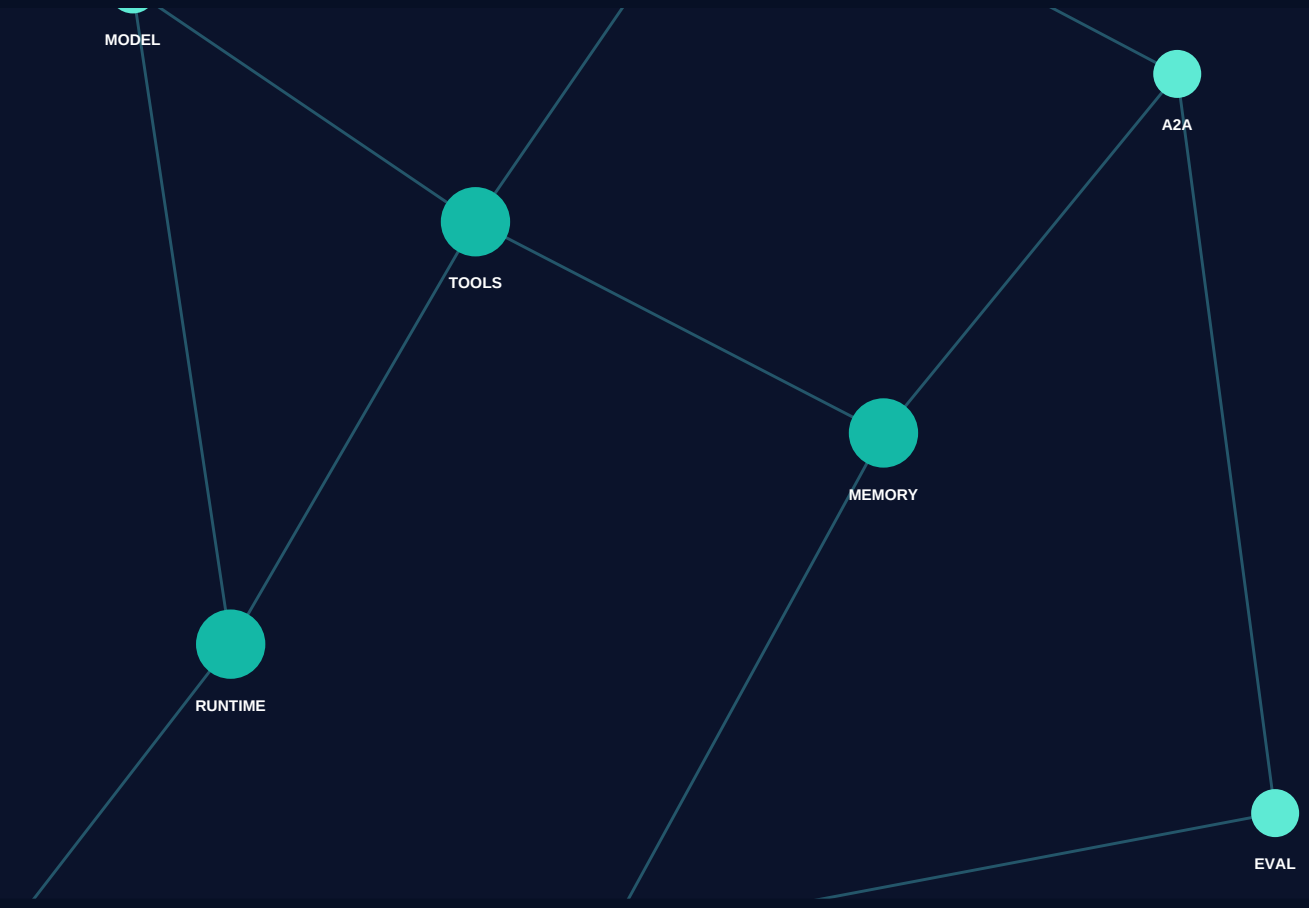
ENGINEERING FIELD GUIDE - 2026

AI Systems Design

A field guide to model APIs, agent runtimes, MCP, A2A, and production engineering

FOR HANDS-ON AI SYSTEMS ENGINEERS



AUTHORS

Aditya Karnam Gururaj Rao

Arjun Jaggi

PUBLISHED

JULY 2026

Architecture-first. Protocol-aware. Reliability and security by construction.

READ THE SYSTEM, THEN THE PARTS

How to Use This Field Guide

This guide assumes software engineering fluency and skips model-training fundamentals. It is organized around the decisions that turn a model endpoint into a dependable product.

Three passes

First, scan the diagrams to build a mental map. Next, read the design rules and failure modes. Finally, use the checklists while designing or reviewing a real system.

- Mental model: what the component is responsible for.
- Mechanism: messages, state transitions, and control flow.
- Production test: what must remain true under failure and attack.

Recurring notation

Solid boundaries represent processes or services. Dashed boundaries represent trust zones. Cyan paths carry requests or data; amber paths require approval or policy; red paths indicate failure or rejected authority.

The Quecto lens

Quecto appears as a runtime lens: a place to map loop ownership, tool dispatch, context assembly, policy, tracing, and checkpoints. The architecture remains portable to custom runtimes and other agent frameworks.

ENGINEERING CHECK

- [ ] Name the owner of every loop and state transition.
- [ ] Distinguish protocol capability from product permission.
- [ ] Treat external content and model output as untrusted.

FIVE PARTS, ONE OPERATING MODEL

Contents

Move from the model boundary to protocols, orchestration, production controls, and a complete reference architecture.

The spine of the guide

Every chapter answers one systems question: who owns the decision, what crosses the boundary, where state lives, how authority is checked, and how the run can be understood or recovered later.

SEPARATE THE LAYERS

The AI Systems Map

An agent product is a distributed control system around a probabilistic component. Reliability comes from the deterministic layers that constrain, observe, and recover that component.

The key separation

The model proposes. The runtime decides when and how to call it. The tool plane performs effects. Protocols connect capabilities. State systems preserve continuity. Policy decides authority. Observability and evaluation create evidence.

Two directions of integration

MCP connects an AI application to tools and context. A2A connects one independently operated agent system to another. Native APIs remain appropriate for stable, tightly owned service contracts.

DESIGN RULE

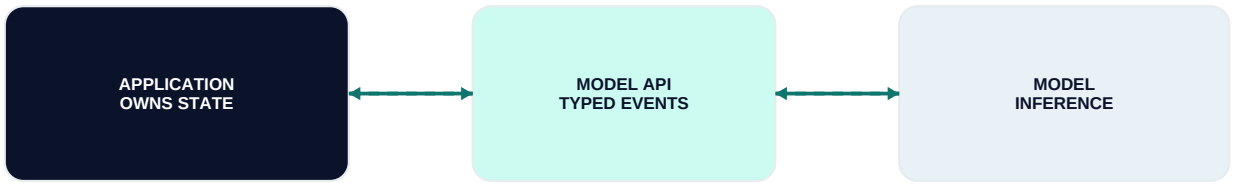
Do not hide policy, persistence, or side effects inside the model call.

THE INTELLIGENCE BOUNDARY

A Model API Is Not an Agent

A model API transforms input into typed output items or events. It does not automatically own retries, tools, permissions, durable state, or task completion.

MODEL APIS + RUNTIMES



What the API can return

Depending on the provider and configuration, one response may contain text, structured data, tool calls, citations, refusals, usage, and lifecycle metadata. Streaming exposes these as ordered events rather than one final string.

What your application still owns

- Input assembly, tenant identity, and data boundaries.
- Tool execution, authorization, and result validation.
- Retry policy, deadlines, cancellation, and budgets.
- Persistent sessions, checkpoints, and final task status.

Why the distinction matters

If the model call is treated as the whole system, production concerns become prompt text. Prompts can influence behavior, but they cannot enforce filesystem isolation, token audience, transactionality, or idempotency.

QUECTO LENS

Quecto is closer to the runtime around the model than to the model API itself.

STREAM STATE, NOT JUST TOKENS

Responses Are Event Lifecycles

A robust client consumes a response as a stateful event stream: created, in progress, item added, deltas, item completed, response completed - or an explicit failure.

Build an event reducer

Treat each event as input to a deterministic reducer that updates the run view. A reducer can handle reordering guards, duplicate delivery, partial output, progress UI, and recovery after reconnect.

Separate transport completion from task completion

A closed HTTP stream does not prove that the business task succeeded. Persist a terminal status only after required output items pass schema validation and all necessary tool effects are confirmed.

PYTHON

```
for event in stream:
state = reduce_event(state, event)
persist_cursor(run_id, event.sequence_number)
if state.terminal:
validate_terminal_state(state)
```

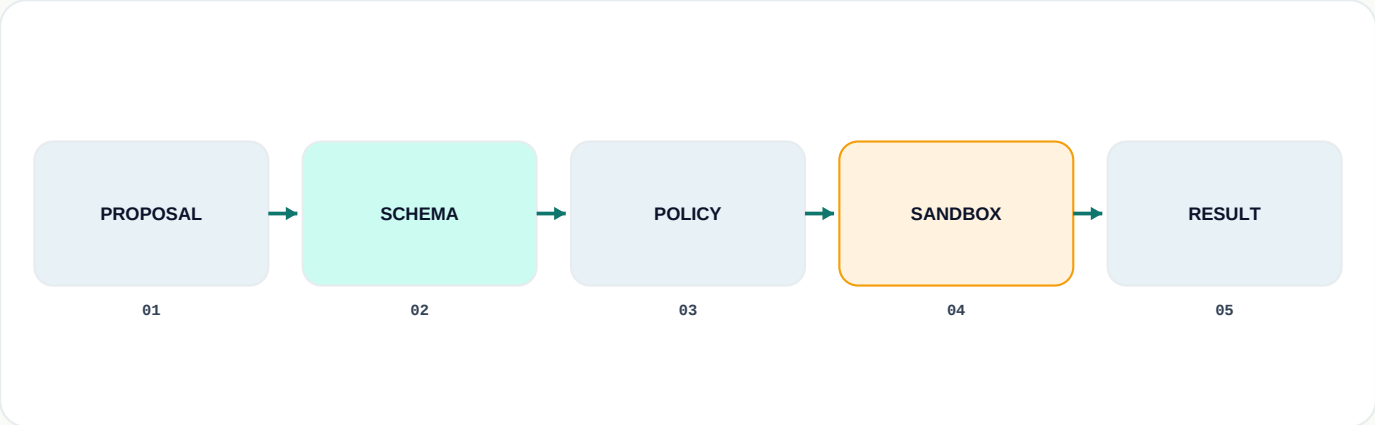
FAILURE MODE

Rendering deltas directly without a reducer makes reconnects and duplicate events corrupt visible state.

THE APPLICATION EXECUTES

Tool Calling Is a Proposal

A tool call is structured model output. The runtime must validate the schema, resolve the tool, authorize the principal, apply policy, execute in an isolation boundary, and return a typed result.



The execution gate

Never dispatch directly from a tool name emitted by the model. Resolve against a server-side registry, reject unknown fields, normalize types, and bind authorization to the authenticated user and current task.

Result envelopes

Return machine-readable status, bounded content, error class, retryability, provenance, and artifact references. Model-visible error text should be useful without leaking secrets or internal stack traces.

