GIT IN THE AGE OF AI AGENTS

Free sample — Foreword and Chapters 1–2

Victor PEREZ

FREE SAMPLE

Multi-agent engineering · Git · Integration · Recovery

Foreword

Code can now be produced faster than a human team can understand, review, and integrate it.

This book is for engineers, tech leads, staff engineers, and managers who use agents and subagents on the same application. It does not attempt to teach Git from scratch. It provides the operational methods teams need when code production changes scale.

The practices are vendor-independent. They can be applied with Codex, GitHub Copilot, other agents, or an internal platform. Products evolve; the invariants remain: an identifiable baseline, an isolated workspace, an explicit scope, reproducible evidence, controlled integration, and human accountability.

> **HOW TO USE THIS BOOK**
>
> **Every chapter leads to an actionable decision**
>
> You will find YAML contracts, policies, prompts, commands, checklists, and dashboard templates. Adapt the thresholds to your team, but preserve the control function they provide.

Table of Contents

PART I

WHAT AGENTS CHANGE ABOUT USING GIT

CHAPTER 1 — WHEN A DEVELOPER BECOMES RESPONSIBLE FOR MULTIPLE CODE PRODUCERS

The change of scale that Git cannot absorb on its own

The concrete problem

A developer traditionally works in a fairly easy-to-represent loop: they inspect the project, modify a few files, run tests, and review their diff before creating a commit. Even when they collaborate with a team, they usually know the intent behind the changes in their own working tree.

With AI agents, this loop changes scale. The same developer can assign a bug fix to one agent, a migration to another and a performance analysis to a third. Each agent may delegate part of its work to sub-agents. Within minutes, several producers write code, add tests, move files or propose commits.

The developer is no longer just the author of changes. They have become responsible for a small production system.

> **MENTAL MODEL**
>
> **From the workbench to the workshop**
>
> With a single developer, Git resembles a workbench: the tools and parts remain visible to the person doing the work. With several agents, Git resembles a workshop with multiple stations. Each station can produce a correct part, but someone has to distribute the plans, prevent two workers from using the same machine, inspect the parts, and decide on the assembly order.

Git remains indispensable in this workshop. It keeps the versions, compares the states and connects the commits together. But Git does not distribute the missions and does not know the team's intention.

What actually changes

The most visible change is the volume. An agent can rapidly produce a large diff. The most dangerous change is less dramatic: several results can be correct separately and incompatible together.

Let us imagine four missions launched since the same commit:

- Agent A renames a service and adapts its tests;
- Agent B adds a call to the old name in a new job;
- Agent C modifies the structure of a table used by both services;
- Agent D updates the documentation and a configuration file.

Git may detect a conflict if two missions change the same lines. It will not necessarily signal that Agent B's job now calls an interface deleted by Agent A. Nor will it know whether the migration of Agent C should be deployed before or after the code.

So the problem is not just: "Do several agents change the same file? The question becomes: "Do several missions change the same system, the same rule or the same order of execution? "

The three capacities not to be confused

To lead several code producers, the team must separate three capacities.

Producing means writing code, tests, migrations, or documentation.

Integrating means constructing a combined state, examining this result and deciding that it can join the shared branch.

Validating means providing evidence that this combined result satisfies the requirements and constraints and the actual state of the application.

An agent can be very effective at producing. This effectiveness does not automatically give it the authority to integrate or the ability to validate the whole system alone.

> **KEY TAKEAWAY**
>
> **Code throughput is not delivery throughput**
>
> A team can complete 20 missions and correctly integrate only three changes. The backlog of pending branches, diffs and decisions is then larger than the number of tasks announced as completed.

The new role of the responsible developer

The developer supervising the agents performs at least seven functions:

1. they define the starting point for each mission;
2. they assign a file and functional scope;
3. they limit delegation to sub-agents;
4. they preserve the provenance of the results;
5. they review diffs and evidence;
6. they organize the integration order;
7. they decide whether to accept, revise, or abandon a result.

This responsibility does not mean they must reread each character manually. It means they must design a process in which a significant difference, an extension of scope or absent evidence becomes visible before integration.

A Tansa scenario, without having to know Tansa

Tansa is organized into successive modules. Layer1 acquires Bitcoin facts. Cluster then builds prudent groupings. Other modules produce profiles, behaviors and categories. Each module depends on a certified state of the previous module.

