

Agentic Code Surgery for Brownfield Systems

Vivek Ganesan, Ampyard, vivek.ganesan@ampyard.com, github.com/vivganes

Kamal Raj Sekar, Ampyard, kamalraj.sekar@ampyard.com, github.com/trycatchkamal

Kiran Kashyap, Independent, kiran.kashyap.ds@gmail.com, github.com/kirankashyap

Abstract

AI coding assistants are more helpful in greenfield development than for modifying brownfield code — large, undertested, poorly-maintained systems that make up the majority of professional programming. Left to their defaults, these assistants read a few files, guess at intent, and *edit first, verify later*: precisely the failure mode Michael Feathers warned against in *Working Effectively with Legacy Code* [1]. We propose a seven-agent workflow — **Plan, Map, Break, Cover, Implement, Refactor, Finish** — that forces an AI assistant to follow Feathers’ discipline: characterize existing behavior with tests **before** touching code. Each agent has a narrow scope, an explicit exit contract, and a file-based handoff to the next, with human review at every boundary. Applied to a real brownfield codebase, this workflow produced 43 new passing tests (raising statement coverage from 0.85% to 16.78%) against zero new tests and 0.82% coverage for a regular (plan and implement) approach, and **avoided** all critical and major bugs the regular approach introduced.

TL;DR — The Seven Agents

- **Plan** — converts vague requirements into a concrete, scoped change plan; produces `plan/plan.md`.
 - **Map** — surveys the codebase for seams and testing obstacles without touching code.
 - **Break** — minimally restructures dependencies so that tests can be written.
 - **Cover** — writes characterization tests that pin current behavior before anything changes.
 - **Implement** — adds new functionality using Sprout/Wrap, with all tests green.
 - **Refactor** — improves structure, never behavior, with tests green throughout.
 - **Finish** — runs full verification and writes design documentation for the next developer.
-

1. Introduction

AI coding assistants break things. Not so much in greenfield repos with good tests — there, they are useful and mostly useful, if not completely safe. They break things in the codebases that matter

most: the old, undertested, mission-critical systems where the majority of professional programming actually happens.

A substantial fraction of professional programming time is spent on **brownfield code**: systems that are years or decades old, written by people who have long since left, with little or no automated test coverage, tangled dependencies, and business rules that live only in the code itself.

Michael Feathers gave this kind of code a precise and uncomfortable definition [1]:

“Legacy code is code without tests.”

Under this definition, legacy code is not an aesthetic judgment about style or age. It is an operational property: there is no safety net, so every change is a bet. Feathers’ book is essentially a set of techniques for *earning* that safety net before making the change you actually wanted to make [1].

When an AI coding assistant is pointed at such a codebase, two things tend to go wrong:

1. **It reads too little.** The assistant treats the user’s ticket as the specification, inspects few files, and proposes an edit. It does not discover the hidden callers, the implicit coupling, or the undocumented behavior that the code encodes.
2. **It skips the test step.** Because writing tests for untested code is hard — the seams do not exist yet, dependencies are concrete, and the behavior is unclear — the assistant quietly drops the step and proceeds straight to “Implement.” The user gets a plausible-looking diff with no protection against regression.

These failures are not primarily a model-capability problem. A sufficiently capable model *can* follow Feathers’ discipline. The problem is that nothing in the default prompting setup *requires* it to. This paper describes a simple structural remedy: rather than one “do the task” agent, we decompose the work into a pipeline of seven narrow agents, each of which is only allowed to do one step of Feathers’ methodology, and each of which must produce a named artifact before control passes to the next.

2. Background

2.1 Feathers’ core idea

Feathers’ recommended loop [2] for a change to legacy code can be compressed into five steps:

1. Identify the change points.
2. Find places where tests can be written (test points).
3. Break dependencies that prevent testing.
4. Write characterization tests that pin down current behavior.
5. Make the change, then refactor, with the tests as a safety net.

The order matters. Steps 3 and 4 are the ones developers — and language models — most want to skip, because they feel like unrelated work. They are also the steps that make the final change safe.

2.2 Why AI assistants default to skipping them?

A language model prompted with “add feature X to this repository” is under no obligation to write tests first. Its training distribution rewards *visible progress* on the stated task. Characterization tests are invisible progress: they do not move the diff toward the feature the user asked for. Unless the process around the model forces the test step to happen, it usually will not.

