Engineering Claude Code for Long-Lived, Multi-Session Projects

Giuseppe Destefanis
University College London | g.destefanis@ucl.ac.uk

Abstract

AI coding agents reason over a bounded context window. When a session is reset, summarised, or reaches a usage limit, any state held only in the conversation is lost, and two separate sessions cannot read what each other holds. The practices that keep a long project workable under these constraints are not specific to AI tools. They are the engineering disciplines any long-lived project already requires, namely explicit state, version control, separation of concerns, and reproducibility from recorded inputs. What the tools change is the cost of skipping them: output arrives fast enough that speed stands in for structure, and a project of any length does not survive the substitution. This document describes how these disciplines apply to multi-session work with agents such as Claude Code. The state that matters is kept in files on disk, so that any session can resume from the project folder alone. The work is divided into two roles with disjoint file ownership: a Producer that maintains source material and project memory, and a Composer that writes the deliverable, with information passing between them through a person. The document separates two arrangements that are often conflated. The first is agent orchestration, in which a single session coordinates subagents programmatically. The second is multi-terminal operation, in which independent sessions run side by side and a person mediates between them. The two address different problems and serve different purposes, and the concurrent operation of two terminals is treated as a distinct case with its own coordination and its own risks. Resumable long-running jobs, a set of supporting conventions, and a worked example complete the account.

1 Introduction

Tools such as Claude Code lower the cost of producing code and prose, since a request returns a working function or a drafted section in seconds, and that speed can stand in for structure. Over a single task this rarely matters, over a project that runs for weeks it does, because output accumulates faster than anyone can track it, and decisions taken in conversation are recorded nowhere durable. The practices in this document apply ordinary engineering principles to this setting: state is made explicit and kept under version control, concerns are separated so that independent parts do not interfere, work is reproducible from recorded inputs rather than from the memory of a session. None of these principles is specific to AI tools: what the tools change is that they make it easy to skip the principles, and a project of any length does not survive skipping them.

The constraint that makes the discipline necessary is a property of the agent itself. An AI coding agent reasons over a context window, a finite buffer that holds the recent conversation. The buffer is fast to use, but it has three properties that affect long projects: (I) it is finite, so that once it is full, older content is dropped or summarised in a step called compaction, which loses detail, (II) it is volatile, so that closing the session, reaching a usage limit, or resetting removes the working memory, (III) it is private to one session, so that a second session cannot read what the first one holds.

These properties produce four recurring failures:

- a decision explained earlier is compacted away, and the agent later contradicts it;

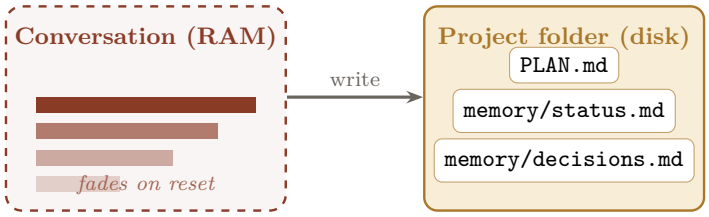

Figure 1: Important state is written from the volatile conversation to durable storage.

- each new session rediscovers the same facts, repeating work and consuming tokens;
- two sessions, or two days of work, develop different models of the same project and diverge;
- a crash during a task discards reasoning that was never written down.

One property of the project folder removes all four, because files on disk survive resets, can be read by every session, and can be opened by a new agent with no history. The practices in this document follow from a single decision: move the state that matters out of the conversation and into files, while the work proceeds rather than only at its end. Each practice that follows is an instance of a principle that predates these tools, and the text names the principle where it applies, so that the case throughout is the same one: the tools raise the cost of ignoring engineering, they do not remove the need for it.

The remainder of the document is organised as follows. Section 2 describes the file-based memory used within a single session. Section 3 distinguishes agent orchestration from multi-terminal operation. Section 4 describes the two-terminal division of work and the relay between terminals. Section 5 treats the concurrent operation of two terminals as a separate case. Section 6 covers long-running jobs, Section 7 the supporting conventions, Section 8 a worked example, and Section 9 the common ways the practices break. Appendix A gives template files and Appendix B a glossary.

