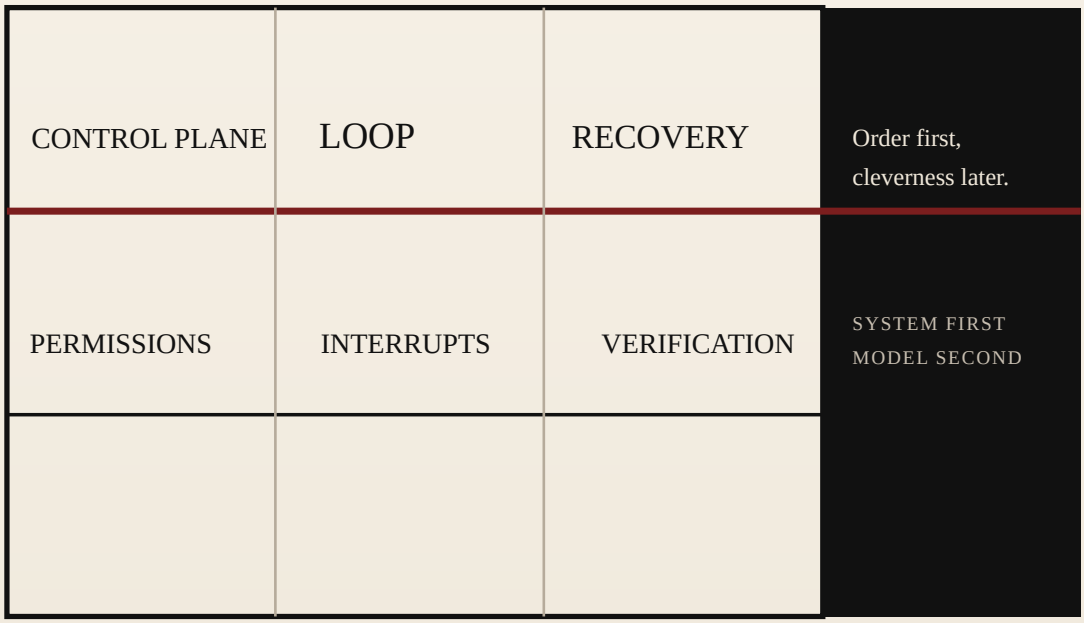
AGENTWAY / HARNESS BOOKS



Harness Engineering

A Design Guide to Claude Code

A system is judged reliable
not by how smoothly it speaks,
but by how it restores order
when execution goes wrong.

CONTROL PLANE / QUERY LOOP / RECOVERY / VERIFICATION

Engineering principles for controllable AI coding systems
distilled from source-level runtime design

agentway.dev

From using agents to shipping an agent PoC

https://agentway.dev

2026-04-01 · rev fbf2b4

Contents

Introduction 1

Preface 3

Chapter 1 Why Harness Engineering Matters 6

1.1 The core problem is keeping the model from going off script . . 6

1.2 Claude Code's first harness layer: a constrained conversation system 6

1.3 The second harness layer: the agent depends on a continuous loop 7

1.4 The third harness layer: tool calls must obey scheduling discipline 8

1.5 The fourth harness layer: the most dangerous tool needs the strictest rules 9

1.6 The fifth harness layer: errors are part of the main path 10

1.7 The first principle extractable from source 10

Chapter 2 Prompt Is Not Personality 12

2.1 Treating prompt as persona is a common misunderstanding . . 12

2.2 In source, Claude Code prompt is layered from the beginning . . 12

2.3 Prompt value is not text itself, but precedence structure 13

2.4 Prompt is not static copy; it is connected to memory systems . . 14

2.5 A real control plane must include caching and compute cost . . 15

2.6 Users can override prompt, but cannot bypass this structure . . 16

2.7 Why prompt here is closer to constitution than dialogue lines . . 17

2.8 The second principle extractable from source 17

Chapter 3 Query Loop 19

3.1 To judge maturity, first ask whether the system has a loop 19

3.2 State is core business, not baggage 19

3.3 The first duty of query loop is input governance 22

3.4 Model invocation is one phase of the loop, not the loop itself . . 22

3.5 A heartbeat must handle interrupts, or it is just inertia 23

3.6 A heartbeat must also recover, or it is fragile repetition 24

3.7 Stop conditions cannot be singular, or failure and completion get

3.8 QueryEngine proves this belongs to conversation lifecycle 26
3.9 The third principle extractable from source 29

Chapter 4 Tools, Permissions, and Interrupts **30**

4.1 Once models call tools, the nature of risk changes 30
4.2 Tool orchestration is part of behavioral constitution 30
4.3 A lot happens before a tool actually runs 31
4.4 Permission comes before capability 31
4.5 Permission results are runtime semantics, not booleans 32
4.6 StreamingToolExecutor proves interrupt is first-class semantics . 34
4.7 Why Bash is always more suspect than other tools 37
4.8 Tool systems protect users and protect runtime itself 37
4.9 The fourth principle extractable from source 38

Chapter 5 Context Governance **39**

5.1 As context grows, systems develop a low-level illusion 39
5.2 The `CLAUDE.md` system: long-lived instructions cannot be mixed with ad hoc chat 39
5.3 `MEMORY.md` is an index, not a diary 40
5.4 Session memory: short-term continuity cannot be brute-forced by chat logs 41
5.5 Autocompact: context governance is budget governance first . . 42
5.6 `compactConversation()` shows summaries must rebuild working context 43
5.7 Context governance is about preserving work semantics 45
5.8 The fifth principle extractable from source 45

Chapter 6 Errors and Recovery **47**

6.1 The least trustworthy sentence in engineering is "under normal conditions" 47
6.2 `prompt too long` is seasonal, not exceptional 47
6.3 Reactive compact: recovery must not become dead-loop machinery 48

6.4 `max_output_tokens`: recovery should prioritize continuation . . 49
6.5 Auto-compact circuit breaker: recovery systems need governance too 49
6.6 Compact itself can fail, so recovery action needs its own recovery 51
6.7 Abort semantics: interrupts are part of recovery 54
6.8 Error handling protects narrative consistency of execution . . . 54
6.9 The sixth principle extractable from source 55

Chapter 7 Multi-Agent Work and Verification **56**

7.1 Past a certain point, the question is not "can one agent do it" but "how should work be split" 56
7.2 The first principle of forked agents is cache safety 56
7.3 State isolation: reduce contamination before sharing anything . 58
7.4 Coordinator mode: synthesis is the scarce capability 59
7.5 Verification must be independent, or implementation completion will impersonate problem solved 59
7.6 Hooks and task lifecycle: spawning a subagent is only the beginning 60
7.7 Verification applies to memory and recommendations, not only code 63
7.8 Multi-agent really solves uncertainty partitioning 63
7.9 The seventh principle extractable from source 63

Chapter 8 Team Adoption **65**

8.1 Expert success does not automatically turn into safe team reuse . 65
8.2 The first team step is to make minimum boundaries clear 66
8.3 `CLAUDE.md` matters because it stays stable, layered, and low-dispute 67
8.4 Verification definition usually matters earlier than skill count . . 68
8.5 Skills are better understood as workflow modules than as institutions themselves 69
8.6 Approval works best when it is tiered by risk 69
8.7 Hooks are powerful, but usually belong later in the rollout . . . 70

8.8 Replayability matters, but teams should separate baseline traces from advanced audit trails 71
8.9 The eighth principle we can extract from source 71

Chapter 9 Ten Principles of Harness Engineering **73**
9.1 Treat models as unstable components, not teammates 73
9.2 Prompt is part of the control plane 74
9.3 Query loop is the heartbeat of agent systems 74
9.4 Tools are managed execution interfaces 74
9.5 Context is working memory 74
9.6 Error paths are main paths 74
9.7 Recovery should optimize for continuation 75
9.8 Multi-agent matters because it partitions uncertainty 75
9.9 Verification must be independent 75
9.10 Team institutions matter more than personal tricks 75
9.11 One final sentence 76

Appendix A Checklists **77**
A.1 Agent runtime design checklist 77
A.2 Prompt design checklist 78
A.3 Tool and permission design checklist 78
A.4 Context governance checklist 79
A.5 Error recovery checklist 79
A.6 Multi-agent checklist 80
A.7 Team adoption checklist 80
A.8 Review question set 81
A.9 Final short checklist 82
A.10 Implementation seeds (pseudocode stubs) 82

Appendix B Diagrams **84**
B.1 Diagram 1: Claude Code global control plane 84
B.2 Diagram 2: Query loop main cycle and recovery branches 88

B.3 Diagram 3: Tool batch ordering and StreamingToolExecutor . . 88
B.4 Diagram 4: Context sources and compact rebuild 92
B.5 Diagram 5: Coordinator-worker flow and verification separation 92
B.6 Diagram 6: Team governance map 92

Appendix C Source Map **96**

C.1 Chapter 1 Why Harness Engineering Matters 96
C.2 Chapter 2 Prompt Is the Control Plane, Not Persona Decoration . 97
C.3 Chapter 3 Query Loop: The Heartbeat of an Agent System 97
C.4 Chapter 4 Tools, Permissions, and Interrupts 98
C.5 Chapter 5 Context Governance: Memory, CLAUDE.md, and Compact 98
C.6 Chapter 6 Errors and Recovery 99
C.7 Chapter 7 Multi-Agent and Verification 99
C.8 Chapter 8 Team Adoption 100
C.9 Chapter 9 Ten Principles 100

Introduction

> This book is not asking whether a model can write code. It asks how to keep a code-writing model from steering an engineering system off course once it is plugged into terminals, repositories, and team workflows.

This is not a stitched set of source comments, and not a feature tour. It focuses on how Claude Code constrains an unstable model into a sustainable engineering order, where control plane, main loop, tool permissions, context governance, recovery paths, multi-agent verification, and team practices form one coherent skeleton.

This book starts from three reading assumptions:

- The center of gravity is not model capability, but how the harness organizes constraints and execution
- The goal is not line-by-line function narration, but why the runtime had to grow this way
- The value is not personal trick collection, but turning structure into reusable team practice

Recommended reading order:

1. Preface: Harness, Terminals, and Engineering Constraints
2. Chapter 1 Why Harness Engineering Matters
3. Chapter 2 Prompt Is Not Personality, Prompt Is the Control Plane
4. Chapter 3 Query Loop: The Heartbeat of an Agent System
5. Chapter 4 Tools, Permissions, and Interrupts: Why Agents Cannot Touch the World Directly
6. Chapter 5 Context Governance: Memory, CLAUDE.md, and Compact as a Budgeting Regime

7. Chapter 6 Errors and Recovery: An Agent System That Keeps Working After Failure
8. Chapter 7 Multi-Agent Work and Verification: Managing Instability Through Division of Labor
9. Chapter 8 Team Adoption: Turning a Smart Tool into a Reusable Institution
10. Chapter 9 Ten Principles of Harness Engineering
11. Appendix A Checklists: Turning Principles into Executable Constraints
12. Appendix B Diagrams: Drawing the Runtime Skeleton
13. Appendix C Source Map: Which Files Ground Each Chapter

If you only want the consolidated judgment first, jump to Chapter 9.

Preface: Harness, Terminals, and Engineering Constraints

In recent years people have liked to call code-writing models "agents."The term carries visible optimism, as if reading a repo, calling tools, and producing a decent patch were enough for independent work in engineering environments. But engineering environments have consequences. Terminals, filesystems, and Git history are not abstract space; every change leaves a trace.

A model that only outputs text mostly creates interpretation cost when it fails. A model that can run commands, write files, access networks, and modify repositories leaves execution artifacts when it fails. Directories change, processes break, configuration gets damaged, and history becomes difficult to audit. At that point the key question is no longer whether the model is smart enough, but whether the system imposes enough constraint.

That is what this book discusses.

I call it Harness Engineering. Here, harness means continuously active control structures that bound model behavior in engineering environments. For AI coding agents, unconstrained capability only expands blast radius.

This is not a line-by-line source walkthrough of Claude Code. Source matters, but if we only follow directories and explain function by function, we end up with comment compilation. That can tell us what functions do, but not why the system had to grow this shape. To understand systems like Claude Code, knowing `queryLoop()`, `compactConversation()`, and `runTools()` exists is not enough. The harder question is: why does a "code-writing" system eventually require prompt layering, permission checks, state machines, compact logic, recovery branches, subagent lifecycle control, verification stages, and team process?

The answer is not complicated: models are unstable.

That judgment is not always pleasing, but engineering systems cannot run on optimistic narratives. If a core component is unstable by nature, the system must be designed around that fact. Otherwise the problem clusters in incident retrospectives.

Claude Code is worth studying because its implementation stays deliberately restrained:

- It does not assume model correctness is sustained, so it uses a query loop to manage state
- It does not assume tool calls are naturally safe, so it constrains tools through permission and scheduling
- It does not assume more context is always better, so it introduces memory, `CLAUDE.md`, compact, and session memory
- It does not treat errors as rare events, so it designs recovery paths for prompt-too-long, max output tokens, interrupts, and hook loops
- It does not equate multi-agent with stronger capability, so it separates synthesis and verification to avoid self-endorsement

Taken together, this is the agent. The model is just the most eloquent and most unstable component inside it.

So this book keeps one fixed position:

> Prompt determines how it speaks. Harness determines how it acts.

Harness here is not an accessory layer and not emotional defense against model capability. It is the precondition for putting a model into engineering environments. Without that layer, risk gets transferred to users, teams, and future maintainers.

One boundary up front: this book does not include Claude Code source code, and does not reproduce large source passages. The reason is simple: copyright boundaries. What we can do is extract design principles, runtime mechanisms, and methodological judgments through fair engineering analysis and limited citation, not republish protected implementation text.

This book tries to do two things.

First, using Claude Code source structure, it explains the structures that actually determine reliability. The focus is why context governance must be a

primary path, why multi-agent solves role partitioning, and why team process must attach to lifecycle checkpoints, rather than simply listing "there is compact," "there is subagent," "there is hook."

Second, it extracts broader engineering principles behind those implementation choices. Specific code versions change, function names change, product surfaces change. But as long as people keep integrating unstable models into real workflows, certain principles remain valid. For example:

- Error paths must be designed as first-class paths
- Verification must be part of the definition of done
- Permission is an organ of the system, not an accessory feature
- Context is a resource, not a junk drawer
- Multi-agent depends on role separation, not headcount tactics
- Team institutions matter more than personal tricks

If those judgments hold, Claude Code is best treated as a reference specimen. Its value is not teaching people to clone one exact CLI, but showing how an AI agent facing real engineering conditions evolves toward stricter constraint structure.

Put more directly, this book is not about packaging a model into an "engineer-like" illusion. It is about constructing a runnable engineering system despite the model lacking engineer-grade stability.

This work is rarely flashy. Rollback, approval, permission, verification, compact, and orphan-process cleanup are not flashy. Yet long-term system stability usually depends on exactly these pieces. If you over-prioritize "human-like naturalness," the common outcome is a system that inherits human-like error patterns without human accountability.

Given that, we begin with constraints.

Across the next nine chapters, we discuss how this harness structure grows, why it must grow this way, and how a team turns individual experience into reusable engineering institutions.

@wquguru
2026-04-01
On the day of the Claude Code source leak, April Fool's Day

You can also read the online edition at harness-books.agentway.dev/book1-claude-code for a better reading experience.

Chapter 1 Why Harness Engineering Matters

1.1 The core problem is keeping the model from going off script

Once a speaking probability distribution can touch shell, Git, networks, and local files, the problem shifts from "the answer is not good enough" to "execution caused real damage." Harness Engineering is about constraining that unstable core component into a manageable system. It addresses one practical fact: models are not trustworthy by default.

That judgment is not always pleasant, but it is usually useful. Ignore it, and the problem usually resurfaces in logs and incident reports.

1.2 Claude Code's first harness layer: a constrained conversation system

On the surface, Claude Code looks like a CLI that talks with users and can patch code. In implementation terms, it was never designed as a bare model wrapper. It was designed as a conversation system with context boundaries, runtime state, and behavior rules.

You can see this immediately in how the system prompt is organized.

- Starting at `src/constants/prompts.ts:175`, the system defines identity and top-level mission
- Starting at `src/constants/prompts.ts:186`, it adds system-level rules for tools, permissions, reminders, and context compression

- Starting at `src/constants/prompts.ts:199`, it adds engineering constraints for task execution, such as avoiding unauthorized edits, not claiming validation is done when it is not, and not inventing abstractions for convenience

Pause on this point. Many prompt discussions still sit at the rhetorical level of "what kind of assistant are you." Claude Code places prompt inside runtime control structure. These texts define execution boundaries, failure behavior, and reporting responsibility.

More importantly, this prompt is assembled in segments. In `getSystemPrompt()` at `src/constants/prompts.ts:444`, static and dynamic parts are explicitly split, with memory, language, output style, MCP instructions, and scratchpad injected by section. In `src/utils/systemPrompt.ts:28`, default prompt, custom prompt, agent prompt, and appended prompt are then composed via explicit precedence rules.

This reflects a plain engineering fact: a truly usable agent system cannot rely on one "universal prompt" to solve everything. It must split control into layers, then split layers into responsibilities. Otherwise every new reminder and prohibition quickly conflicts with others, and behavior becomes unpredictable.

Invariants: three hard constraints on the control plane

```
assert prompt.layers ⊇ {default, project, custom, agent, append}   # identity s
assert ∀ tool_call t: scheduler.decides_concurrency(t) before exec(t) # tools
assert on recoverable_error: route ∈ {recover, terminate_clean}    # errors ar
path
```

The moment any one of these is bypassed, the query loop, tools, context, and recovery mechanisms in later chapters all start to look suspect.

1.3 The second harness layer: the agent depends on a continuous loop

If prompt defines what it should be, the query loop defines how it actually runs.

Claude Code's core is not one isolated API call. It is `query()` starting at `src/query.ts:219` and especially `queryLoop()` starting at `src/query.ts:241`. The key point in this implementation is explicit acknowledgment that agent systems depend on stateful multi-turn execution.

At `src/query.ts:268`, the system puts `messages`, `toolUseContext`, `autoCompactTracking`, `maxOutputTokensRecoveryCount`, `hasAttemptedReactiveCompact`, `pendingToolUseSummary`, `turnCount`, `transition`, and more into one cross-iteration state object. Once a conversation system is designed this way, it is formally admitting that unresolved issues from the last turn enter the next turn, and the system must continue from there.

That is the core harness mindset. The real question is whether behavior remains consistent across continuous turns:

- Is there a budget concept?
- Is there a recovery concept?
- Is there a self-rescue mechanism after context expansion?
- Can the task continue after tool failure?

Without these structures, an agent is just an unstable executor.

After `src/query.ts:365`, the loop also handles message slicing, tool result budget, history snip, microcompact, context collapse, and autocompact before each model call. There are many details, but they point to one thing: Claude Code tries to pull control back to runtime before model invocation.

This is why Harness Engineering cannot be reduced to prompt engineering. The former governs state machines; the latter governs wording. Wording matters, but state machines decide who is accountable for behavior.

1.4 The third harness layer: tool calls must obey scheduling discipline

When a model can only output text, at worst it sounds overconfident. Once it can call tools, risk shifts from rhetoric risk to execution risk. Then the key question is: who decides how tools run?

Claude Code's answer is direct. Runtime chooses parallel or serial execution according to tool properties.

In `runTools()` at `src/services/tools/toolOrchestration.ts:19`, tool calls are first grouped by `partitionToolCalls()`. At `src/services/tools/toolOrchestration` the system reads tool schema and calls `isConcurrencySafe()` to decide whether a tool is safe for parallel execution. Concurrency-safe calls run in batches; unsafe calls run one by one. In parallel paths, context modifiers are buffered first and then replayed in original block order, see `src/services/tools/toolOrchestration.ts:31` to `:63`.

This is representative. It shows Claude Code does not treat tools as natural extension of model capability, but as managed execution units requiring scheduling discipline. A tool system without scheduling discipline only amplifies model instability into the external world.

Unconstrained parallelism increases blast radius. Claude Code takes a conservative strategy here. In environments touching files, terminals, and permission boundaries, that conservatism is usually more reliable.

1.5 The fourth harness layer: the most dangerous tool needs the strictest rules

Among all tools, Bash is the one to distrust most. It barely has domain boundaries: it can touch files, processes, networks, and Git repositories directly, with shell redirection and pipeline semantics on top. Any system that overtrusts Bash usually gets concrete consequences.

Claude Code's position is very clear in `src/tools/BashTool/prompt.ts:42` and below. It contains extensive operating rules, especially around Git and PRs: do not randomly modify git config, do not skip hooks, do not casually `git add .`, do not use `--amend` to fold previous commits after pre-commit failures, do not commit unless explicitly asked, and do not default to push.

Some people call this too granular. But high-risk interfaces usually require high-density constraints. Once Bash enters real workflow, many rules must be explicit.

One key Harness Engineering principle is to package high-risk capability as high-constraint capability. The stronger the capability, the finer the control.

External systems do not forgive bad execution just because model tone sounds confident.

1.6 The fifth harness layer: errors are part of the main path

Many software systems treat failure paths as exceptions and success paths as the main text. Agent systems cannot do that. Failures here are not occasional; they are structurally present. Models hit token limits, trigger `prompt too long`, hit `max_output_tokens`, face tool denial, user interrupts, hook blockage, API retries, and more. If all these are handled as afterthought `catch` blocks, the system may look alive while constantly rolling trouble forward.

Claude Code does not do this in query loop. Look at autocompact handling after `src/query.ts:453`, and comments around context limit and blocking logic after `src/query.ts:592`: failures are handled as persistent structural conditions.

This is one major difference between harnesses and ordinary assistants. Ordinary assistants often follow "answer first, apologize if wrong." Harnesses emphasize constrain first, execute next; and if errors happen, route through recovery paths rather than improvising.

A system that can apologize is not necessarily mature. A system that knows when not to start, when to retry, when to terminate, and how to report failure accurately is closer to maturity.

1.7 The first principle extractable from source

By now, Chapter 1 really says one thing:

> The key capability of an agent system is constrained execution.