JSON

```
{
"tool": "search_code",
"call_id": "call_42",
"arguments": {"query": "auth policy"},
"policy": {"tenant": "t_7", "mode": "read_only"}
}
```

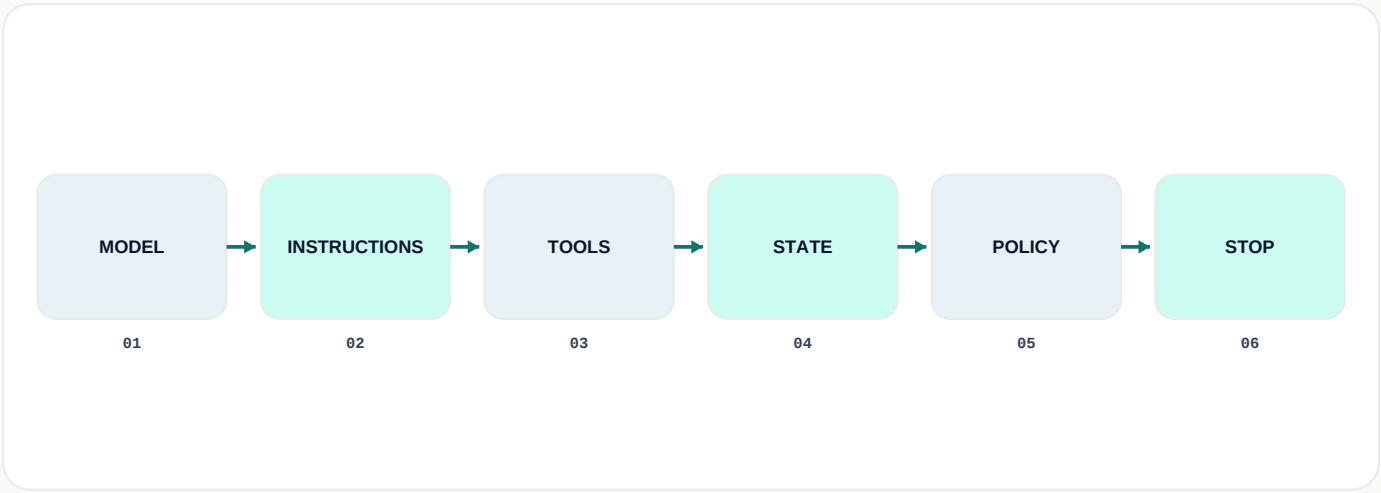
ENGINEERING CHECK

- [ ] Schema-valid arguments
- [ ] Authorized principal and tenant
- [ ] Deadline and resource budget
- [ ] Approval for consequential effects
- [ ] Sanitized result envelope

MODEL PLUS CONTROL SYSTEM

The Agent Equation

An agent is not a personality wrapper. It is a model placed inside an execution system with instructions, tools, state, policies, budgets, and termination rules.



A useful decomposition

Agent = model + instructions + tools + loop + state + policy + termination. Each term changes system behavior independently and should be configurable, observable, and testable.

The model is only one dependency

The runtime can substitute a model, restrict a tool, replay state, or stop a runaway sequence. This separation is the foundation for provider portability, deterministic testing, and safe degraded modes.

Instructions are policy hints

Instructions shape model choices but are not an enforcement layer. Security policy must execute outside the model and remain correct even if the model ignores or misunderstands a prompt.

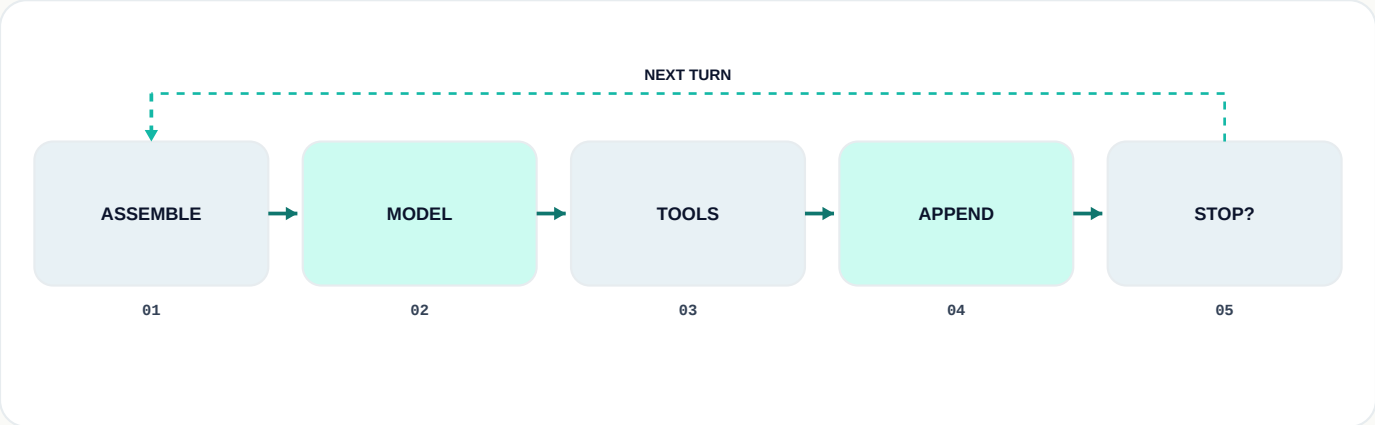
DESIGN RULE

Anything that must always happen belongs in code or infrastructure, not only in instructions.

ONE LOOP, EXPLICIT OWNERSHIP

The Minimal Agent Loop

The runtime repeatedly calls the model, interprets typed output, executes approved tools, appends results, and stops on a valid final answer or a deterministic limit.



The heart of the runtime

A small explicit loop is easier to secure and observe than a maze of callbacks. Every turn should carry a run ID, deadline, remaining step budget, cost budget, trace context, and cancellation token.

Terminal conditions

Stop on a valid final output, user cancellation, deadline, budget exhaustion, policy rejection, unrecoverable dependency failure, or a cycle detector. A friendly final message is not the same as a terminal state.

PYTHON

```
while budget.allow_turn():
out = model.respond(context,
tools=registry.schemas())
if out.final:
return validate_final(out)
for call in out.tool_calls:
result = dispatcher.execute(call, run_policy)
context.append(result)
raise RunStopped(budget.reason)
```

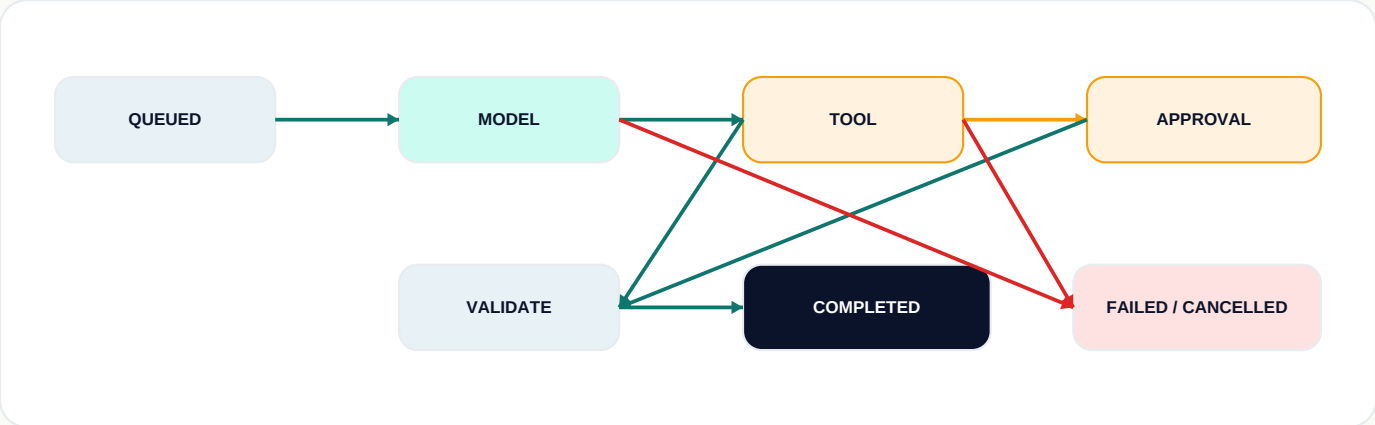
QUECTO LENS

Make loop ownership visible: one component advances the run and persists every transition.

MAKE TRANSITIONS EXPLICIT

Runtime State Is a State Machine

A production run is a durable state machine, not an in-memory while loop. Explicit transitions make cancellation, resumption, approval, and audit reliable.



Recommended states

Use states such as queued, assembling_context, calling_model, awaiting_tool, awaiting_approval, executing_tool, validating, completed, failed, and cancelled. Persist both the state and the event that caused it.

Transition guards

Only valid transitions should commit. Compare-and-swap versions prevent two workers from advancing the same run. Terminal states must reject late tool results and duplicate callbacks.

Budgets are state

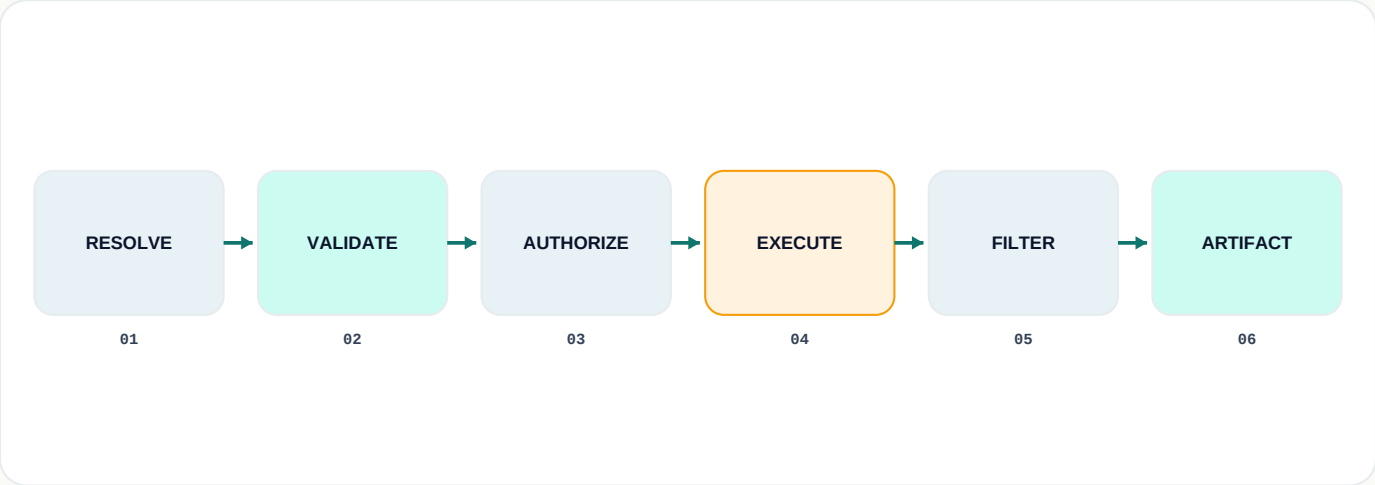
Track turns, tokens, elapsed time, tool CPU, network bytes, and monetary cost as first-class counters. The runtime should stop before a provider or infrastructure hard limit does.

Guard	Prevents	Evidence
Version check	Double advance	State version
Deadline	Hung run	Monotonic timestamp
Step budget	Tool loop	Remaining turns
Terminal lock	Late mutation	Final event

EFFECTS LIVE OUTSIDE THE MODEL

Design the Tool Execution Plane

The execution plane turns proposed calls into controlled effects. It should be narrow, typed, identity-aware, observable, and replaceable without changing the agent loop.



The dispatch pipeline

Resolve tool identity, validate input, authorize, classify risk, request approval if needed, allocate a sandbox, execute with a deadline, filter output, persist an artifact, and emit a trace span.

Risk classes

- Read-only and bounded: search, inspect, calculate.
- Reversible mutation: create draft, stage change, open branch.
- External or consequential: send, deploy, purchase, delete, publish.

Parallel calls need a join policy

Independent read tools can run concurrently. Mutations require declared conflict domains. The join must define whether one failure cancels siblings, returns partial results, or triggers compensation.