2 State on disk: the memory model

A useful analogy comes from computer architecture, where the conversation behaves like RAM: fast, in context, and cleared on power loss, while the project folder behaves like disk: slower to write, explicit, and persistent. The recommended habit is the one engineers already apply to RAM and disk, which is to write important data to disk rather than holding it only in fast, volatile memory. The same habit appears in write-ahead logging, where a change is recorded durably before it is acted on, so that an interruption leaves a record from which the work continues. The memory model here applies that practice to a conversation.

2.1 Layout

The project folder holds the memory in two parts: one file is a static charter, and a small folder holds living state.

```
# Static charter, read first by every session
PLAN.md          the goal, scope, conventions,
                 and the memory protocol itself

memory/
  status.md      # done / in progress / next
  decisions.md   # dated log, with rationale
  open-questions.md  # unresolved choices
```

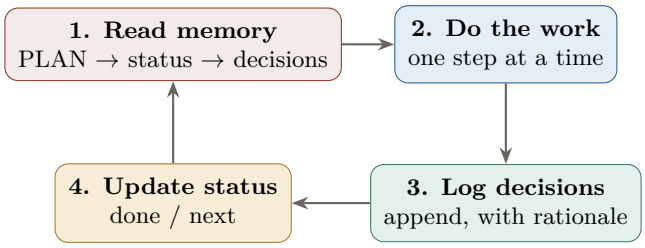

Figure 2: The session loop. Each pass ends by writing to disk.

```
    components.md  # parts and their locations
```

`PLAN.md` changes rarely and deliberately, while the files in `memory/` change constantly. The separation tells a reader where the fixed ground truth ends and the moving state begins. The decisions file is append-only and dated; rewriting its history removes the record of why earlier choices were made.

2.2 The session loop

Each working session follows the same four steps, shown in Figure 2. It reads the memory files in a fixed order, does one unit of work, records any decision with its reasoning, and updates the status before stopping. Because each pass ends by writing to disk, a reset between passes loses nothing.

Two further practices support the loop. The first concerns resets: rather than relying on automatic compaction, the operator reports to the user when the context is becoming large and lets the user decide when to reset, and because the state is on disk, the timing of a reset does not affect correctness. The second is a check, in which opening a new session, reading only `PLAN.md` and `memory/`, and attempting to resume shows whether the files are sufficient to continue without the conversation. When they are not, the files are incomplete and are updated. This cold-start check is the clearest test of whether the memory discipline is being followed.

3 Terminals and agents

There are two ways to run more than one Claude process on a project, and they are often treated as the same thing. The first is agent orchestration, in which a single session, the orchestrator, spawns subagents through an agent or task tool, gives each a part of the work, and collects their results, where the number of subagents can be one or many, and a single run can fan out to a large number of them. The second is multi-terminal operation, in which several independent sessions run side by side, each with its own context window, and a person passes information between them, where here too the number can be one or many. The two are separated by the mechanism that connects the processes rather than by their number, and that mechanism determines what each arrangement is suited to, as Figure 3 contrasts.

In orchestration, the subagents are created and discarded within a single run. Coordination is automatic: the orchestrator decides what each subagent does and combines what they return. The subagents derive from one controlling context and report back into it, so the shared state is that context together with the messages passed through it. Adding subagents scales the work that one controlling session can carry out at once. The controlling session remains a single point of failure, and its context has a bound, so the results returning from many subagents can refill the buffer.