Claude Code source points to the same conclusion in key places:

- `constants/prompts.ts` shows prompt is part of the control plane, not personality decoration

- `utils/systemPrompt.ts` shows behavior must follow explicit layered precedence
- `query.ts` shows agent runtime depends on continuous loop state, not one-shot Q&A
- `services/tools/toolOrchestration.ts` shows tool calls must obey scheduling discipline
- `tools/BashTool/prompt.ts` shows high-risk tools require high-density constraints

Seen together, Harness Engineering is not mysterious. It just insists on engineering common sense that is often ignored:

- Models make mistakes
- Tools amplify consequence
- Context expands
- State contaminates later turns
- Users interrupt
- Failures recur

Given that, systems cannot preserve order through "smartness." They preserve order through structure. Structure is less flashy than smartness, but usually more reliable.

The next chapter turns to the most misunderstood layer in this structure: system prompt. Many people treat it as persona text. We will show it is closer to institutional policy in an operating system. Persona improves feel; policy constrains machines.

Chapter 2 Prompt Is Not Personality, Prompt Is the Control Plane

2.1 Treating prompt as persona is a common misunderstanding

When people hear "system prompt," they first think of familiar rhetoric: who you are, what you are good at, ideally with a stable personality. That framing is enough for chat-only systems and clearly insufficient for an agent that reads files, calls tools, touches shell, handles permissions, and executes across turns. Persona answers "what it looks like"; a control plane answers "what it can do, when, what happens on failure, who owns fallback." They are different layers.

Claude Code implementation shows this directly: its system prompt is a layered assembly of behavioral blocks—closer to a runtime protocol than to a character biography.

2.2 In source, Claude Code prompt is layered from the beginning

In `getSystemPrompt()` at `src/constants/prompts.ts:444`, Claude Code returns an array of sections, not one complete string. This detail matters. Once prompt becomes multi-block, the system formally acknowledges internally distinct constraints with distinct duties.

These sections include at least several categories.

First is identity and mission. At `src/constants/prompts.ts:175`, the system states it is an interactive agent that uses available tools to complete software engineering tasks, including safety constraints such as not guessing URLs.

Then come system-level rules. Starting at `src/constants/prompts.ts:186`, the system explicitly defines:

- Which text users can see
- That tool calls may trigger permission approval
- That denied actions must not be mechanically retried
- That system reminders may appear inside tool results and user messages
- That context may be automatically compacted

These rules share one obvious property: they do not care whether the model "looks like a smart assistant." They care whether it is a disciplined executor. That is control-plane tone; its primary mission is boundary definition.

Then, starting at `src/constants/prompts.ts:199`, come engineering instructions for doing work: do not add requirements casually, do not optimize beyond authorization, do not hide failed validation just to look polished, and do not invent abstractions unnecessarily. They may look like writing-style guidance, but they are tightly coupled to engineering constraints. A model that "helpfully optimizes everything" may look enthusiastic product-wise and still be dangerous engineering-wise.

So source structure alone already shows this: Claude Code prompt is designed to keep model behavior within boundaries in complex runtime.

2.3 Prompt value is not text itself, but precedence structure

If prompt just exists as text, that still does not prove control-plane status. What decides that is strict precedence.

You can see this in `buildEffectiveSystemPrompt()` at `src/utils/systemPrompt.ts:28`, where prompt sources are composed in explicit order:

1. override system prompt

2. coordinator system prompt
3. agent system prompt
4. custom system prompt
5. default system prompt

Then `appendSystemPrompt` is uniformly appended.

The proactive-mode handling is even more direct. After `src/utils/systemPrompt.ts:99`, when agent prompt and proactive mode co-exist, agent prompt no longer replaces default prompt; it is appended after. A general constitution can be extended by a job description, but must not be wiped out by it.

Skeleton: the prompt-assembly precedence chain

```
// skeleton: buildEffectiveSystemPrompt() (src/utils/systemPrompt.ts:28)
sources = [override, coordinator, agent, custom, default]
base = first_present(sources)        // higher precedence replaces default who
if proactive_mode and agent:
   base = default + agent          // special case: layer, not replace
return base + appendSystemPrompt        // append is always last

assert exists unique base in sources           # baseline must be unique
assert precedence(override) > precedence(default)  # order is hardcoded, not "
assert appendSystemPrompt never replaces base     # append can only append
```

Break any one of these invariants and prompt degrades into a graffiti board where whoever writes last is in charge.

2.4 Prompt is not static copy; it is connected to memory systems

If the above already sounds like a runtime handbook, Claude Code's handling of memory and `CLAUDE.md` makes it even clearer: prompt here is an entry point for full context governance, not just "a paragraph shown to the model."

In `getClaudeMds()` at `src/utils/claudemd.ts:1153`, the system gathers project instructions, local instructions, team memory, and auto memory into

a unified format and then merges them into prompt-adjacent context. It even labels provenance explicitly: project-level instructions, user-private project instructions, shared team memory, or cross-session persisted auto memory.

In `buildMemoryLines()` at `src/memdir/memdir.ts:187`, the system turns memory-writing rules themselves into part of prompt:

- Memory is a file-based persistence system
- `MEMORY.md` is an index, not body text
- Frontmatter should be written in specific form
- Certain information should not be saved
- Plans and tasks should not be misused as memory entries

This is crucial. It extends prompt duty from "constrain current behavior" to "constrain how future knowledge is deposited." This is beyond ordinary prompting and closer to a knowledge-governance protocol for runtime participants.

In other words, Claude Code does not only use prompt to define "how to speak this turn." It uses prompt to define "how long-lived memory is formed." Once a system reaches this stage, prompt can no longer be merely tone; it becomes institution.

2.5 A real control plane must include caching and compute cost

Most prompt discussions ignore performance. The common idea is: prompt is just text fed to models, so write it and done. Claude Code is more pragmatic: prompt is also compute cost. The more complex and volatile it is, the worse cache hit rate becomes, and the more expensive and slower runtime gets.

In `src/constants/systemPromptSections.ts:16` and below, prompt sections are split into:

- cacheable `systemPromptSection`
- cache-breaking `DANGEROUS_uncachedSystemPromptSection`

`resolveSystemPromptSections()` prefers cache reuse and recomputes only when necessary. `clearSystemPromptSections()` clears section state after `/clear` or `/compact`.

This may look like optimization, but it is still control-plane design. A production prompt system cannot optimize only expressive power while ignoring throughput, latency, and cache behavior. In `getSystemPrompt()`, Claude Code even places explicit boundaries between static and dynamic prompt segments after `src/constants/prompts.ts:560`. That means design explicitly assumes some content is turn-stable while other content changes per turn; they must not be mixed in a way that burns cache.

Once a system starts asking "which part of prompt invalidates cache," it has stopped treating prompt as copywriting. Copywriting optimizes completeness of expression. Control planes optimize governability, reuse, and predictable behavior cost.

2.6 Users can override prompt, but cannot bypass this structure

Claude Code does not lock users into default prompt. CLI explicitly supports override and append.

After `src/main.tsx:1342`, the system handles options like `--system-prompt`, `--system-prompt-file`, `--append-system-prompt`, and `--append-system-prompt-file`. Users can absolutely bring custom rules.

But one key point remains: regardless of override or append, final assembly still goes through `buildEffectiveSystemPrompt()`. So customization is allowed, order is not abandoned. Users can change content; system retains structure.

Customization without structure usually decays into arbitrary drift: one section added today, another removed tomorrow, and some agent replacing baseline constraints the day after. Behavior then starts looking like ad hoc oral announcements. Claude Code chooses a different path: let users modify, but force modifications through fixed layering and precedence.

2.7 Why prompt here is closer to constitution than dialogue lines

Put previous sections together and the conclusion is clear:

Claude Code prompt is constitution-like.

Dialogue lines are what characters say on stage. A constitution defines power boundaries, duty relations, and exception handling. Claude Code prompt is closer to the latter because it satisfies structural conditions:

- It is layered, not monolithic
- It has precedence, not "latest write wins"
- It forms a full control plane together with memory, `CLAUDE.md`, agent instructions, and MCP instructions
- It has cache and dynamic section mechanics, not ad hoc text concatenation
- It is tightly coupled to runtime, not floating outside the system as decoration

That is also why "writing a good prompt" in isolation has limited value. The harder question is where prompt sits in system architecture, which modules it coordinates with, and whether it participates in governance of permissions, state, context, and long-term memory. Without those answers, a so-called good prompt is usually only temporarily correct in one smooth scenario.

2.8 The second principle extractable from source

This chapter can be compressed into one sentence:

> Prompt is valuable only when it is integrated into explicit control structure.

Claude Code source proves this across several modules:

- `constants/prompts.ts` structures prompt as segmented control blocks rather than one declaration

- `utils/systemPrompt.ts` defines strict source precedence
- `utils/claudemd.ts` integrates project and long-term memory into context assembly
- `memdir/memdir.ts` uses prompt to define long-term memory write protocol
- `constants/systemPromptSections.ts` turns prompt into runtime objects that are cacheable, invalidatable, and recomputable by section

So in mature agent systems, prompt should not be understood as "opening lines that put the model in character." It is a live institutional text. Institutional text can be clear, but its decisive feature is enforceability.

Next chapter moves to an even harder structural bone: the query loop. Even the best control plane must eventually land inside execution cycles. Prompt defines boundaries; loops decide fate. What a system ultimately becomes is usually revealed by how each turn continues, interrupts, and recovers.

Chapter 3 Query Loop: The Heartbeat of an Agent System

3.1 To judge maturity, first ask whether the system has a loop

Imagining a code-writing model as an upgraded Q&A API is the easiest mistake: once the system starts calling tools, spanning turns, handling interrupts, persisting state, and compacting context, one-shot Q&A collapses. Claude Code structurally admits that agents depend on continuous, stateful execution.

`query()` at `src/query.ts:219` is shell; `queryLoop()` at `src/query.ts:241` is core. It maintains cross-iteration state, performs pre-governance, streams model output, then decides whether to execute tools, recover, compact, continue, or terminate.

The key word is lifecycle. Whether a system deserves to be called an agent is determined less by how it talks and more by whether it still knows what it is doing several turns later.

3.2 State is core business, not baggage

Many systems initially treat state as a burden and statelessness as elegance. For agent systems this preference has limited value. Once an agent enters real workflows, state emerges naturally. Ignoring state does not remove it; it only makes it return in less manageable form.

Claude Code is explicit here. In `src/query.ts:203` to `:217`, mutable query-loop state is clearly defined:

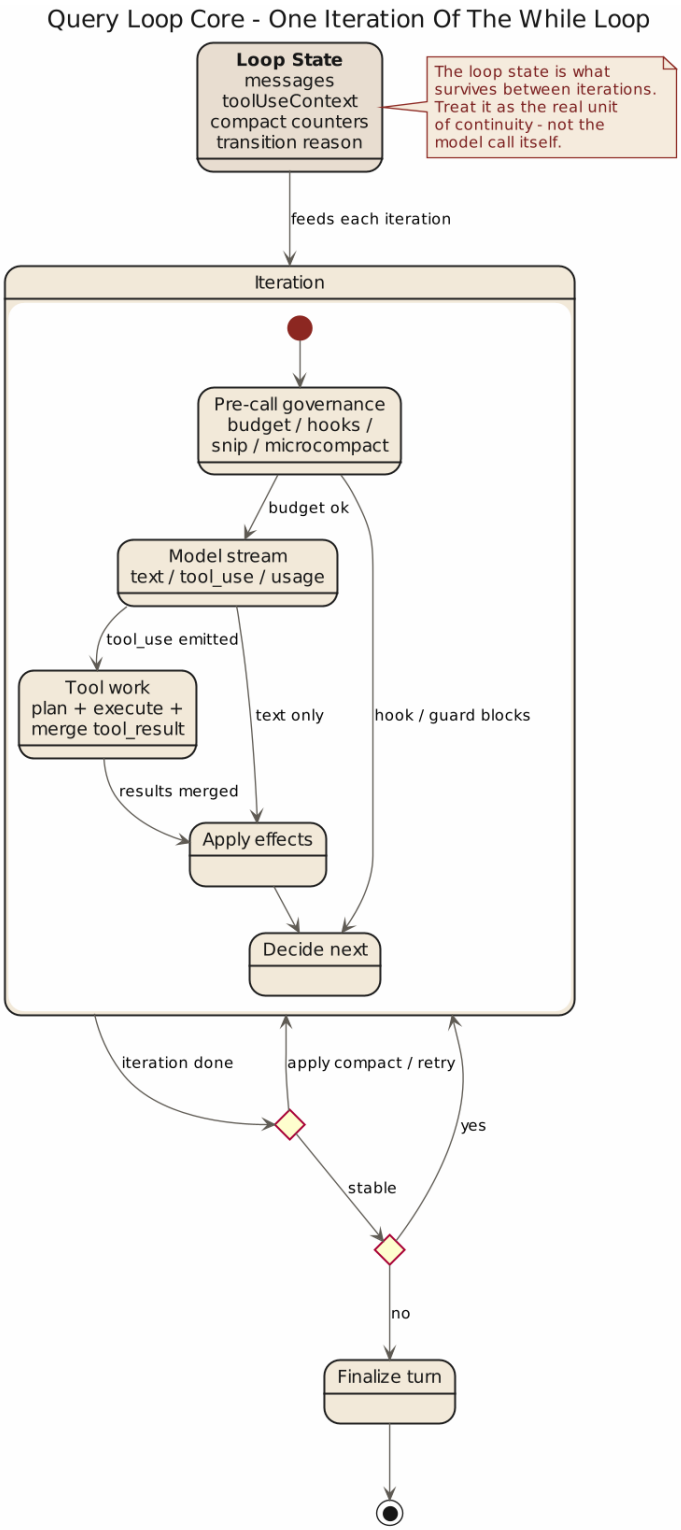


Figure 1: Claude Code Query Loop Core

- `messages`
- `toolUseContext`
- `autoCompactTracking`
- `maxOutputTokensRecoveryCount`
- `hasAttemptedReactiveCompact`
- `pendingToolUseSummary`
- `stopHookActive`
- `turnCount`
- `transition`

At `src/query.ts:268`, these are assembled into one `State` object when query loop starts, and updated as a whole through continue branches.

Claude Code does not scatter recovery, compaction, budget, hooks, and turn counting across ad hoc local booleans. Scripts ask only whether this step finished. Agent systems must also ask whether the next step can continue from the state left by a failed step.

Skeleton: queryLoop()

```
// skeleton: queryLoop()  (src/query.ts:203–1305)
state = { messages, toolUseContext, autoCompactTracking,
        maxOutputTokensRecoveryCount, hasAttemptedReactiveCompact,
       pendingToolUseSummary, stopHookActive, turnCount, transition }

while not done(state):
   govern_input(state)            // prefetch / snip / microcompact / collapse
    events = stream_model(state)
    for e in events:
        if e.is(tool_use):  schedule(e, state.toolUseContext)
        if e.is(api_error): return surface(e)
    if interrupted:
        drain_tools_with_synthetic_results(state); break
    state = advance(state, recover_if_needed(state))

assert state.turnCount_{t+1} >= state.turnCount_t        # state is monotonic
assert every emitted tool_use has a matching tool_result    # ledger is closed
```

```
assert hasAttemptedReactiveCompact ⇒ no further compact    # no self-loop
```

3.3 The first duty of query loop is input governance

From outside, people often assume the core action of an agent is "call the model." In engineering, what matters more is often the long preparation before that call. Claude Code makes this explicit in `queryLoop()`.

Before entering model stage, runtime does all of the following:

- Start memory prefetch, see `src/query.ts:297`
- Prefetch skill discovery, see `src/query.ts:323`
- Slice valid messages after compact boundary, see `src/query.ts:365`
- Apply tool result budget, see `src/query.ts:369`
- Perform history snip, see `src/query.ts:396`
- Perform microcompact, see `src/query.ts:412`
- Perform context collapse, see `src/query.ts:428`
- Attempt autocompact last, see `src/query.ts:453`

This sequence is an architectural statement by itself. It tells you Claude Code puts context governance before model reasoning. In other words, it does not delegate "turn chaos into order" to the model. Runtime governs first, then passes cleaner inputs to the model.

This is critical because many systems do the opposite: stuff huge context in, then hope the model will decide what matters. That looks convenient but shifts runtime responsibility onto probability distributions.

Claude Code takes a more traditional engineering posture: clean the site first, then execute. It is less elegant, but usually more stable.

3.4 Model invocation is one phase of the loop, not the loop itself

Only after those governance steps does Claude Code enter model invocation, around `src/query.ts:652`. One detail is worth isolating: output is consumed with `for await` streaming, not as one synchronous complete response.

That means model output in Claude Code is an event stream, not just final prose. Events may contain:

- assistant text
- `tool_use` blocks
- usage updates
- stop reasons
- API errors

This is particularly clear after `src/query.ts:826`. The system stores assistant messages, extracts `tool_use` blocks, decides follow-up needs, and can dispatch tools to `StreamingToolExecutor` while streaming is still in progress.

Engineering-wise, this is a structural shift. Once output becomes event stream, architecture is no longer plain request-response, but a drive-schedule-feedback process. Streaming is not just about seeing words sooner; it lets runtime schedule next actions before model output fully ends.

That is why query loop, not model call, is the heartbeat. Model call is one contraction inside heartbeat. What keeps the system alive is the full cycle: input intake, stream consumption, tool scheduling, failure recovery, and turn continuation.

3.5 A heartbeat must handle interrupts, or it is just inertia

A real heartbeat not only keeps beating, it can also stop when necessary. If it cannot stop, what remains is inertia.

Claude Code handles interrupts very concretely. After `src/query.ts:1011`, streaming abort is handled first. If `streamingToolExecutor` is active, remaining results must be consumed and synthetic tool results generated, so issued `tool_use` blocks are not left without paired results. Otherwise `yieldMissingToolResultBlocks()` fills missing interruption records.

This follows a basic engineering principle: once the system has committed to execution externally, interruption still requires ledger closure. You cannot pretend earlier `tool_use` calls never happened just because the user interrupted

later. External systems, UI, and transcripts need a consistent causal chain. Even "interrupted" must be complete interruption.

This matters because once an agent enters multi-tool, multi-turn mode, external expectations become more than "did it answer." The system must leave interpretable traces. Non-interpretable traces eventually become operations problems, audit problems, or latent team liabilities no one can explain.

So interrupt handling is core runtime duty. Started actions need closure, even if closure says "not finished."

3.6 A heartbeat must also recover, or it is fragile repetition

If interrupts are external shocks, recovery is internal slack. Loops without recovery capability eventually expose one common flaw: they treated luck as design.

Claude Code's recovery is layered, not simplistic retry. Prompt-too-long and max-output-tokens are typical cases.

After `src/query.ts:1065`, the system checks whether the last assistant message is a withheld `prompt too long`. If yes, it first tries to flush staged context collapse (`:1086` to `:1116`); if still insufficient, it enters reactive compact (`:1119` to `:1166`). So recovery proceeds from lower cost and lower destructiveness upward.

`max_output_tokens` is handled similarly. After `src/query.ts:1185`, runtime first tries raising token cap; if still insufficient, it appends a meta message telling the model to continue exactly from truncation point, not apologize, recap, or write polite filler.

This reveals Claude Code's design posture. Recovery is part of main runtime path, not post-failure etiquette. Recovery exists to preserve ability to continue work. In real engineering, continuity is usually more valuable than surface polish.

3.7 Stop conditions cannot be singular, or failure and completion get conflated

In ordinary chat systems, stop condition is simple: answer exists, end. Agent systems cannot be this lazy. In one session, "this turn ended" does not equal "task completed," and does not equal "system succeeded."

Claude Code query loop distinguishes at least:

- Streaming completed with `tool_use`, requiring follow-up
- No `tool_use`, entering stop hooks and further checks
- User interruption
- Prompt-too-long recovery branch
- Max-output-tokens recovery branch
- Stop-hook blockage requiring loop re-entry
- API errors returning directly

You can see this from `src/query.ts:1062` through `:1305`. The stop-hooks section around `:1267` to `:1305` is especially important: it explicitly prevents dead loops like "still too long after compact, blocked by hook, compact again forever."

This is worth attention. Many systems have only one naive rule: retry if failed. Claude Code admits retry itself needs governance. Runtime must know why retrying, what already tried, which protection state cannot reset, and what patterns lead to infinity. These decisions separate systems that "keep trying" from systems that "know when not to try again."

Stop-condition failure matrix

Event order	Pre-state	Trigger	Next
stream done + `tool_use`	pending tool_use present	stop reason	follow-up, execute tools
stream done, no `tool_use`	no pending tool	stop reason	enter stop hooks

Event order	Pre-state	Trigger	Next
user interrupt	any	abort signal	drain remaining results, yield synthetic tool_result
prompt_too_long	compact not yet tried	recoverable error	collapse drain / reactive compact
max_output_tokens	cap < MAX	stop reason	raise `maxOutputTokensOverride`, re-run
max_output_tokens	cap = MAX	stop reason	append meta user msg, continue writing
stop hook block + PTL recurs	`hasAttemptedReactiveCompact` = true	overflow failure	skip stop hooks, surface error
API error	—	api_error	return directly, no retry

3.8 QueryEngine proves this belongs to conversation lifecycle

If `queryLoop()` were not enough evidence, `QueryEngine` makes it explicit.

At `src/QueryEngine.ts:176`, source states:

> QueryEngine owns the query lifecycle and session state for a conversation.

This line states the chapter's point directly. QueryEngine owns conversation lifecycle, not one call. At `src/QueryEngine.ts:180`, source further states one `QueryEngine` maps to one conversation, and each `submitMessage()` opens a new turn inside that conversation while preserving state.

After `src/QueryEngine.ts:675`, QueryEngine hands prepared `messages`, `systemPrompt`, `userContext`, `systemContext`, and `toolUseContext` to `query()`, then writes assistant/user/compact-boundary messages back into transcript.

