Appendix: Agentic Workflow Design

Most agent failures are architecture failures, not model failures.

A team picks a model, connects it to tools, writes a system prompt, and launches. The agent hallucinates a tool call. It loops on a failed action. It burns 200,000 tokens chasing a dead end. The team blames the model, swaps in a better one, and watches the same failures recur with more eloquent reasoning.

The model was never the problem. The architecture was. You cannot govern a badly designed agent into reliability any more than you can inspect quality into a badly designed factory. The governance has to be designed in — at every decision point, at every handoff, at every place the system can go wrong. This chapter is about how to build agent systems that are governable by construction, not by hope.

START SIMPLE: WORKFLOW BEFORE AGENT

Here is the most expensive mistake in agentic AI: building an agent when a workflow would do.

Anthropic draws a sharp line. A **workflow** is a system where LLMs and tools are orchestrated through predefined code paths. The developer controls the flow. An **agent** is a system where the LLM autonomously directs processes and tool usage, maintaining control over how tasks are accomplished. The developer sets the goal and the guardrails. The LLM decides the steps.

The difference matters because governance complexity scales with autonomy. A workflow has a finite number of paths. You can test them. You can trace them. You can put a gate at every junction. An agent has an unbounded number of paths. You can constrain them, but you cannot enumerate them.

Think of it like a GPS versus a taxi driver. The GPS gives you turn-by-turn directions. You know exactly where you are going, and you can verify each turn. The taxi driver takes you where you asked to go, and you trust them to figure out the route. Both get you there. One is dramatically easier to audit.

Anthropic's guidance is blunt: "Start with simple prompts, optimize them with comprehensive evaluation, and add multi-step agentic systems only when simpler solutions fall short." Most teams do the opposite. They jump to agents because agents feel more impressive. The result is unpredictable systems that are expensive to test and hard to debug.

When you actually need an agent: the task has no predictable step sequence. The number and type of subtasks cannot be determined in advance. The system needs to recover from errors in ways you

cannot enumerate. If you can draw a flowchart of the task, you do not need an agent. You need a workflow.

THE FIVE WORKFLOW PATTERNS

These five patterns come from Anthropic's guidance and recur across every major implementation. They are composable building blocks. Each one has a natural place for governance gates.

Prompt chaining. Decompose a task into sequential steps. Each LLM call processes the output of the previous one. Best for fixed subtasks that require accuracy over speed. The governance gate fits between every step — a programmatic checkpoint that validates the output before passing it forward. This is where your **contract gates** live: did the output match the expected schema? Did it respect the constraints from the spec?

Routing. Classify the input and direct it to a specialized downstream process. A customer service query goes to billing, technical support, or account management based on classification. Best for tasks with distinct categories requiring different handling. The governance gate is the classifier itself — an **invariant gate** that ensures every input gets routed and no input falls through to an unhandled path. Misrouted inputs are the silent killer here.

Parallelization. Run independent subtasks concurrently, or run the same task multiple times for confidence. Best for speed improvement or diverse verification. Two variants: sectioning (divide the work) and voting (repeat the work). The governance gate is the aggregator — a **policy gate** that defines how conflicting outputs are resolved. If three parallel safety checks disagree, what wins? That policy must be deterministic, not left to another LLM call.

Orchestrator-workers. A central LLM dynamically decomposes a task and delegates to worker LLMs. Best for tasks where subtask requirements are unpredictable, like multi-file code changes. The governance gate sits between the orchestrator's plan and the workers' execution — a **contract gate** that validates each delegated task against the original spec. The orchestrator proposes. The gate checks. The workers execute.

Evaluator-optimizer. One LLM generates, another evaluates and critiques. Iterative refinement in a loop. Best for tasks with clear evaluation criteria. The governance gate is the exit condition — an **invariant gate** that determines when the output is good enough and caps the number of iterations. Without a hard cap, this pattern will loop until you run out of tokens or patience.