In multi-terminal operation, each session persists across the project. There is no programmatic channel between sessions, so coordination passes through people and through the

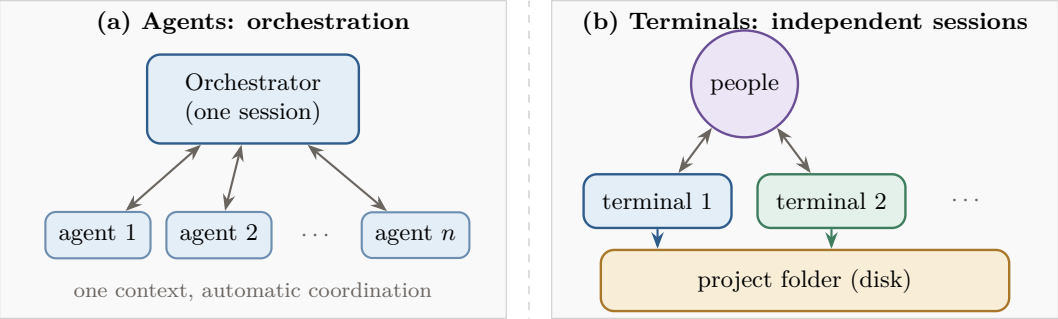

Figure 3: Two ways to run more than one process. (a) An orchestrator coordinates subagents inside one session, and a run can fan out to many. (b) Independent terminals run separate sessions whose shared substrate is the project folder, with people between them.

	Agents (orchestration)	**Terminals (sessions)**
Unit	a subagent within one run	a session across the project
Coordination	automatic, by the orchestrator	through people and the filesystem
Context	derived from one controlling context	one independent context each
Shared state	the orchestrator context and its messages	files on disk
Lifetime	ephemeral	persistent
Scaling	more subagents within a run	more workstreams at once
Purpose	parallel parts of one task	separate long-running workstreams
Recovery	handled by the orchestrator	each session recovers from disk

Table 1: Agent orchestration and multi-terminal operation compared. Both scale to many; they differ in mechanism.

shared files on disk. Each session has its own independent context, and the shared substrate is the project folder rather than any session's memory. Adding terminals scales the number of separate workstreams that proceed at once, each owning a region of the folder that the others do not write. Each session recovers on its own through the cold-start check, so no session depends on another's history.

The distinction is therefore one of mechanism: agents scale work inside one controlling context through automatic coordination, while terminals scale work across independent contexts coordinated by people and by the filesystem. Because the mechanisms differ, so do their properties, summarised in Table 1: coordination is automatic for agents and human for terminals; subagents are ephemeral while terminals persist; the shared state is a context and its messages for agents and the files on disk for terminals; and recovery is the orchestrator's responsibility for agents while each terminal recovers on its own.

Using one mechanism in place of the other leads to predictable mistakes. Expecting subagents to hold state that must survive across days places durable state in a context that will be reset. Opening separate terminals to divide the parts of a single task pays the cost of human coordination where automatic coordination would serve. The two also combine, since one terminal can orchestrate subagents for its current task, which is independent of how many terminals are running.

The rest of this document uses a configuration of two terminals as a worked instance of

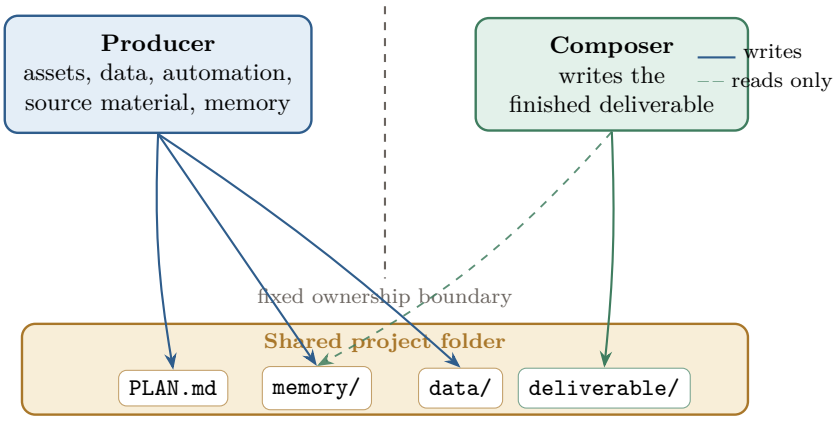

Figure 4: Disjoint ownership over one shared folder. The Composer reads memory and data and writes only the deliverable.