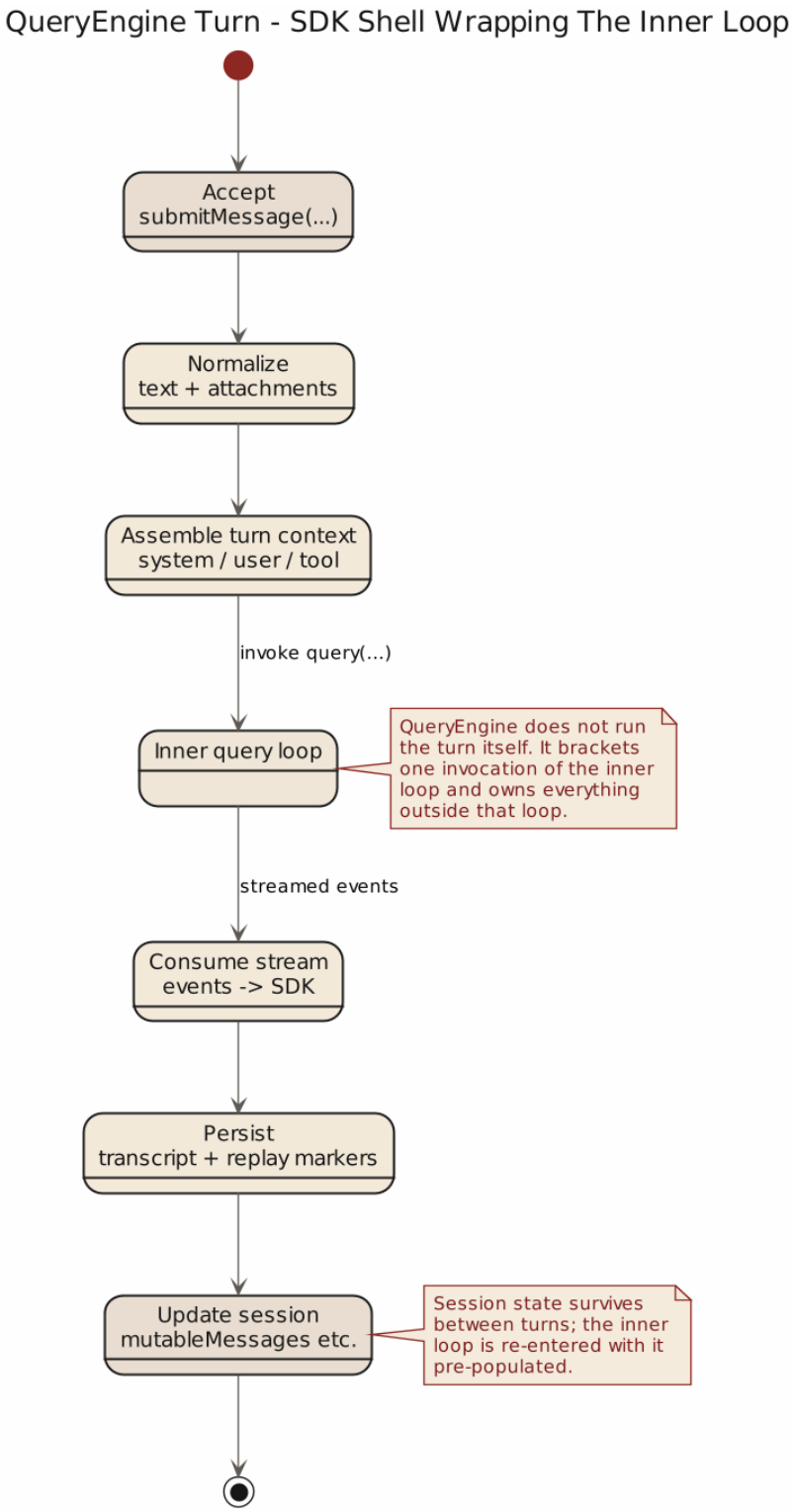


Figure 2: Claude Code QueryEngine Turn Flow

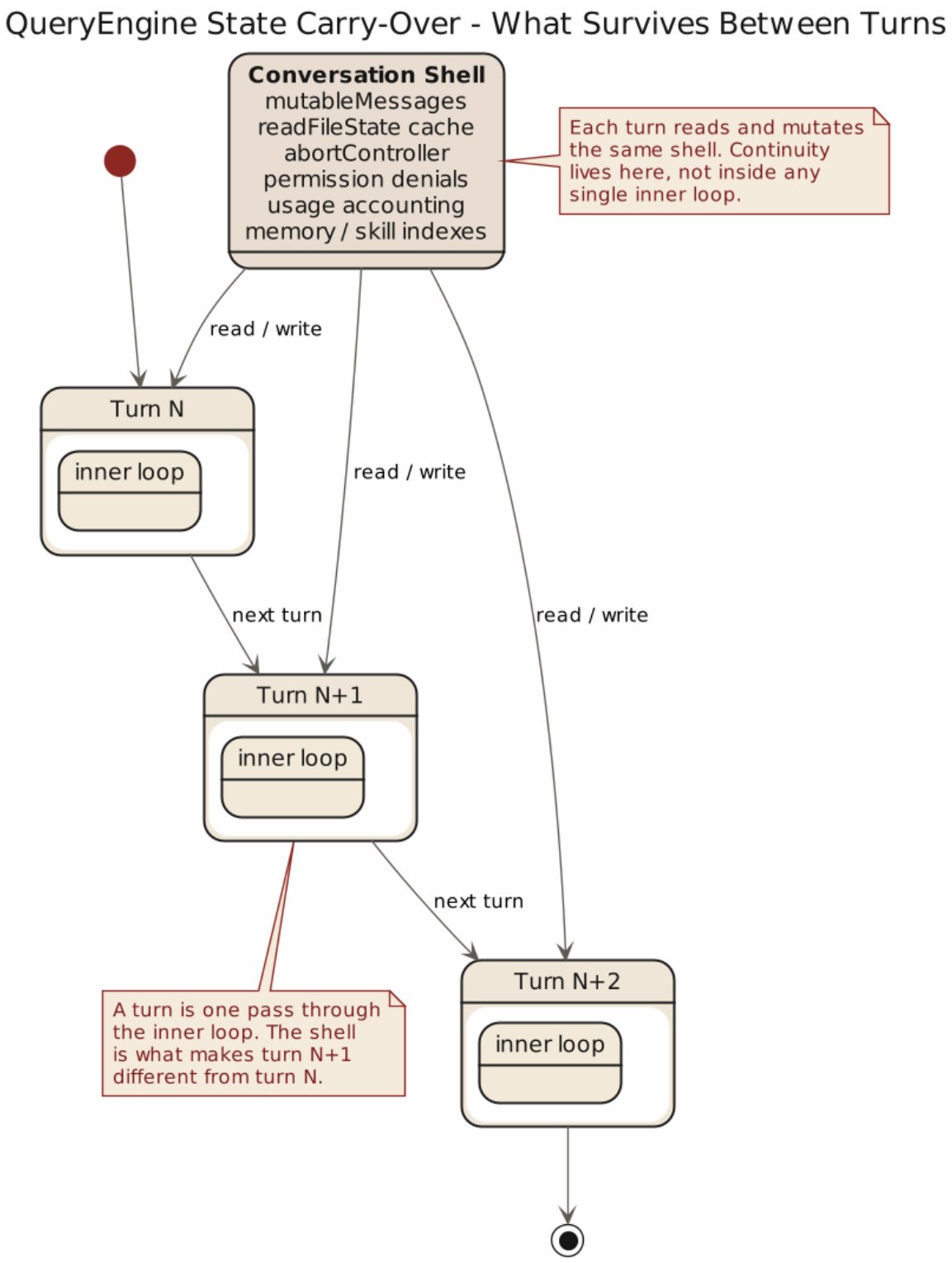


Figure 3: Claude Code QueryEngine State Carry-Over

This means query loop is the actual execution center of conversation runtime. Outer UI, SDK, and session persistence all orbit it. To understand Claude Code you cannot stop at listing tools or reading prompt text. You must inspect how this loop turns those constraints into continuous behavior.

3.9 The third principle extractable from source

This chapter can be compressed into one line:

> The core capability of an agent system is maintaining a recoverable execution loop.

The skeleton, invariants, and stop-condition matrix above already line up the evidence. One line suffices to close: a mature agent heartbeat must simultaneously govern cross-turn state, input governance, streaming consumption, the interrupt ledger, and the distinction between {completion, failure, recovery, continuation}.

Without these structures, systems may still produce attractive demos, but they are closer to staged performances than runtimes. Performances have value, but they do not replace order.

Next chapter moves to where heartbeat most directly touches the external world: tools, permissions, and interrupts. This chapter explained why loops exist. Next we explain why once loops own tools, they must also learn restraint.

Chapter 4 Tools, Permissions, and Interrupts: Why Agents Cannot Touch the World Directly

4.1 Once models call tools, the nature of risk changes

A text-only model mostly costs communication time when wrong. Once it starts calling tools, risk changes in kind: tools are not opinions but actions, and actions touch the real world. A wrong explanation stays at understanding level; a wrong command deletes files, kills processes, mangles Git history. Capability growth comes with consequence growth.

So the central question is simple: who constrains these tools? Claude Code's answer is to turn tools into managed execution interfaces, not let models reach out directly.

4.2 Tool orchestration is part of behavioral constitution

`runTools()` at `src/services/tools/toolOrchestration.ts:19` does something representative: it does not directly execute the `tool_use` list —it batches by concurrency safety. `partitionToolCalls()` at `:91` reads each tool's `inputSchema` and calls `isConcurrencySafe()`; safe calls go into parallel batches, unsafe calls split into serial units.

This looks like performance tuning but is really consistency design. Once concurrency is allowed, an old question returns: who decides context evolution, and in what order? In parallel paths (`:31–:63`), Claude Code does not let the

fastest tool mutate context first; it buffers `contextModifier` callbacks and replays them in original block order. Execution is parallel while semantic context evolution stays deterministic.

Classic engineering conservatism: concurrency may improve throughput but must not break causality. Mature agent systems do not idolize parallelism — they treat it as an exception that must prove safety, not a default freedom.

4.3 A lot happens before a tool actually runs

Many people assume that once `tool_use` appears, execution naturally follows. Robust systems do not work that way. After `src/services/tools/toolExecution.ts:30`, `runToolUse()` is already wrapped with permission logic, hooks, telemetry, and synthetic error materialization —pre-checks, in-flight events, post-execution correction, and failure compensation.

Tools here are not ordinary internal library functions. Internal functions assume stable callers who own consequence; tool interfaces sit between models and the external world, so runtime cannot assume stable caller judgment. That is why so many wrappers exist: caller is the most unstable variable. Tools should not be modeled as "extensions of model capability" but as external capabilities whose risk must be runtime-managed. Once you accept that, permission, hooks, interrupts, and synthetic results become basics, not burdens.

4.4 Permission comes before capability

Claude Code's permission entry point is after `src/hooks/useCanUseTool.tsx:27`. The very existence of `CanUseToolFn` says something important: tool allowance is not decided by model intent alone; it goes through an authorization chain.

Inside `useCanUseTool()`, runtime does not auto-run just because the model requested a tool. It first calls `hasPermissionsToUseTool(...)`, see `useCanUseTool.tsx:37`. Results are split into `allow`, `deny`, or `ask`. This may look ordinary, but it is fundamental. Mature authorization systems need more than yes/no; they need a third state where system itself should not decide on behalf of user.

After `useCanUseTool.tsx:64`, branches continue:

- `deny`: reject directly
- `ask`: route to coordinator, swarm worker, classifier, or interactive approval
- `allow`: execute

This structurally rejects a common dangerous idea: if the model understood user intent, it has authority to execute. It does not. Intent understanding is not authorization, and certainly not persistent authorization. Systems must split "can do" from "may do."

From this angle, permission clarifies agent role: model can propose actions, but runtime, rules, and user decide release. Capability and authority are deliberately separated.

Skeleton: the permission chain

```
// skeleton: useCanUseTool()  (src/hooks/useCanUseTool.tsx:27)
decision = hasPermissionsToUseTool(tool, input, ctx)
match decision:
    case allow:  return exec(tool, input)
    case deny:   return reject(reason)
   case ask:   return route_to(coordinator | swarm_worker | classifier | intera

assert decision ∈ {allow, deny, ask}                        # three-
valued, no boolean shortcuts
assert ask never auto-escalates to allow           # no unauthorized escalatio
assert deny is sticky for this tool_use_id          # no silent retry to allow
```

4.5 Permission results are runtime semantics, not booleans

After `src/utils/permissions/PermissionResult.ts:23`, Claude Code even defines explicit describers for permission semantics: `allow`, `deny`, and `ask`. This detail matters. Permission is not an internal bool; it is a runtime object with independent meaning.

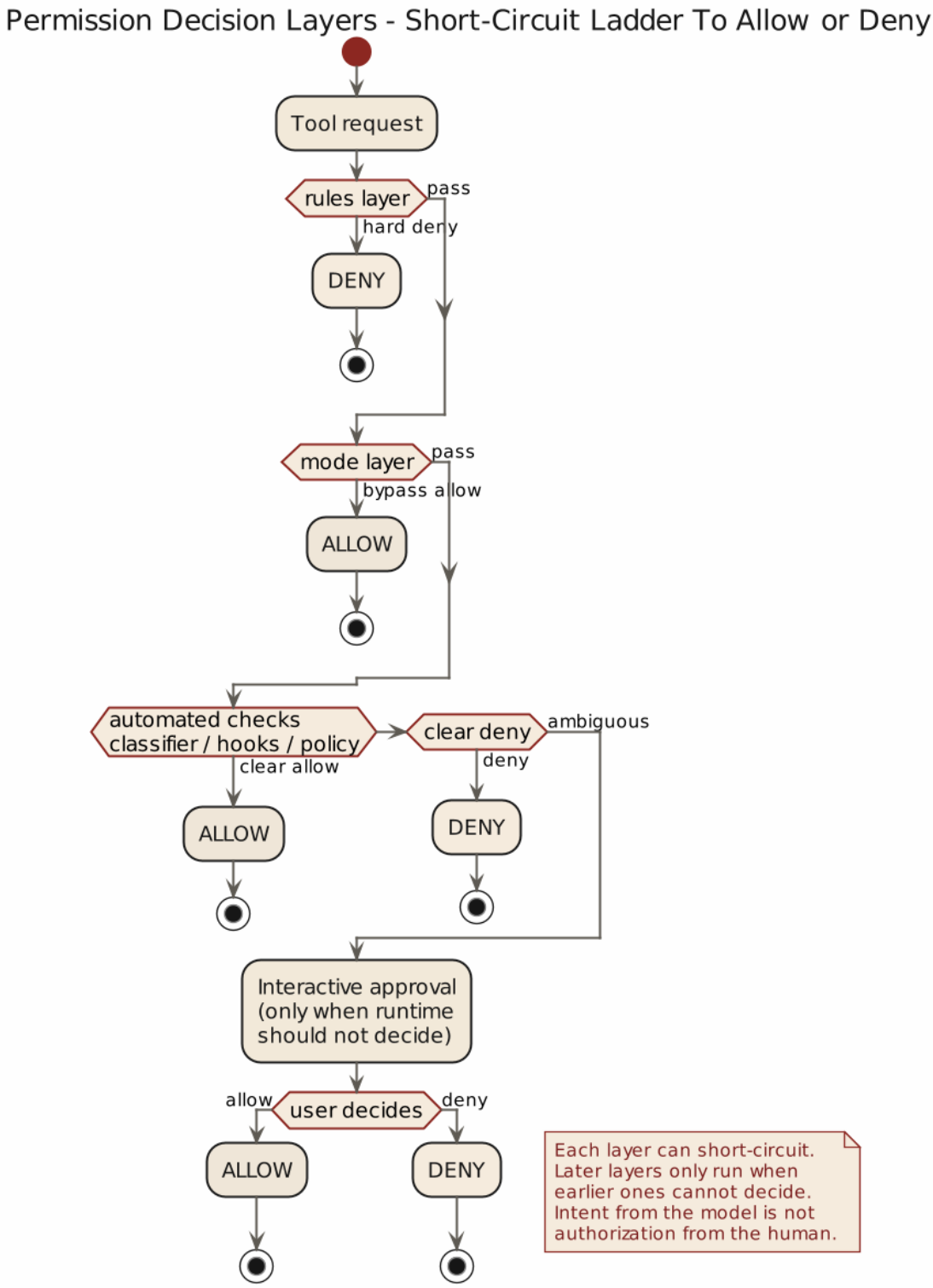


Figure 4: Claude Code Permission Decision Layers

Why this matters: a permission system must let runtime clearly express why a step did not continue. When an agent says "I need confirmation," it is declaring responsibility boundaries. Once boundaries are explicit, refusal, approval, cache rules, temporary grant, and persistent grant all have a place.

Put simply, if an agent cannot distinguish "I can do this," "I cannot do this," and "I must ask," it should not touch a terminal. Terminals do not fill missing semantics. Terminals only execute.

4.6 StreamingToolExecutor proves interrupt is first-class semantics

Once tools are parallel and streaming, interrupt handling becomes immediately complex. Runtime now faces a queue with states like queued, executing, completed, yielded, not one single action.

After `src/services/tools/StreamingToolExecutor.ts:34`, Claude Code explicitly makes this a dedicated streaming executor. Most important is how it handles interruption and discard.

At `StreamingToolExecutor.ts:64` to `:70`, runtime can discard the current tool set during streaming fallback. At `:153` to `:205`, it generates synthetic error messages for different causes:

- sibling error
- user interrupted
- streaming fallback

After `:210`, it further distinguishes interruption causes:

- cancelled due to sibling tool failure
- cancelled due to user interrupt
- dropped due to fallback

At `:233` and below, each tool can define `interruptBehavior`, deciding whether to `cancel` or `block` when user interjects.

This design matters because Claude Code does not treat interruption as "a special execution error." It treats interruption as semantics equal in importance

to execution itself. Runtime must know not only whether a tool may start, but how it ends when interrupted, how results are closed, and whether new messages can interleave.

That is a core Harness Engineering trait: design start and stop both. Execution systems without stop semantics eventually depend on user-side hard interruption to finish design.

Concurrency & interrupt failure matrix

Event order	Pre-state	Trigger	Next
one tool fails in parallel batch	siblings still executing	sibling error	keep others; replay `contextModifier` in block order
user interjects + `interruptBehavior=cancel`	tool not yet done	user interrupt	cancel, emit `user interrupted` synthetic result
user interjects + `interruptBehavior=block`	tool not yet done	user interrupt	finish tool, block new message until done
streaming fallback	batch queued/executing	fallback	discard batch, yield fallback synthetic results in order
abort with pending `tool_use`	any	abort signal	synthesize `tool_result`, close the ledger
Bash compound command exceeds subcommand cap	classifier rejects	safety check	`deny`; do not route to `ask`

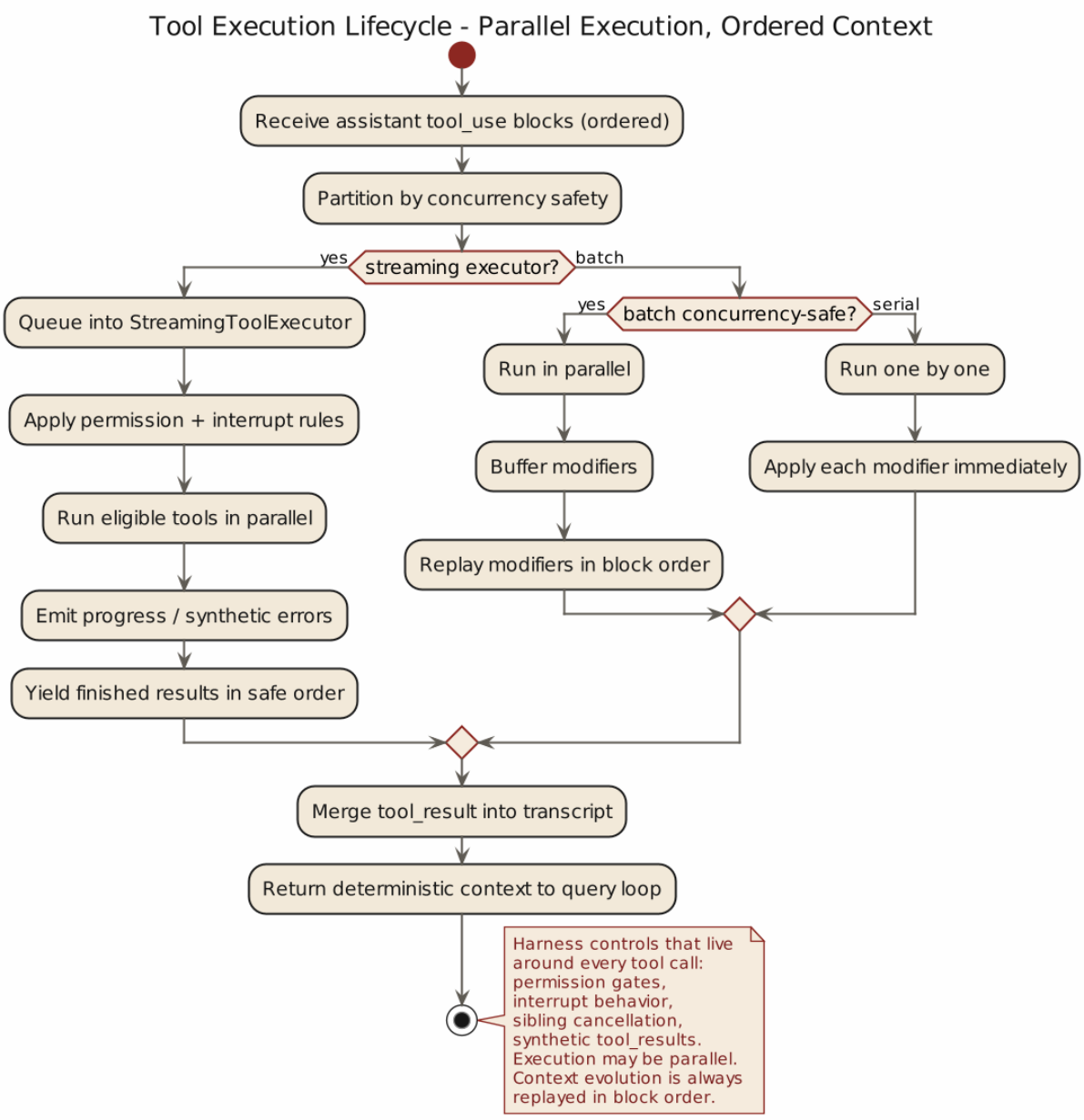


Figure 5: Claude Code Tool Execution Lifecycle

4.7 Why Bash is always more suspect than other tools

In Claude Code's tool world, Bash is a risk amplifier, not a normal tool. It is too general —a file-read tool will not casually kill processes, a grep tool will not secretly push commits, but Bash can do almost everything. Claude Code's distrust is explicit.