PRODUCTION CHECK

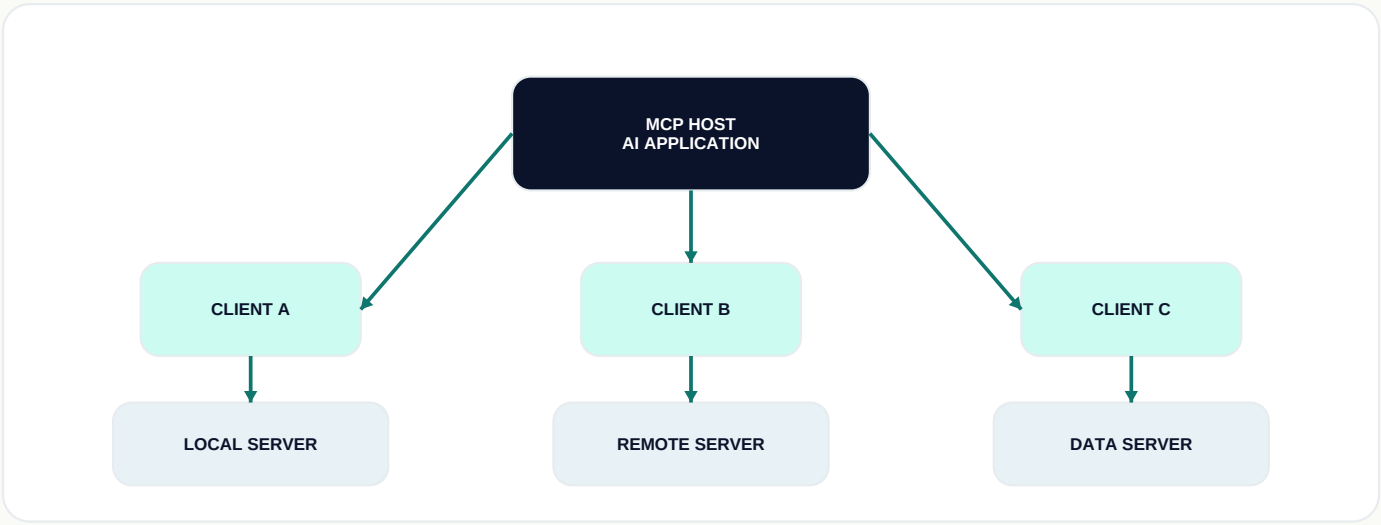
Can an operator explain exactly which principal caused each external effect?

CAPABILITY INTEROPERABILITY

MCP: Host, Client, Server

MCP uses a client-host-server architecture. The host owns the AI application and permissions; it creates one MCP client connection for each server that exposes capabilities.

MCP + A2A



Participant responsibilities

The host coordinates user intent, model access, connection policy, and capability presentation. Each client maintains protocol state for one server. The server publishes tools, resources, prompts, and optional capabilities.

A protocol, not an agent runtime

MCP defines discovery and invocation across a boundary. It does not decide when the model should stop, how memory is compacted, or whether a tool request is allowed for a particular user.

Local and remote are trust decisions

A local stdio process can still be malicious. A remote server can be strongly authenticated. Placement changes attack surfaces, but trust must be established through identity, provenance, policy, and isolation.

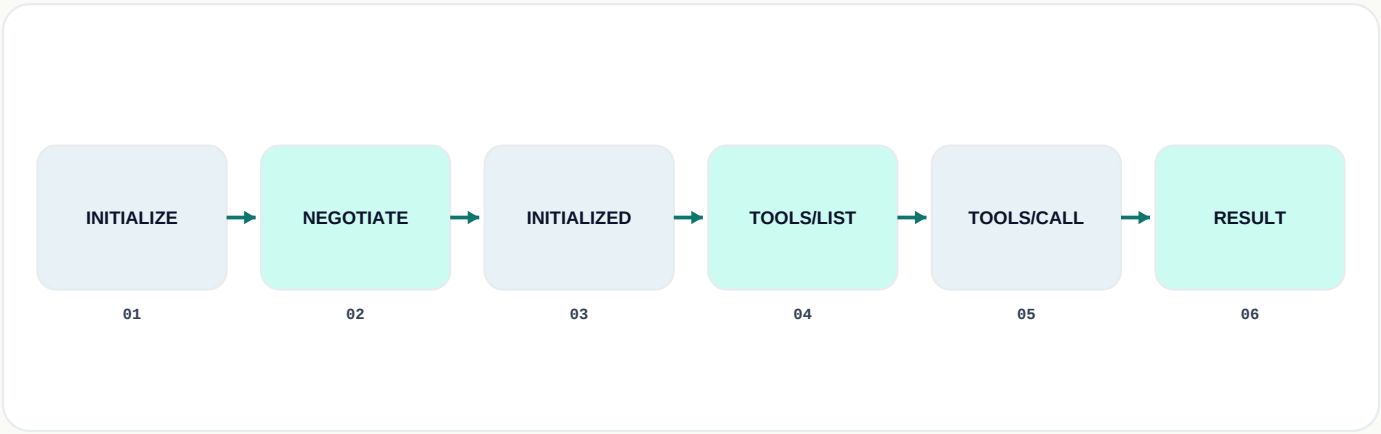
DESIGN RULE

The host may expose a capability to the model only after product policy approves it.

AGREE BEFORE OPERATING

MCP Lifecycle and Negotiation

Initialization establishes protocol version, implementation identity, and supported capabilities before normal requests begin.



The handshake

The client sends `initialize` with its supported protocol version, capabilities, and implementation metadata. The server returns its selected version and capabilities. The client then sends `notifications/initialized`.

Capability discipline

Never call or emit a feature that was not negotiated. Capability flags are compatibility contracts, not authorization grants. Product policy still decides which negotiated features may be used.

Timeout and recovery

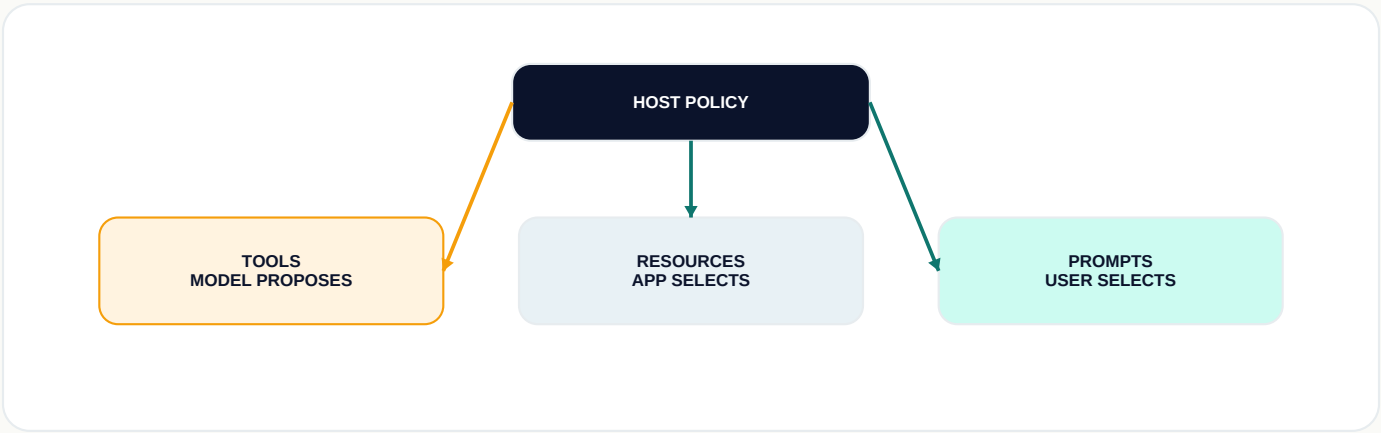
Every request needs a timeout. A timed-out request should be cancelled and locally completed as failed or unknown. For HTTP sessions, a lost session may require a new initialization.

```json
{
"jsonrpc": "2.0",
"id": 1,
"method": "initialize",
"params": {
"protocolVersion": "2025-06-18",
"capabilities": {"roots": {}},
"clientInfo": {"name": "runtime", "version": "1.0"}
}
}
```

THREE DIFFERENT CONTROL MODELS

MCP Primitives: Tools, Resources, Prompts

Tools, resources, and prompts are not interchangeable. They carry different control expectations and should map to different product UX and security policy.



Tools

Executable capabilities, typically proposed by the model. Tool schemas describe shape, not trust. The host validates, authorizes, displays, and may require confirmation before execution.

Resources

Application-selected data such as files, records, or schemas. Resource content is untrusted input when it originated outside the application trust zone, even when delivered through a trusted server.

Prompts

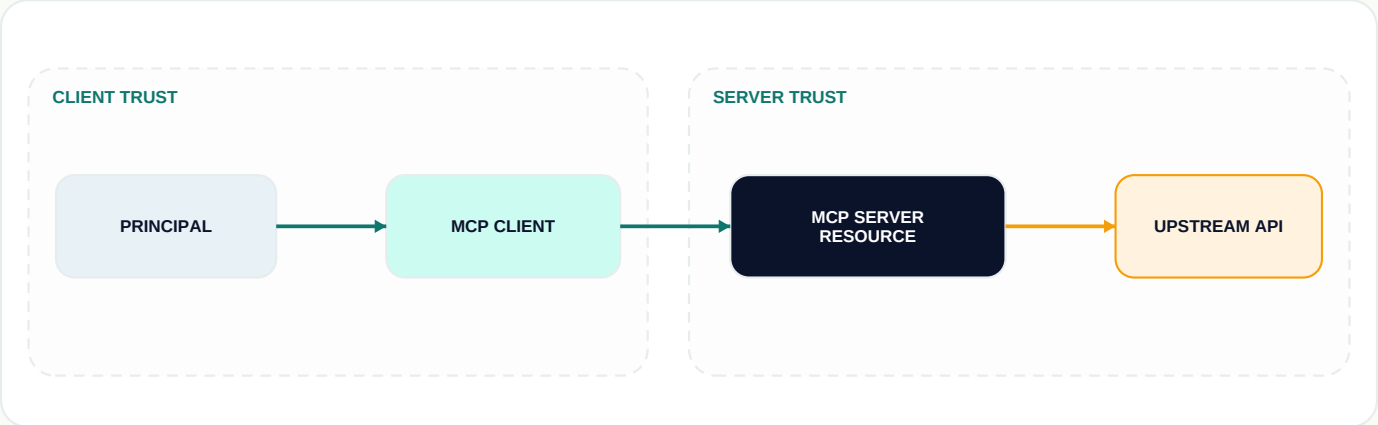
Reusable interaction templates selected by a user or application. Prompt text can improve consistency, but a server-provided prompt must not override host security policy.

Primitive	Primary	Typical use	Key risk
Tool	Model proposes	Perform action	Excessive agency
Resource	Application selects	Provide context	Injection/data leak
Prompt	User selects	Start workflow	Instruction override

CONNECTION SECURITY IS NOT TOOL AUTHORITY

MCP Transport and Authorization

MCP supports stdio and Streamable HTTP. Transport authentication identifies the connection; tool authorization must still evaluate the user, tenant, requested action, resource, and current policy.



Local stdio

Launch the server with an explicit executable path, minimal environment, fixed working directory, restricted filesystem, controlled network egress, and bounded resources. Do not treat locality as trust.

Streamable HTTP

Use HTTPS, validate Origin where required, authenticate every request, validate session ownership, and bind locally hosted servers to loopback unless remote exposure is intentional.

OAuth boundaries

Bind access tokens to the intended MCP server, validate token audience, use PKCE, and obtain separate credentials for downstream APIs. Passing the client token through to another service breaks accountability and creates confused-deputy risk.