	Producer	**Composer**
Owns	source material, scripts, data, tests, `memory/`, version control	the deliverable folder only
Reads only	—	`memory/` and data
Does not	edit the deliverable	run jobs, edit `memory/`, run version control

Table 2: The ownership contract between the two roles.

multi-terminal operation: a Producer that prepares material and a Composer that writes the deliverable. The same reasoning applies to more terminals when a project has more independent workstreams, with one caveat. Coordination runs through a person, so the relay cost grows with the number of terminals a person must mediate. Two terminals are the case where the benefit of parallel workstreams is clear and the mediation is still light. Beyond that, the added workstreams have to be weighed against the attention the relay demands.

4 The two-terminal division of work

The two-terminal practice divides a project into two roles with disjoint file ownership. This is separation of concerns applied to the project folder, with a single-writer rule for each region. The Producer does the heavy lifting, including assets, data, automation, and source material, and maintains the project memory, while the Composer turns that raw material into the finished deliverable. The boundary between them is fixed: each role owns a region of the project folder that the other does not write. Figure 4 shows the arrangement and Table 2 states the ownership contract.

The division has four effects. Each session holds a shallow context, because the production transcript does not enter the writing session and the writing does not enter the production session. Overwrite conflicts are avoided, because two sessions never write the same file. Accidental actions are bounded, because the Composer's read-only scope means it cannot start an expensive job and the Producer cannot change the finished copy. Recovery is independent, because either session rebuilds from files and neither depends on the other's history.

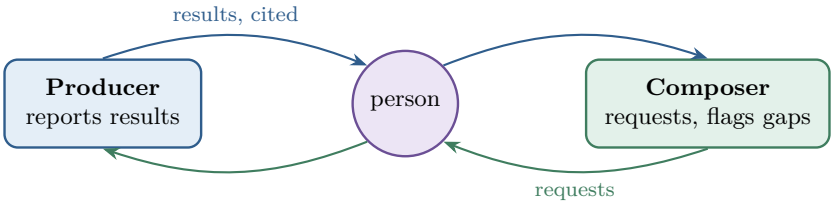

Figure 5: Information passes through a person, in one direction at a time.

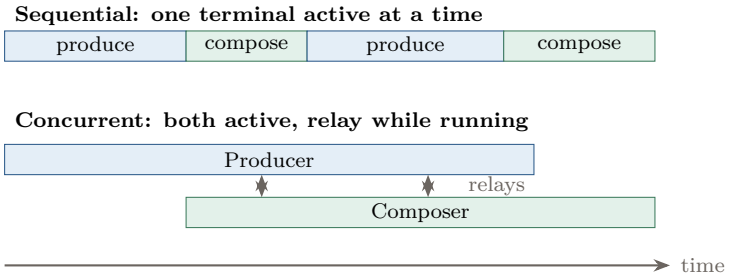

Figure 6: Sequential and concurrent use of the two terminals.

Because the two sessions share no conversation state, information passes between them through a person, in one direction at a time, as shown in Figure 5. The relay follows three conventions. Facts are reported with their source, such as a file, a row, or an asset, and the Composer records them with that source, so that a later change is traceable and does not propagate silently into the deliverable. The Composer stops rather than improvising: when it needs material that does not yet exist, it reports that it cannot proceed, and the person routes the request to the Producer, which keeps the underlying material stable. Producer-side changes follow a fixed order: make the change, regenerate the material, log it in `decisions.md`, report the difference to the person, who relays it to the Composer.

The division is a way of organising ownership and does not require both sessions to run at the same time. In its simplest use, the two roles are exercised one after another: the Producer prepares material, then the Composer writes from it. The case in which both run at the same time is treated separately in Section 5, because it adds coordination and risks that the sequential use does not have.

5 Concurrent operation of two terminals

Running the two terminals at the same time is a specific case of the division in Section 4, worth doing when the heavy work runs long enough that the Composer can write while the Producer continues, for example while a long batch is running. Figure 6 contrasts the sequential use, in which one terminal is active at a time, with the concurrent use, in which both are active and the relay operates while they run.