For each pattern, notice the same principle: the gate is deterministic code, not another LLM call. The LLM does the creative work. The gate does the verification. Mix them and you lose the ability to reason about what your system will do.

THE BLUEPRINT PATTERN

Stripe merges over 1,300 agent-produced PRs per week. The architecture that makes this possible is the blueprint: a state machine that alternates deterministic and LLM steps.

Picture an assembly line. Some stations are robots — fast, creative, unpredictable. Other stations are jigs and fixtures — dumb, rigid, absolutely reliable. The jigs do not make anything. They verify that what the robot made is the right shape before it moves to the next station. That is the blueprint pattern.

A Stripe minion blueprint looks like this:

1. **Pre-push linting** (deterministic) — no LLM involvement. Code must pass static analysis.
2. **Implementation** (agentic) — LLM writes or modifies code.
3. **Autofix application** (deterministic) — automated formatting and style corrections.
4. **CI iteration** (hybrid) — run tests. If they fail, the agent gets up to two attempts to fix.
5. **Human handoff** (deterministic) — after two push attempts, code returns to a human. Every PR gets human review regardless.

The deterministic nodes save tokens, reduce error surface, and enforce compliance. The agentic nodes do the creative work that requires reasoning. The alternation is the key insight: never let the LLM do two creative steps in a row without a deterministic check between them.

Where the three gate types fit in a blueprint:

- **Contract gates** between Steps 2 and 3: does the implementation match the interface spec? Are the function signatures correct? Do the types align?
- **Invariant gates** in Step 4: do the tests pass? Do the business rules hold? Does the change preserve idempotency where required?
- **Policy gates** before Step 5: no secrets in the diff. No PII in logs. No unauthorized access patterns. Compliance rules that are non-negotiable regardless of code quality.

The blueprint pattern is the most production-proven architecture for agentic work at scale. It works because it treats the LLM as a powerful but unreliable collaborator that needs a jig at every step.

AGENT DESIGN WHEN YOU ACTUALLY NEED AGENTS

You have exhausted the workflow patterns. The task genuinely requires autonomous decision-making. Before you build, answer three questions.

What is the exit condition? An agent without an exit condition is a while loop without a break statement. It will run until something external stops it. Define what "done" looks like in terms the agent and your governance system can evaluate. "The tests pass and the diff is under 500 lines" is an exit condition. "The code is good" is not.

What is the blast radius? If the agent makes the worst possible decision at its most autonomous moment, what happens? Stripe's answer: nothing catastrophic, because each minion runs on an isolated devbox with no internet and no production access. The blast radius is bounded by design, not by the agent's judgment. If you cannot describe the worst case and live with it, you need tighter constraints.

What happens when the agent is wrong? Not if. When. Design the recovery path before you design the happy path. Circuit breakers that cut off after N failures. Human escalation that packages the agent's context and reasoning for a human to inspect. Rollback mechanisms that undo what the agent did. The recovery path must be deterministic. If you need another LLM call to recover from an LLM failure, you have compounded the problem.

Tool design is where most agent implementations break. Anthropic's guidance: self-contained tools with minimal functional overlap, token-efficient outputs, descriptive parameter names. Stripe limits its agents from 400+ available tools to roughly 15 relevant ones per task. Too many tools cause "token

paralysis" — the agent spends its reasoning budget deciding which tool to use rather than using tools. Curate aggressively. A **policy gate** on tool selection prevents the agent from calling tools outside its approved set.

Error recovery follows a hierarchy: retry with backoff for transient errors, semantic fallback for output quality failures, circuit breaker for persistent failures, human escalation as the last resort. Stripe caps at two push attempts, then escalates. The cap is a **contract gate** — a hard limit that prevents diminishing-return loops.

MULTI-AGENT TOPOLOGIES

When a single agent is not enough, you need multiple agents coordinating. Three topologies dominate production use.

Centralized supervisor. One orchestrator agent decomposes the task and delegates to specialized workers. Anthropic's Claude Code Review uses this: a dispatcher sends a team of review agents to examine a PR from different angles, then synthesizes their findings. Best for tasks with clear decomposition. The governance gate is the supervisor itself — but do not trust it blindly. Put a **contract gate** between the supervisor's plan and the workers' execution.