2.3 Agents as a scaffolding mechanism

GitHub Copilot supports *custom chat agents* — markdown files that specify a system prompt, a toolset, and a scope of responsibility. A single agent can be instructed to refuse work outside its remit. This is the lever we use: instead of asking one generalist agent to “be careful,” we build seven specialists, each of which is structurally unable to cut corners because the scissors are in a different agent’s drawer.

3. The Seven-Agent Workflow

The workflow maps Feathers’ five steps [2] onto seven agents. The mapping is not one-to-one: we split “Identify” into a dedicated **Plan** phase and a separate **Map** phase, because in our experience the question “what am I changing?” and the question “where can I test?” draw on very different skills. We also add a final **Finish** agent whose only job is verification and documentation — the step most often skipped by both humans and models.

Expected outcomes of the seven-agent workflow

The workflow is designed to produce two kinds of artifacts before the final implementation is accepted:

- **Code + tests for new functionality** — new behavior is added in a test-driven fashion, with tests written as part of the implementation process.
- **Tests for integration points with existing legacy code** — the workflow ensures coverage at the points where the new code interacts with the already existing old code, protecting the change from regressions.

The Workflow at a Glance

See the picture in the next page for a pictorial view of the workflow.

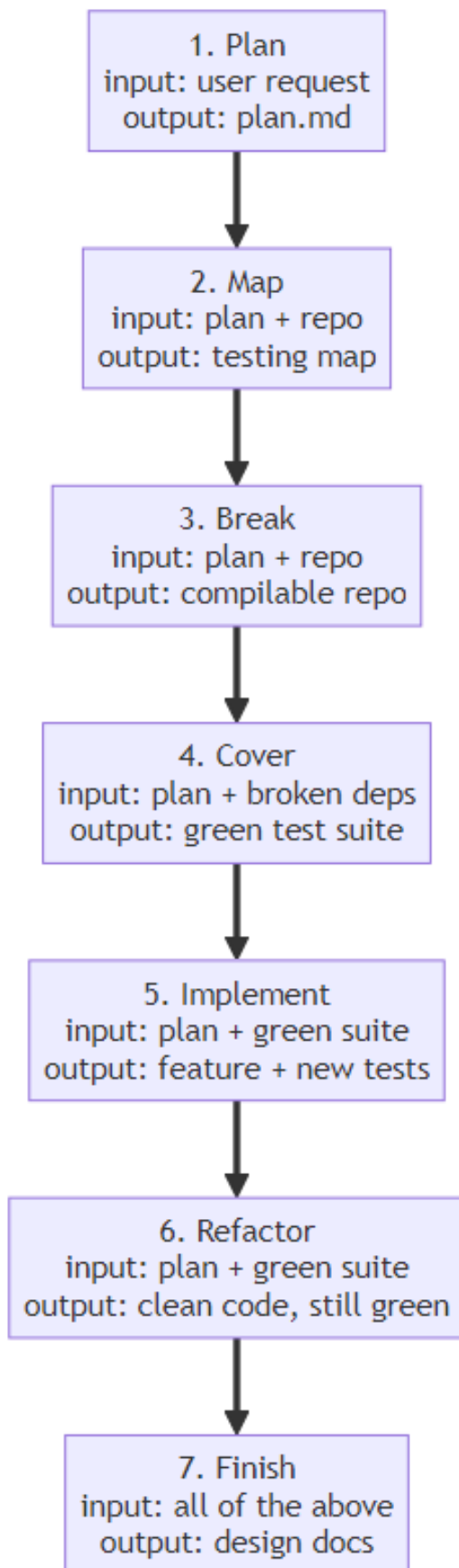


Figure 1 - The seven step agent workflow

- **Two durable, file-based handoffs anchor the pipeline.** The first is `plan/plan.md`, produced by Plan and read as input by every subsequent agent: the *intent* of the change is captured out of band, on disk, so that later agents do not have to reconstruct it from a chat history that may have been truncated or forgotten. The second is `plan/seams-and-dependencies.md`, produced by Map and read by Break as its starting point. The remaining handoffs ride on the code and tests themselves — Break leaves a compilable repository, Cover leaves a green suite, Implement leaves a green suite with new tests — and Finish writes design documentation that records what was learned about the codebase for the next developer. The human developer reviews each artifact between agents.

