strict schemas/enums/absolute paths, and failures returned as structured `is_error: true` signals?

□ **6. Context Budget & Cache Optimized:** Are static instructions and tool definitions placed before dynamic per-request data for cache stability, and is noisy exploration isolated in subagents?

□ **7. Handoff Contracts Typed:** Do subagents run in fresh, task-scoped context windows and return compact, schema-validated `HandoffContract` objects (with URIs for large artifacts)?

□ **8. Separation of Authority & Trajectory Evals:** Are high-stakes outputs verified by an independent critic + deterministic code checks, and does CI evaluate both final answers and tool trajectories?

08. Glossary

- **ACI (Agent-Computer Interface):** The design, naming, schema constraints, error semantics, and documentation of tools exposed to a model—the agentic equivalent of HCI/UX.
- **Agent Theater:** Anti-pattern where multiple persona agents with identical tools and no context isolation or independent verification are chained together, inflating cost and latency for zero architectural benefit.
- **Architectural Evaluation Criteria:** Decision rubric comparing Single Call, Composable Workflow, and Autonomous Agent across path predictability, latency budget, error blast radius, token cost ceiling (~1× vs. 3–5× vs. ~15×), and auditability.
- **Augmented LLM:** The foundational building block of agentic systems: a single LLM equipped with retrieval, tools (often via MCP), and memory.
- **Compaction:** Context compression technique that summarizes older conversation turns into a compact block so long sessions stay within token budgets; effective for *context confusion*, but unsafe for *context poisoning*.
- **Context Isolation Rule:** Principle that subagents execute in fresh, task-scoped context windows rather than inheriting the coordinator's full noisy history, returning only a structured summary (`HandoffContract`).
- **Context Rot (Poisoning / Confusion / Clash):** Degradation of model accuracy as context grows: **poisoning** (hallucinated premise persists in history), **confusion** (bulky tool outputs/schemas dilute attention), and **clash** (contradictory instructions).
- **Deterministic Invariant Degradation (Deterministic Drift):** Reliability failure caused by delegating exact arithmetic, schema validation, or policy enforcement to probabilistic prompt instructions instead of deterministic code.
- **Epistemic Gating:** Deterministic pre-LLM guardrail that scores input/extraction integrity (OCR confidence, lexical ratio, structural completeness) and blocks or escalates corrupted inputs before spending model tokens.
- **Evaluator–Optimizer:** Workflow pattern pairing a generator LLM with an independent critic/auditor LLM (plus deterministic checks) in a rubric-driven loop bounded by `max_iterations`.
- **Gate (`gate_check`):** Deterministic plain-code check between workflow steps that validates intermediate schemas, invariants, or citations before allowing execution to proceed.
- **HandoffContract:** Strict structured schema (status, confidence band, compact summary, verified citations, and `artifact_uri` pointer) returned by a subagent to its coordinator.
- **HITL (Human-in-the-Loop):** Control pattern that pauses execution and persists state before an irreversible or high-stakes action commits, presenting concrete tool arguments for human approval, modification, or rejection.
- **LLMCompiler / ReWOO:** Plan-and-execute architectures that decouple upfront planning (emitting symbolic variable plans or task DAGs) from parallel tool execution to eliminate sequential ReAct turn latency.
- **Orchestrator–Workers:** Workflow pattern where a central LLM dynamically decomposes an unpredictable task at runtime, dispatches scoped workers, and synthesizes their results.
- **Parallelization (Sectioning vs. Voting):** Running concurrent LLM calls either on independent sub-parts of a task for speed/focus (*Sectioning*) or repeating the same task for consensus/high-recall verification (*Voting*).
- **Poka-Yoke:** Mistake-proofing tool definitions (e.g., requiring absolute file paths, strict enums, and self-contained schemas) so invalid model calls become structurally impossible.
- **Prompt Chaining:** Sequential workflow decomposing a predictable task into fixed LLM steps separated by deterministic code gates.
- **ReAct (Reason + Act):** Core autonomous agent loop that interleaves reasoning (*Thought*), tool execution (*Action*), and environmental ground truth (*Observation*) until a stopping condition is met.
- **Routing:** Workflow pattern that classifies an input's intent or complexity and dispatches it to a specialized prompt, toolset, or cost-appropriate model tier.
- **Separation of Authority:** Decoupling generation from verification by running an independent, cold-context critic/auditor alongside deterministic code checks to eliminate self-grading confirmation bias.
- **Sequential Decision Framework:** Ordered architectural checklist run before prompt writing: (1) Problem decomposition & execution engine triage → (2) Work-shape selection → (3) Routing & hosting → (4) Guardrails & verification.

- **Tripartite Execution Regimes:** Triage framework dividing system logic into **Regime 1 (Deterministic Code)** for math/validation/RBAC, **Regime 2 (Probabilistic LLM)** for unstructured synthesis/classification/planning, and **Regime 3 (Human-in-the-Loop)** for irreversible, regulated, or accountability-bearing decisions.