First layer is prompt guidance in `src/tools/BashTool/prompt.ts:42` and below: detailed rules for git, PRs, dangerous commands, hooks, force push, and interactive flags —disciplined verbosity where consequences are large. Second layer is permission and safety classification. `src/tools/BashTool/bashPermissions.ts:1` handles shell semantics, command prefixes, redirection, wrappers, safe env vars, classifier routing, and rule matching; after `:95` a subcommand-count cap prevents compound commands from escaping checks.

Bash is a dangerous channel requiring special governance, not a generic command interface. A reusable judgment: high-risk capability deserves special governance, not generic capability treatment; treating Bash as ordinary is usually design laziness.

4.8 Tool systems protect users and protect runtime itself

Permissions, scheduling, and interrupts look like user protection mechanisms, but they also protect runtime consistency itself. If an agent system allows unresolved problems such as incomplete `tool_result`, out-of-order context mutation, unbounded parallel side effects, and ambiguous interrupt semantics, the first thing to break is internal consistency.

This is tightly coupled across `query.ts` and tool execution layers. Chapter 3 discussed how query loop synthesizes missing tool results on interrupt. This chapter shows StreamingToolExecutor has its own discarded/errored/sibling abort/interrupt behavior mechanics. Together they preserve a traceable causal chain for:

- what was executed

- what was unfinished
- why it stopped

That is another core harness meaning: preserving order for the system itself. Many constraints look like misoperation prevention on the surface, but at deeper level they prevent runtime from collapsing into unexplainable state fragments.

4.9 The fourth principle extractable from source

This chapter can be compressed to one line:

> Tools are managed execution interfaces; permission is an organ of the agent system.

Claude Code source supports this jointly:

- `toolOrchestration.ts` batches before execution, so scheduling comes before impulse
- `toolExecution.ts` wraps hooks, permission, telemetry, and synthetic errors around execution, so calls are never bare
- `useCanUseTool.tsx` splits authorization into `allow / deny / ask`, making authority a first-class semantic branch
- `StreamingToolExecutor.ts` defines interruption, fallback, and sibling-failure semantics, making stopping as important as starting
- `BashTool/prompt.ts` and `bashPermissions.ts` apply high-pressure special governance to Bash, proving high-risk capability must carry denser constraints

As portable engineering principles: model proposes, runtime authorizes; preserve causal order even under parallelism; interrupt is first-class, not a generic exception; treat Bash-class tools as explicit exceptions; a tool system protects both users and runtime.

Next chapter addresses another common illusion in this system: "more context is always better." Claude Code implementation says the opposite. Experienced systems treat context as a resource, not a warehouse. We now turn to how memory, `CLAUDE.md`, and compact form one context-governance regime.

Chapter 5 Context Governance: Memory, CLAUDE.md, and Compact as a Budgeting Regime

5.1 As context grows, systems develop a low-level illusion

"More information makes systems smarter"is a dangerous myth. Agent systems are not libraries; context is not a warehouse where "stored" equals "owned." It is an expensive, inflation-prone, self-polluting budget. Claude Code source is unsentimental about this: what to load, what to truncate, what to preserve long-term, and what to summarize short-term are serious runtime governance decisions.

So this chapter asks: how does Claude Code avoid being crushed by what it remembers? "Remember more" and "govern memory" look close but engineering-wise are different regimes.

5.2 The `CLAUDE.md` system: long-lived instructions cannot be mixed with ad hoc chat

At the top of `src/utils/claudemd.ts`, Claude Code already defines memory layers clearly. Instruction sources are split into:

- managed memory, such as `/etc/claude-code/CLAUDE.md`
- user memory, such as `~/.claude/CLAUDE.md`

- project memory, such as root `CLAUDE.md`, `.claude/CLAUDE.md`, `.claude/rules/*.md`
- local memory, such as `CLAUDE.local.md`

These files are loaded by precedence and directory proximity. Project rules closer to current working directory get higher priority. More private and local rules are loaded later, thus placed closer to model attention front.

This is critical. It shows Claude Code refuses to blend long-lived collaboration policy with temporary chat context. Team norms, personal preferences, and repository constraints outlive any single user message. If you force all of them into chat history, systems oscillate between two bad extremes: either reinject everything every turn (context waste), or rely on model recollection (eventual failure).

`claudemd.ts` solves this by making stable rules discoverable, layered, and composable persistent instruction systems. One detail is revealing: it supports `@include`, but only for explicitly allowed text extensions. This shows design balancing convenience against a common failure mode: accidental inclusion of binaries, giant docs, or prompt-toxic artifacts.

That is practical restraint. The system asks first: what deserves memory entry, and what becomes pollution if admitted?

5.3 `MEMORY.md` is an index, not a diary

If `CLAUDE.md` governs policy layer, `memdir` governs fine-grained persistent memory. In `src/memdir/memdir.ts`, one design decision is worth revisiting: `ENTRYPOINT_NAME` is `MEMORY.md`, but it is not intended for direct content dumping. It is explicitly an index.

`buildMemoryLines()` states memory writing as two steps:

1. write concrete memory into dedicated files
2. add one-line pointers in `MEMORY.md`

Why this extra step? Because entrypoint files are loaded frequently. Once they bloat, context is gradually dragged by index weight.

That is why `memdir.ts` sets hard limits: `MAX_ENTRYPOINT_LINES = 200` and `MAX_ENTRYPOINT_BYTES = 25_000`. Beyond those, runtime triggers `truncateEntrypointContent()` and appends an explicit warning: only partial load performed; move details into topic files.

This feels like design from someone who has seen too many runaway indexes. It does not trust everyone to stay concise, so it enforces "entrypoint must stay short." Once an entry file tries to be both table of contents and full text, it eventually becomes neither.

From a Harness Engineering perspective, the principle is clear: long-term memory must split into entrypoint and body. Entrypoint gives low-cost addressing; body carries dense content. Mix them and entrypoint fails, then the memory system degrades into decoration.

5.4 Session memory: short-term continuity cannot be brute-forced by chat logs

Long-term memory alone is insufficient. The hardest real issue in agent sessions is often "where exactly were we before this turn?" That is a continuity problem inside one session.

Claude Code builds dedicated templates for this in `src/services/SessionMemory/prompt`
Default sections include:

- `Current State`
- `Task specification`
- `Files and Functions`
- `Workflow`
- `Errors & Corrections`
- `Codebase and System Documentation`
- `Learnings`
- `Key results`
- `Worklog`

These are clearly not for journaling. They track status, pitfalls, changed files, and next actionable continuation. The update prompt tone is equally explicit:

- Use Edit tool only to update notes file
- Do not talk about note-taking itself
- Do not alter template structure
- Keep `Current State` aligned with latest work
- Keep every section dense but budget-aware

This means session memory in Claude Code is not "save another copy of chat history." It distills the session into an operational continuation brief. It does not seek full replay; it seeks the minimum structure needed to continue working.

There is also a highly practical detail: `prompts.ts` sets `MAX_SECTION_LENGTH = 2000` and `MAX_TOTAL_SESSION_MEMORY_TOKENS = 12000`. Beyond budget, runtime asks for aggressive condensation, with explicit priority on `Current State` and `Errors & Corrections`.

This says a lot. Mature systems treat "preserve what is most useful for continuation" as virtue. Context budget is working memory, and working memory must stay operable.

5.5 Autocompact: context governance is budget governance first

By now we have long-term rules, persistent memory, and session memory. Context still inflates. So in `src/services/compact/autoCompact.ts`, Claude Code admits another reality: regardless of organization quality, long enough sessions approach window limits.

`getEffectiveContextWindowSize()` first subtracts reserved output budget for summary generation. `MAX_OUTPUT_TOKENS_FOR_SUMMARY` reserves 20,000 tokens directly. Runtime assumes compact itself costs budget and never waits until oxygen is almost gone.

Then `getAutoCompactThreshold()` subtracts another `AUTOCOMPACT_BUFFER_TOKENS = 13_000`. Warning thresholds, error thresholds, and manual compact reserve all have dedicated buffers.

The logic is simple: context governance must pre-reserve room for failure and recovery. Systems that reserve nothing may look frugal in normal cases but simply defer risk billing to later turns.

Budget thresholds

Name	Value	Purpose	Source
`MAX_ENTRYPOINT_LINES`	`200`	line cap for `MEMORY.md` index	`memdir/memdir.ts`
`MAX_ENTRYPOINT_BYTES`	`25_000`	byte cap for the index file	`memdir/memdir.ts`
`MAX_SECTION_LENGTH`	`2_000`	per-section cap in session memory	`SessionMemory/prompts.ts`
`MAX_TOTAL_SESSION_MEMORY_TOKENS`	`12_000`	total budget for session memory	`SessionMemory/prompts.ts`
`MAX_OUTPUT_TOKENS_FOR_SUMMARY`	`20_000`	output reserve for compact summary	`compact/autoCompact.ts`
`AUTOCOMPACT_BUFFER_TOKENS`	`13_000`	autocompact early-warning buffer	`compact/autoCompact.ts`
`MAX_CONSECUTIVE_AUTOCOMPACT_FAILURES`	`3`	circuit-breaker threshold	`compact/autoCompact.ts`

`AutoCompactTrackingState` is also revealing. It tracks not only `compacted`, but also `turnCounter`, `turnId`, and `consecutiveFailures`. So autocompact is a tracked runtime behavior that can fail and be rate-limited.

Source even notes a hard lesson: large amounts of API calls were once wasted on repeated autocompact failure, therefore `MAX_CONSECUTIVE_AUTOCOMPACT_FAILURES = 3` with circuit breaker after that. You may fail, but you may not fail infinitely without memory.

5.6 `compactConversation()` shows summaries must rebuild working context

Many people hear compact and assume "summarize the previous chat." Claude Code does much more. In `compactConversation()` within `src/services/compact/compa`

runtime disassembles old context, summarizes, then reinjects required runtime attachments to rebuild a post-compact world that can still work.

Look at pre-summary cleaning first. `stripImagesFromMessages()` replaces images and docs with markers like `[image]` and `[document]`. `stripReinjectedAttachments()` removes attachments that will be reinjected anyway, to avoid token waste. These two actions alone show compact explicitly drops content with high token cost and low summary value.

Now look at summary failure handling. Source includes `truncateHeadForPTLRetry()` for the awkward case where compact request itself hits prompt-too-long. The system admits not only main path can explode, but rescue tooling can also explode.

After compact succeeds, Claude Code does not keep only one summary line. It also:

- clears stale `readFileState`
- regenerates post-compact file attachments
- reinjects plan attachments
- reinjects plan mode attachments
- reinjects invoked-skills attachments
- reinjects deferred tools, agent listing, and MCP instruction delta attachments
- runs session start hooks and post-compact hooks
- writes compact boundary messages with pre-compact token counts and boundary metadata

Taken together, compact in Claude Code is controlled reboot, not chat recap. Old context is translated into new operating substrate. Many systems do only the first half, then discover that after compact they "remember roughly" but lost tool state, plan state, and attachment state, and must spend turns rediscovering themselves.

5.7 Context governance is about preserving work semantics

If you read the latter half of `compact.ts`, one consistent preference appears: Claude Code preserves working semantics.

It restores recently accessed file attachments because they encode local working reality. It restores plan mode because otherwise models may forget they still operate under planning discipline. It keeps invoked-skill content but token-caps each skill so post-compact stage is not hijacked by skill payloads.

One source phrase captures this well: per-skill truncation beats dropping. Even when cutting, keep critical leading constraints rather than dropping whole blocks. That is governance, not brute throttling. Throttling cuts blindly. Governance decides what to cut and what to preserve.

A stable lesson follows: context systems should prioritize preserving action semantics, not preserving the highest apparent information volume. File specifics, current plans, error corrections, and skill constraints directly determine whether next action can be correct. Long repetitive chat history and easily retrievable runtime data do not deserve permanent seats.

5.8 The fifth principle extractable from source

This chapter can be compressed into one sentence:

> Context is working memory. Governance exists to keep the system able to continue work.

Claude Code source supports this on several layers:

- `claudemd.ts` loads long-lived instructions in layers, separating stable rules from temporary dialogue
- `memdir.ts` defines `MEMORY.md` as index and hard-truncates it, forcing entrypoint brevity and addressability
- `SessionMemory/prompts.ts` structures short-term continuity with fixed template plus section/global budgets

- `autoCompact.ts` reserves summary budget, buffers, and failure circuit breaker, treating window operation as risk-managed budgeting
- `compact.ts` restores plans, files, skills, tool attachments, and hook state after summarization, showing compact aims to rebuild working semantics, not produce pretty summaries

As portable engineering principles: layer long-term rules, persistent memory, and session continuity rather than mixing them; keep index-like artifacts small; session summaries serve continuation, not full recollection; compact is a primary path; post-compact context must preserve runtime semantics, not just language surface.

Next chapter asks what happens when this governance system meets hard limits: prompt too long, max output tokens, hook dead loops, and competing recovery branches. That is where you can finally tell whether a system is "hoping nothing breaks" or "designed to keep operating after breakage."

Chapter 6 Errors and Recovery: An Agent System That Keeps Working After Failure

6.1 The least trustworthy sentence in engineering is "under normal conditions"

Many design docs describe only the "normal path," as if a pretty happy-path flow could make errors secondary. Once agent systems enter real runtime, this breaks fast: models get truncated, requests go too long, hooks create loops, tools get interrupted, fallback paths trigger, and recovery logic itself fails.

Maturity cannot be judged by how human-like the system sounds when smooth; it must be judged by whether failures look like system behavior. Claude Code's strength here is not pretending errors are rare. Source repeatedly reflects a calm assumption: errors are part of the main path, and recovery must be predesigned runtime mechanism.

6.2 `prompt too long` is seasonal, not exceptional

For long-session agents, `prompt too long` is not an edge case. It is a season that eventually arrives. Treating it as rare exception is an invitation to be corrected by production.

Claude Code `query.ts` does not treat it as accidental. It can temporarily withhold such errors instead of surfacing immediately. During streaming, withheld logic identifies recoverable classes including:

- prompt too long
- media size error
- max output tokens

Meaning: some errors should be handed to recovery first, then surfaced only if recovery fails. Users usually care less about raw error type than whether work can continue.

For prompt-too-long specifically, Claude Code first tries cheaper, less destructive recovery. If context collapse is enabled, it first calls `recoverFromOverflow()` to flush staged collapse; only if insufficient does it call `reactiveCompact.tryReactiveCompact()`. Recovery is layered: clear known backlog first, then do heavier full compaction.

This sequence is highly practical. Good recovery does not hit every error with one giant hammer. It tries to preserve fine-grained context first and escalates only when required.

6.3 Reactive compact: recovery must not become dead-loop machinery

A common mistake in recovery systems is both naive and costly: once an error is "recoverable," keep retrying until recoverable turns into resource disaster.

Claude Code is explicitly defensive against this. Two places in `query.ts` show it.

First is `hasAttemptedReactiveCompact`. Once reactive compact has already been attempted, same-class failures are not blindly retried. If compact did not help, repeating compact often just replays the same failure in different posture.

Second is stop-hook dead-loop guards. Source comments are direct: if stop hooks run after unrecoverable prompt-too-long states, death spirals become possible:

error -> hook blocking -> retry -> error -> hook blocking

This is not literary writing; it is brutally honest engineering. The most dangerous errors are often branches where failure and recovery start reproducing each other.

So when prompt-too-long cannot recover, Claude Code surfaces the error directly and skips stop hooks. Continuing formal process there only ritualizes failure.

6.4 `max_output_tokens`: recovery should prioritize continuation

Many model products handle truncation with polite waste: "Sorry, I was cut off, let me summarize." It sounds nice and helps little.

Claude Code behavior after `src/query.ts:1185` is far more engineering-aligned. It first tries lower-cost recovery: if current cap is conservative, raise `maxOutputTokensOverride` and rerun the same request. No meta message, no courtesy preface, just one more chance to finish the task.

If higher cap still fails, it escalates to second layer: append a concise meta user message that says, in effect:

continue directly; no apology; no recap; if cut mid-sentence, continue from that half sentence; split remaining work into smaller chunks.

This instruction is highly instructive. Claude Code treats recovery as preserving task continuity, not preserving social polish. In long tasks this difference is huge. Each truncation recap burns additional budget and increases semantic drift. Eventually system spends turns recapping itself instead of doing the task.

For `max_output_tokens`, good recovery is usually continuation-first. Claude Code explicitly optimizes for that.

6.5 Auto-compact circuit breaker: recovery systems need governance too

Previous sections are about "how to recover one failure."`src/services/compact/autoCom` tackles another layer: what if recovery mechanism itself keeps failing?

Source answer is simple and correct: stop retrying forever.

`AutoCompactTrackingState` tracks `consecutiveFailures`. Once failure count exceeds threshold, even if `shouldAutoCompact` says compact is due, system

skips compact directly. Source comments reference historical waste: sessions once burned large amounts of API calls on repeated autocompact failure, so circuit breaker was required.

Circuit breaking means admitting current recovery method is ineffective in current state. Mature systems must not only record success metrics; they must know when to back off under repeated failure. Recovery systems without brakes are like vehicles without brakes.

Harness Engineering principle here is hard and clear: any automated recovery must be countable, rate-limited, and breakable.

Circuit-breaker invariants

```
assert withheld_error ∈ {prompt_too_long, media_size, max_output_tokens}  # re
assert hasAttemptedReactiveCompact ⇒ skip further reactive compact      # no s
loop
assert consecutiveFailures < MAX_CONSECUTIVE_AUTOCOMPACT_FAILURES       # bre
assert compact aborted by user ⇒/ summary_success                    # abort ≠ s
assert every withheld recoverable_error surfaces iff recovery exhausted   # wi
```

Recovery-path failure matrix

Event order	Pre-state	Trigger	Next	Threshold
PTL → collapse	`stagedCollapses > 0`	`prompt_too_long`	`recoverFromOverflow()`	—
PTL → compact	`stagedCollapses = 0`	`prompt_too_long`	`tryReactiveCompact()`	once per turn
PTL → surface	`hasAttemptedReactiveCompact`	`prompt_too_long`	surface directly; skip stop hooks	—
compact PTL	compact input too long	inner `prompt_too_long`	`truncateHeadForPTLRetry()`	drop PTL API rounds in chunks

Event order	Pre-state	Trigger	Next	Threshold
MOT → cap ↑	cap < MAX	`max_output_tokens`	retry with `maxOutputTokensOverride`	cap ∈ {conservative, max}
MOT → continue	cap = MAX	`max_output_tokens`	append meta user msg, continue	no recap, no apology
autocompact streak	`consecutiveFailures` ⩾ threshold	next trigger	skip compact, surface	`MAX_CONSECUTIVE_AUTOC = 3`
user abort	streaming with pending `tool_use`	Esc	consume remaining + synthetic `tool_result`	ledger must close

6.6 Compact itself can fail, so recovery action needs its own recovery

`compactConversation()` contains a very realistic moment: compact requests themselves can hit `prompt too long`.

There is black humor here. System sends summary requests to reduce context, then summary requests fail because context is too large. Many designs hide this scenario because it is embarrassing. Engineering systems prioritize survival over elegance.

Claude Code handles this with `truncateHeadForPTLRetry()` in `compact.ts`. When compact input is too large, it strips older API rounds in chunks from the head and retries compact, preventing users from being stuck in "cannot compact compact."

Trade-off is explicit: this fallback is lossy and may drop history, but it prioritizes not deadlocking users. Source comments describe it as last-resort escape hatch.

Value of this choice: it does not deny hard limits. When the system is choking, first priority is restoring breathing, then preserving high-fidelity history.