3.1 Agent 1 — Plan: understand the change

The **Plan** agent is forbidden from editing code. Its only output is `plan/plan.md`, which must answer:

- What is the functional requirement?
- Which classes or modules are the change points?
- What is the scope of impact and who depends on these change points?
- What could go wrong, and how will we know?

The agent is explicitly instructed to push back on “rewrite everything” framings and to ask clarifying questions when requirements are vague. This is important because an AI assistant that *can* plausibly generate a rewrite will generate one if asked — the Plan agent’s job is to convert an ambitious prompt into an incremental one before any code is touched.

3.2 Agent 2 — Map: find test points and seams

The **Map** agent reads the plan and surveys the codebase for:

- **Test points:** places where a test could plausibly be written given current structure.
- **Seams:** in Feathers’ sense, places where behavior can be altered without editing the code at that location — constructor parameters, overridable methods, interface boundaries.
- **Testing obstacles:** hidden `new` calls, static singletons, global state, god objects.

The Map agent is explicitly told *not* to fix anything. Its output is a testing map, not a patch. This separation matters because the most common failure mode we observed during design was agents that, having identified an obstacle, immediately “helpfully” refactored it — skipping past Break, Cover, and Implement in one silent move.

3.3 Agent 3 — Break: break dependencies, minimally

The **Break** agent is given a clear and narrow charter: *Break dependencies to enable testing, NOT to improve design*. It selects from a catalog of Feathers’ dependency-breaking techniques — Parameterize Constructor, Extract Interface, Extract and Override Method, Parameterize Method, Replace Global Reference, Introduce Static Setter — and applies the minimal technique that unblocks the next step.

Two rules are enforced at the agent level:

1. **One dependency at a time.** The agent must stop and compile between breaks.

2. **No new logic.** The agent may only restructure existing code. Bug fixes, cleanups, and “while I’m here” improvements are forbidden and must be deferred to Refactor.

The second rule is the one most worth enforcing on a language model, because an LLM’s prior toward “make this better” is strong, and unsupervised it will silently rewrite a method it finds ugly.

3.4 Agent 4 — Cover: characterize existing behavior

The **Cover** agent writes tests for **existing behavior**, not new behavior. This is the step that most resembles Feathers’ characterization tests:

1. Write a test that exercises the code.
2. Assert what you *think* happens.
3. Run it, observe the failure, and **update the assertion to match reality**.

The Cover agent is explicitly instructed that if a test documents something that looks like a bug, it should pin the bug down, not fix it. Fixes — if warranted — belong in Implement or Refactor, under the protection of the tests that Cover just wrote.

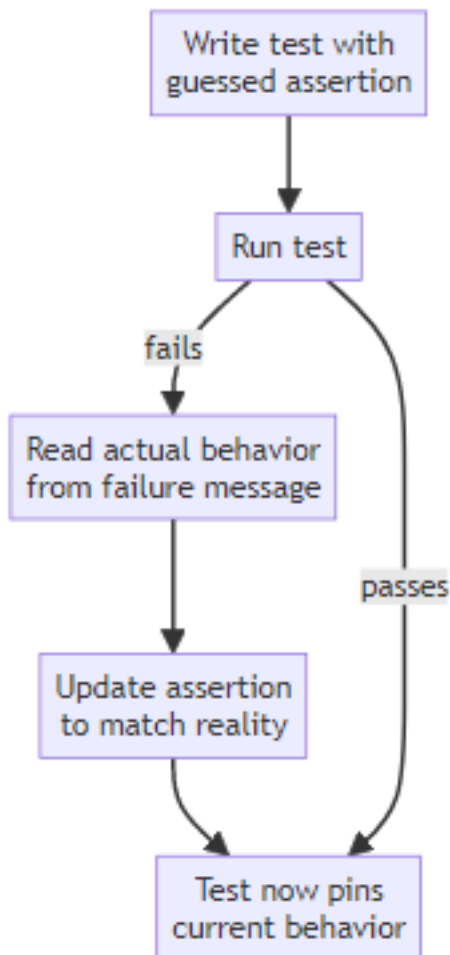


Figure 2 - The process of creating characterization tests using AI coding agents

3.5 Agent 5 — Implement: add the new functionality

Only now, with a passing test suite around the change points, is the **Implement** agent permitted to change behavior. It is instructed to use **Sprout** (method or class) and **Wrap** (method or class) techniques by default, rather than in-place edits, so that:

- New code lives in new, fully-tested units.
- Existing legacy code is touched in as few lines as possible — often a single call site.
- The new functionality can be removed or replaced without unwinding a tangle.