Suppose that one agent optimizes Layer1 and another modifies Cluster in parallel. They can work in completely different files. Git can merge their branches without textual conflict. However, the second change may be based on a structure or guarantee that the first agent has just changed.

The lesson is not specific to Bitcoin:

Two missions may be in conflict when they affect the same responsibility or contract, even if they do not share any lines of code.

In an online shop, the same problem may arise between price calculation and billing. In a medical application, it may appear between data import and a rule engine. In a logistics hub, it can link stock and order preparation.

The first method: one brief per task

Before launching an agent, the developer creates a task brief. This is not a long specification. It is a minimum boundary that allows the work to be allocated and later to understand what has been produced.

```
mission_id: TANSA-L1-042
owner: victor
agent: agent-layer1
base_commit: 4a1b2c3d4e5f...
branch: agent/layer1-output-audit
worktree: ../worktrees/layer1-output-audit
objective: "Add an inspection of the number of outputs processed"
allowed_paths:
  - app/services/layer1/
  - test/services/layer1/
forbidden_paths:
  - db/migrate/
  - app/services/cluster/
impact_surfaces:
  - layer1_certification_contract
subagents:
  allowed: true
  maximum: 1
tests_required:
  - test/services/layer1/output_audit_test.rb
stop_if:
  - "migration becomes necessary"
  - "the contract read by Cluster must change"
delivery:
  commit_required: false
  report_required: true
```

Values are examples. The important point is the relationship between the fields.

The `base_commit` identifies the state from which the agent reasons. Authorized paths limit its writing space. Impact surfaces describe the responsibilities that can be affected beyond the files. The conditions for stopping prevent a silent extension of the mission.

Before the first mission: observe without change

The developer must first know which directory and state they are working in. The following commands are all presented separately because they answer different questions.

Command 1 — In which folder am I?

Why use it: a terminal can stay open in an old clone or in the worktree of another mission. This command displays the current folder.

Exact effect: `pwd` is an operating-system command, not a Git command. It reads the current path and does not change anything.

Expected output: a path, for example `/projets/tansa`.

Limit: This path does not yet prove that this is the root of the right repository.

Risk: R0 - observation.

```
pwd
```

Example:

```
/home/victor/worktrees/layer1-output-audit
```

This output indicates the current directory. If the expected path was `/home/victor/tansa`, stop and understand why the terminal is in another worktree.

Command 2 — What repository contains this folder?

Why use it: a path resembling the project can be an independent copy or a subdirectory. This command asks Git the root of the working tree to which the terminal belongs.

Exact effect: `git rev-parse --show-toplevel` queries the repository structure and displays the path of its root. It does not change files or references.

Option explained: `--show-toplevel` asks for the top-level directory of the current worktree.

Expected output: an absolute path. An error like `not a git repository` means that Git does not recognize any repository from this location.

Risk: R0 - observation.

```
git rev-parse --show-toplevel
```

Example:

```
/home/victor/worktrees/layer1-output-audit
```

In a linked worktree, this root is the worktree directory, not necessarily the directory where the repository was originally cloned.

Command 3 — What branch does this folder display?

Why use it: the branch name provides a useful human reference for assigning the workstream.

Exact effect: `git branch --show-current` reads the branch associated with `HEAD`. It does not create, move or remove any branch.

Option explained: `--show-current` only asks for the short name of the current branch.

Expected output: a name like `agent/layer1-output-audit`. An empty output may indicate that `HEAD` is detached: the folder then displays a commit directly without being positioned on a local branch.

Limit: the name of a branch is mobile. It does not replace the hash of the commit.

Risk: R0 - observation.

```
git branch --show-current
```

Command 4 — What exact commit is displayed?

Why use it: two people can talk about the same branch at different times. The hash identifies the actually observed commit object.

Exact effect: `git rev-parse --verify HEAD` Resolves `HEAD` and verifies that this reference exists. It doesn't move `HEAD`.

Options explained: `--verify` requires a valid reference; `HEAD` denotes the commit currently displayed in this worktree.

Expected output: a complete hash. An error means that the reference is not resolved, for example in a repository that still has no commit.

Risk: R0 - observation.

```
git rev-parse --verify HEAD
```

Example:

```
4a1b2c3d4e5f678901234567890abcdef1234567
```

This hash can be copied into the task brief. It does not change when the branch advances; it continues to designate the same commit as long as the object remains available.

