carrick

FIELD GUIDE

How coding agents fail

Six structural failure modes of frontier coding agents on production codebases, and the explicit rules that mitigate them.

While building Carrick, we ran controlled test suites observing frontier coding agents execute tickets on production estates ranging from single repositories to multi-repo systems. The most critical failure modes are structurally silent: the tests run green and the diffs read clean, while duplicate implementations, unnotified consumer breaks, and unbudgeted review overhead accumulate underneath.

TL;DR: skip to the rules file.

1 · Silent duplication

When assigned a task whose underlying behaviour already exists, agents frequently build a redundant implementation instead of reusing code. In our benchmark runs, unaided agents shipped parallel implementations 54% of the time without flagging the overlap.

This duplication is rarely flagged during initial review. The engineering cost accumulates over time as the two implementations drift, each gathering its own dependent consumers.

> **Action:** Make *"does this duplicate existing logic?"* a standing review question on any agent PR that adds helpers, parsers, validators, or formatting functions. Before the agent creates a new function, require it to search likely names and their synonyms, to read the module where that behaviour would live, and to run a semantic code search where one is available. Require a single-line explanation whenever existing code was found and bypassed. `grep` can prove a name is absent; it cannot prove the behaviour is.

2 · The file that was never opened

When tasked with enumerating components across a codebase ("list every service in this workflow" or "find all consumers of this event"), agent omissions correspond directly to unread files. In a 43-service monorepo audit, unaided agents missed nearly 25% of participating services. Every omission resulted from the agent failing to open the file; none were due to unreachable or ambiguous code.

Agents search using their initial context and assumed naming conventions. A service registered under an unexpected name remains practically invisible. Because this is a failure of omission rather than fabrication, the output appears plausible, leaving no erroneous lines for a reviewer to catch.

> **Action:** For enumeration tasks, require agents to query a mechanically listable source of truth, such as a service registry, route table, directory listing, or dependency graph, and explicitly cite that source rather than relying on unstructured search. If no single source of truth exists, creating one benefits human maintainers as well.

---

3 · Repository blindness

If required context resides in a repository the agent has not checked out, it will not initiate cross-repository discovery. Instead, it relies on local context and fills information gaps with plausible assumptions. In multi-repository benchmark runs, unaided agents routinely missed cross-repo dependencies despite having git access to all repositories, simply because they lacked targeted navigational context.

> **Action:** Provide explicit architectural mapping in the agent rules file, defining domain ownership, contract locations, and cross-service consumers. For tasks spanning repository boundaries, ensure relevant repositories are checked out side-by-side and documented in the instructions.

---

4 · Trusting the drifted copy

Codebases frequently contain local duplicates of external service contracts, such as mirrored types, manual API clients, or static constant files. Agents treat the nearest local file as authoritative. During testing, agents regularly generated code based on outdated local constants without cross-checking the producing service's primary definition.

Action: Enforce an authority rule: local copies of external contracts are prone to drift; the producing service remains the definitive source of truth, and schema conflicts must always resolve in favour of the producer. Systematically eliminate local contract mirrors to reduce maintenance overhead for both agents and human developers.

---

5 · Confidently, identically wrong

On a complex cross-service task, repeated runs produced identical incorrect output in every case. Engineers often misinterpret convergent AI outputs as validation. In practice, it indicates that independent runs encountered the same misleading signal.

Action: Never treat consensus (between runs, distinct agents, or agent-reviewer pairs) as verification. Validate against ground truth by executing tests and inspecting producer contracts directly. Allocate review depth based on system blast radius rather than output unanimity, and treat agent self-attestations (”verified”, ”tests pass”) as unverified claims.

---

6 · Finding is not adopting

Agents do not duplicate code solely due to retrieval failure. In reuse test cases, agents successfully located the existing implementation, quoted it in their execution summaries, and still re-implemented the logic or copied internal code line-by-line into a new module. Retrieval succeeded, but adoption failed. An agent summary stating ”already implements identical logic” does not guarantee proper code reuse in the pull request.

This behaviour stems from two main causes: a bias towards direct authorship (interpreting minor signature or naming mismatches as authorisation to rewrite rather than wrap) and premature search termination (treating a near-miss function as a signal to build from scratch).