The **Implement** agent follows a red-green-refactor loop, but the refactor in that loop is scoped to the new code only. Refactoring of the legacy code it touched is explicitly deferred to the next agent.

3.6 Agent 6 — Refactor: improve structure, not behavior

The **Refactor** agent is permitted to improve the shape of code that was touched during the **Implement** step — and only that code. It does not refactor legacy code that was not part of the change. It operates under one inviolable rule: *all tests remain green, all the time*. If a refactoring causes a red test, the change is reverted and attempted in smaller steps. The agent works from a catalog of familiar refactorings (Extract Method, Extract Class, Introduce Parameter Object, Replace Conditional with Polymorphism, Introduce Value Object) and is instructed to prefer IDE-backed automated refactorings over freehand edits whenever possible.

3.7 Agent 7 — Finish: verify and document

The **Finish** agent runs the full test suite, checks coverage on the changed area, lints, and verifies documentation. It checks that new or modified code is accompanied by appropriate documentation — creating a `docs/` folder and adding design notes if none exist — so that future changes to the same area begin with a written record of what the code was found to do, not just what it was supposed to do.

4. An analogy: heart surgery

The workflow maps onto cardiac bypass surgery phase by phase. A surgeon does not open the chest and start cutting: they plan the incision, map the vasculature, place the patient on bypass so the organs under change can be safely stopped, attach continuous monitoring, perform the graft, close cleanly, and write a medical record.

The mapping is not merely metaphorical. **Breaking dependencies** is structurally the same move as placing a patient on bypass: real collaborators (databases, singletons) are replaced by controllable test doubles, just as the real heart is replaced by a controllable pump. The substitution is temporary; the safety of every subsequent step depends on it.

The table below states the phase-by-phase correspondence. The surgeon's post-phase claim and the engineer's post-phase claim describe the same structural guarantee in two different domains.

Step	Goal	Once done, heart surgeon says	Once done, software engineer says
1. PLAN	Understand what needs to change.	“Now, we have the exact incision point; we know where to operate and what the likely complications are.”	“Now, we have a clear scope and change points; we know which classes and methods will be modified.”
2. MAP	Find where tests can go.	“Now, we have the entire vascular map; we know where every external connection is and where our life support can attach.”	“Now, we have the seams and dependencies identified; we know exactly what is preventing immediate testing.”
3. BREAK	Make code testable.	“Now, we have the patient stabilized on life support; all external systems are controllable and replaceable.”	“Now, we have a testable system; all hard-coded dependencies are broken and injectable using test doubles.”
4. COVER	Write tests for existing behavior.	“Now, we have a constant, reliable readout of the patient’s vitals; any change in existing function will immediately trigger an alarm.”	“Now, we have a safety net of passing characterization tests; we are protected from accidentally changing existing behavior (regression).”
5. IMPLEMENT	Add new functionality with TDD.	“Now, we have successfully grafted the new vessel; the patient’s new circulation path is working perfectly.”	“Now, we have the new feature fully implemented and proven; all new and old tests are passing.”
6. REFACTOR	Improve design safely.	“Now, we have closed the incision cleanly; the surgery is a success, and the new structure promotes long-term health.”	“Now, we have clean, organized, and maintainable code; the design is improved and all tests are still green.”
7. FINISH	Verify and document.	“Now, we have a full medical record; the patient is stable, and the entire procedure is documented for future care.”	“Now, we have a complete, verified, and documented feature; the work is ready for deployment and is safe for the next developer.”

Two failure modes map cleanly onto this frame: the *cowboy surgeon* opens the chest and starts

cutting without bypass (AI edits code without breaking dependencies or writing tests), and the *unmonitored operation* performs a technically correct graft with no vital signs attached (code is changed but nothing detects a regression). Both failures are prevented by enforcing the order of operations — which is what the seven agents do.