Concurrency does not change the ownership boundary, but it adds three considerations.

5.1 The relay becomes live

In sequential use, the relay is a hand-off at a role change. In concurrent use, it runs while both sessions work, so requests and results are exchanged repeatedly during a session. The conventions of Section 4 still apply, and they matter more: a result that is relayed without its source is harder to trace once both sessions have moved on.

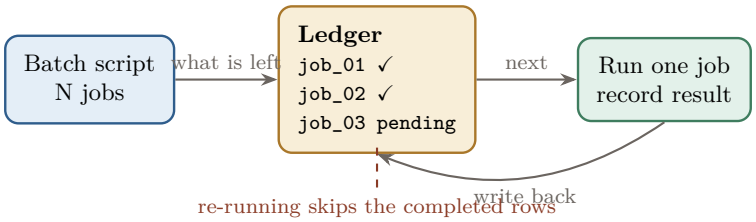

Figure 7: The ledger holds the state of the batch, so re-running after an interruption resumes where it stopped.

5.2 Shared memory needs a single writer

Context resets remain per session, so one session resetting does not affect the other, and the shared point is the project folder. The ownership boundary already prevents the two sessions from writing each other's regions, so the remaining hazard is the case where both attempt to write the same file at the same time. In this practice the Producer is the only writer of `memory/`, which removes the hazard for the files that matter most. Where a tool keeps a project-wide automatic memory that both sessions can write, concurrent writes are possible, so that store is left to one session or avoided for shared state.

5.3 Pacing shared usage

When both sessions draw on the same usage allowance, they compete for it, and long batches in the Producer can consume the allowance that the Composer also needs. Making batch progress readable from a file, as described in Section 6, lets the person see how much work remains and decide how to pace the two sessions without either guessing the other's state.

Concurrency is therefore an option rather than a requirement, used when parallel progress is available and the coordination cost is justified, while the sequential use is preferred when it is not.

6 Long-running jobs

Large automated runs are where a setup that is not pinned wastes time and the usage allowance. Four techniques keep them reliable. The run setup is pinned, so that the way jobs authenticate and execute is fixed in the launcher and every run is identical and reproducible. Each batch writes a progress ledger keyed by job identity, so that an interrupted batch resumes by re-running the same script and completed items are skipped (Figure 7). The runner detects failure cascades, such as several jobs failing within seconds, and pauses rather than filling the ledger with failed rows. Large runs are spread across usage windows, and progress is made readable from a file so that either terminal can check status without asking the other.

A related convention concerns artefacts that exist in more than one form. Where both a machine form and a human-readable form are kept, both are placed under version control, and the canonical form is declared so that the two do not drift apart.

7 Supporting conventions

Several conventions hold the structure together. Work proceeds one component at a time, and a component is finished and checked before the next begins. The folder is built for hand-off from the start: dependencies are pinned, assets and scripts are under version control, and the structure allows another person to continue from it. Choices about scope, sample size, tooling, and parallelism are argued on their merits. Framing a choice around a deadline tends to favour shortcuts and adds little to the reasoning. The folder is the source of truth, and

nothing important is kept only in a conversation.

8 A worked example

Setting

A small team must produce a data-backed report. The work has three parts: gather a dataset, run an analysis over many inputs, and write the report.

The project begins with one terminal, since there is nothing to compose yet and a single terminal is enough. On day zero, `PLAN.md` is written with the goal, the ownership boundary, and the memory protocol, and `memory/` is created with its four files. Over the next three days the Producer builds the analysis pipeline one component at a time and logs each design choice in `decisions.md`, for example the choice of CSV over JSON for the ledger because it is easier to inspect and to compare, and a long batch starts and writes a ledger, so an overnight interruption costs nothing.