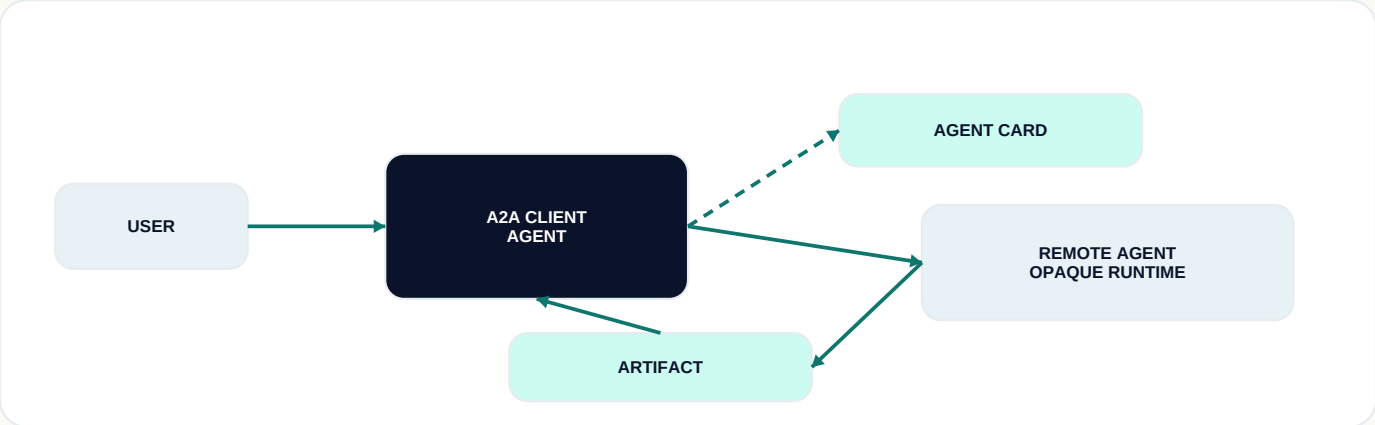
ENGINEERING CHECK

- [ ] Server identity and provenance verified
- [ ] Transport authenticated and encrypted
- [ ] Token audience validated
- [ ] Per-tool authorization enforced
- [ ] Session ID not used as authentication

INTEROPERABILITY ACROSS OPAQUE RUNTIMES

A2A: Independent Agent Systems

A2A connects a client agent to a remote agent system. The remote system is opaque: it exposes identity, capabilities, skills, interaction requirements, and task results - not its internal tools or memory.



Core objects

An Agent Card describes the remote agent. A Message carries a conversational turn made of Parts. A Task represents stateful work. Artifacts are the durable outputs produced by that task.

Why use A2A

Use it when ownership, deployment, framework, or trust boundaries make another agent a true remote service rather than an internal function. The protocol supports discovery and long-running coordination without exposing internal implementation.

Message or task

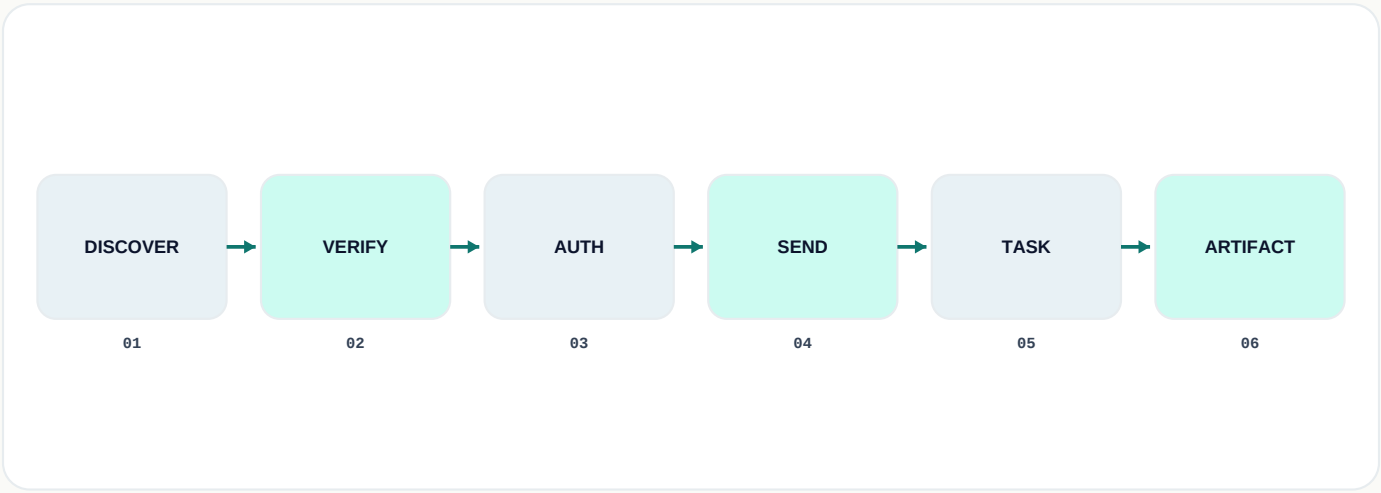
A simple interaction may return a direct Message. Long-running or stateful work returns a Task and later status or artifact updates. Clients must handle both forms.

```json
{
"message": {
"role": "user",
"messageId": "msg_7",
"parts": [{"text": "Review this design"}]
},
"configuration": {"acceptedOutputModes":
["text/markdown"]}
}
```

DISCOVER, AUTHORIZE, TRACK, RECEIVE

A2A Discovery and Task Lifecycle

A client discovers an Agent Card, selects a compatible interface, authenticates out of band, sends a message, and receives either a direct message or a task that advances toward a terminal state.



Discovery is input, not trust

Agent Cards advertise endpoints, skills, capabilities, and authentication requirements. Validate domain, provenance, signature when available, allowlist policy, and freshness before selecting an agent.

Task semantics

Track the server-issued task ID and context ID. Persist status updates and artifacts separately. Messages communicate; artifacts deliver results. Reconnect logic should fetch current task state instead of assuming all streaming events were received.

Terminal handling

Completed, failed, cancelled, and rejected are terminal outcomes. A client must reject late mutation, reconcile duplicate artifact updates, and surface whether partial artifacts are usable.

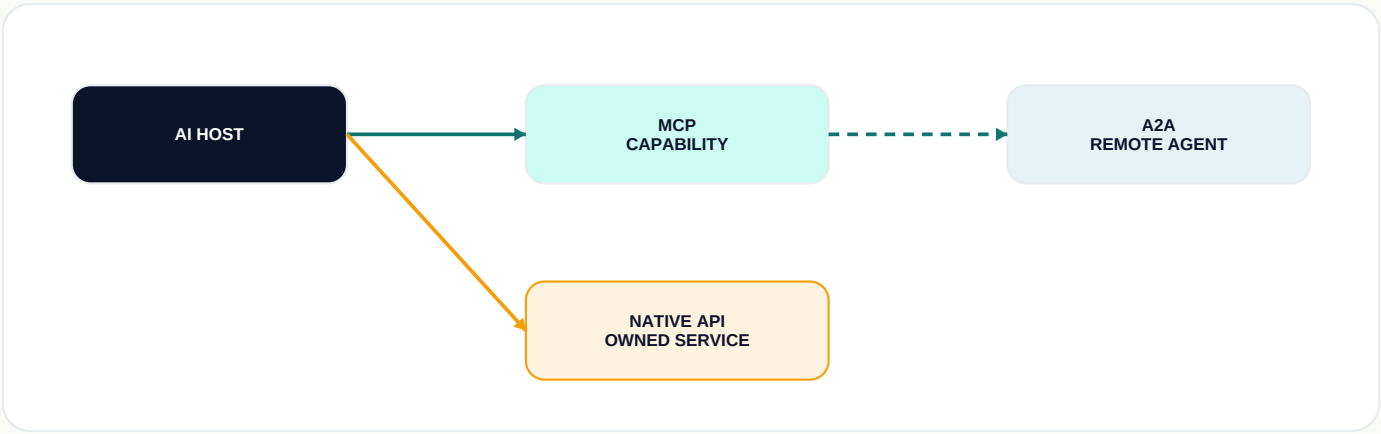
RELIABILITY RULE

Streaming improves latency, but the task resource is the durable source of truth.

CHOOSE THE RIGHT BOUNDARY

MCP vs A2A vs Native API

These interfaces solve different integration problems. Select by ownership, autonomy, discovery needs, task duration, and the state contract - not by trend.



A practical rule

Use a native function inside one process. Use a typed service API across stable internal services. Use MCP when several AI hosts should consume the same capability surface. Use A2A when another independently operated agent owns planning and task execution.

They compose

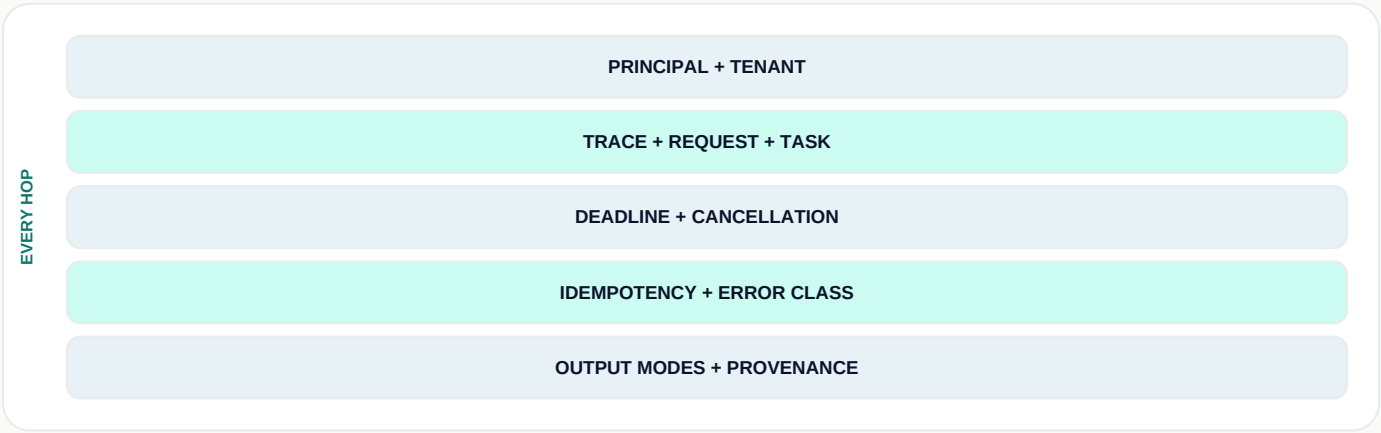
An A2A remote agent may internally use MCP servers. A host may expose a native service through MCP. Composition is healthy when each boundary preserves identity, deadline, trace context, and idempotency.

Dimension	MCP	A2A	Native API
Connects	Host to capability	Agent to remote agent	Service to service
Discovery	Tools/resources /prompts	Agent Card/skills	Docs or schema
State	Connection/session	Task lifecycle	Application-defined
Best for	Tool interoperability	Opaque delegation	Stable owned contract
Main risk	Capability overreach	Distributed task drift	Tight coupling

PRESERVE SEMANTICS END TO END

Design Contracts at Protocol Boundaries

Interoperability fails when a gateway translates syntax but drops identity, deadlines, provenance, cancellation, or error meaning.



The universal envelope

Carry request ID, task ID, tenant and principal, trace context, absolute deadline, idempotency key, data classification, policy version, and accepted output modes across every hop.

Errors need classes

Separate invalid input, unauthorized, forbidden, conflict, unavailable, deadline exceeded, cancelled, rate limited, dependency failure, and internal failure. Mark whether retry is safe and whether the prior effect is known.

Artifacts beat giant messages

Large results should become immutable artifacts with content type, hash, provenance, retention, and access policy. Messages can reference artifacts without duplicating sensitive content through every context window.

JSON

```
{
"request_id": "req_91",
"principal": "user_17",
"tenant": "acme",
"deadline": "2026-08-05T20:00:00Z",
"idempotency_key": "review:design:v3",
"traceparent": "00-..."
}
```

ADD AGENTS ONLY FOR A REASON

The Topology Ladder

Start with one agent and tools. Add planners, specialists, handoffs, or remote agents only when the new boundary improves isolation, ownership, parallelism, or measurable quality.