Action: Add an explicit code adoption directive to agent rules. If an existing function provides the required logic, agents must import or extend it. Signature or parameter mismatches require a thin wrapper rather than a re-implementation. Any decision to bypass existing code must be explicitly logged with a reason from a pre-approved list (e.g. distinct domain logic, import cycle constraints, or explicit deprecation). Naming differences or parameter mismatches are not valid reasons for non-adoption.

When agents get it right

When the task lined up with existing, well-named modules, agents got it right in 100% of test runs: they consolidated the duplicate logic, moved every consumer onto a single implementation, and deleted the redundant modules.

The failures in this guide cluster where search, naming, domain ownership, and authority are ambiguous. They are properties of the codebase rather than of the agent, which is why engineering conventions can mitigate them.

The rules file

Drop this into the instructions file your agents read (`CLAUDE.md`, `AGENTS.md` or equivalent) and edit the bracketed parts.

```
# Working in this codebase

## Contracts and other services
If the answer is defined by another service (its endpoints, its real
request/response types, who consumes it, or any local type, constant,
or SDK helper that mirrors it), verify against the producing service
before you rely on anything local. Local copies of another service's
contract drift and lie; the producing service defines the contract,
and if two definitions disagree, the producer's wins.

## Before writing new code
Inside this repo, grep and read are fast; use them. One exception:
before writing any new helper, parser, validator, or domain function,
check whether the behaviour already exists. Search likely names and
their synonyms, and read the module where that behaviour would live;
if you have a semantic or intent-based code search, query it with a
plain-English description of the behaviour instead. Grep can prove a
name is absent; it can never prove the behaviour is. Run that check
before your first search on the task, not once you are about to
write. By the time you have a design in mind, a near-miss reads as a
reason to start fresh.

## Adopting what you find
If an existing function implements the behaviour you need, import it
or extend its module. A signature mismatch is a reason for a thin
wrapper, not a rewrite. If you deliberately do not adopt something
you found, say so in one line and give one of these reasons: the
behaviour genuinely differs, the module cannot be imported from
here, or it is deprecated. A signature mismatch, a naming
```

```
difference, or an extra parameter are not on that list. Those are
wrapper cases.

## Enumeration
For any "find all X" task, enumerate from a listable source of truth
(registry, routes table, directory listing, dependency graph) and
cite it. Do not answer enumeration questions from search results.

## Where things live
[Repo/domain map: which repo owns which domain, where contracts are
defined, which repos to check out side by side for cross-repo work.]

## Verification
Agreement between runs, agents, or reviewers is not verification.
Verify against ground truth: run the code, read the producing
service's actual contract.
```

Where the rules stop working

Rules files have two ceilings.

- **The vocabulary cap.** Rules cannot hand an agent a name it never thought to search for. In the 43-service census, every miss was a file no agent opened. Better instructions reduce this; they cannot eliminate it, because the failure happens before any rule about verification can apply.
- **The address gap.** Rules can say that other repositories matter, and even list them. They cannot say which unfamiliar repository holds the one fact an agent needs today. In our cross-repo runs, agents had identical git access. What differed was whether something told them where to look.

Both ceilings are failures of knowing what exists and what it is called, across a boundary the agent cannot see over. That is an index problem, not an instructions problem.

About the evidence

- Hundreds of controlled test runs of frontier coding agents (Claude Sonnet and Opus class models) working real tickets on production-scale codebases, scored against known ground truth.
- Your results will differ. The variant space is large, spanning model, prompt, ticket wording, existing context, docs, and whatever rules files are already in the repo. Treat every percentage as evidence of a mechanism, not a benchmark.
- The tasks behind the 54% figure pinned exact function signatures so they could be scored mechanically, which plausibly nudges agents towards fresh code. The real-world rate may be lower.

About Carrick

Carrick exists for the two ceilings above. It parses your source in CI, so every merge rebuilds one index of the whole system across your organisation's TypeScript repositories. Each function carries a plain-English description of its behaviour, each endpoint carries its compiler-resolved request and response types, and the index records which services consume which contracts. Your agents query it over MCP and search by what code does rather than what it is called, so "does this behaviour already exist?" and "which service owns this contract?" are answered from an index of what actually exists rather than from whatever the agent happened to search for.

The same index works in review: the Carrick GitHub App flags duplicate functions, contract mismatches and dependency version conflicts in the pull request, before they merge.

Apply for free access at carrick.tools/apply, or read the docs at docs.carrick.tools.

Published by Far Harbour B.V. · carrick.tools