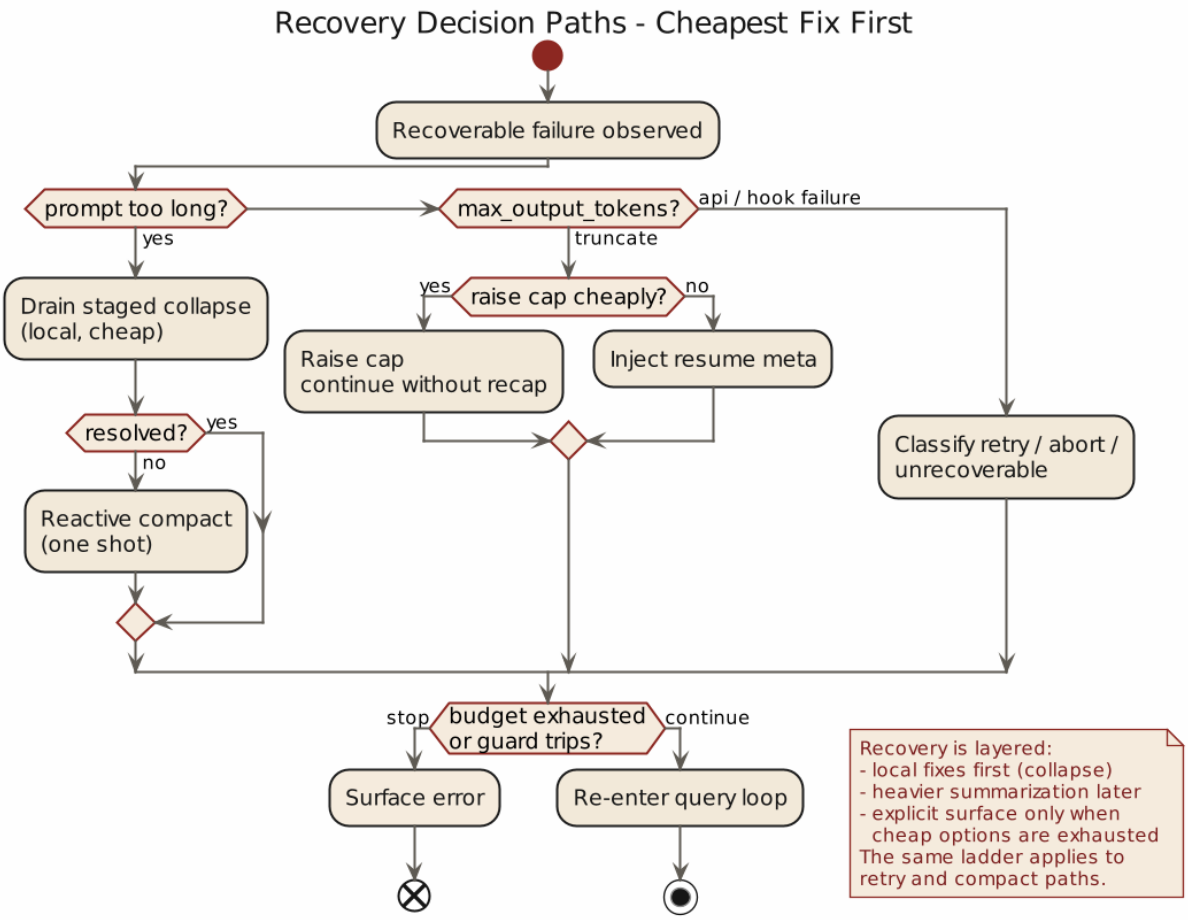


Figure 6: Claude Code Recovery Decision Paths

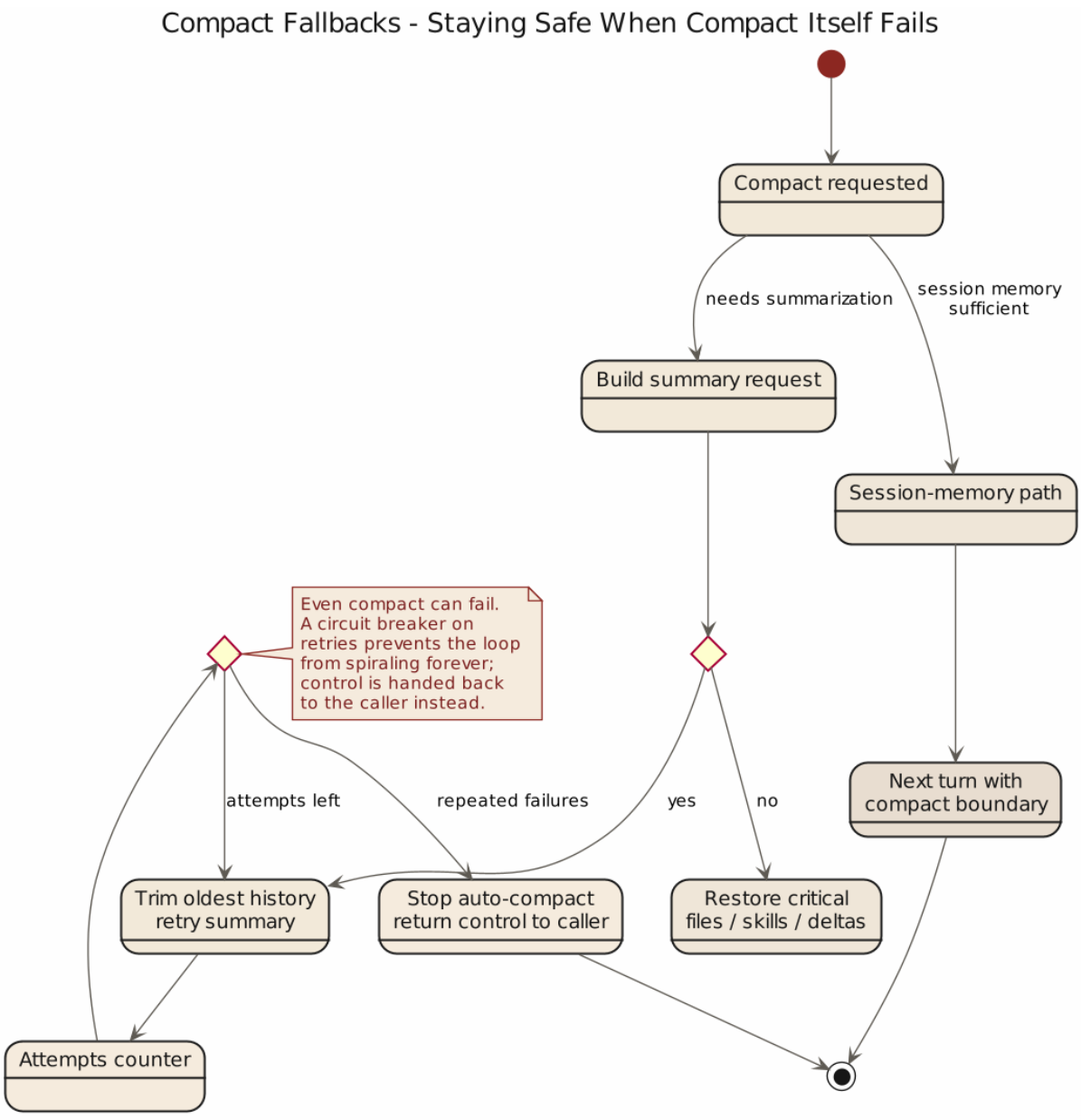


Figure 7: Claude Code Compact Fallbacks

6.7 Abort semantics: interrupts are part of recovery

Many teams classify abort under UX only. Runtime-wise, interrupts are failure states requiring semantic closure.

Claude Code handles this at two layers. In `query.ts`, streaming interrupt consumes `StreamingToolExecutor.getRemainingResults()` and synthesizes tool results for issued-but-unfinished calls, preventing dangling commitments. In `compact.ts`, the compact abort controller is passed into the forked agent and `APIUserAbortError` is handled explicitly, preventing "compact cancelled by user Esc" from being counted as a successful summary.

Interruption is not merely "user stopped reading"; it is a state transition requiring correct closure. Recovery that handles exceptions but ignores interruption leaves semantically half-broken execution traces.

6.8 Error handling protects narrative consistency of execution

Claude Code's recovery philosophy has one often-overlooked goal: preserve narrative consistency of execution —whether the system can still explain what it attempted, why it failed, which recovery path was used, and whether it is now continuing, stopping, or rerouting.

Fields like `transition.reason`, `maxOutputTokensRecoveryCount`, `hasAttemptedReactiveCompact`, compact boundaries, and synthetic error messages in `query.ts` exist to keep this narrative unbroken —they are anti-amnesia mechanisms. Without narrative consistency, systems keep outputting text while internally decomposing: users see filler, ops sees hook-retry and compact-retry loops with unclear causality, and teams can no longer explain what the system actually went through. Recovery fixes not only errors but the system's self-explainability; once explainability breaks, the engineering object degrades into opaque magic.

6.9 The sixth principle extractable from source

This chapter can be compressed into one sentence:

> An agent system shows its reliability by maintaining explainable, bounded, and resumable execution order after failure.

Claude Code source supports this across key points:

- `query.ts` withholds recoverable errors for branch-level transformation before surfacing
- Prompt-too-long recovery layers collapse drain before reactive compact, ordered by cost and destructiveness
- `hasAttemptedReactiveCompact` and stop-hook guards prevent recovery self-loops
- `max_output_tokens` handling escalates cap first and prefers direct continuation over polite recap
- `autoCompact.ts` tracks consecutive failures and enforces circuit breaking
- `compact.ts` includes fallback for compaction' s own prompt-too-long failure

As portable principles: layer recovery paths rather than using one heavy hammer; recovery logic must be loop-safe; automated recovery needs counters and circuit breakers; after truncation, continuation beats summary; interruptions are semantic failure states requiring closure; reliability is proven by whether the system can still explain itself after errors.

Next chapter turns to a harder class of problems: multi-agent and verification. When systems move from "one agent recovers itself" to "one agent delegates and another verifies," error and recovery stop being purely single-thread concerns and become organizational design.

Chapter 7 Multi-Agent Work and Verification: Managing Instability Through Division of Labor

7.1 Past a certain point, the question is not “can one agent do it” but “how should work be split”

Single-agent friction can be hidden by patience, but once tasks grow, research, implementation, and verification all crowd one context chain, competing for budget, attention, and narrative center. Multi-agent looks like an obvious answer, but it is not cheap: unconstrained parallelism just copies single-agent disorder. The real difficulty is isolating per-agent instability while recombining outputs coherently.

Claude Code source is clear-eyed here: it does not treat subagents as “another chat window.” It treats them as managed execution processes with explicit cache boundaries, state boundaries, verification duties, and cleanup responsibilities.

7.2 The first principle of forked agents is cache safety

At the top of `src/utils/forkedAgent.ts`, a comment precisely states forked-agent utility duties:

1. share cache-critical params with parent to ensure prompt-cache hits
2. track usage across the whole query loop
3. record metrics

4. isolate mutable state to avoid disturbing parent loop

Among these, cache-critical sharing appears first. That is not accidental. It means fork is runtime-controlled branching. If it is branching, parameters that must remain parent-consistent are crucial; otherwise prompt-cache reuse fails and both cost and latency degrade.

`CacheSafeParams` explicitly includes:

- `systemPrompt`
- `userContext`
- `systemContext`
- `toolUseContext`
- `forkContextMessages`

Source also warns against casually changing `maxOutputTokens`, because thinking configuration may change and thinking config is part of cache keys.

This shows multi-agent is first a runtime economics problem. If each child agent re-burns parent context from scratch, what looks like parallel acceleration is parallel waste. Claude Code's first concern in fork paths is preserving cache discipline.

Skeleton: forkAgent()

```
// skeleton: forkAgent() (src/utils/forkedAgent.ts, src/utils/createSubagent
params = CacheSafeParams {
    systemPrompt, userContext, systemContext,
   toolUseContext, forkContextMessages     // must match parent to keep prompt
cache hits
}
ctx = createSubagentContext(parent):
   readFileState  := clone(parent.readFileState)          // isolated by defau
   abortController := new ChildAbortController(parent)     // child controlle
    getAppState    := wrap(parent.getAppState, suppress_prompt)
    setAppState    := noop                                  // read-
only by default
    nestedMemoryAttachmentTriggers := new Set()
```

```
    loadedNestedMemoryPaths         := new Set()
if opt_in.shareAbortController:  ctx.abortController := parent.abortControlle
if opt_in.shareSetAppState:    ctx.setAppState    := parent.setAppState
hooks.fire(SubagentStart, { agent_id, agent_type })
track_usage(parent, child)
defer hooks.fire(SubagentStop, { agent_transcript_path })    // lifecycle mus
```

Invariants

```
assert child.CacheSafeParams == parent.CacheSafeParams      # fork does not bi
assert child.mutable_state isolated unless opt_in.share_*     # sharing must b
assert parent.abort ⇒ propagate(child.abort)               # parent dies, chil
assert SubagentStart fired ⇒ SubagentStop fired eventually    # lifecycle clo
assert verification_worker ≠ implementation_worker          # verify and imple
```

7.3 State isolation: reduce contamination before sharing anything

The second key is `createSubagentContext()`: by default, all mutable state is isolated to avoid parent interference. Defaults clone `readFileState`, create child abort controllers, wrap `getAppState` to suppress permission prompts, make `setAppState` a no-op, and recreate sets like `nestedMemoryAttachmentTriggers` and `loadedNestedMemoryPaths`. Sharing happens only via explicit opt-in flags such as `shareSetAppState`, `shareSetResponseLength`, `shareAbortController`.

The reason matters: the main value of child agents is containing local chaos away from main thread. Research misreads, temporary file observations, exploratory reasoning branches, and in-flight tool decisions should not be blindly written back. Claude Code's stance is explicit: sharing requires consent, isolation is default ethics —closer to transactional database design than chat toys.

7.4 Coordinator mode: synthesis is the scarce capability

In `src/coordinator/coordinatorMode.ts`, coordinator expectations are disciplined:

- help users reach goals
- direct workers for research, implementation, and verification
- synthesize findings and communicate with users
- answer directly when delegation is unnecessary

The most important line in section 5 of the prompt is: **Always synthesize**. When workers return findings, coordinator must digest and convert them into concrete prompts; it must not forward raw findings and outsource understanding again.

This is the throat of multi-agent design. The scarce capability is not parallel output; it is recompressing distributed local knowledge into actionable, verifiable next steps. Without this layer, multi-agent degrades into polite task forwarding. Claude Code handles this at prompt level: research and synthesis are separated, coordinator owns interpretation, and follow-up prompts must mention concrete files, locations, and changes rather than abstract "based on above findings." Research can be distributed; understanding must reconverge.

7.5 Verification must be independent, or implementation completion will impersonate problem solved

`coordinatorMode.ts` also contains a section worth copying verbatim in spirit. It separates common task flow into:

- Research
- Synthesis
- Implementation
- Verification

And explicitly states verification proves effectiveness, not merely code existence:

- run tests with the feature enabled
- investigate errors instead of dismissing them as unrelated
- stay skeptical
- test independently, do not rubber-stamp

This shows Claude Code treats verification as second-layer quality gate, not a side effect of implementation: prompt explicitly stacks implementation self-check plus independent verification worker.

Why it matters: "I changed code" and "the change is correct" are separated by a wide river, and models are good at building paper bridges over it —changes, explanations, plausible test-like output, none of which guarantees real correctness. Independent verification prevents "can modify code" from impersonating "can deliver outcomes" ; in multi-agent work this role separation is central.

7.6 Hooks and task lifecycle: spawning a subagent is only the beginning

In multi-agent systems, a commonly ignored fact is that spawn is start, not end.

`src/utils/hooks/hooksConfigManager.ts` defines `SubagentStart` and `SubagentStop` hooks. Start events include `agent_id` and `agent_type`. Stop events include `agent_transcript_path`, and even allow exit code 2 so stderr can be fed back and let subagent continue.

This means subagents are explicit lifecycle objects in Claude Code. Start can be observed, pre-stop can be intercepted, transcript paths are traceable. "Subagent end" is itself a managed event.

At another layer, `registerAsyncAgent()` in `src/tasks/LocalAgentTask/LocalAgentTask` shows each async agent has cleanup handlers. Parent aborts propagate to child abort controllers. On task completion, outputs can be evicted, state updated, and cleanup callbacks unregistered.

This resembles operating-system lifecycle control, not chat tabs:

- is the agent still running
- should child terminate when parent terminates
- should output artifacts be retained

- were cleanup handlers leaked

Many multi-agent demos stop at "I can start another agent." Claude Code takes one more crucial step: it treats agents as runtime entities that can leak resources, leave state residue, and become orphans after parent death.

Subagent lifecycle failure matrix

Event order	Pre-state	Trigger	Next	Threshold
parent abort + child in-flight	child unfinished	parent abort	propagate `child.abort`, wait cleanup	no orphans allowed
child crash	child exits abnormally	`SubagentStop` exit ≠ 0	evict output, fire stop hook	exit 2 feeds stderr back to continue
child timeout	child runs too long	`registerAsyncAgent` timeout	abort child, write synthetic result	—
cache key drift	`CacheSafeParams` mutated	fork	refuse fork or rebuild cache	must align fully
undeclared share	opt-in not enabled	child writes `setAppState`	no-op, parent untouched	isolation is default
stale memory conflict	memory disagrees with reality	old record read	trust reality; update/delete memory	verify first
leaked cleanup	task ends without unregister	finalize	force unregister + evict	no dangling handlers

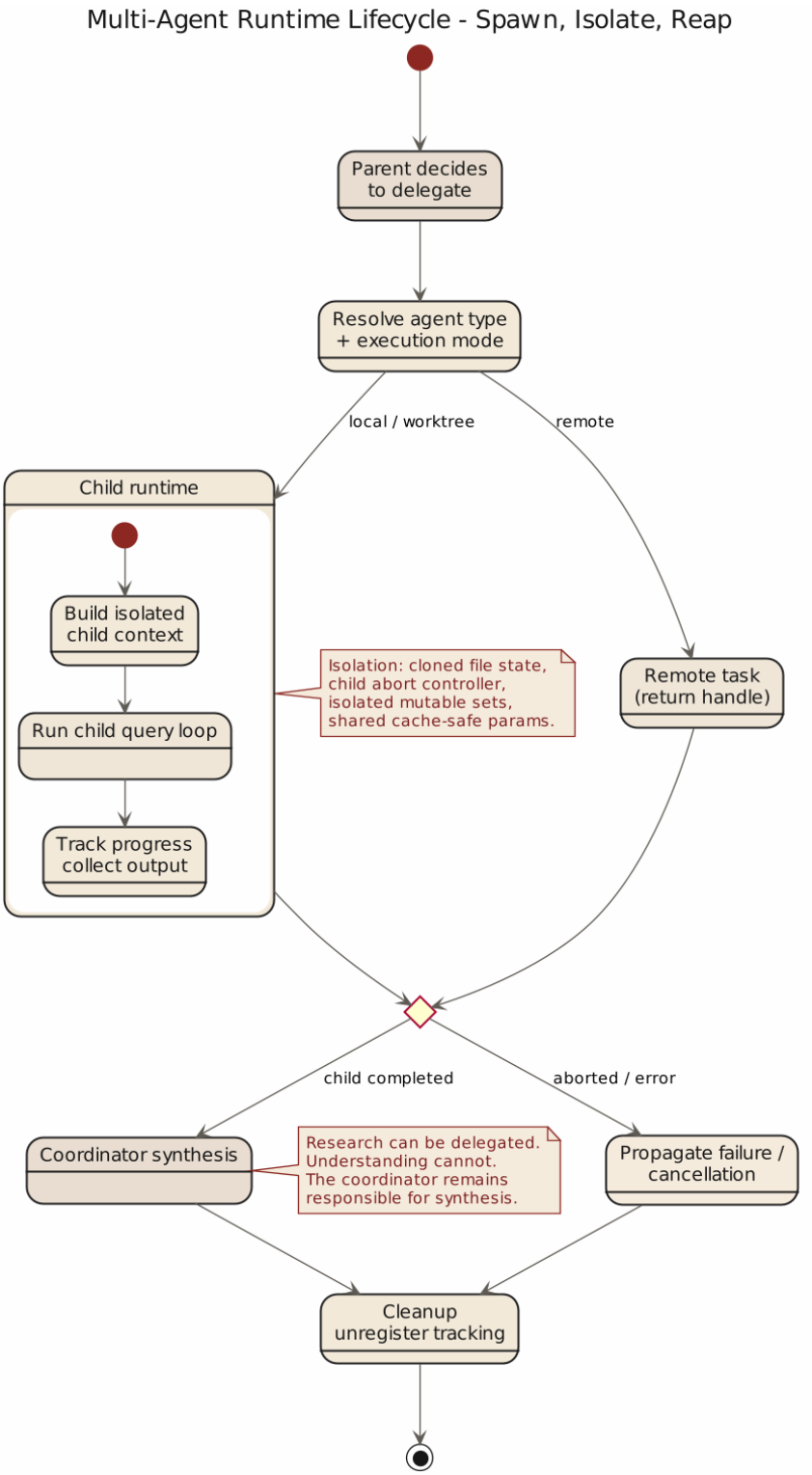


Figure 8: Claude Code Multi-Agent Runtime Lifecycle

7.7 Verification applies to memory and recommendations, not only code

Multi-agent verification is not only a post-code-change activity. `src/memdir/memoryTypes` warns explicitly: memory records can become stale; before giving recommendations from memory, verify current state; when memory conflicts with present reality, trust the current observed state and update or delete stale memory.

The broader fact: verification is the system's baseline habit against temporal and context drift. If a system verifies newly written code but not old memory, assumptions, or indexes, history will still steer it wrong. Verification is both a skill and an institutional discipline: you can delegate work, persist information, and let other agents run ahead, but before users act on outputs, someone must return to current reality and revalidate.

7.8 Multi-agent really solves uncertainty partitioning

Claude Code's multi-agent design centers on a plain target: partition uncertainty. Research workers explore in local contexts, implementation workers focus on edits, verification workers specialize in skepticism, and coordinator sits in the middle to converge and communicate.

The main benefit is clear responsibilities, so failures can be located: did research miss a key signal, did synthesis fail to digest findings, did implementation introduce defects, did verification undercheck? A single agent delivering "one thick soup" cannot be debugged by layer. Multi-agent's value is less about speed and more about placing different uncertainties in different containers, then recombining them through coordinator synthesis.

7.9 The seventh principle extractable from source

This chapter can be compressed into one sentence:

> Multi-agent systems depend on clear division of labor: research, implementation, verification, and synthesis must run under different constraint containers, then be stitched back into deliverable outcomes by a coordinator.

Claude Code source supports this jointly:

- `forkedAgent.ts` prioritizes cache-safe params, usage tracking, and state isolation, making fork a runtime-control problem first
- `createSubagentContext()` isolates mutable state by default and only shares via explicit opt-in
- `coordinatorMode.ts` requires synthesis instead of forwarding, so understanding cannot be outsourced
- verification is explicitly separated from implementation and must independently prove effectiveness
- `hooksConfigManager.ts` provides `SubagentStart` and `SubagentStop`, making subagents observable lifecycle objects
- `LocalAgentTask.tsx` handles parent abort propagation, cleanup, and output eviction

As portable principles: fork first for cache and state boundaries, then for "persona specialization"; isolate child agents by default and share only by explicit declaration; research can be delegated, synthesis cannot; verification must be role-separated from implementation; agent lifecycle must be observable, interruptible, and reclaimable; the real parallel value is clearer responsibilities, not raw speed.