Command 5 — What local changes are already present?

Why use it: throwing an agent into an already modified file mixes responsibilities. Differences present in advance of the mission need to be identified.

Exact effect: `git status --short --branch` compares the commit `HEAD`, the index and the working tree. According to the official documentation, `git status` also shows untracked files that are not ignored.

Options explained: `--short` produces a compact output; `--branch` adds the information on the branch.

Expected output: a first branch line, then possibly file lines. The two status columns represent the index and the working tree, respectively.

Important limit: A state without a modified file only means that no tracked or untracked visible differences are reported. This does not prove that the right commit, the right database or the right application is being used. In addition, Git may update certain information cached from the index during `status` in highly concurrent automation, the documentation recommends examining the use of `--no-optional-locks`.

Risk: R0 - functional observation.

```
git status --short --branch
```

Example:

```
## agent/layer1-output-audit
 M app/services/layer1/output_audit.rb
?? notes/measurement.txt
```

`M` indicates here a working-tree change that is not staged in the index. `??` denotes an untracked file. Before assigning the file to an agent, the team must identify the owner of these two elements.

Command 6 — What other worktrees use this repository?

Why use it: multiple agents can already work in files related to the same repository. This command produces the inventory.

Exact effect: `git worktree list --porcelain` reads the metadata of the worktrees. It does not create or remove any worktree.

Option explained: `--porcelain` requires a stable format, designed to be analyzed by a tool. Each record contains, in particular, the path, the hash of `HEAD` and the associated branch.

Expected output: several blocks separated by an empty line. `detached` indicates a `HEAD` detached; `locked` a locked worktree; `prunable` an entry whose associated path no longer seems to exist.

Limit: the list says where the known worktrees are located. It does not attribute their contents to a human or agent and does not explain their mission.

Risk: R0 - observation.

```
git worktree list --porcelain
```

Simplified example:

```
worktree /home/victor/tansa
HEAD 91aa22bb33cc44dd55ee66ff7788990011223344
branch refs/heads/main

worktree /home/victor/worktrees/layer1-output-audit
HEAD 4a1b2c3d4e5f678901234567890abcdef1234567
branch refs/heads/agent/layer1-output-audit
```

This snapshot establishes two distinct states. It does not yet say whether the task branch points to the expected commit; this comparison will be dealt with in a later chapter.

Initial observation prompt

The following prompt authorizes only the observation commands explained above. It does not ask the agent to “prepare” or “clean” the repository.

```
You must establish the starting point of this mission without changing
the repository.

Authorized commands, and only these:
- pwd: display the current folder;
- git rev-parse --show-toplevel: identify the root of the worktree;
- git branch --show-current: display the current branch;
- git rev-parse --verify HEAD: get the full hash of the displayed
  commit;
- git status --short --branch: inventory visible local differences;
- git worktree list --porcelain: inventorying related worktrees.

Do not execute any other commands. Do not change any file, index,
branch, or reference. Do not create a commit or stash. Do not switch
branches.

Report separately:
1. the facts observed;
2. anomalies in relation to the mission;
3. the missing information;
4. the reasons for not starting.
```

The prompt does not magically make the controls safe. It makes the authorization comprehensible and verifiable. The execution environment and permissions must, where possible, apply the same limits.

Common mistakes

Launch all agents since `main`. The name `main` does not guarantee that all start at the same hash. The branch can move between two launches.

Use the same working tree for several agents. The agents then share the same index and the same files. It becomes difficult to attribute a difference, and a branch change command can disrupt everyone.

Confuse a completed task with an integrated result. An agent may have satisfied his or her local objective while being based on an outdated base or by changing a reserved responsibility.

Allow sub-agents without limit. The delegation could dilute the scope and make the provenance difficult to rebuild.

Request a backup commit. A commit assigns and writes a state. It must not be used as an automatic reaction to a situation that no one yet understands.

Checklist before launching several agents

- Each mission has an identifier and a human manager.
- The starting commit is recorded by its full hash.
- [- The working tree is identified.

- [. Pre-existing differences have an owner.
- The permitted and prohibited paths are written.
- [- The affected functional surfaces are reported.
- [The number of sub-agents authorized is defined.
- [The tests and the expected report are specified.
- The conditions for stopping are explicit.
- The path of integration is known before the production of the code.