Decentralized peer-to-peer. Agents hand off control directly to each other. OpenAI describes this as effective for conversation triage. Agent A determines the query is a billing issue and hands off to Agent B. No central coordinator. Best for routing scenarios. The governance gate is the handoff protocol — a **policy gate** that validates each handoff and prevents circular delegation.

Hierarchical. Manager agents delegate to team leads, who delegate to workers. Best for large-scale decomposition. Most complex to govern because failures cascade through the hierarchy. This is where the research gets sobering.

A study of 150+ execution traces across five multi-agent frameworks found 14 distinct failure modes. No single error category dominates. Failures are systemically diverse. A Cooperative AI Foundation study tested 180 configurations and found that unstructured multi-agent networks amplify errors up to 17.2x compared to single-agent baselines. Adding agents does not add reliability. It multiplies failure modes.

Cascade prevention: put a circuit breaker at every agent boundary. If Agent B fails, Agent A gets a structured error, not a garbled context window. Every inter-agent message should be schema-validated — a **contract gate** that prevents malformed output from one agent from corrupting the next. Tag data provenance so you can trace which agent produced which output when the post-incident review starts.

The honest guidance: use the fewest agents possible. Every agent you add is another source of non-determinism, another context window to manage, another failure mode to test. If you can solve the problem with one agent and three tools, do not build three agents with one tool each.

CONTEXT MANAGEMENT

Context is the practical problem that kills most agents. Not capability. Not reasoning. Context.

An agent starts a task with a clear goal and relevant information. Fifty tool calls later, the context window is packed with results, errors, retries, and intermediate reasoning. The agent starts losing track

of its original goal. It repeats actions it already tried. It contradicts decisions it made twenty steps ago. The model did not get dumber. The context got noisy.

Anthropic's guiding principle: "Find the smallest set of high-signal tokens that maximize the likelihood of your desired outcome." Every token in the context window competes for the model's attention. Irrelevant tokens dilute relevant ones.

Summarization. Compress older conversation segments while preserving architectural decisions and unresolved issues. Manus, Anthropic, and Stripe all converge on this. The benefit goes beyond staying within token limits — context growth is actively distracting. Models rely less on their training as the context grows.

Sliding window. Fixed-size context buffer. New information enters, old exits. Assumption: recent context matters most. Simple and effective for short-horizon tasks. Dangerous for long-horizon tasks where early decisions constrain later ones.

RAG for tool selection. Apply retrieval-augmented generation to tool descriptions. Fetch the most relevant tools based on semantic similarity to the current task. Research shows 3x improvement in tool selection accuracy. This is how Stripe gets from 400+ tools to 15 relevant ones per minion.

Task recitation. Manus constantly rewrites the to-do list to push the global plan into the model's recent attention span. This counters the "lost-in-the-middle" problem where information in the center of a long context gets less attention than information at the beginning or end.

The 50% signal. When your context window is more than half full, agent performance starts degrading. This is the point to compact, summarize, or spawn a sub-agent with a fresh context that receives a condensed briefing (1,000-2,000 tokens) from the parent.

An **invariant gate** on context health: monitor context utilization per agent turn. When it crosses your threshold, trigger compaction automatically. Do not leave this to the agent's judgment. The agent cannot reliably assess its own context quality.

SCOPING FOR UNATTENDED RUNS

The overnight agent is the organizational design question from Finding 4 applied to architecture. You queue a task at 6 PM. The agent works overnight. You review the result at 9 AM. This only works if the task is scoped correctly.

What makes a task safe for unattended execution:

- The blast radius is bounded. The agent operates in an isolated environment. Stripe's devboxes: no internet, no production access.
- The exit condition is machine-evaluable. Tests pass or they do not. The diff is within scope or it is not.
- The rollback path is automatic. If the agent produces garbage, a git reset restores the starting state.
- The permission set is minimal. Just-in-time access granted for the task, revoked immediately after.
- The iteration cap is hard. Two attempts at CI, then stop. Not "keep trying until it works."