Next chapter moves from runtime design to organization design: how `CLAUDE.md`, skills, approvals, hooks, and memory become reusable team institutions instead of private expert tricks.

Chapter 8 Team Adoption: Turning a Smart Tool into a Sustainable Workflow

8.1 Expert success does not automatically turn into safe team reuse

Many AI coding tools look impressive in expert hands: skilled users know when to patch context, when to watch the model closely, and when a single "do not touch this directory" holds behavior in place. That easily creates an illusion — if power users drive it smoothly, team rollout is just more best-practice notes.

The problem is that personal technique works precisely because it depends on continuous supervision, background knowledge, and situational judgment. Once a team adopts the tool, assumptions change: you can no longer assume everyone knows which commands are dangerous, which memories are stale, which skills fork subagents, which steps may skip approval, and which must not. The real team problem is turning order that lived in expert heads into a workflow ordinary contributors can repeat without heroics.

Claude Code source is useful here because it makes expert behavior explicit: instructions have loading layers, permissions have decision points, subagents have isolation boundaries, runtime exposes lifecycle hooks. Adopting a coding agent is both introducing a smarter autocomplete and rearranging who may do what inside which boundary.

8.2 The first team step is to make minimum boundaries clear

This chapter is easiest to misread if team adoption is imagined as a large governance project from day one. In reality, most teams do not begin with hooks, audit chains, and elaborate skill catalogs. Their starting point is usually much smaller:

- which task classes may involve the agent at all
- which changes still require human review
- what minimum verification must happen before work is considered done
- which resources are simply off limits

These sound basic, but they matter more than ambitious slogans. Early on, teams usually benefit most from having a clearly defined minimum control boundary.

If that boundary is wrong, every later automation layer becomes distorted:

- if allowed usage is vague, people will use the agent in places that were never meant to be automated
- if review responsibility is vague, nobody knows who owned the last meaningful checkpoint
- if verification is vague, the system learns to satisfy the weakest standard in the room
- if forbidden zones are vague, efficiency gains become blast-radius expansion

So the more realistic rollout order is rarely "build many skills first, then add governance later." It is usually:

1. define acceptable usage scope
2. define review and verification expectations
3. then decide which recurring workflows are worth formal reuse

Teams often fail because they skipped this step, even when the agent itself is quite capable.

Staged rollout checklist (builder checklist)

```
- [ ] Week 1: layered CLAUDE.md live — team / personal / project all verifiable
- [ ] Week 1: shared verification defined (lint / type / test commands in CLAUDE
- [ ] Week 1: forbidden zones encoded as repo-level hard constraints (dirs, dang
- [ ] Week 2: approvals tiered by consequence (allow / deny / ask), Bash has its
- [ ] Week 2: first ≤3 skills shipped, each with precondition / postcondition /
- [ ] Week 2: explicit policy for "done with known issues"
- [ ] Week 3: stop / post-tool-use hooks landed, capturing transcript + task out
- [ ] Week 3: monthly stale-memory maintenance in effect
- [ ] Week 3: baseline replay (Git diff / PR / CI) gap-
free before pursuing advanced audit
Gate: a newcomer can use it without an expert standing by — the workflow is matu
```

8.3 `CLAUDE.md` matters because it stays stable, layered, and low-dispute

Earlier chapters covered layered loading in `claudemd.ts`. That still matters during team adoption, but it should be interpreted with some restraint.

Team-level `CLAUDE.md` is best used for stable rules. It does not need to carry every procedural detail. Examples include:

- repository-level hard constraints, such as forbidden directories or dangerous command classes
- shared verification expectations, such as the minimum checks that must run
- collaboration discipline, such as do not overwrite user-unrequested changes and do not reset dirty worktrees without explicit instruction
- output discipline, such as findings-first review reporting

What does not fit well: rapidly changing temporary processes, instructions relevant only to a narrow task subset, and details better expressed as commands, skills, or scripts. Once `CLAUDE.md` becomes encyclopedic, it loses stability and credibility at once: team members stop knowing whether a rule is current or leftover from months ago, and the system learns to treat obsolete norms as active law.

So the ideal team `CLAUDE.md` is not simply "large." It should be stable enough that people rarely need to debate it. It functions like foundation, not bulletin board.

8.4 Verification definition usually matters earlier than skill count

One of the most common failures in coding-agent rollout is that the team has no shared definition of done.

One person thinks "it runs" is enough. Another accepts something half-tested. Another accepts a plausible-looking explanation from the model. In that environment, even a smart system learns to satisfy the weakest bar available.

Claude Code repeatedly pushes against that slide. Earlier we saw coordinator mode separate verification into its own stage; verification-related instructions are not just about proving files exist, but about proving the change actually works.

This matters especially for teams because skills can replicate process, but only verification definitions replicate quality.

A more realistic team move is to define these three things before building many skills:

- which task classes require independent verification
- what minimum actions verification must include, such as tests, local runs, logs, or human acceptance
- whether failed verification may still be marked as "done with known issues"

If these stay unclear, any later automation only speeds up ambiguity. If they become clear early, then even a team with very few skills can still hold a consistent quality floor.

So the more durable rollout order is usually:

- standardize verification definition first
- then package recurring workflows into skills or commands
- only then consider more complex automation

8.5 Skills are better understood as workflow modules than as institutions themselves

When teams first design skills, they drift into two bad models: either a skill is "just a long prompt template," or it is over-elevated into an institutional object. Neither fits.

Claude Code clearly treats skills as more than prompt snippets: if intent matches a skill, the Skill tool must be called; already-loaded skills must not be reloaded; a skill can run in a forked subagent context with its own boundary and tool set. So a skill is at least an executable workflow module. But for team practice, the safer interpretation stays narrower —skills should first solve "how to reuse this recurring workflow reliably," not encode the whole organization into agent form.

The questions worth answering during skill design: what task class does it serve, what tools may it use by default, should it execute directly or fork to a subagent, what verifiable result must exist afterward. Unanswered, skills decay into well-named but semantically vague slogans.

8.6 Approval works best when it is tiered by risk

Claude Code keeps returning to one distinction: being able to do something differs from being authorized to do it. Solo use underestimates this because individuals grant themselves ad hoc authority; teams cannot. Once an agent writes files, mutates Git state, touches networks or external systems, each step is also a movement of responsibility.

Teams often make the opposite mistake: design an elaborate approval taxonomy immediately, and the rollout cost becomes unsustainable. More realistic is to tier by risk, not tool name —reads / listings / pure analysis are lower risk; workspace mutation and write operations are materially higher; Git push, external network, and sensitive environments are higher again. What teams really need to control is irreversibility and environment sensitivity, not the button label. Once those boundaries are clear, automation is less likely to amplify damage in the wrong places.

Approval rule template

```
// template: approval rule (mirrors useCanUseTool's three-
valued decision)
rule {
    name:           <short>
    match:          { tool, args_pattern, cwd_scope }
   risk_tier:    read | write | irreversible    // tier by consequence, not too
    decision:       allow | deny | ask
    ask_to:         coordinator | reviewer | operator
    ttl:            session | turn | persistent
    audit:          transcript + PR comment + CI log
}
assert every irreversible action ⇒ decision ∈ {deny, ask}   # irreversible mus
assert Bash compound commands > subcommand_cap ⇒ deny      # bundled commands
assert ask never auto-escalates to allow                    # three-
valued never collapses
```

8.7 Hooks are powerful, but usually belong later in the rollout

`hooksConfigManager.ts` exposes many lifecycle events: `SessionStart`, `SessionEnd`, `SubagentStart`, `SubagentStop`, `PreCompact`, `PostCompact`, `FileChanged`, `DirectoryChange`, and others. Read together, they make the real value of hooks obvious: rules can happen at the right moment.

Examples include:

- injecting organization-level context when instruction files load
- running one more verification step before a subagent stops
- recording summary around compact boundaries
- doing archival or cleanup work at session end

All of that is useful. The more important judgment is that useful does not necessarily mean first.

For most ordinary engineering teams, the first move is usually simpler: repository-level instruction files, code-review rules, CI and test expectations,

and a small set of recurring commands or skills. Hooks start to pay off when scale, risk, or compliance pressure rises further; before that, they often introduce more moving parts than the team is ready to own —unmaintained scripts, unclear trigger timing, and debugging costs that exceed the manual step they replaced.

So the more mature conclusion is that hooks are an advanced automation interface. They fit timing-bound actions and usually belong after baseline governance is already stable.

8.8 Replayability matters, but teams should separate baseline traces from advanced audit trails

Observability and auditability are the easiest things to overstate in this chapter. Of course teams want to know why something happened, where drift started, and who authorized the consequential step. Claude Code's logs, telemetry, task outputs, transcript paths, hook events, and agent notifications do create stronger replayability.

But two layers must be separated. The baseline layer —Git diffs and commit history, PR comments and reviewer decisions, CI and test logs, issue trackers and acceptance notes —most teams already have, and it is usually enough for day-to-day reconstruction. The advanced layer —transcript paths, tool-call records, hook events, compact summaries, subagent usage and state transitions —pays off when scale or compliance justifies it. The key is to make layer one gap-free before investing in layer two; many teams lack a clean review or verification standard, and chasing full agent auditability too early turns governance into an expensive display.

So baseline replayability is a requirement for team adoption; advanced audit trails are an enhancement for high-risk, high-scale, or compliance-heavy teams. That distinction prevents platform-team governance from being miswritten as everyone's starting point.

8.9 The eighth principle we can extract from source

The closing sentence of this chapter is more accurate in this form:

> Team adoption works best when acceptable boundaries, verification standards, and recurring workflows become stable early.

What Claude Code source actually suggests, in portable form, is closer to this:

- instructions must be layered so that stable rules and temporary process details do not collapse into each other
- skills are useful for recurring workflows, but only when applicability, tool scope, and outputs are explicit
- approvals should be tiered by risk and environment rather than crudely sorted by tool names
- hooks are powerful, but they belong after baseline governance is already stable
- replayability should be built in layers: baseline traceability first, advanced auditability when the context justifies it

Translated into team-operable principles: define acceptable usage scope before large-scale rollout; standardize verification definitions before increasing skill count; use review, CI, and a small set of stable instruction files to set the floor before adding hooks and orchestration; require every automation path to be explainable without demanding full heavy auditability from day one; optimize not for maximum institutional weight but for clearer boundaries and a more bearable system.

The next chapter closes the book. Across the previous chapters, one judgment keeps returning: the model is the least stable component, so what we really design is how to keep the surrounding system bearable, verifiable, and correctable when the model is not reliable.

Chapter 9 Ten Principles of Harness Engineering

Technical writing has a bad habit: after presenting enough detail, it avoids making judgments, as if showing complexity exempts conclusion responsibility. It does not. Complexity remains complexity, but judgment is still required. Teams carry principles forward, not function names from one source version.

Across the previous eight chapters, we kept circling one fact: models are unreliable, but systems can still be reliable if reliability is built into harness rather than delegated to models.

This chapter does not re-argue details. It compresses the book into ten principles. They are not slogans. They come from working judgments reflected in Claude Code source and engineering structure.

9.1 Treat models as unstable components, not teammates

Teammates can be trusted with stable responsibility. Models cannot. Models may speak like teammates, but they do not automatically gain teammate-grade stability, accountability, or sustained judgment. The earlier this is acknowledged, the earlier systems grow permission boundaries, recovery paths, verification gates, and rollback ability.

9.2 Prompt is part of the control plane

System prompts define behavioral protocol. Together with runtime, tool schema, memory, and hooks, they form the control plane. If prompt is treated as persona decoration, you get systems that perform well rhetorically and behave without discipline.

9.3 Query loop is the heartbeat of agent systems

Real agents depend on continuous execution loops. Input governance, stream consumption, tool scheduling, recovery branches, and stop conditions all belong to heartbeat. Without query loop, systems may demo well but are not runtimes.

9.4 Tools are managed execution interfaces

Once models touch shell, filesystem, Git, and networks, the question shifts from "can it say it" to "can it leave consequences." Therefore tools must be scheduled, authorized, interruptible, and ledger-closed. The more dangerous a tool is, the less it should be treated like ordinary capability.

9.5 Context is working memory

Being able to stuff context does not mean context should be stuffed. Long-lived rules, persistent memory, session continuity, and temporary dialogue should be governed in layers. Compact exists to preserve semantic substrate for continued work. Good context management optimizes not for "more," but for "governable."

9.6 Error paths are main paths

Prompt-too-long, max-output-tokens, interrupts, hook loops, and compact failures are ordinary weather for long-session agents. Recovery, circuit breaking,

retry limits, and anti-loop guards must exist at design time, not post-incident patch time.

9.7 Recovery should optimize for continuation

After truncation, continuation is usually better than recap. When compaction fails, first restore breathing. True engineering politeness is not trapping users inside failure states.

9.8 Multi-agent matters because it partitions uncertainty

Multi-agent places research, implementation, verification, and synthesis into different responsibility containers. State is isolated, roles are separated, and coordinator reconverges understanding. The real value of parallelism is not speed alone, but sharper responsibility boundaries.

9.9 Verification must be independent

Implementers naturally overtrust their own changes. Models do so even more. For important work, verification should be a dedicated independent phase, ideally with independent role ownership. Otherwise "done" quickly degrades into "written and feels fine."

9.10 Team institutions matter more than personal tricks

An expert can tame an agent by experience. A team cannot rely on that. Teams need:

- layered `CLAUDE.md`
- explicit approval boundaries

- executable skills
- lifecycle hooks
- traceable transcripts
- unified verification definitions

Only when personal technique is institutionalized can agent systems become organizational capability rather than personal craft.

9.11 One final sentence

If the whole book must be compressed one more time:

> Harness Engineering asks how systems can still behave like engineering systems when models themselves are not reliable.

What Claude Code source really teaches is restraint: treat instability as known input, then design prompt, loop, tools, memory, compact, recovery, verification, and team workflow around that input. That is exactly why it is worth studying as a design specimen.

At this point the conclusion is simpler than the path. The hard part is not saying principles, but accepting them: harness over excitement, institutions over cleverness, and verification over confidence. Teams that internalize these three are already standing at the door of Harness Engineering.

Appendix A Checklists: Turning Principles into Executable Constraints

Previous chapters discussed principles. If principles cannot be turned into checklists, they often decay into judgments that sound right but do not hold up in practice. This appendix turns easy-to-say, hard-to-sustain judgments into directly usable lists.

These checklists do not guarantee automatic improvement. They mainly prevent common and repetitive mistakes from recurring. In engineering, many improvements come from repeating fewer avoidable errors.

A.1 Agent runtime design checklist

If an AI coding agent is going into real engineering workflow, it should answer at least:

- Is there an explicit query loop instead of treating each turn as isolated Q&A?
- Is there a cross-turn state object recording recovery, budget, compaction, hooks, and turn counters?
- Is model output handled as event stream instead of final prose blob?
- Can interrupted tool calls be closed with synthetic results to keep execution ledger complete?
- Are completion, failure, recovery, and continuation modeled as distinct stop semantics?

- Is there context budgeting for long sessions, rather than ad hoc emergency handling only after overflow?

If two or three of these cannot be answered clearly, the system is likely still in "demo-capable" stage rather than "workflow-capable" stage.

A.2 Prompt design checklist

System prompt should not be merely long. It should be layered with explicit duties.

At minimum, check:

- Are identity, behavior rules, tool constraints, and output discipline organized separately?
- Is prompt source precedence explicit (default, project, custom, append, agent-specific)?
- Are dangerous operations, unauthorized behavior, and verification discipline written as explicit rules rather than vague hints?
- Is prompt prevented from carrying duties that belong to runtime enforcement?
- Can team maintain it stably, rather than appending emergency text after every bug?

A practical test: if removing one section causes structural behavior change, that section is likely true control-plane logic. If behavior barely changes, it may be decoration.

A.3 Tool and permission design checklist

Any system that lets models touch the world should ask first:

- Do tool calls pass through unified orchestration rather than direct model invocation?
- Does concurrency require explicit safety proof rather than default allowance?

- Is there semantic branching like `allow / deny / ask`?
- Are high-risk tools treated as special cases rather than identical to ordinary tools?
- Can interrupts, fallbacks, and sibling failures produce explicit closure semantics?
- Can execution preserve causal chain and avoid dangling `tool_use` blocks?

If Bash and ReadTool are governed almost identically, risk understanding is usually insufficient.

A.4 Context governance checklist

Any long-session agent will eventually be disciplined by context limits. Early governance is cheaper.

At minimum, check:

- Are long-lived rules, persistent memory, session continuity, and temporary dialogue layered?
- Is there explicit separation between entrypoint files and body files to prevent index bloat?
- Are there token budgets for memory, session memory, and skill attachments?
- Is compact output space pre-reserved instead of waiting until the window is full?
- After compact, are work semantics restored (plans, skills, key files, tool state)?
- Is there recovery strategy when compact itself fails?

Systems with good context governance often look somewhat frugal. That frugality is usually a strength.

A.5 Error recovery checklist

Recovery design has two common failures: no design, and dead-loop design.

At minimum, check:

- Are recoverable errors routed to recovery branches before being surfaced immediately?
- Are recovery paths layered from low-destructiveness to high-destructiveness?
- Are there guards preventing reactive-compact, stop-hook, and retry loops from biting each other?
- After `max_output_tokens`, is continuation prioritized over recap?
- Do automated recoveries include counters, retry caps, and circuit breakers?
- Are interrupts treated as failure states requiring semantic closure?

A recovery system that never stops can be as dangerous as a recovery system that never starts.

A.6 Multi-agent checklist

Multi-agent design should organize uncertainty, not multiply confusion. Check:

- Does fork design preserve prompt-cache consistency through cache-safe params?
- Is mutable state isolated by default for child agents?
- Are research, implementation, verification, and synthesis roles explicitly separated?
- Does coordinator truly synthesize instead of only forwarding worker output?
- Is verification independent from implementation?
- Is agent lifecycle observable, interruptible, and reclaimable?
- Does parent abort propagate to children to prevent orphan tasks?

If a system claims multi-agent but all agents do similar work and nobody owns synthesis or verification, it is usually parallelized disorder.

A.7 Team adoption checklist

The most common rollout misread is mistaking personal proficiency for institutional maturity.

Before rollout, verify:

- Is layered `CLAUDE.md` in place, with clear guidance on what belongs and what does not?
- Is verification definition standardized before scaling skill volume?
- Are approval boundaries tiered by consequence and environment sensitivity?
- Are key institutions attached to appropriate hook timing instead of stuffed into static docs?
- Are transcript, task output, and hook events retained as replay evidence?
- Is there maintenance policy for stale memory, obsolete rules, and invalid skills?

Teams that can sustainably use agents are usually teams where ordinary members can operate correctly within institutions, not teams that depend on a few experts.

A.8 Review question set

To review an AI coding-agent proposal, ask directly:

- Which behaviors are constrained by prompt and which are enforced by runtime?
- Who blocks tool misuse, and at what layer?
- When is context compacted, and how are work semantics reconstructed afterward?
- How are prompt-too-long and max-output-tokens recovered differently?
- After interruption, how is transcript consistency maintained with tool results?
- In multi-agent flows, who owns synthesis and who owns verification?
- Does failure recovery include circuit breakers and anti-loop guards?
- How does the team audit what the agent did and why?

If answers frequently reduce to “we can add that later,” runtime is likely not truly designed yet.

A.9 Final short checklist

If everything above feels too long, keep these six: design permission before capability; rollback before autonomy; verification before delivery; context budgets before long dialogue; lifecycle before multi-agent; institutions before expecting team proficiency. Meeting them does not guarantee excellence; missing them usually means the system has just not failed yet.

A.10 Implementation seeds (pseudocode stubs)

The skeletons from earlier chapters collapsed into reusable starting points. Full forms live in their chapters; this keeps only the minimal spine.

A.10.1 queryLoop (skeleton, from Chapter 3)

```
state = { messages, toolUseContext, autoCompactTracking, turnCount, transitio
while not done(state):
   govern_input(state)            // memory / snip / collapse / autocompact
    events = stream_model(state)
    for e in events:
        if e.is(tool_use): schedule(e, state.toolUseContext)
        if e.is(api_error): return surface(e)
   if interrupted: drain_tools_with_synthetic_results(state); break
    state = advance(state, recover_if_needed(state))
assert state.turnCount monotonic ∧ every tool_use has tool_result
```

A.10.2 permission decision (skeleton, from Chapter 4)

```
decision = hasPermissionsToUseTool(tool, input, ctx)
match decision:
    allow: exec(tool, input)
    deny:  reject(reason)
   ask:   route_to(coordinator | worker | classifier | interactive)
assert decision ∈ {allow, deny, ask}                # three-
valued, never collapses
```

```
assert ask never auto-escalates to allow   # no unauthorized escalation
```

A.10.3 forkAgent (skeleton, from Chapter 7)

```
params = CacheSafeParams { systemPrompt, userContext, systemContext, toolUseC
ctx   = createSubagentContext(parent)     // mutable state isolated by default
hooks.fire(SubagentStart, { agent_id, agent_type })
defer hooks.fire(SubagentStop, { agent_transcript_path })
assert parent.abort ⇒ propagate(child.abort)
```

A.10.4 recoverFromError (skeleton, from Chapter 6)

```
on recoverable_error(e):
    if e.is(prompt_too_long):
        if stagedCollapse > 0: recoverFromOverflow()
        elif not hasAttemptedReactiveCompact: tryReactiveCompact()
        else: surface(e); skip_stop_hooks()
    if e.is(max_output_tokens):
        if cap < MAX: raise(maxOutputTokensOverride); retry()
        else: append(meta_continue_msg); retry()
assert consecutiveFailures < MAX_CONSECUTIVE_AUTOCOMPACT_FAILURES
```

Appendix B Diagrams: Drawing the Runtime Skeleton

Previous chapters explained runtime structure in text. Text is necessary, but some structures become clearer as diagrams: Claude Code is an explicit state-machine system, not "a pile of prompts plus a few tools."