ORCHESTRATION + MEMORY

Five levels

- Deterministic workflow with model steps.
- Single agent with tools.
- Manager with specialist tools or agents-as-tools.
- Handoffs between peer specialists.
- Independent remote agents connected through A2A.

The cost of each level

More agents add model calls, context duplication, coordination messages, ambiguous ownership, and partial-failure surfaces. Architecture should justify each added loop with a measurable requirement.

Default bias

Keep orchestration deterministic where sequence is known. Give the model discretion only where classification, synthesis, or planning benefits from it.

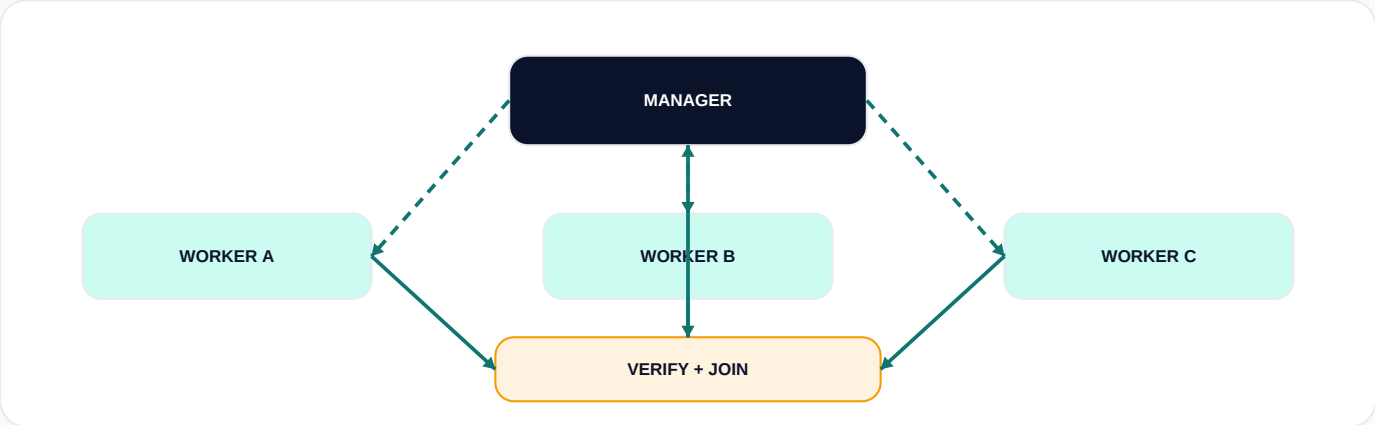
DESIGN RULE

A new agent must own a distinct responsibility and a distinct acceptance test.

CENTRAL CONTROL, SPECIALIZED EXECUTION

Manager-Worker Orchestration

A manager decomposes work, delegates bounded tasks, monitors progress, and combines results. Workers return evidence and artifacts rather than ungrounded confidence.



Manager responsibilities

Define the work graph, select workers, pass the minimum context, set acceptance criteria, assign budgets, cancel unnecessary work, resolve conflicts, and own the final answer.

Worker contract

Each worker receives a goal, scope, inputs, allowed tools, deadline, output schema, and evidence requirement. It returns status, artifacts, citations or observations, cost, and unresolved risks.

Fan-out and fan-in

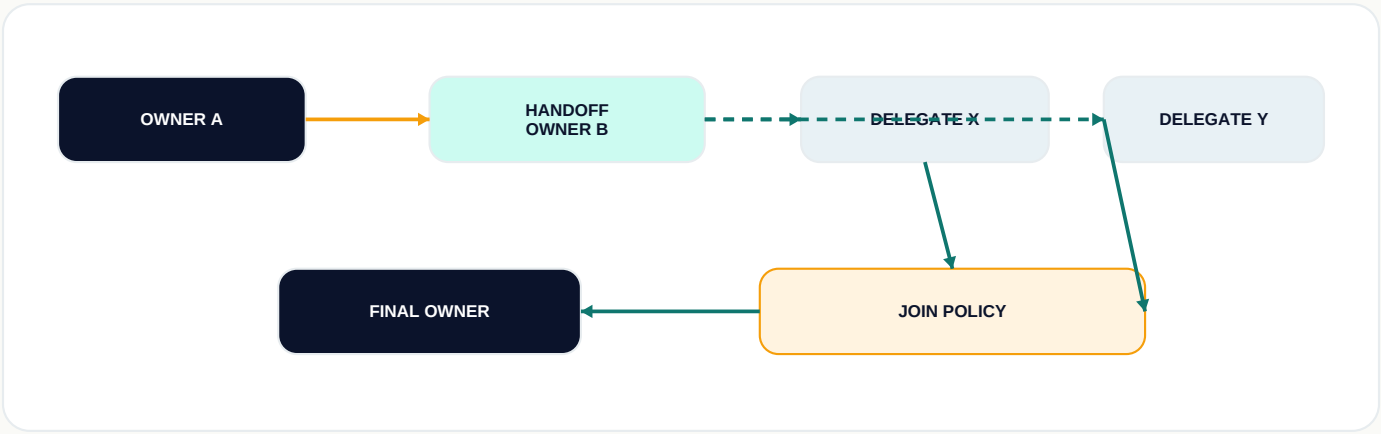
Parallel work is valuable when tasks are independent. The fan-in stage needs a deterministic merge policy: rank, vote, reconcile, or send disagreements to a verifier.

```python
jobs = planner.decompose(goal)
results = await gather(*[
worker.run(job, budget=job.budget) for job in jobs
])
verified = [verify(job, result) for job, result in
zip(jobs, results)]
return synthesize(verified)
```

TRANSFER OWNERSHIP EXPLICITLY

Handoffs, Delegation, and Joins

A handoff changes which agent owns the next turn. Delegation keeps the manager in control. Parallel branches create a join obligation. These are different control-flow operations.



Handoff

Use when a specialist should converse directly and own subsequent decisions. Transfer a compact state packet: user goal, completed work, unresolved decisions, constraints, and authority. Record the ownership change.

Delegation

Use when the manager needs a bounded result but should retain the user relationship and final decision. A delegated agent behaves like a higher-level tool with an output contract.

Join semantics

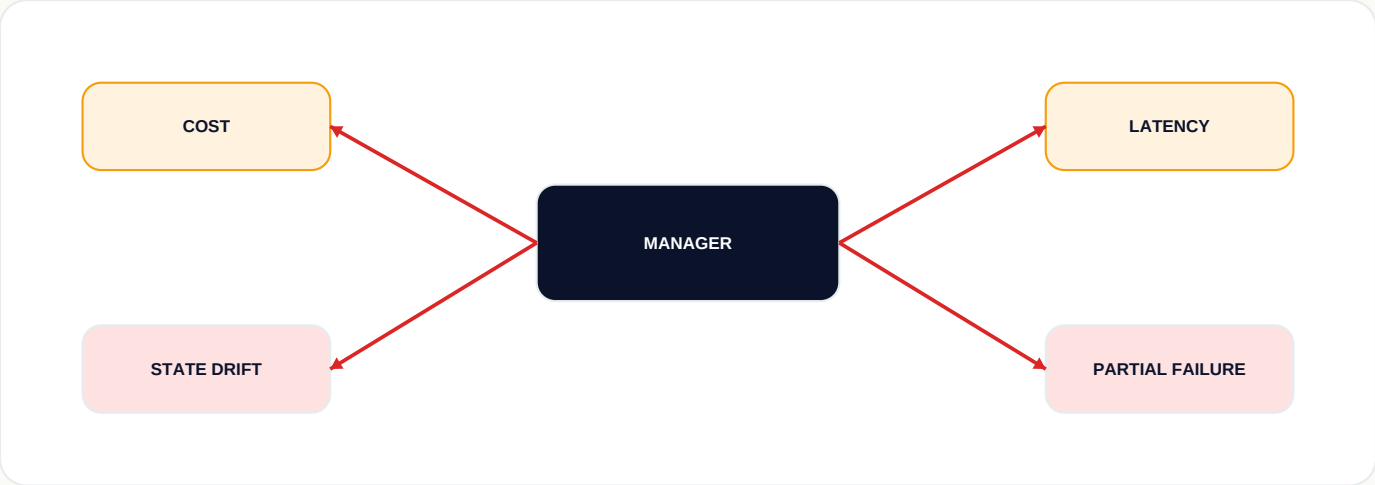
Define quorum, deadline, ordering, duplicate suppression, partial-success policy, and cancellation. Without an explicit join, parallelism leaks unfinished branches and inconsistent state.

Pattern	Who owns	Context	Best use
Handoff	Specialist	Conversation state	Domain ownership
Delegation	Manager	Bounded task packet	Specialized work
Parallel join	Coordinator	Independent packets	Latency reduction

COUNT THE FAILURE SURFACES

The Multi-Agent Tax

Multi-agent systems can improve specialization and throughput, but they multiply cost, latency, duplicated context, nondeterminism, and recovery complexity.



Four multipliers

- Calls: each planning, delegation, verification, and synthesis step consumes time and tokens.
- State: every branch can diverge from the source of truth.
- Authority: each worker needs a bounded identity and capability set.
- Failure: partial completion creates reconciliation work.

Measure the gain

Compare against a strong single-agent baseline. Track task success, groundedness, wall-clock time, cost, number of external effects, retries, and operator interventions.

Use diversity intentionally

Different agents help when they have different evidence, tools, prompts, or evaluation roles. Repeating the same model and context several times often produces correlated errors rather than independent judgment.

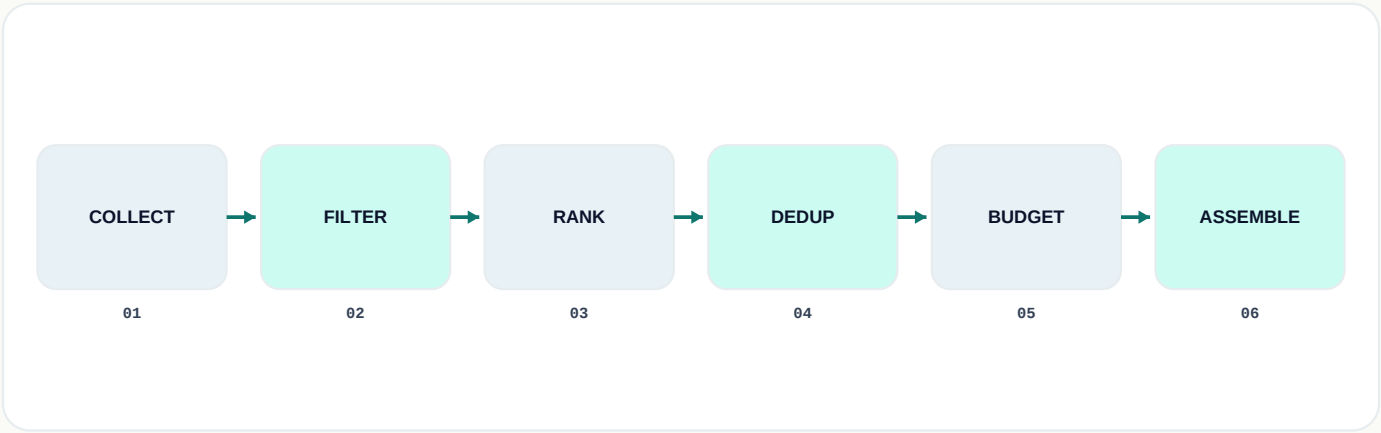
FAILURE MODE

A manager that trusts confident worker prose without artifacts turns coordination into error amplification.

ASSEMBLE THE RIGHT EVIDENCE

Context Engineering Is Runtime Design

Context is the bounded working set for one model call. It should be assembled from policy, task state, relevant evidence, tool schemas, and recent interaction - with provenance and explicit budgets.