The spec template for an unattended run:

```
Task:           [One-sentence description]
Scope:          [Files/modules in scope]
Non-goals:      [What the agent must NOT touch]
Exit criteria:  [Machine-evaluable conditions]
Iteration cap:  [Maximum attempts]
Blast radius:   [What is the worst that can happen]
Rollback:       [How to undo everything]
```

Every field in that template maps to a governance gate. The scope is a **policy gate** — the agent cannot modify files outside the declared scope. The exit criteria are **contract gates** — did the output meet the spec? The iteration cap is an **invariant gate** — the system stops regardless of the agent's confidence.

Anthropic's long-running harness uses a two-agent pattern: an initializer agent creates the setup script, progress file, initial commit, and structured feature list. A coding agent in subsequent sessions reads the progress files, works on single features incrementally, commits changes, and updates progress. Git history, progress files, and feature JSON bridge sessions explicitly. The context window alone is not enough.

Critical insight from Anthropic's research: agents must be explicitly prompted to test as a human would — browser automation, end-to-end testing — rather than declaring features complete without validation. An agent that says "done" without running the tests is not done. That is a **contract gate** failure in your harness design.

THE HANDOFF: AGENT OUTPUT TO HUMAN REVIEW

The interface between agent and human determines whether your human reviewers are effective or just tired.

Approval gate. The agent completes work and enters a holding state until a human signals approval. Every content generation and financial transaction should use this pattern. The gate is deterministic: work does not proceed without explicit human action.

Escalation trigger. The agent monitors its own confidence. When uncertainty exceeds a threshold, it escalates. Anthropic's research shows Claude Code asks clarification questions more than twice as often as humans interrupt — self-aware uncertainty is an important safety mechanism. The trigger threshold is an **invariant gate**: set it in code, not in the prompt.

Active handoff. The agent pauses, packages its current intent, reasoning, and context as structured data, and hands control to a human. This is not "here is 500 lines of conversation." This is: "I was trying to do X. I got stuck on Y. Here are the three options I considered and why I recommend Z." Stripe's model: after two push attempts with CI failures, the code returns to a human with full context. The human does not start from scratch.

The handoff quality determines your human review cost. A bad handoff dumps raw context on a reviewer. A good handoff is a structured brief. Design the handoff format as carefully as you design the agent's tools.

CROSS-ORGANIZATION AGENT INTERACTIONS

This is the frontier problem. Be honest: it is unsolved.

The vast majority of multi-agent deployments in 2025-2026 are intra-organizational. Cross-organizational agent-to-agent interaction is largely still in pilot phase. The one major exception is commerce: Visa's Trusted Agent Protocol and Mastercard's Agent Pay have completed hundreds of secure agent-initiated transactions with early partners. These are tightly constrained — purchase transactions with cryptographic verification — not general-purpose collaboration.

Why payment networks are ahead: they already had the trust infrastructure. Visa and Mastercard have spent decades building a network where every participant is enrolled, verified, and bound by contract. Adding agent-to-agent transactions is an extension of an existing trust layer, not a new one.

For everyone else, the gaps are real. 81% of enterprises lack documented governance for machine-to-machine interactions. Cross-organizational prompt injection attacks succeed at rates exceeding 85% when adaptive strategies are used. Delegation chains — Agent A acts on behalf of User X, calls Agent B, which calls Agent C — have no production-ready verification mechanism.

What the responsible approach looks like in 2026:

1. Do not allow general-purpose agent-to-agent communication across organizational boundaries. Constrain to specific, pre-defined interaction types with schema validation.
2. Use gateway infrastructure on both sides. Every interaction proxied through a policy enforcement layer.
3. Require cryptographic agent identity. Never trust a claim without proof.
4. Human-in-the-loop for novel interactions. Anything outside pre-approved patterns requires human approval.
5. Assume prompt injection will succeed. Design so a compromised agent cannot cause catastrophic damage.
6. Start with read-only interactions. Let agents query each other before they transact.