B.1 Diagram 1: Claude Code global control plane

This should not be drawn as a children-book flowchart like "user -> model -> tool -> output," because that hides the actual organs. A more accurate view is five layers:

1. user interaction layer
2. control plane layer
3. execution loop layer
4. external capability layer
5. persistence and observability layer

The point is not listing every module. The point is:

- model is neither topmost nor bottommost
- model is one phase inside query loop
- control and recovery planes are what hold the system together

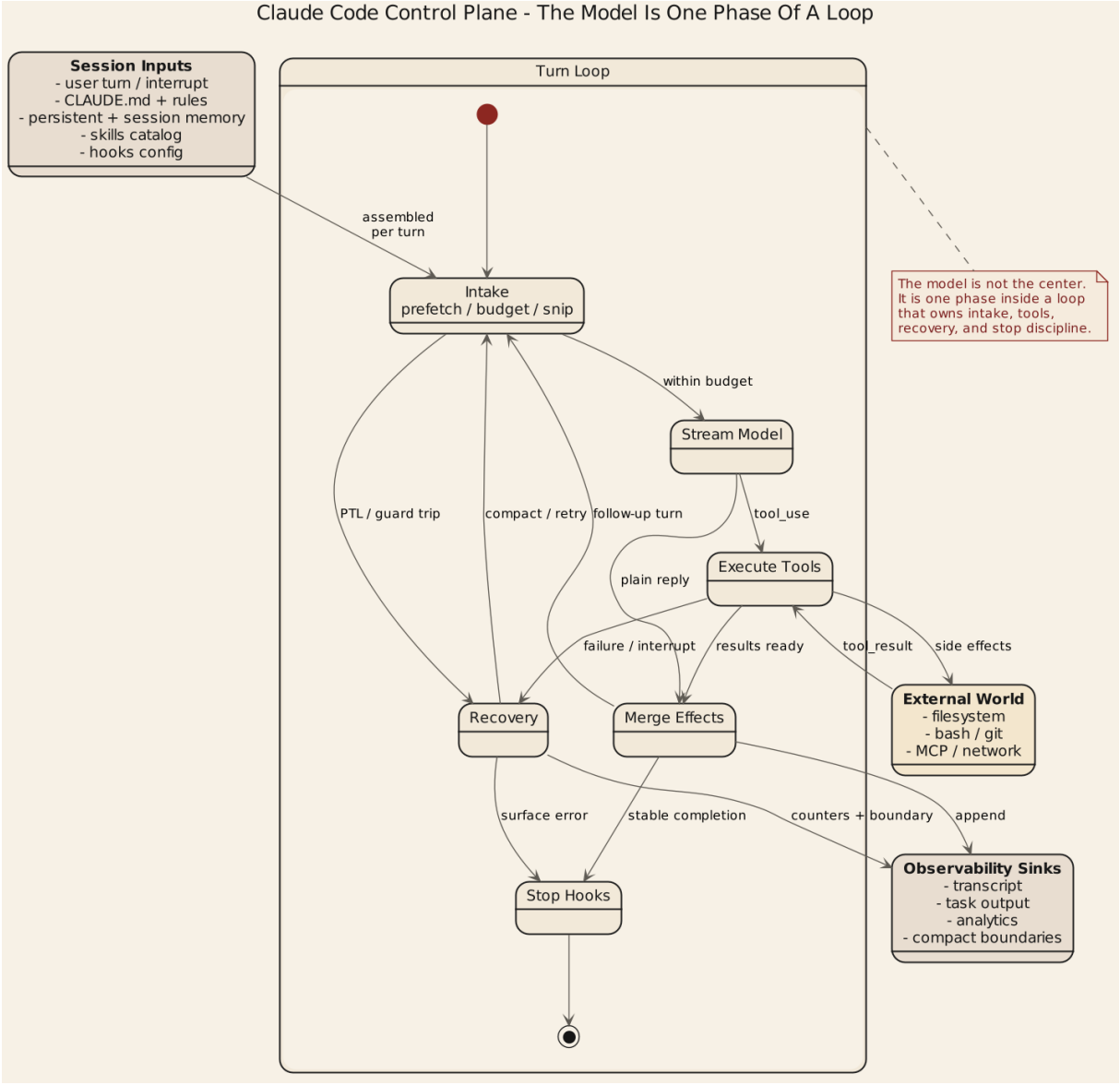


Figure 9: Claude Code global control plane

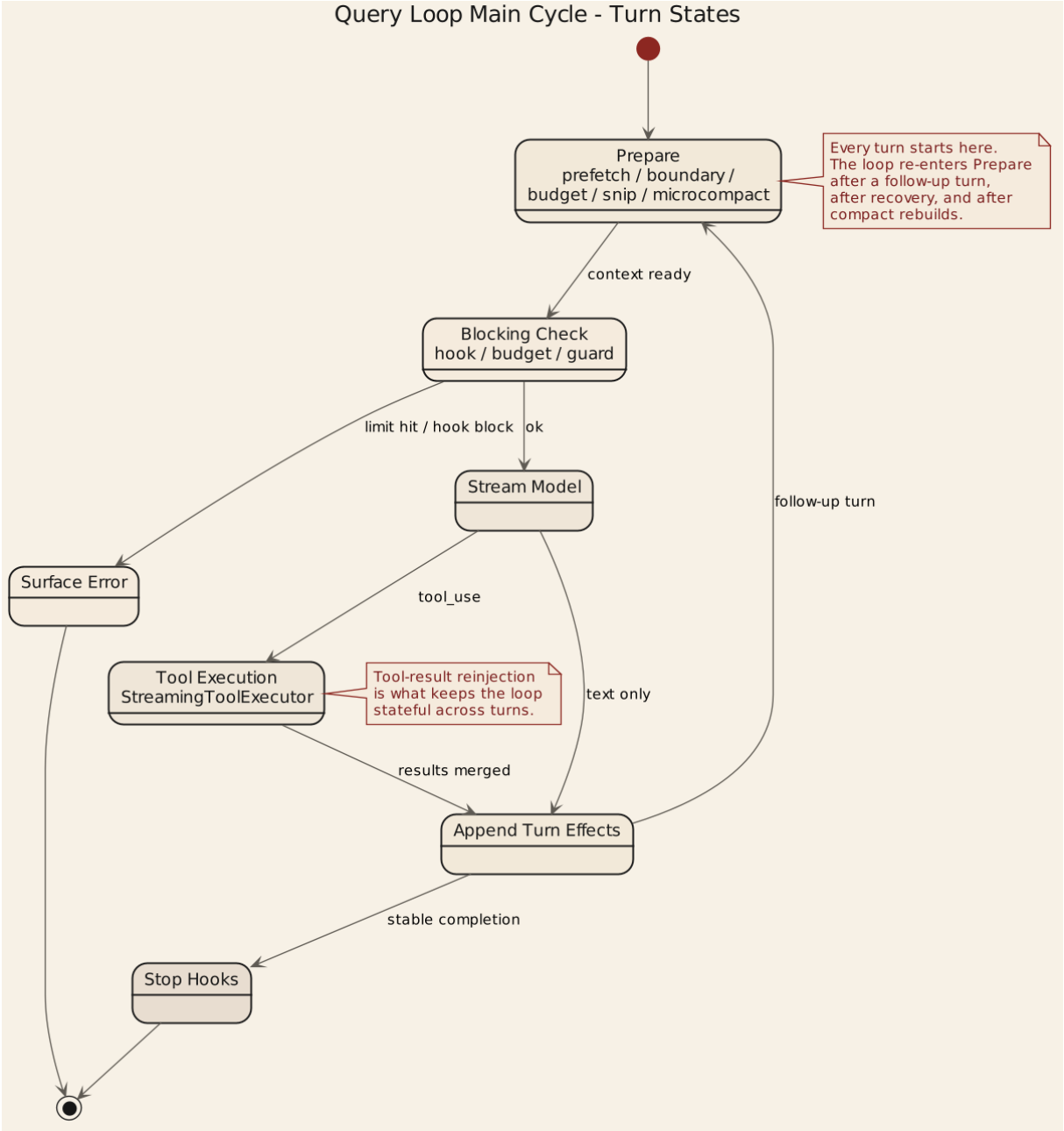


Figure 10: Query loop main cycle

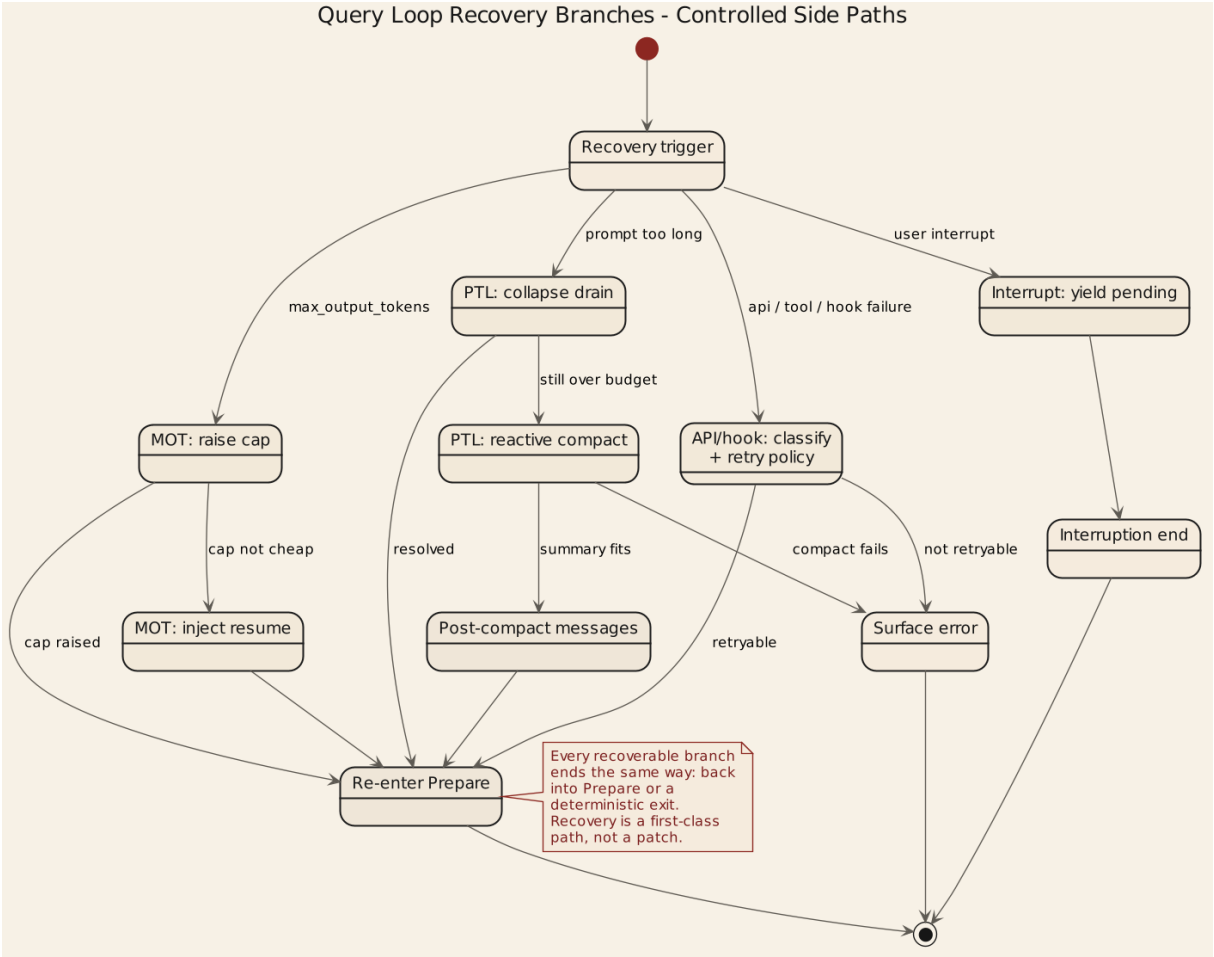


Figure 11: Query loop recovery branches

B.2 Diagram 2: Query loop main cycle and recovery branches

B.3 Diagram 3: Tool batch ordering and StreamingToolExecutor

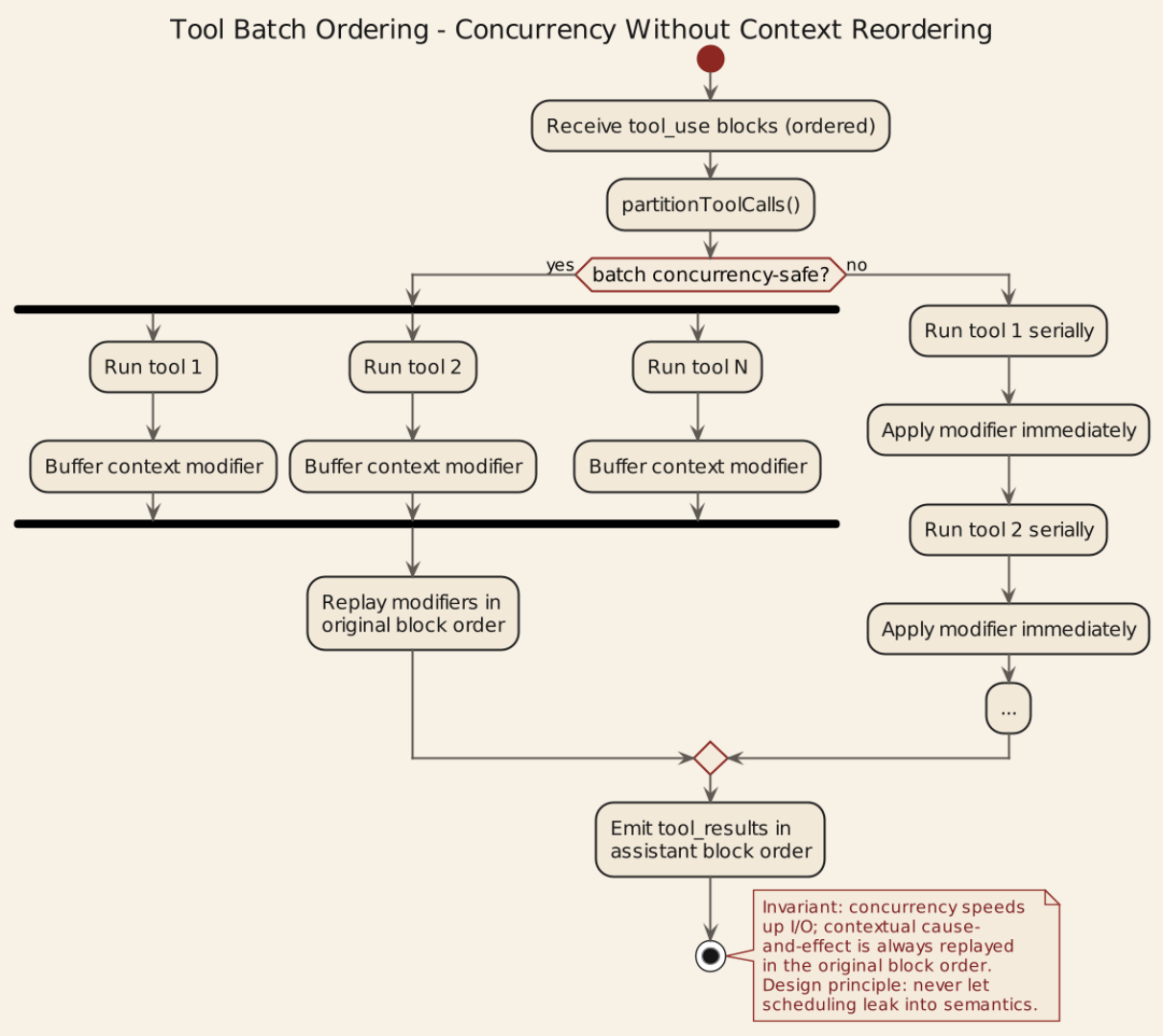


Figure 12: Tool batch ordering

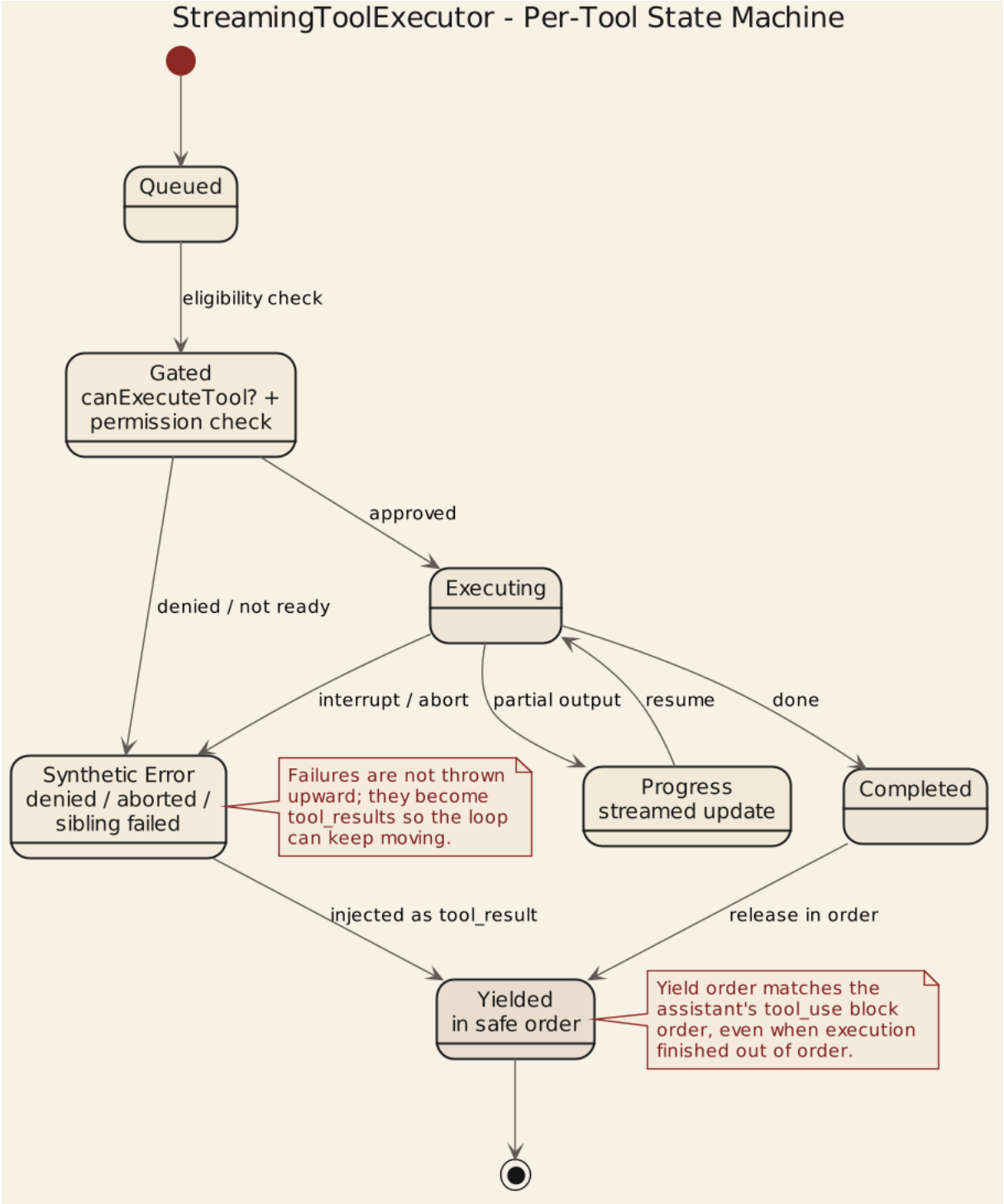


Figure 13: StreamingToolExecutor lifecycle

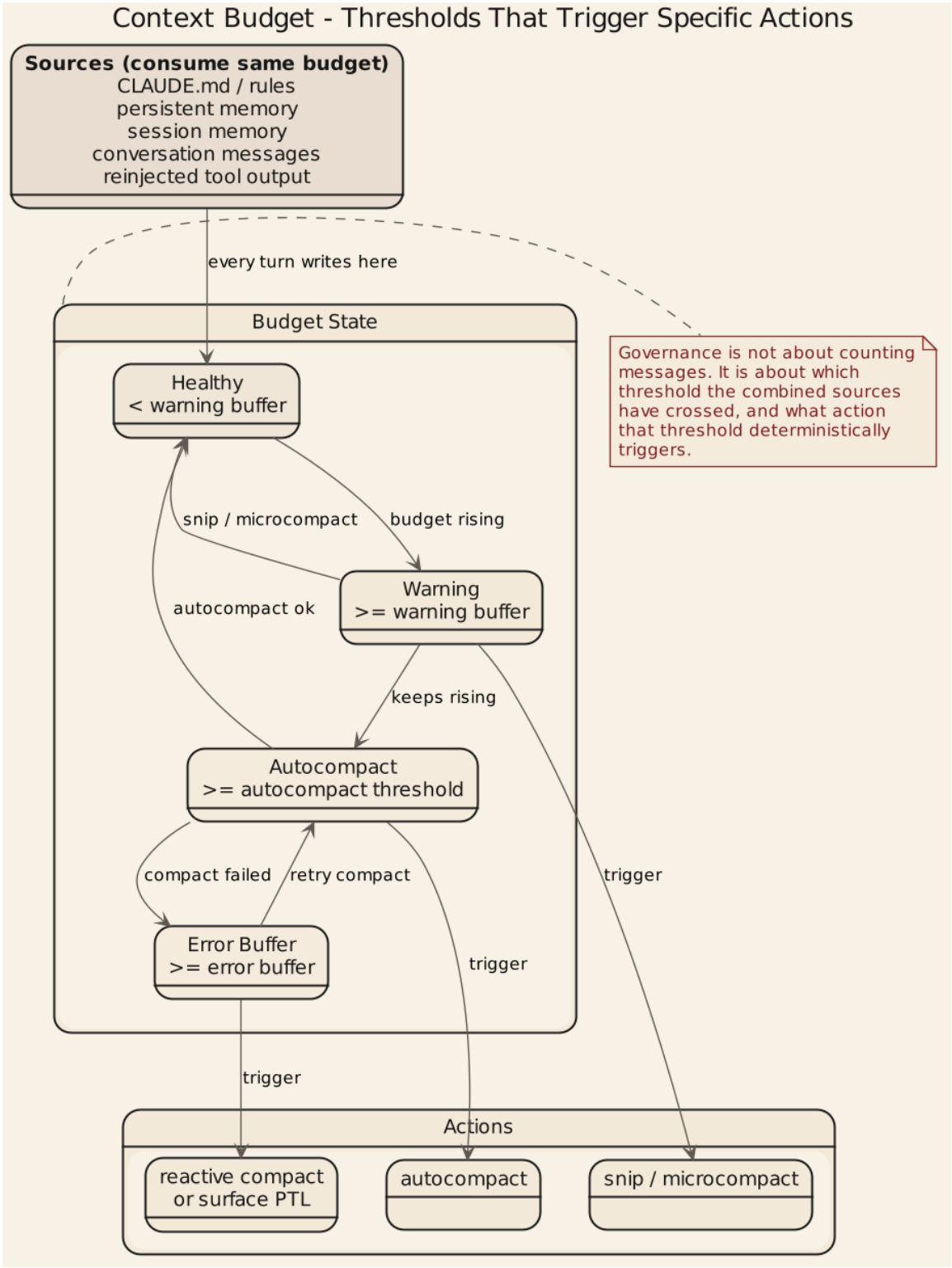


Figure 14: Context sources and budget

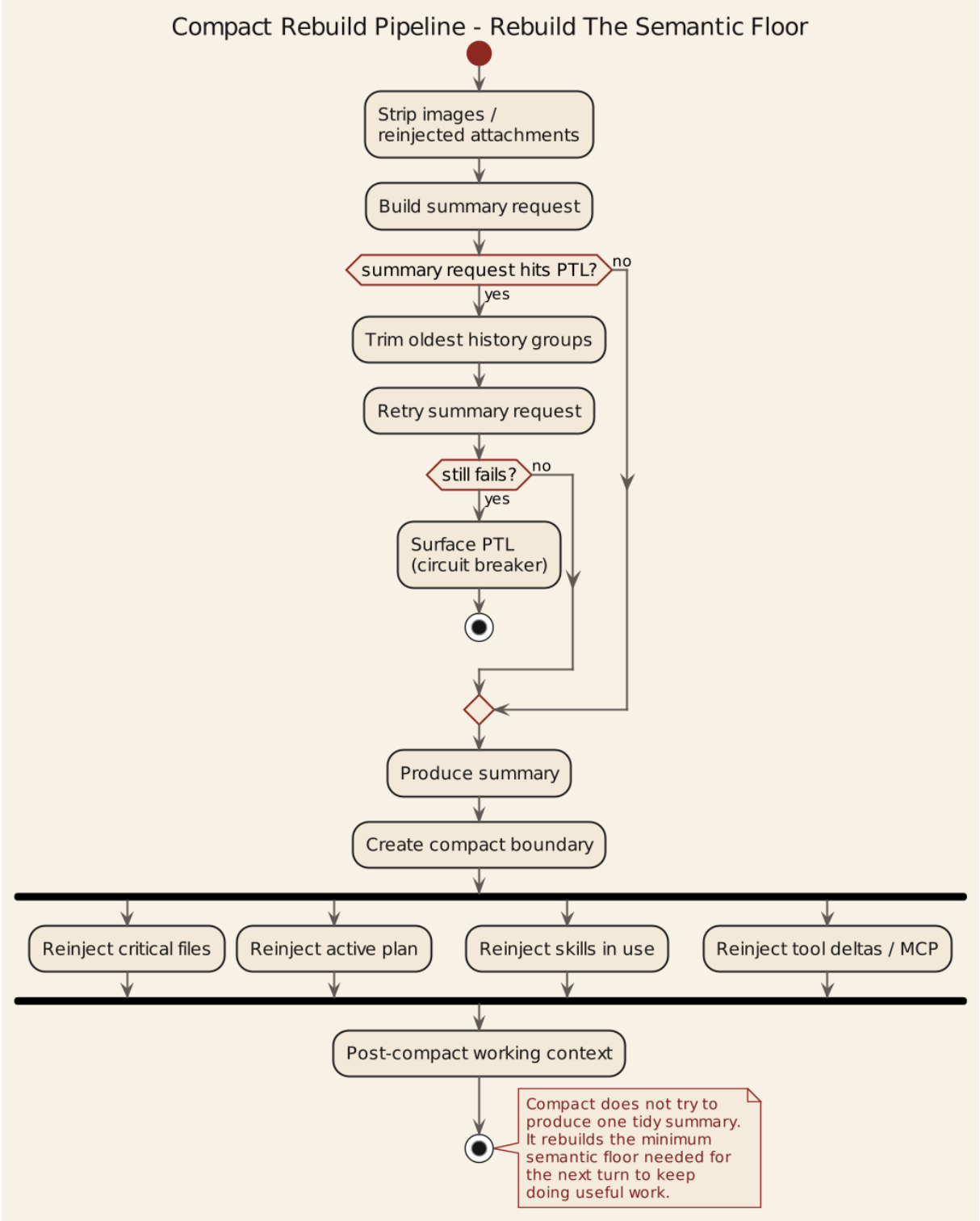


Figure 15: Compact rebuild pipeline

B.4 Diagram 4: Context sources and compact rebuild

B.5 Diagram 5: Coordinator-worker flow and verification separation

B.6 Diagram 6: Team governance map

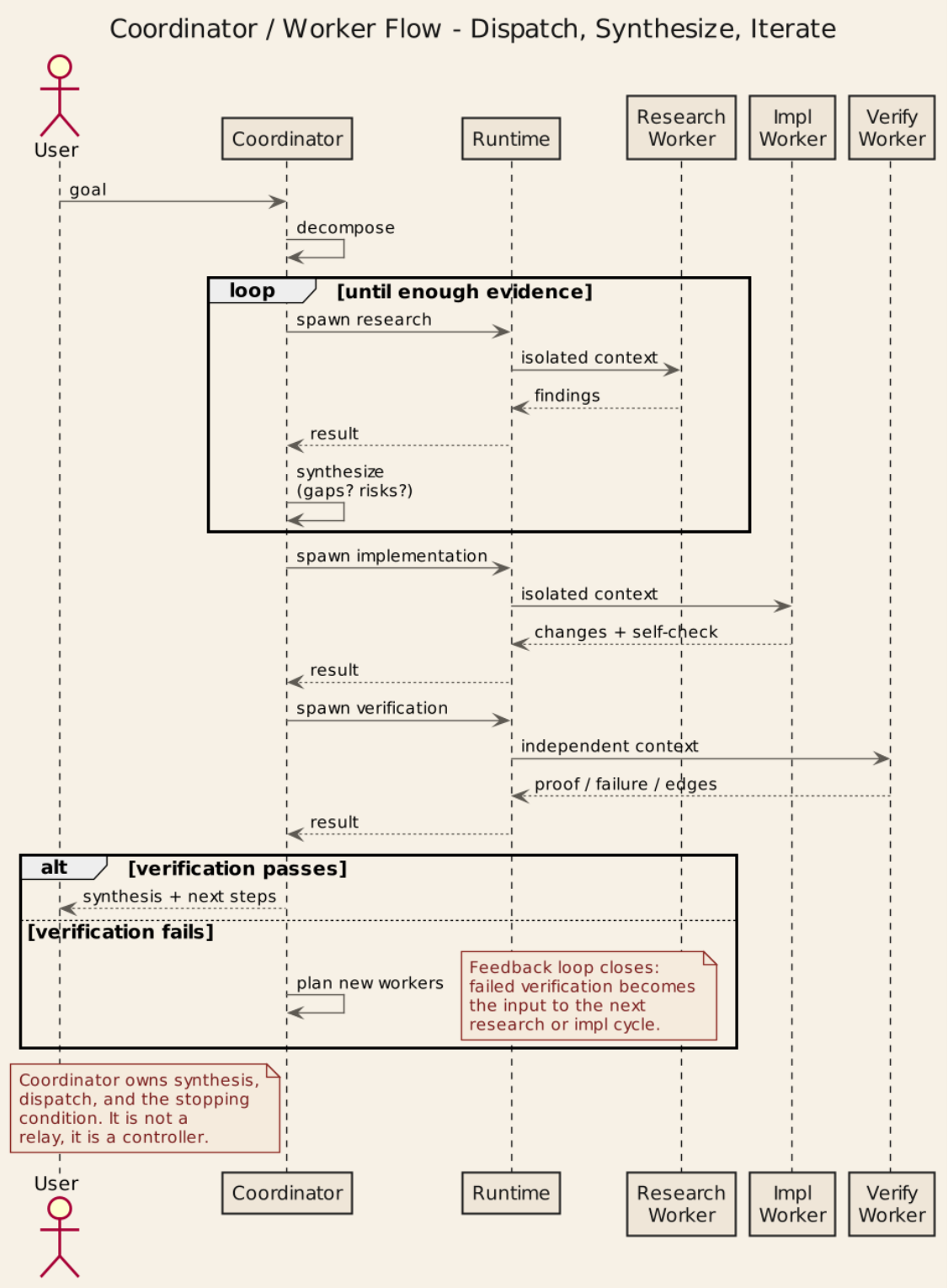


Figure 16: Coordinator and worker flow

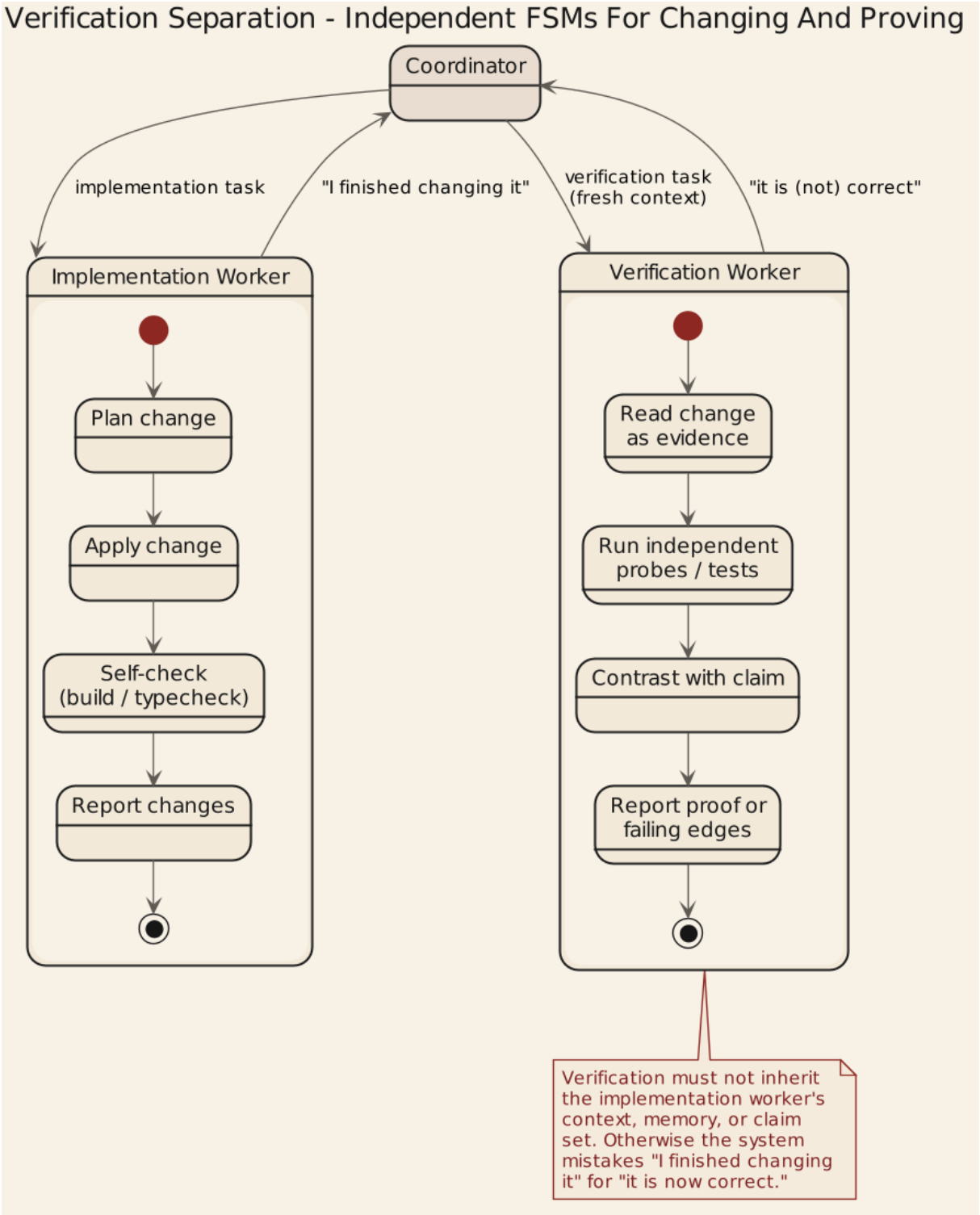


Figure 17: Verification separation

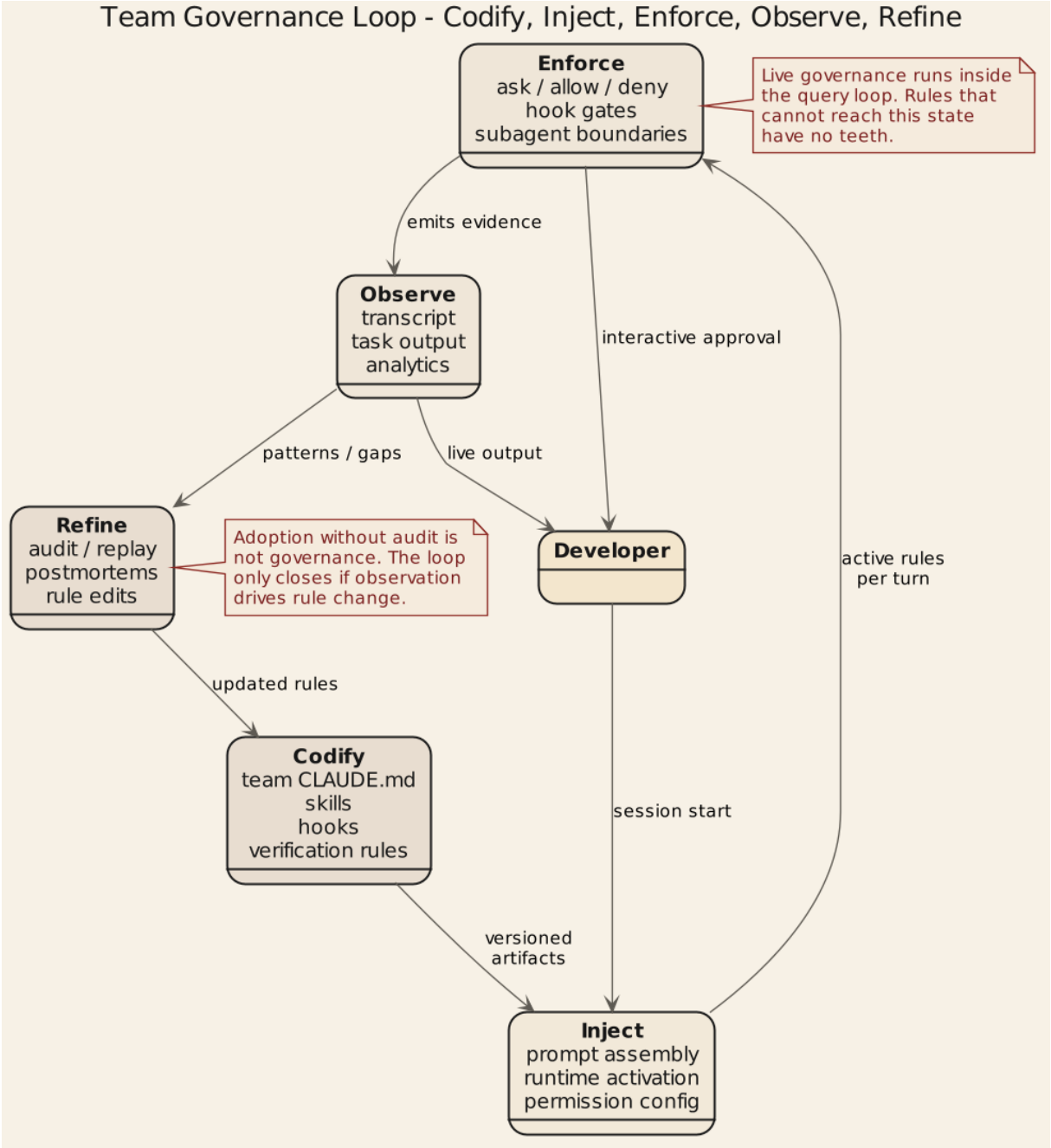


Figure 18: Team governance map

Appendix C Source Map: Which Files Ground Each Chapter

This book is not a full source walkthrough, but it is source-grounded. This appendix maps each chapter to its primary supporting files.

This is not a complete index. It lists only files directly relevant to the book's arguments.

One copyright boundary note: this map is for citing analytical basis, not for promising source-text reproduction. It preserves necessary engineering references, module positioning, and structural analysis, without shipping source copies or long implementation excerpts.

C.1 Chapter 1 Why Harness Engineering Matters

Core files:

- `src/constants/prompts.ts`
- `src/utils/systemPrompt.ts`
- `src/query.ts`
- `src/services/tools/toolOrchestration.ts`
- `src/tools/BashTool/prompt.ts`

Primary argument basis:

- prompt as control-plane component rather than personality packaging
- query loop as agent-system skeleton
- tool and Bash risk profiles as proof harness is necessary

C.2 Chapter 2 Prompt Is the Control Plane, Not Persona Decoration

Core files:

- `src/constants/prompts.ts`
- `src/utils/systemPrompt.ts`
- `src/utils/claudemd.ts`
- `src/memdir/memdir.ts`
- `src/constants/systemPromptSections.ts`
- `src/main.tsx`

Primary argument basis:

- layered system-prompt assembly
- `CLAUDE.md` and memory as control-plane inputs
- dynamic reminders and context injection

C.3 Chapter 3 Query Loop: The Heartbeat of an Agent System

Core files:

- `src/query.ts`
- `src/QueryEngine.ts`