5. Why seven agents instead of one prompt?

A reasonable objection is that everything above could be collapsed into a single system prompt: “follow Feathers’ methodology; write tests first; use Sprout/Wrap.” We tried the collapsed form. It fails in a predictable way: the model writes a plausible plan, writes a plausible test stub, and then, under the gravitational pull of the actual user request, skips to Implement. The seven-agent form works better for four structural reasons:

1. **Scope narrowing reduces drift.** Each agent’s system prompt describes only one step, so the “attractor” of the conversation is that step, not the final feature.
2. **A written plan survives context loss.** `plan/plan.md`, produced by Plan and read by every subsequent agent, lets later agents recover the intent of the change even if the chat window has been compressed. Intermediate handoffs ride on the code and tests themselves — a compilable repo after Break, a green suite after Cover — which are likewise durable across context resets.
3. **The human has seven checkpoints.** A developer using the workflow reviews an artifact between every pair of agents. Mistakes are caught before they compound.
4. **Forbidden moves are enforced at the agent boundary.** “Do not refactor here” is easier to obey when the agent has no refactoring remit at all than when it is one bullet in a long system prompt.

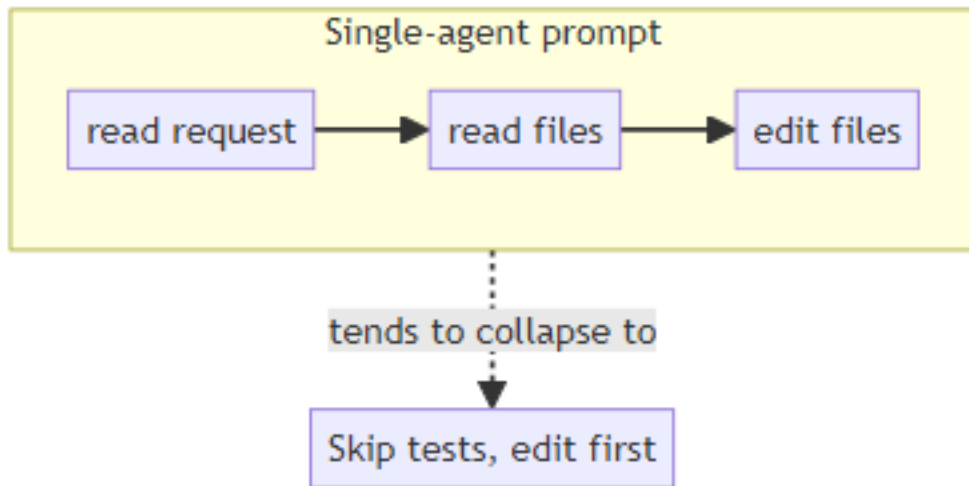


Figure 3 - Problem with using a single prompt

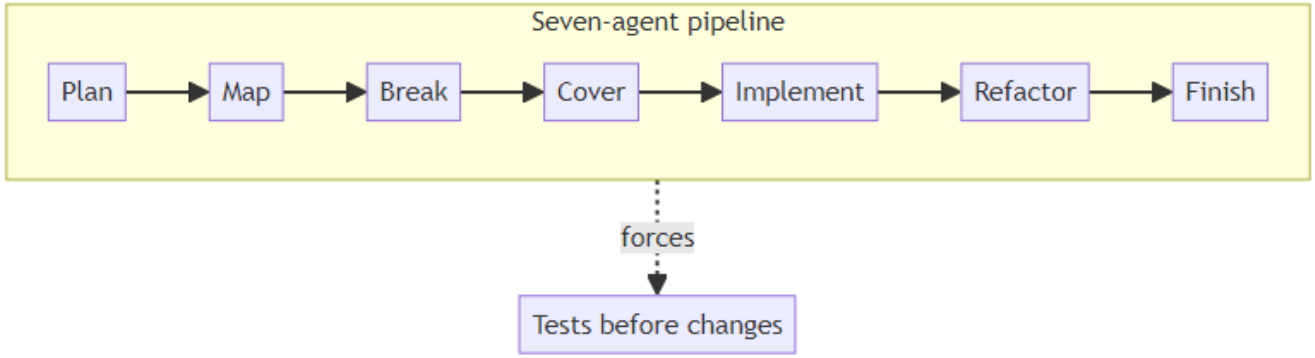


Figure 4 - The solution: The seven step agent workflow

6. Experiment: Seven-Step vs. Regular Usage on a Real Brownfield Codebase

6.1 Setup

To validate the workflow in practice, we applied it to **kanbanstr** — a opensource, production Nostr-based Kanban application written in Svelte and TypeScript — targeting a non-trivial brownfield feature: threaded card comments. The codebase had existing test infrastructure but no tests for the components and stores we needed to touch.