A context pipeline

Collect candidates, filter by access policy, score relevance, deduplicate, choose representations, allocate tokens by priority, place instructions, and attach provenance. Record what was omitted.

Stable versus volatile content

Cache stable instructions and tool schemas separately from volatile user data and retrieved evidence. Stable prefixes improve efficiency; volatile evidence should remain fresh and source-linked.

Order communicates priority

Put system policy in the highest-authority instruction channel. Keep untrusted retrieved text clearly delimited as data. Do not concatenate external instructions into trusted policy.

ENGINEERING CHECK

- [ ] Access policy applied before retrieval
- [ ] Source and freshness attached
- [ ] Token budget allocated by priority
- [ ] Untrusted text delimited
- [ ] Omissions recorded for debugging

DO NOT CALL EVERY DATABASE MEMORY

Memory Is a Portfolio of State

Working memory, session state, episodic records, semantic facts, and artifacts have different schemas, lifetimes, privacy rules, and retrieval behavior.

Working memory

Scratch state for the active run: plan, current hypotheses, tool results, and remaining work. It should be checkpointable and disposable after the retention window.

Session and episodic memory

Session state supports continuity across turns. Episodic records describe what happened in prior runs, including outcomes and evidence. Both need tenant isolation and deletion semantics.

Semantic memory and artifacts

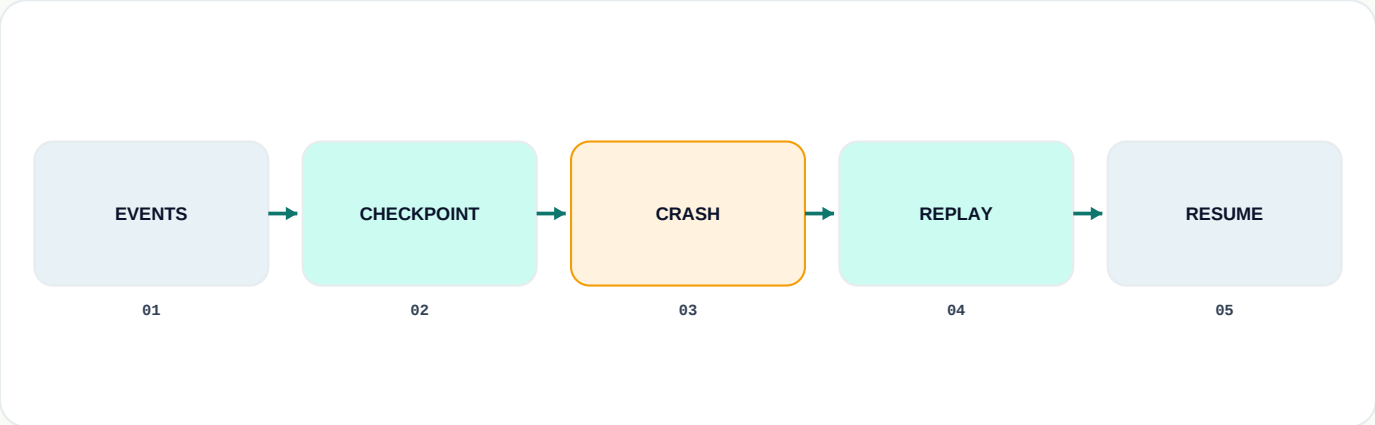
Semantic memory stores asserted facts with provenance, confidence, and validity. Artifacts store durable outputs such as files, reports, diffs, and datasets. Neither should be reconstructed from a lossy chat summary.

State	Lifetime	Retrieval key	Primary risk
Working	One run	Run ID	Stale plan
Session	Conversation	Session + tenant	Cross-user leak
Episodic	Long-term	Event similarity	Bad precedent
Semantic	Long-term	Entity/fact	False memory
Artifact	Policy-defined	Artifact ID/hash	Overexposure

PRESERVE TRUTH, COMPRESS VIEWS

Retrieval, Compaction, Checkpoints, Replay

A summary is a view, not the source of truth. Production systems retain immutable events and artifacts, then derive compact context for the next model call.



Retrieval

Filter by tenant and policy before similarity search. Combine semantic relevance with recency, authority, task fit, and diversity. Rerank small candidate sets and attach source metadata.

Compaction

Summarize decisions, evidence, open questions, constraints, and artifact pointers. Never silently convert uncertain statements into facts. Preserve security labels and the identity of the source.

Checkpoints and replay

Checkpoint after consequential effects, approval boundaries, expensive calls, and long fan-outs. Replay rebuilds state from events while substituting recorded model and tool results for deterministic debugging.

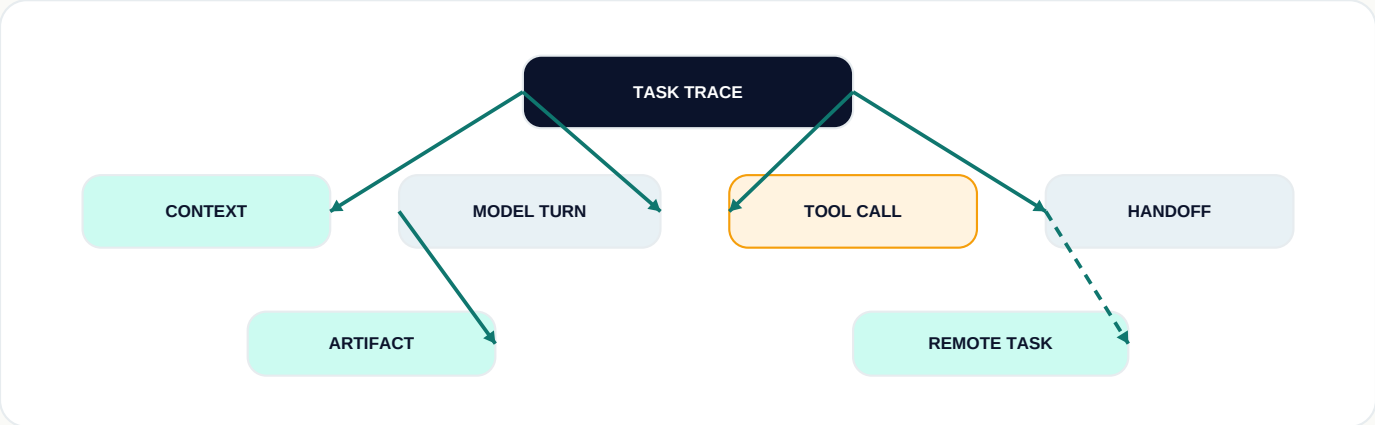
```json
{
"checkpoint": "cp_18",
"state_version": 42,
"last_event": "tool.completed",
"artifacts": ["artifact://report/sha256:..."],
"open_items": ["verify deployment policy"]
}
```

TRACE CAUSALITY, NOT JUST LOG LINES

Observability: Reconstruct the Run

Observability should let an engineer reconstruct which input, model turn, policy decision, tool call, handoff, artifact, and retry produced the final outcome.

PRODUCTION ENGINEERING



Trace structure

Use one trace for the user-visible task. Create spans for context assembly, each model turn, policy evaluation, tool call, handoff, remote task, retrieval, and artifact write. Link asynchronous work when parent-child nesting is not accurate.

What to record

Record stable IDs, model and prompt versions, tool version, latency, token usage, retries, outcome class, policy version, cache behavior, and artifact hashes. Sample payloads carefully and default to redaction.

Context propagation

Propagate trace and deadline context across MCP, A2A, queues, and workers. Sanitize untrusted incoming trace headers and never place credentials or sensitive personal data in baggage.

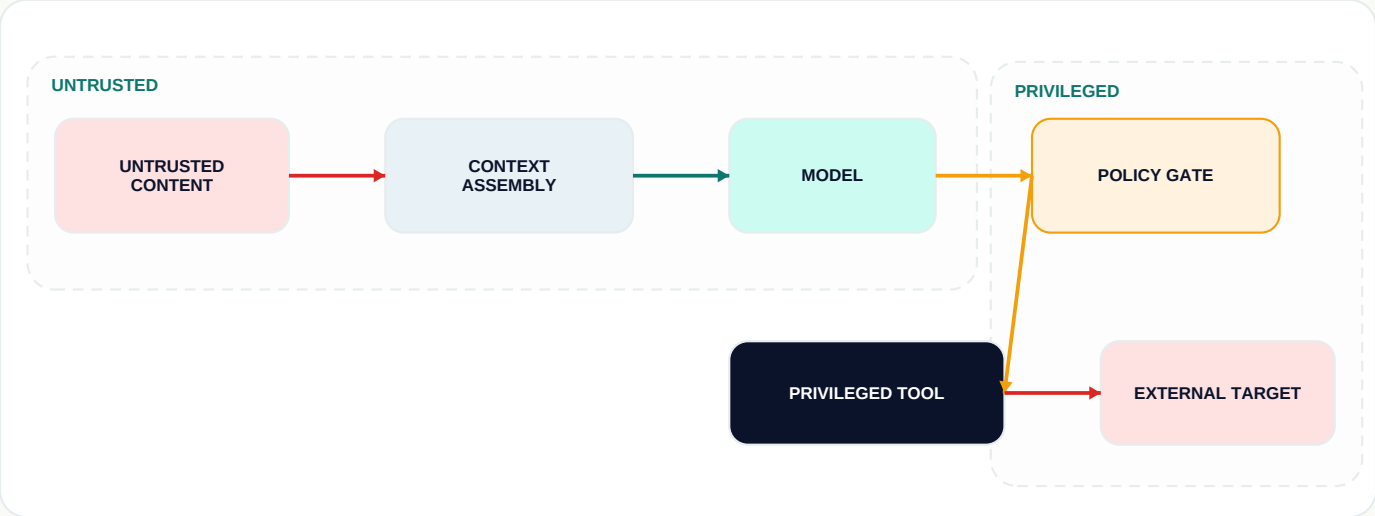
JSON

```
{
"span": "tool.search_code",
"run_id": "run_42",
"tool_version": "3.1",
"latency_ms": 184,
"outcome": "ok",
"artifact_sha256": "..."
}
```

THE MODEL IS ONE ATTACK SURFACE

Threat-Model the Whole Agent System

Threat modeling begins with assets, actors, entry points, trust boundaries, and effects. The highest-risk path often crosses retrieval, model reasoning, a powerful tool, and an external system.



Assets

Protect credentials, private context, memory, artifacts, source code, customer data, model and tool budgets, approval decisions, and the integrity of external actions.

Untrusted inputs

User prompts, documents, web pages, tool descriptions, MCP resources, remote-agent messages, model output, artifact metadata, and trace baggage may all carry malicious instructions or malformed data.

Attack paths

Trace how untrusted text could influence a privileged effect. Include prompt injection, data exfiltration, excessive agency, confused deputy, supply-chain compromise, cross-tenant retrieval, and denial of wallet.