Team deliverable: register of active missions

```
repository: application
reference_branch: main
observed_reference_commit: FULL_HASH
missions:
  - id: MISSION-001
    owner: HUMAN_OWNER
    agent: AGENT_ID
    worktree: ABSOLUTE_PATH
    branch: agent/mission-001
    base_commit: FULL_HASH
    allowed_paths: []
    impact_surfaces: []
    subagents:
      maximum: 0
    status: ready
    integration_order: undecided
```

This register does not replace Git. It provides Git with the organizational information that Git does not have: mission, owner, limits and expected decision.

Success criteria

The system is ready for multi-agent work when each future result can be linked to a mission, manager, worktree, branch and starting commit. If the team discovers a modified file without an owner, an agent with no precise base or an unknown sub-agent, it does not increase the parallelism: it first restores traceability.

CHAPTER 2 — WHY GIT DOES NOT AUTOMATICALLY COORDINATE AGENTS

Git writes states; team organizes work

The initial misunderstanding

Git is often presented as a tool for collaboration. This formula is true, but it can lead to excessive expectations. Git allows multiple people to store, compare and combine versions. It does not know why those people are working, what missions depended on each other or what results should be deployed first.

Let's replace people with agents and add sub-agents: the limit becomes immediately visible.

Git can answer the question: "Which files differ between these two commits?" It cannot answer this question on its own, "Do these two changes still follow the same business contract?" ”

> **MENTAL MODEL**
>
> **Git is the register, not the workstream manager**
>
> A register may indicate which documents have been received, their version and origin. It does not decide which teams should manufacture them, in what order they must assemble them or whether the building obtained complies with the plan. These decisions belong to the coordination system built around the register.

What Git really knows

Git mainly represents objects and references.

A blob keeps the content of a file. A tree describes a directory and connects names to objects. A commit connects a tree structure to one or more parents and adds metadata. A reference, like a branch, contains a name that designates an object, usually a commit.

In a worktree, Git also connects three key states:

1. the commit designated by `HEAD`;
2. the index, which prepares the content of the next commit;
3. the working tree, which the tools and agents may modify.

Through these structures, Git can comput differences, follow a story, and build mergers.

What Git doesn't know

Git does not, of course, contain:

- the objective produced by the mission;
- the human manager of the agent;
- the number of sub-agents authorized;
- components reserved by another mission;
- the order required between migration and deployment;
- the significance of a business test;
- the state of the database;
- the configuration actually loaded in production;
- the commit actually deployed, unless the organisation registers it;
- the decision to accept or abandon a result.

This lack is not a defect from Git. It defines the boundary of the tool.

Four different conflicts

When two agents work in parallel, the word “conflict” may refer to at least four realities.

1. the textual conflict

Two changes affect lines that Git cannot automatically combine. This is the most visible conflict. Markers appear and a human or assisted decision is required.

2. Semantic conflict

Files combine, but intentions are incompatible. One agent renames one method while another adds a new call to the old contract in another file. Git can merge texts and produce incorrect software.

3. temporal conflict

The mission was correct when it started, but its base became too old. Another development has changed a necessary hypothesis. The absence of a textual conflict does not automatically render the current result.

4. the operational conflict

The code and history are consistent, but the actual execution order is bad: destructive migration too early, old instance still active, variable absent or unprepared data.

> **KEY TAKEAWAY**
>
> **a successful merger validates only a Git operation**
>
> It does not prove the business compatibility, the validity of the data or the success of a deployment.

Example Tansa: different files, the same chain of trust

In Tansa, Layer1 provides certified facts to Cluster. Cluster must not work beyond the reliable checkpoint of Layer1. Susposes two missions:

- Agent A modifies the way in which Layer1 certifies a height;
- Agent B optimises the selection of approved blocks in Cluster.

The paths may be distinct. However, missions affect the same area: the certification contract between the modules.

The solution is not to teach Git what a Bitcoin checkpoint is. It consists in declaring this area in mission contracts:

```
mission_a:
  allowed_paths:
    - app/services/layer1/
  impact_surfaces:
    - certified_checkpoint_contract

mission_b:
  allowed_paths:
    - app/services/cluster/
  impact_surfaces:
    - certified_checkpoint_contract
```

An orchestration tool or a human journal can then detect logical overlap before fusion.

In another application, `certified_checkpoint_contract` could become `payment_state_contract`, `patient_record_version` or `inventory_reservation_rule`. The method remains the same.