At the time of the experiment, the codebase had 6 passing tests across 2 test files, with 0.85% statement coverage — typical of brownfield code where tests exist for utilities but not for the core business logic. We ran two parallel implementations of the same feature:

- **Seven-Step (PR #32):** The full seven-agent workflow — Plan, Map, Break, Cover, Implement, Refactor, Finish — with human review between each step.
- **Regular Usage (PR #33):** A condensed two-step approach — a single planning prompt followed by a single implementation prompt — mirroring the common default usage of an AI coding assistant.

The repository is public and can be accessed at <https://github.com/vivganes/kanbanstr>

Both PRs were then independently reviewed by LLMs and the results were compared.

6.2 Results

Dimension	Regular Usage (PR #33)	Seven-Step (PR #32)
Model used	Claude Sonnet 4.6 (Medium) in Github Copilot	Claude Sonnet 4.6 (Medium) in Github Copilot
Interactions	2 (plan → implement)	7 (one per agent)
Tests written	None (0 new)	43 new (49 total)
Statement coverage	0.82%	16.78%
<code>kanban.ts</code> coverage	0%	20.83%
<code>CardDetails.svelte</code> coverage	0%	100%
Critical bugs	3	0
Major bugs	1	0

Dimension	Regular Usage (PR #33)	Seven-Step (PR #32)
Medium issues	0	3
Process artifacts	plan.md only	plan.md, seams-and-dependencies.md, design docs
Human review checkpoints	0	6
Commits	1	4 (one per phase boundary)

The three critical bugs in the regular usage output were:

1. **Memory leak** — a `kanbanStore.subscribe` call with no corresponding `onDestroy` cleanup, causing subscriptions to accumulate on every card open/close.
2. **Silent error swallowing** — `try/finally` without `catch`, meaning publish failures produced no user feedback.
3. **Unstable Map key** — comments keyed on `card.id` (an event hash), which changes when the card is edited; the correct key is `card.dTag`.

The seven-step output had none of these. Its issues were minor polish items — a missing toast notification and a fragile test mock — the kind a normal code review would catch.

6.3 Observations

6.3.1 Tests were the sharpest differentiator. The regular usage agent produced a plan that explicitly mentioned tests were needed, then silently dropped them during implementation — leaving statement coverage at 0.82%, essentially unchanged from the 0.85% baseline. The seven-step workflow made this structurally impossible: the Cover agent runs before Implement and is forbidden from writing implementation code. The result was 43 new tests across two new test files, raising statement coverage to 16.78% and achieving 100% statement coverage on `CardDetails.svelte` — the primary component touched by the feature.

6.3.2 Seam analysis prevented the memory leak. The `seams-and-dependencies.md` produced by the Map agent explicitly listed `kanbanStore.subscribe` as a lifecycle seam requiring cleanup. The regular usage agent read the same codebase and skipped this analysis; the memory leak followed directly.

6.3.3 Planning quality was comparable; execution quality was not. Both approaches produced a solid plan that correctly identified the change points, the Nostr tag schema, and the risk of unstable Map keys. The divergence happened during implementation. This confirms the central claim: the seven-step workflow’s value is in structured execution, not in smarter planning.

6.3.4 Severity distribution follows process discipline. The regular usage output contained 2 critical and 1 major bug with 0 new tests and sub-1% statement coverage. The seven-step output contained 0 critical or major bugs with 43 new tests and 16.78% statement coverage. The remaining seven-step issues are all in the minor category — the type that surfaces in normal code review rather than in production.

6.3.5 The commit history is itself a quality signal. The seven-step PR’s commits tell a traceable story: `changes-upto-4-cover` → `code after 5-implement` → `code after 6-refactor` → `after finish`. Each commit corresponds to a verifiable state. The regular usage PR has a single commit with no intermediate checkpoints.

6.3.6 Agent instructions were obeyed partially, not universally. Even within the structured pipeline, the model drifted from its instructions in three observable ways.