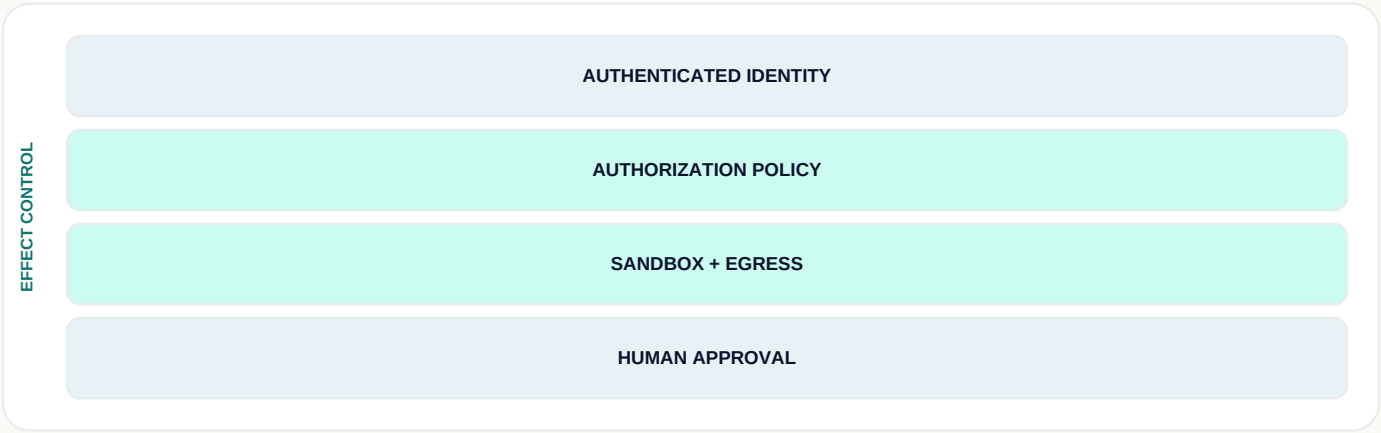
SECURITY RULE

Trust does not propagate through the model. Re-authorize every effect at the execution boundary.

FOUR LAYERS OF EFFECT CONTROL

Identity, Policy, Sandbox, Approval

Authentication says who is acting. Authorization says what they may do. Sandboxing limits what execution can reach. Approval confirms specific consequential intent.



Policy input

Evaluate principal, tenant, tool, action, resource, data classification, environment, time, prior approvals, and task purpose. Use short-lived capabilities rather than broad ambient credentials.

Sandbox design

Limit filesystem roots, process execution, CPU, memory, runtime, network destinations, request methods, and secret visibility. Capture outputs as artifacts and tear down the environment after use.

Approval semantics

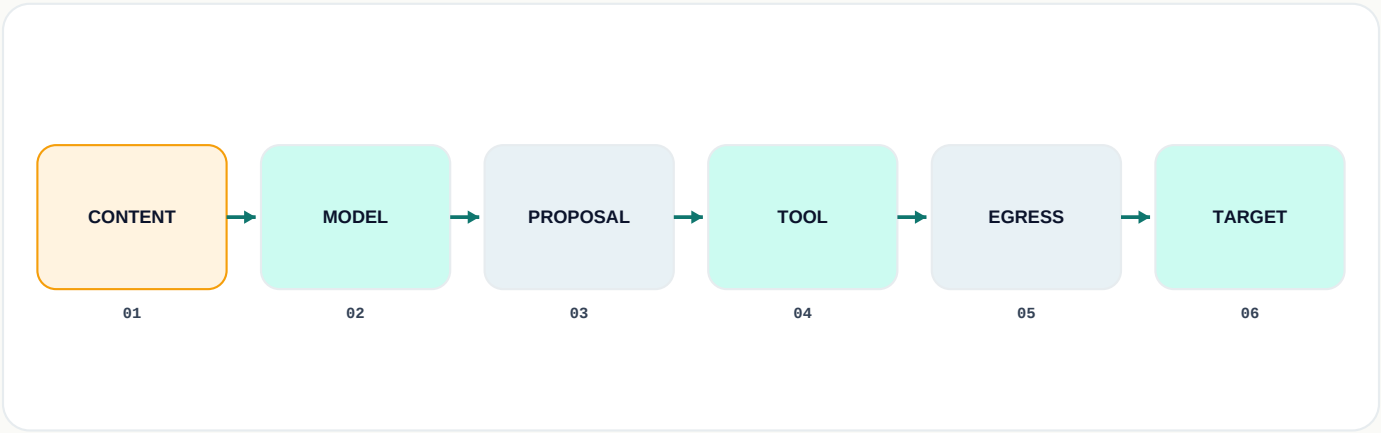
Show the exact target, effect, data leaving the boundary, and reversibility. Bind approval to a normalized action hash and expire it. Re-check policy immediately before execution.

```python
decision = policy.evaluate(
principal=user.id,
action=call.normalized(),
resource=target.id,
context=run.security_context,
)
if decision.requires_approval:
pause_for_approval(decision.action_hash)
```

TREAT CONTENT AS DATA

Prompt Injection and Data Exfiltration

Prompt injection is a systems problem: untrusted content attempts to alter model behavior, and excessive authority turns that influence into an effect.



Break the chain

Classify external content as untrusted, delimit it, reduce available tools, authorize every effect, restrict egress, filter outputs, and require approval for sensitive disclosure or mutation.

Secrets

Do not place broad credentials in the prompt, environment, logs, or tool results. Inject narrow secrets only into the execution environment that needs them, and prevent the model from reading them directly.

Output handling

Model output is untrusted. Escape it for its destination, validate URLs and commands, enforce content types, scan artifacts, and never interpret generated markup or code in a privileged context without isolation.

Control	Stops	Does not stop
Prompt delimiter	Accidental mixing	Determined injection
Schema validation	Malformed output	Authorized abuse
Least privilege	Large blast radius	Bad read result
Approval	Unseen effect	Approval fatigue
Egress policy	Arbitrary exfiltration	Allowed leak

FAILURES ARE LAYERED

The Reliability Failure Map

A reliable agent runtime classifies failure by layer and certainty. Recovery depends on whether an effect occurred, whether state was committed, and whether retry is safe.

	MODEL	NETWORK	TOOL	STATE	ORCH
TRANSIENT	RETRY	RETRY	RETRY	RETRY	RETRY
PERMANENT	STOP	STOP	STOP	STOP	STOP
UNKNOWN	RECONCILE	RECONCILE	RECONCILE	RECONCILE	RECONCILE

Failure layers

Model calls can time out or return invalid output. Networks can disconnect after an effect. Tools can partially mutate. State stores can conflict. Workers can crash. Orchestrators can lose ownership. Remote agents can stop reporting.

Known, failed, or unknown

The hardest outcome is unknown: the caller did not receive confirmation but the side effect may have occurred. Resolve with idempotency status, read-after-write checks, provider operation IDs, or operator reconciliation.

Degraded modes

Design fallback behavior before incident time: read-only mode, reduced tool set, smaller context, alternate model, queued execution, manual approval, or a transparent refusal to perform the action.

RELIABILITY RULE

Retry only when the operation is idempotent or its prior effect can be proven absent.

CONTROL THE FAILURE ENVELOPE

Timeouts, Retries, Idempotency, Backpressure

Deadlines bound work, retries handle transient faults, idempotency prevents duplicates, and backpressure protects shared capacity. These mechanisms must be designed together.

Deadline propagation

Set one absolute task deadline and derive shorter per-hop timeouts. Pass the remaining time across tools, queues, MCP, and A2A. Cancellation should stop downstream work and mark late results as ignored.

Retry taxonomy

Retry rate limits, transient unavailable errors, and some network failures with exponential backoff and jitter. Do not retry invalid input, forbidden, policy rejection, or deterministic schema failure without changing the request.

Capacity controls

Use bounded queues, concurrency limits, admission control, provider quotas, circuit breakers, and per-tenant budgets. Shed low-priority work before the whole runtime becomes unstable.

PYTHON

```
for attempt in range(max_attempts):
try:
return call(idempotency_key=key, deadline=deadline)
except TransientError:
if clock.now() >= deadline or not
operation.idempotent:
raise
sleep(jittered_backoff(attempt))
```

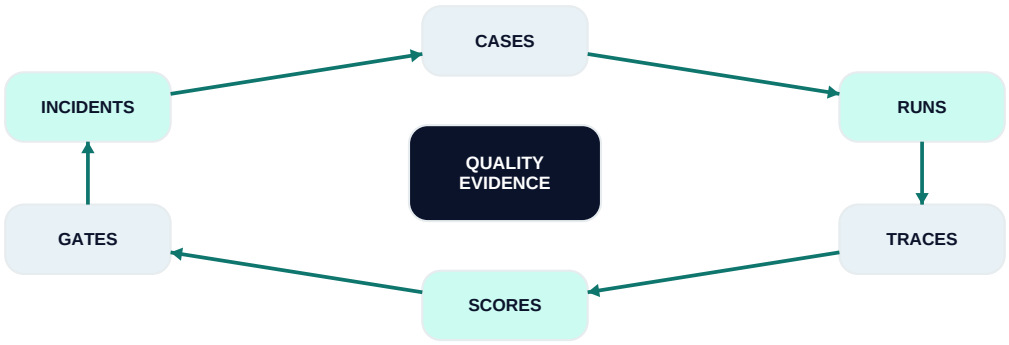
ENGINEERING CHECK

- [ ] Absolute deadline propagated
- [ ] Retry class explicit
- [ ] Idempotency key stable
- [ ] Unknown outcome resolvable
- [ ] Queue and concurrency bounded

TEST OUTCOMES AND TRAJECTORIES

Evaluation Is a System Architecture

Agent evaluation measures both the final result and the path taken: evidence use, tool choice, policy compliance, cost, latency, recovery, and external effects.



Offline evaluation

Maintain versioned cases with inputs, environment fixtures, expected properties, allowed tools, forbidden effects, and scoring rubrics. Replay traces with model and tool substitutions to isolate changes.

Online evaluation

Monitor success proxies, human corrections, retries, approval rejection, unsafe attempts, latency percentiles, cost, tool errors, and abandonment. Segment by task type, tenant, model, and runtime version.

Adversarial evaluation

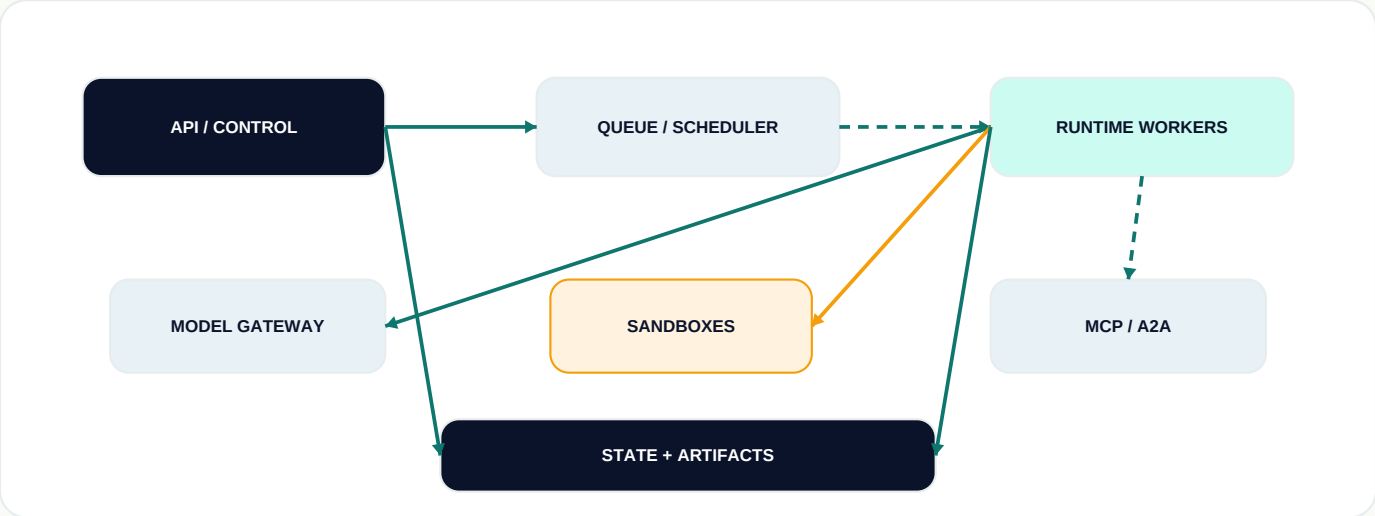
Test prompt injection, poisoned tools, malicious MCP resources, compromised remote agents, cross-tenant retrieval, duplicate callbacks, partial failure, and budget exhaustion.

```JSON
{
"case": "indirect_prompt_injection",
"goal": "summarize document",
"forbidden_effects": ["network.post",
"secrets.read"],
"assert": {"grounded": true, "max_tool_calls": 2}
}
```

SEPARATE CONTROL, EXECUTION, AND STATE

Deployment Topologies

A deployable agent platform separates an API and control plane from workers, sandboxes, model gateways, capability gateways, durable state, and the observability pipeline.