The four-card method

To complete what Git does not know, the team maintains four light maps.

The mission map links each agent to its purpose, base and manager.

The map of spaces connects each mission to a worktree, branch and paths allowed.

The dependency map indicates the shared surfaces and the order between the missions.

The integration map shall indicate which result should be combined, tested and approved.

These maps can live in a YAML file, an internal database or a tracking tool. The most important thing is that they are observable and connected to the Git hashes.

Observe Git relations without changing them

The commands in this chapter are only used to understand existing states. They do not create a branch and do not merge anything.

Command 1 — See recent history and visible branches

Why use it: before coordinating missions, we need to see how recent commits are connected to each other.

Exact effect: `git log --oneline --decorate --graph --all -20` traverses commits reachable from local references and displays them. It does not move any reference.

Options explained: `--oneline` condenses each commit; `--decorate` shows the reference names; `--graph` draws on relations; `--all` includes all known local references; `-20` limits the display to 20 commits.

Expected output: lines containing a graph symbol, a short hash, possible names and a message.

Limits: `--all` shows the known references in the local repository. It does not contact the server. A reference `origin/main` may be old if no `fetch` recent developments have not been carried out. The graph does not show the tasks or occupational dependencies.

Risk: R0 - observation.

```
git log --oneline --decorate --graph --all -20
```

Simplified example:

```
* a812fd3 (agent/cluster-window) Optimize cluster window
"92ce441 (agent/layer1-audit) Add Layer1 output audit
|/
* 4a1b2c3 (hand) Certify pipeline checkpoint
```

The two visible branches here share the same parent `4a1b2c3`. This relationship does not say whether their changes are compatible.

Command 2 — Find the common ancestor of two results

Why use it: two branches can have clear names but be part of different basics. The common ancestor makes it possible to locate their separation.

Exact effect: `git merge-base A B` calculates a better common ancestor between the two revisions. It does not fuse the branches.

Parameters explained: `A` and `B` should be replaced by two references or, preferably in a report, by two confirmed complete hashes.

Expected outing: a commit hash. An error or an absent exit must be examined; stories may not share exploitable ancestors.

Limit: A common ancestor describes the Git topology, not the basis stated in the contract of assignment. A mission may have been recreated or incorporated other changes.

Risk: R0 - observation.

```
git merge-base A B
```

Example:

```
4a1b2c3d4e5f678901234567890abcdef1234567
```

The report can compare this result with `base_commit` declared. A difference is not automatically a fault; it asks for an explanation.

3 — count the specific commits on each side

Why use it: the team wants a first measure of the gap between two states.

Exact effect: `git rev-list --left-right --count A...B` goes through history and counts the commits accessible only from each side.

Syntax explained: `A...B` with three points requires the symmetric difference; `--left-right` keeps the side of membership; `--count` replaces the list with two numbers.

Expected output: two numbers. The first corresponds to the left side `A`, the second on the right-hand side `B`.

Limit: the number of commits does not measure either the number of lines or the severity of the changes. A single commit may modify a critical contract.

Risk: R0 - observation.

```
git rev-list --left-right --count A...B
```

Example:

```
2       5
```

`A` here has two commits that are not in `B`, and `B` has five which are not in `A`. This output justifies an analysis; it does not decide which branch is correct.

Command 4 — List the different files between a base and a result

Why use it: the mission map declares paths allowed. It must be compared to the paths that are really modified.

Exact effect: `git diff --name-status BASE...RESULT` lists the paths and their type of change between the result and their common ancestor with the base.

Options and parameters: `--name-status` displays status and path without showing the content; `BASE` and `RESULT` should be replaced by verified hashes or references; the three points require a comparison from the common ancestor.

Expected output: a letter and then a path. `A` means added, `M` amended, `D` deleted and `R` renamed with a possible score.

Limits: The command does not show the files that are ignored or untracked by another worktree. Nor does it reveal the logical responsibilities affected by these files.

Risk: R0 - observation.

```
git diff --name-status BASE...RESULT
```

Example:

```
M       app/services/cluster/window_selector.rb
A       test/services/cluster/window_selector_test.rb
```

If the contract allowed only `app/services/cluster/` and `test/services/cluster/`, the paths respect the physical border. The functional surface has yet to be examined.

Command 5 — Preview a merge without touching the worktree

Why use it: Before trying a merge in a folder, the team can ask Git to calculate a potential result and report textual conflicts.