On day four there is stable material to write about, so a second terminal opens as the Composer with a scope charter: it owns `report/`, may read `memory/` and `data/`, and runs nothing, and the two terminals now run concurrently. The Composer drafts a section and then stops, because it needs a figure that does not exist, and reports the gap, which the person relays to the Producer, which generates the figure, logs the change, and reports back the source and the value, for example `data/fig-03.csv`, mean 4.1, with twenty observations. The Composer records that source in the draft, so that when the Producer later refines the number, the recorded source shows the one place to update.

On day eight the Producer's session reaches a usage limit during a task, and a new session reads `PLAN.md` and `memory/` and resumes within a few minutes, while the Composer is unaffected.

Outcome

No work was lost at the interruption. There was no overwrite conflict between the writing and the analysis. Every figure in the report was traceable to a file, and a new team member could continue from the folder.

9 Anti-patterns

Several failures recur, often while the team believes it is following the practices. Decisions are discussed but never written to `decisions.md`, so the folder looks tidy while the reasoning is lost at the next reset. The Composer edits a data file once, as an exception, after which two sessions write the same file and diverge. Two terminals run from day zero, before there is anything to compose, which only adds overhead. A status file is updated but unspecific, such as a note that progress was made, and so fails the cold-start check. The operator lets the tool summarise the context instead of choosing when to reset. A value is copied into the deliverable without a source, so that when it changes the dependent text is hard to find. In each case the correction is the practice the failure omits: record decisions when they are made, keep the ownership boundary without exceptions, split only when parallel work exists, record the next concrete action in the status file, reset deliberately, and record the source with every value.

10 Conclusion

The practices in this document are not new, and that is the point. Explicit state, version control, separation of concerns, and reproducibility from recorded inputs are the disciplines any long-lived project already requires, and they follow here from one constraint and one

property: the constraint is that an AI agent's working memory is bounded and volatile, while the property is that the project folder is not. Keeping the state that matters on disk lets any session resume from files, and dividing the work into two roles with disjoint ownership keeps two long workstreams from mixing. Multi-terminal operation is distinct from multi-agent orchestration: the first separates persistent workstreams coordinated by a person, while the second divides one task among subagents coordinated automatically, and the two are used for different purposes. Running two terminals concurrently is an option within the division, used when parallel progress is available and its coordination cost is justified. The tools lower the cost of producing work, but they do not lower the cost of keeping a long project coherent, and that cost is paid in engineering.

A Template files

PLAN.md

```
# Project plan: <project name>

## Goal
One paragraph stating what "done" looks like.

## Scope and non-goals
What is in scope. What is out of scope.

## Terminal ownership
- Producer owns: agent code, scripts/, data/,
  tests/, memory/, version control.
- Composer owns: <deliverable>/ only.
  Reads memory/ and data/. Runs nothing.

## Memory protocol
At session start, read PLAN.md, then
memory/status.md, then memory/decisions.md.
During the work, log decisions in
memory/decisions.md, dated, with rationale.
At session end, update memory/status.md.
Report context size; reset deliberately.

## Conventions
Naming, the canonical-form rule, and the
citation rule: every value carries its source.
```

memory/status.md

```
# Status, updated <date>

## Done
- ...

## In progress
- ...  (file: ...,  next action: ...)

## Next
- ...
```

memory/decisions.md

```
# Decisions (append-only)

## <date>: <short title>
Decision: ...
Rationale: ...
Alternatives considered: ...
```

B Glossary

Context window The finite buffer of recent conversation an AI agent reasons over.

Compaction Dropping or summarising older context when the buffer fills. It loses detail.

Reset Restarting a session, which clears the working memory in the context window.

Orchestration One session coordinating subagents that it spawns and discards.

Producer The terminal that maintains source material and project memory.

Composer The terminal that writes only the deliverable, reading memory and data as input.

Charter A short statement of a terminal's scope, given at session start.

Relay The one-way transfer of information between terminals through a person.

Ledger A file recording which jobs in a batch have completed, used to resume the batch.

Cold-start check Resuming a project in a new session from `PLAN.md` and `memory/` alone.

Canonical form The authoritative version of an artefact when two forms exist.