Incomplete TDD granularity. The Implement agent’s prompt specifies strict Red-Green-Refactor: write one failing test, make it pass, refactor, then repeat. In practice the agent wrote all 16 store-layer tests for `loadCommentsForCard` and `publishComment` as a single batch before writing a single line of production code — closer to “test-first” than “test-driven.” The distinction matters: strict TDD uses each failing test to drive the minimal implementation of the next unit of behavior; batch-first testing provides coverage but loses the design feedback that emerges from making each test pass in isolation.

Selective application of TDD. The Implement agent applied even that partial discipline only to the store layer. It then implemented the `CardDetails.svelte` comments UI with no component tests at all. The human reviewer caught this at the Implement→Refactor checkpoint. The agent acknowledged the failure directly: *“I implemented CardDetails.svelte without writing failing component tests first, violating TDD.”* Eighteen component tests were added before Refactor began.

Silent checklist truncation in Finish. The Finish agent’s prompt lists five verification categories, including running `npm run coverage`, running `npm run lint`, and performing manual testing of happy-path and error scenarios. The agent ran the test suite (all passing) and the TypeScript/Svelte type check, then stopped. Coverage was not measured — the coverage dependency had not been installed in the project at that point, and the agent silently skipped the step rather than flagging the gap. Linting and manual testing were not mentioned. Of the five todos the agent created for itself at the start of the step, only four were explicitly completed.

Three observations follow from these incidents:

First, structured agents reduce instruction drift but do not eliminate it — a single agent context window remains subject to the gravitational pull toward visible progress.

Second, the human review checkpoint between agents is the mechanism by which partial compliance is caught and corrected before it compounds; the `CardDetails.svelte` omission would have become a permanent test gap had Refactor proceeded without intervention.

Third, checklist items that require tools or dependencies not yet present in the project will be silently dropped rather than flagged — the Finish agent’s coverage skip is a practical reminder that the agent cannot distinguish “this command would fail” from “this step is not needed.”

6.4 Limitations

Experiment scope. This is a single feature on a single codebase. The regular usage output was generated with a particular model and prompt; a different prompt might produce better results. The comparison was conducted by LLMs whose evaluations carry their own biases. We treat this as a proof of concept that the workflow behaves as designed, not as a controlled study. We invite the readers of this paper to test the workflow using the agent setup detailed in Appendix-B and share their results in any appropriate way.

Known workflow failure modes. The workflow has structural limits that practitioners should understand before applying it:

- **No build environment.** The Break agent compiles between each dependency break; the Cover agent runs tests after each. If the codebase cannot be compiled or the test suite cannot be executed locally, both agents lose their tightest feedback loop. The workflow degrades to a planning exercise without the safety net.
- **Huge codebases.** The Map agent must survey the full codebase for seams. On very large systems (hundreds of modules), this can exceed the agent’s context window. In practice, the Plan agent should scope the map to the relevant subsystem.
- **Break agent getting stuck.** Some dependency tangles cannot be broken with a single Feathers technique — they require several dependent breaks in a specific order. An agent that applies one break and re-reads the full diff can lose track of which breaks remain. Human review between Break invocations is the mitigation.
- **Dynamic languages.** Seams in Python or JavaScript look different from those in Java or C#: duck typing means interfaces don’t need to be extracted, but monkey-patching can introduce unexpected coupling. The agents’ prompts instruct adaptation to the host language, but the technique catalog is biased toward statically-typed languages.

These are not arguments against the workflow; they are conditions under which its benefits shrink and its costs grow. A practitioner who cannot run tests should know this before investing six agent invocations.

7. Discussion

7.1 What this workflow is *not*?

It is not a claim that AI assistants can safely handle arbitrary legacy code unsupervised. It is a claim that, *under human review between steps*, a seven-agent decomposition of Feathers’ methodology reliably produces the artifacts — plan, seams map, broken dependencies, characterization tests, minimal implementation, refactoring, design documentation — that a careful human would produce. The human is still the reviewer; the agents are a disciplined apprentice. They are not a replacement for reading Feathers’ book: without that grounding, users may not fully understand how to review each step or when to accept versus ignore an agent’s recommendation during human-in-the-loop review.

7.2 Costs

The obvious cost is latency: seven serial agent invocations are slower than one. In our experience this is overwhelmingly dominated by the time the developer saves not debugging a change that silently broke an untested call path.