Primary argument basis:

- state-machine properties of query loop
- input governance before model invocation
- streaming event consumption and recovery branches
- QueryEngine ownership of conversation lifecycle

C.4 Chapter 4 Tools, Permissions, and Interrupts

Core files:

- `src/services/tools/toolOrchestration.ts`
- `src/services/tools/toolExecution.ts`
- `src/services/tools/StreamingToolExecutor.ts`
- `src/hooks/useCanUseTool.tsx`
- `src/utils/permissions/PermissionResult.ts`
- `src/tools/BashTool/prompt.ts`
- `src/tools/BashTool/bashPermissions.ts`

Primary argument basis:

- concurrency safety and ordered context replay
- wrapped tool execution stack
- `allow / deny / ask` authorization semantics
- streaming-tool interruption and synthetic closure
- special high-pressure governance for Bash

C.5 Chapter 5 Context Governance: Memory, CLAUDE.md, and Compact

Core files:

- `src/utils/claudemd.ts`
- `src/memdir/memdir.ts`
- `src/services/SessionMemory/prompts.ts`
- `src/services/compact/autoCompact.ts`
- `src/services/compact/compact.ts`
- `src/query.ts`

Primary argument basis:

- layered discovery and loading of `CLAUDE.md`

- `MEMORY.md` as entry index, not content warehouse
- structured continuity in session memory
- autocompact thresholds, buffers, and circuit breaker
- post-compact reconstruction of working semantics

C.6 Chapter 6 Errors and Recovery

Core files:

- `src/query.ts`
- `src/services/compact/autoCompact.ts`
- `src/services/compact/compact.ts`
- `src/services/api/withRetry.ts`

Primary argument basis:

- withheld recoverable errors
- prompt-too-long handling via collapse drain and reactive compact
- `max_output_tokens` escalation and continuation strategy
- autocompact failure circuit breaking
- compaction self-recovery under PTL

C.7 Chapter 7 Multi-Agent and Verification

Core files:

- `src/utils/forkedAgent.ts`
- `src/coordinator/coordinatorMode.ts`
- `src/tasks/LocalAgentTask/LocalAgentTask.tsx`
- `src/utils/hooks/hooksConfigManager.ts`
- `src/skills/bundled/verify.ts`
- `src/memdir/memoryTypes.ts`

Primary argument basis:

- cache-safe fork parameters and state isolation

- coordinator synthesis responsibility
- independent verification stage
- subagent lifecycle hooks
- task cleanup and parent-child abort propagation
- stale-memory verification discipline

C.8 Chapter 8 Team Adoption

Core files:

- `src/utils/claudemd.ts`
- `src/tools/SkillTool/prompt.ts`
- `src/tools/SkillTool/SkillTool.ts`
- `src/utils/forkedAgent.ts`
- `src/utils/hooks/hooksConfigManager.ts`
- `src/main.tsx`

Primary argument basis:

- layered stability in team `CLAUDE.md`
- skills as institutional slices rather than prompt collections
- approval boundaries and allow-rule scoping
- lifecycle governance through hooks
- session startup behavior around instructions and skill loading

C.9 Chapter 9 Ten Principles

Chapter 9 is not derived from one single file. It compresses cross-chapter structures jointly reflected in:

- prompt assembly
- query loop
- tool orchestration
- permission model
- context governance

- recovery system
- multi-agent orchestration
- team governance hooks