The payment network model — enrolled participants, cryptographic verification, constrained interaction types, established liability frameworks — is the blueprint for everyone else. It will take years. Deploying cross-organization agents before the trust infrastructure exists is how you get the MCP supply chain attack timeline, but across company boundaries.

TESTING AGENTS

Testing an agent is not like testing a function. A function has inputs and outputs. An agent has inputs, outputs, and an unbounded number of intermediate decisions. The same input can produce different outputs on different runs. The output might be correct but the reasoning path might be dangerous — a right answer reached through a wrong method that will fail on the next input.

The eval loop from the Verification Triangle chapter applied to agents:

- **Contract evals:** Does the agent's output match the expected schema? Does it respect interface boundaries? Does the API contract hold?
- **Invariant evals:** Do business rules survive the agent's modifications? Does idempotency hold? Are there double charges, negative balances, broken referential integrity?
- **Policy evals:** Did the agent stay within its permission boundaries? Did it access only approved tools? Did it respect data classification rules?

Behavioral testing. SWE-bench and its variants test whether an agent can resolve real GitHub issues. Current leaders exceed 70% on SWE-bench Verified. But benchmarks overestimate capabilities by over 50% compared to how developers actually interact with agents in IDEs. SWE-CI, a new repository-level benchmark, measures something harder: whether an agent's patches support future code advancement, not just current correctness. Even the strongest models struggle to maintain quality over long development periods.

Trace grading. Do not just evaluate the final output. Grade the reasoning trace. Did the agent consider the right alternatives? Did it recover sensibly from errors? Did it escalate when it should have? A right answer via a lucky path is not the same as a right answer via a sound process. Anthropic builds alignment auditing agents that investigate target LLMs for defects — the investigator finds the correct root cause 10-13% of the time alone, but super-agent aggregation improves this to 42%.

The testing gap most teams miss: testing the governance gates themselves. Your circuit breaker should actually fire. Your escalation trigger should actually escalate. Your permission boundary should actually block. Test the gates with adversarial inputs — not just happy-path inputs that never trigger them. A gate that has never fired in testing will not fire reliably in production.

TEN DESIGN PRINCIPLES

1. **Use a workflow until you cannot.** Most tasks do not need an agent. A deterministic orchestration with LLM steps is safer, cheaper, and easier to govern.
2. **Never let the LLM do two creative steps in a row.** Put a deterministic check between every agentic step. The blueprint pattern is the production standard.
3. **Define the exit condition before you start the agent.** An agent without an exit condition is a while loop without a break.
4. **Bound the blast radius by architecture, not by instruction.** Sandboxed environments, scoped permissions, isolated devboxes. The agent's judgment is not a safety mechanism.
5. **Curate tools like you curate dependencies.** Fifteen relevant tools, not four hundred available ones. Token paralysis is real.
6. **Treat context as a finite, depletable resource.** Monitor utilization. Compact aggressively. The 50% mark is when performance starts degrading.
7. **Design the recovery path before the happy path.** Circuit breakers, human escalation, rollback. If recovery requires another LLM call, you have compounded the problem.
8. **Test the gates, not just the agent.** A permission boundary that has never been tested is a suggestion, not a control.
9. **Cap iterations.** Two attempts, then escalate. Diminishing returns on retries cost tokens and hide the real problem.
10. **Infrastructure built for humans works for agents.** Developer experience investments — devboxes, CI, linters, rule files — compound into agent reliability. Build for humans first.

A well-designed agent with no governance drifts silently. A well-governed agent with bad architecture fails loudly and often. You need both: systems that are built to be governed and governance that is built into the system.

The agent is not the product. The agent is a component. The product is the delivery system — spec, agent, gates, traces, review, deployment — that produces trusted changes at speed.

You now have the architecture. You have the governance. The main text puts all of it in front of the people who sign the checks.