The second cost is that the Break and Cover steps sometimes produce ugly intermediate code — parameterized constructors that exist only to enable tests, Extract-and-Override subclasses that will not survive Refactor. Feathers addresses this directly: intermediate ugliness is the price of the safety net, and most of it is removed at the Refactor step. The third cost is that the pipeline does not guarantee compliance: as Section 6.3 shows, the Implement agent drifted from TDD mid-step despite it being the centerpiece of its remit. Human review at each checkpoint is therefore not optional — it is the enforcement mechanism without which the structural guarantees weaken.

7.3 Generalizations

Nothing in the workflow is specific to GitHub Copilot. The same seven agents can be implemented on any assistant that supports per-agent system prompts and file I/O. Nothing is specific to a single language either: each agent’s prompt instructs it to adapt examples to the host language, and the techniques — seams, dependency breaks, Sprout/Wrap — are language-agnostic in Feathers’ original treatment.

8. Related Work

Our workflow is a direct operationalization of Feathers, *Working Effectively with Legacy Code* [1]. The Sprout/Wrap family of techniques is Feathers’; the dependency-breaking catalog is Feathers’; the Red-Green-Refactor cycle used inside Implement is Beck’s [4]. The refactoring catalog used by Refactor derives from Fowler’s *Refactoring* [3].

The contribution of this paper is not the techniques but their **utilization as a pipeline of narrowly-scoped AI agents**, each of which inherits one step of the original methodology and is structurally prevented from stepping on the others.

9. Conclusion

AI coding assistants are at their worst where industrial software is at its worst: in large, old, poorly-tested codebases. The failure is not usually one of model capability; it is one of prompting structure. A single generalist agent, asked to “add the feature,” predictably skips the characterization-test step that makes the change safe.

Splitting the work into seven specialist agents — **Plan, Map, Break, Cover, Implement, Refactor, Finish** — with file-based handoffs and narrow remits restores Feathers’ discipline. Tests get written before changes, dependencies get broken minimally, new functionality sprouts into new units rather than being grafted into tangled ones, and the whole journey is documented.

None of the individual techniques are new. What is new, and what we hope is useful, is the observation that the right way to use an AI assistant on legacy code is to give it less room, not more.

References

- [1] Feathers, M. *Working Effectively with Legacy Code*. Prentice Hall, 2004.
- [2] Feathers, M. *Working Effectively with Legacy Code*. A Short Paper published at <https://objectmentor.com/resources/articles/WorkingEffectivelyWithLegacyCode.pdf>
- [3] Fowler, M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
- [4] Beck, K. *Test-Driven Development: By Example*. Addison-Wesley.
- [5] Feathers, M. Interview on *Working Effectively with Legacy Code*, InfoQ Podcast. <https://www.infoq.com/podcasts/working-effectively-legacy-code/>

Appendix A — Agent Catalog

#	Agent	Input	Output artifact	May edit code?	Forbidden moves
1	Plan	user request	<code>plan/plan.md</code>	No	writing code, proposing rewrites
2	Map	<code>plan/plan.md</code> + repo	<code>plan/seams-and-dependencies.md</code>	No	finding obstacles
3	Break	<code>plan/plan.md</code> + repo	restructured, compilable code	Yes, structure only	new logic, bug fixes, design improvements
4	Cover	<code>plan/plan.md</code> + repo	passing characterization tests	Tests only	fixing bugs it documents
5	Implement	<code>plan/plan.md</code> + green suite	new feature + tests	Yes	large legacy edits (prefer Sprout/Wrap)
6	Refactor	<code>plan/plan.md</code> + green suite	cleaner code, same behavior	Yes	adding features, changing behavior
7	Finish	everything above	verification report, design docs	No	starting new work

Appendix B - Agent Prompts

We have created these agents as Github Copilot `*.agent.md` files. You can find the file contents under `agents` directory of this paper’s Github repository at <https://github.com/ampyard/brownfield-agentic-code-surgery>.

One can easily adapt these files for any coding agent of their choice.

The usage instructions can be found in `how-to-use.md` file in the same repository.

Appendix C - Agent Execution Logs and Supporting Information

You can find the chat logs and other supporting information like coverage report, AI review report, etc. under `experiments` directory of this paper’s Github repository at <https://github.com/ampyard/brownfield-agentic-code-surgery>.