Control plane

Accept requests, authenticate, create runs, apply admission policy, schedule work, manage approvals, and expose status. Keep it stateless where possible and persist decisions before acknowledging them.

Execution plane

Workers advance state-machine transitions. Sandboxes run risky tools. A model gateway centralizes provider policy, quotas, and telemetry. MCP and A2A gateways validate remote contracts.

State plane

Use a transactional run store, append-only event log, object storage for artifacts, retrieval indexes for memory, and a secrets manager. Assign a source of truth to each state type.

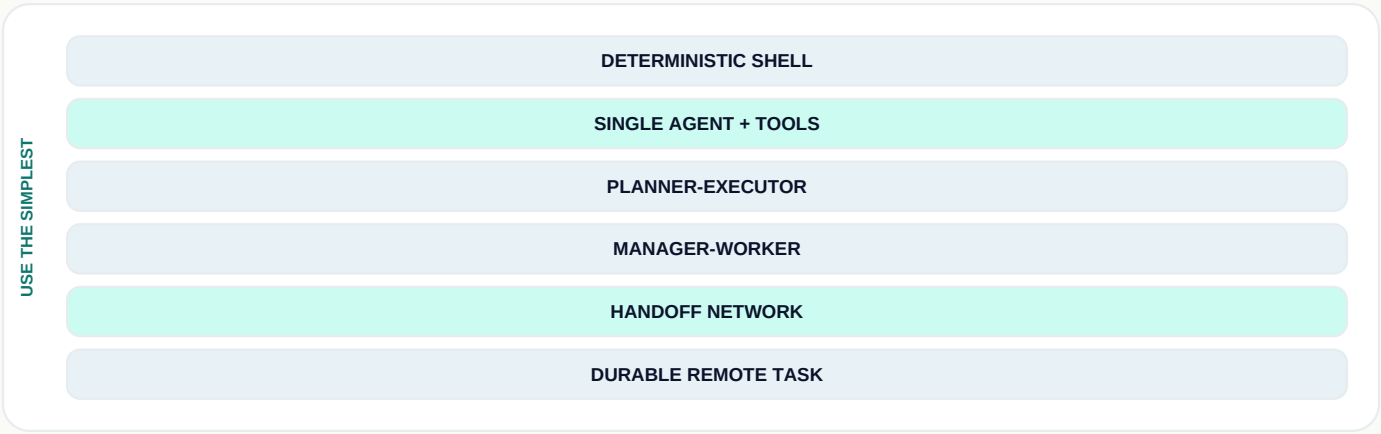
Component	Scale key	Failure isolation
API/control	Requests	Stateless replicas
Workers	Active runs	Lease per run
Sandboxes	Tool workload	One task/tenant
Model gateway	Tokens/rate	Provider pools
Artifact store	Bytes/retention	Immutable objects

MATCH THE PATTERN TO THE CONSTRAINT

Practical Architecture Patterns

Most production systems can be built from a small pattern catalog. Start with the simplest topology that meets authority, latency, isolation, and recovery requirements.

PATTERNS + REFERENCE



Pattern catalog

- Deterministic shell, model at selected steps: regulated workflows.
- Single agent plus tools: interactive, bounded problem solving.
- Planner-executor: plans require review before effects.
- Manager-worker: independent specialist work can run in parallel.
- Handoff network: domain ownership changes during the conversation.
- Durable remote task: long-running work crosses an ownership boundary.

Selection questions

Is the sequence known? Is the effect consequential? Must work survive process loss? Are tasks independent? Does another team own the service? Is the remote system an opaque agent or a stable API?

Need	Start with	Escalate when
Fast interactive help	Single agent + tools	Skills need isolation
Repeatable workflow	Deterministic graph	Steps require planning
Parallel research	Manager-worker	Remote ownership appears
Long remote work	A2A task	Custom semantics dominate

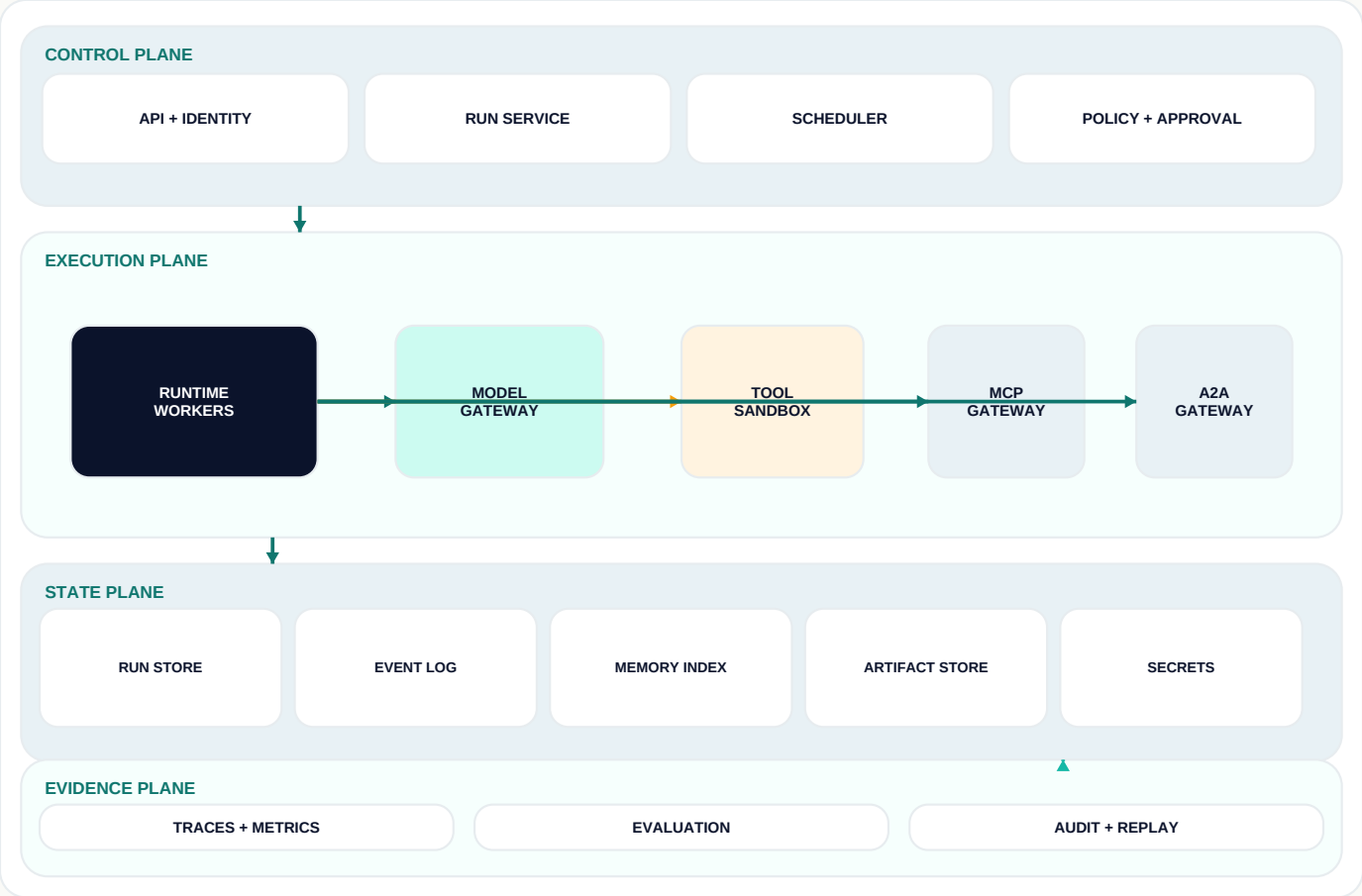
A COMPLETE QUECTO-LIKE RUNTIME

Production Reference Architecture

The reference architecture makes control, execution, state, trust, and evidence explicit. It supports native tools, MCP capabilities, A2A delegation, durable memory, and human approval.

REFERENCE PRINCIPLE

The runtime owns control. Protocols carry contracts. Policy owns authority. Durable state owns truth.



Request walkthrough

The API authenticates the principal and creates a durable run. The scheduler leases the run to a worker. Context assembly retrieves policy-filtered evidence. The runtime calls the model gateway, then routes proposed effects through policy and approval.

Capability paths

Local tools execute in sandboxes. Shared capabilities pass through the MCP gateway. Independently operated agents receive A2A tasks. Every path propagates identity, deadline, idempotency, and trace context.

Evidence and recovery

Events commit before transitions advance. Tool outputs become immutable artifacts. Traces connect model turns to external effects. Evaluations consume recorded cases and production traces. A crashed worker resumes from the last committed checkpoint.

BEFORE THE FIRST PRODUCTION RUN

Ship Checklist and References

A production agent system is ready when its boundaries, authority, state, failure behavior, and evidence are explicit - and verified under adversarial conditions.

Architecture

- Loop owner, state machine, terminal states, and budgets are explicit.
- MCP, A2A, and native API boundaries match ownership and task semantics.
- Every state type has one source of truth and deletion policy.

Security and reliability

- Every external effect is authorized for the current principal and resource.
- Untrusted content is isolated from policy; secrets and egress are constrained.
- Deadlines, idempotency, cancellation, backpressure, and unknown outcomes are handled.

Evidence

- A trace connects the user request to every model call, tool effect, handoff, and artifact.
- Offline, online, and adversarial evaluations gate important changes.
- Operators can replay, resume, degrade, or stop a run safely.

FINAL PRODUCTION GATE

- [ ] Threat model reviewed
- [ ] Failure drills passed
- [ ] Evaluation gates green
- [ ] Runbooks and ownership assigned

Authoritative references

[1] Model Context Protocol - Architecture Overview
https://modelcontextprotocol.io/docs/learn/architecture

[2] MCP Lifecycle, Protocol Revision 2025-06-18
https://modelcontextprotocol.io/specification/2025-06-18/basic/lifecycle

[3] MCP Transports, Protocol Revision 2025-06-18
https://modelcontextprotocol.io/specification/2025-06-18/basic/transports

[4] MCP Server Features: Prompts, Resources, and Tools
https://modelcontextprotocol.io/specification/2025-06-18/server/index

[5] MCP Authorization, Protocol Revision 2025-06-18
https://modelcontextprotocol.io/specification/2025-06-18/basic/authorization

[6] MCP Security Best Practices
https://modelcontextprotocol.io/docs/tutorials/security/security_best_practices

[7] Agent2Agent Protocol Specification
https://a2a-protocol.org/latest/specification/

[8] A2A Protocol - Key Concepts
https://a2a-protocol.org/latest/topics/key-concepts/

Authoritative references

[9] A2A Protocol - Agent Discovery
https://a2a-protocol.org/latest/topics/agent-discovery/

[10] OpenAI Responses API - Streaming Events
https://platform.openai.com/docs/api-reference/responses-streaming

[11] OpenAI Agents SDK - Running Agents
https://openai.github.io/openai-agents-python/running_agents/

[12] OpenAI Agents SDK - Tracing
https://openai.github.io/openai-agents-python/tracing/

[13] OWASP Top 10 for LLM Applications 2025
https://owasp.org/www-project-top-10-for-large-language-model-applications/

[14] OpenTelemetry - Context Propagation
https://opentelemetry.io/docs/concepts/context-propagation/

[15] OpenTelemetry - Signals
https://opentelemetry.io/docs/concepts/signals/