Exact effect: `git merge-tree --write-tree A B` calculates a merge and creates an object tree representing the result. It does not create any commits, does not move any branch or writes or writes the index or working tree. It nevertheless adds an object to the local object base: it is therefore not an observation strictly without internal writing.

Parameters: `A` and `B` should be replaced by the two commits or branches to be compared. The team must check the version of Git used, as the forms and options available from `merge-tree` have evolved.

Expected output: the hash of the calculated tree, followed by conflict information when there is one. The team procedure must document the version of Git and the format it analyses.

Major limit: the absence of a textual conflict does not prove to be any semantic or operational compatibility. The command does not replace the actual candidate's construction and testing.

Risk: R1 - creation of a local object without reference displacement.

```
git merge-tree --write-tree A B
```

If the team does not master the release format of its version of Git, they should not automate a decision from this result. It can later create a dedicated integration worktree and build the candidate in a controlled manner.

The procedure to be followed before parallelizing

1. Record the hash of the reference branch.
2. Create a separate mission for each expected outcome.
3. Declare paths and functional surfaces.
4. Identify the dependencies and possible order.
5. Assign one worktree and one branch per workstream.
6. Define who can delegate and how many sub-agents.
7. Define the path of integration before starting production.
8. Regularly observe the gap between bases and targets.
9. Suspend new missions when journaling or integration saturates.

Prompt comparison of two missions

```
Compare the MISSION-A and MISSION-B missions in read-only mode.

Approved entries:
- commit of MISSION-A: HASH-A;
- MISSION-B commit: HASH-B;
- Contracts of assignment: PATHS-TO-CONTRACTS.

You can only use the reading commands explained in this
chapter: git log, git merge-base, git rev-list, git diff --name-status
and
git merge-tree with the provided parameters.

Launches neither merge, base, checkout, switch, reset, restore, clean,
stash,
commit, fetch, sweater or push. Do not change any files.

Reported separately:
1. historical relationship;
2. modified paths;
3. declared common impact areas;
4. potential textual conflicts;
5. semantic or operational risks to be verified;
6. missing information.

Does not decide to enter and resolve no conflict.
```

Common mistakes

Believe that different branches provide insulation. If agents use the same worktree, they share the file and the index. If their industries touch the same contract, physical isolation does not solve logical dependence.

Use only a list of files. Paths are a first barrier, not a complete representation of architecture.

Allow the producer agent to integrate itself. He becomes a judge of his scope, evidence and conflicts with other missions.

Automatically update all branches. A more recent basis may invalidate the initial reasoning. The update is a new decision, followed by new tests.

Measuring success to the number of completed tasks. The value appears when the combined result is included, validated and delivered.

Coordination checklist

- The missions have explicit starting hashes.
- Each agent and sub-agent is attached to a person responsible.
- [- Worktrees are not shared between active producers.
- [. The paths allowed shall be declared.
- Common functional surfaces are declared.
- [- The order dependencies are visible.
- The main branch is not a space for experimentation.
- The integrator knows the exact candidate to be constructed.
- The tests will be carried out on combined hash.
- [- A mission may be stopped without being merged.

Team deliverable: coordination map

```
reference:
  branch: main
  commit: FULL_HASH

missions:
  MISSION_A:
    owner: HUMAN_A
    result_commit: HASH_A
    paths: []
    impact_surfaces: []
  MISSION_B:
    owner: HUMAN_B
    result_commit: HASH_B
    paths: []
    impact_surfaces: []

dependencies:
  - before: MISSION-A
    after: MISSION_B
    reason: CONTRACT_OR_MIGRATION_REASON

integration:
  owner: HUMAN_INTEGRATOR
  candidate_commit: pending
  isolated_tests: []
  combined_tests: []
  decision: pending
```

Success criteria

The team understood the Git border when no one expects an automatic merger to decide on the validity of the product. Git provides objects, relationships and differences. Coordination adds tasks, responsibilities, dependencies and validation criteria.

The chapter is successfully applied when two results can be compared without modifying them, their non-textual dependencies are visible and a human manager knows what evidence is missing before any integration.

CONTINUE READING

This free sample contains the foreword and the first two chapters of *Git in the Age of AI Agents*.

The complete edition covers 65 operational situations: task contracts, worktrees, scope control, integration, dangerous commands, incident response, recovery, and the organization's playbook.