DuraLoop: Reducing Rework, Cost, and Customer-Visible Errors in AI Coding Agents

A production case study and operational framework for reducing rework in AI coding-agent fleets

Abhishek Sehgal Chitrita Goswami*
Quotech, Bravada Group, Eye Dream (Finsoeasy), Australia

ABSTRACT

AI coding agents matter when they reduce total engineering cost, not when they merely produce patches that pass once. The expensive failure is rework: a customer reports the same problem again, a feature was marked done before it was delivered, or a local fix breaks when the same invariant appears elsewhere. A green regression test is not durability.

We report a named production case study of one instrumented LLM-agent fleet maintaining a live B2B SaaS. The corpus contains **300 production fixes**, **92 logged recurrences**, distilled robustness principles, and agent chat transcripts. We corroborate the defect log against git history (cited commits are real; recurrence-linked fixes re-touch the same files **19**× more than chance), providing a verifiable audit trail.

We define **durability@N** as the probability that a shipped fix survives N days without logged recurrence, and pair it with a five-state delivery lifecycle view.

For teams, the result is a rework-reduction playbook. First, fixes that pass tests can still return: one in seven logged fixes recurs, and **54%** already had a regression test. Second, repeated asks are mostly never-delivered scope or partial delivery, not regressions. Third, incomplete generalization dominates: the agent fixes the reported site, not the shared pattern or contract. This imposes a heavy recurrence tax, of which AI tokens are only **0.2%** (§5.5). We package these findings into **DuraLoop**, an operational framework to measure durability at 7 and 14 days, gate blast-radius changes, and retrieve prior lessons, so teams spend less senior review on low-risk patches and catch agent mistakes before customers do.

KEYWORDS

AI coding agents, software maintenance, fix durability, recurrence, retrieval-augmented generation, empirical software engineering

1 INTRODUCTION

The agentic-software-engineering literature optimises a single-shot metric: given an issue and a hidden test, can the agent produce a passing patch (pass@k [15])? That is useful, but it is not the metric that hurts in a live product. A customer does not care that a patch once passed a test if the same workflow is still missing, only half delivered, or broken again two days later.

We call the missing metric **delivery durability**: after an AI coding agent attempts work, does the intent stay delivered? That forces us to separate five states that are easy to conflate in a busy codebase:

- **Never delivered:** the operator asks again because the work never shipped.
- **Partial delivery:** something shipped, but the missing remainder is the real user need.
- **Delivered then regressed:** the work shipped and later broke; this is true recurrence.
- **Clarification:** the repeat sharpens or changes the original intent rather than proving a failure.
- **Durable delivery:** the work ships and stays correct.

That distinction matters. If every repeated report is called "recurrence", the fix is too narrow: we would write more regression tests when the actual problem was unfinished scope or vague intent. If every repeat is called "backlog", we miss the real regressions that customers are catching. We need a lifecycle view that tells the agent and reviewer which control to apply.

The study comes from an unusually instrumented production setting. LLM agents do most of the coding for a live multi-tenant B2B SaaS and maintain three living ledgers as part of the operating process: a **bug-fix ledger**, a **recurrence log** with a first-principles "why is it back?", and a **robustness-principles** file that turns returns into reusable lessons. With **935 agent chat transcripts**, this yields a longitudinal corpus of fixes, repeated demand, and eventual fates. We corroborate the defect log against the two repositories' git history (§5.1): the commits it cites are real, and recurrence-linked fixes independently land on the same files far more than chance.

We ask five research questions:

- **RQ1.** When a bug is marked fixed, how often does it truly recur, how fast, and who catches it?
- **RQ2.** When operators repeat an ask, is it never-delivered work, partial delivery, true regression, or later-completed work?
- **RQ3.** Which fix-time signals tell us a patch needs deeper review?
- **RQ4.** Does injecting the organisation's own distilled principles improve judged invariant quality in an offline replay?
- **RQ5.** What does non-durable delivery cost, and where should prevention focus?

Contributions.

1. **durability@N**, a deployment-time metric for whether fixes stay fixed.
2. A **delivery-lifecycle dataset**: 300 production fixes, 92 defect recurrences, and 600 prompt-mined demand clusters labelled for re-ask, rectification, and gold-sample lifecycle state.

*ORCID: Abhishek Sehgal 0009-0000-9424-4695, Chitrita Goswami 0000-0001-9849-1095.

3. A practical **failure-to-action mapping**: incomplete generalization, unpinned contracts, site-only fixes, vague intent, and missing shared memory each map to a concrete coding control.
4. A **pre-merge gate**: blast-radius annotation is the robust fix-time signal in this case study, so teams should focus extra review on cross-cutting changes instead of reviewing every patch equally.
5. A **placebo-controlled, independent-model judged replay** showing that structured principle context improves judged invariant quality over no-principle context; whether *relevance-specific* retrieval beats any principle-shaped context is not established at this sample size (n=25). Plus **DuraLoop**, the implementation loop these findings compose into.

2 BACKGROUND AND RELATED WORK

We position the paper against six lines of work, drawn from a corpus of 158 papers mined for this study (Appendix B). The throughline: each measures *whether code is produced correctly once*; none measures whether an agent's fix *stays* fixed in a live repository, which is what durability@N targets.

Agentic software engineering and its metrics. Code generation is scored at pass@k, the chance that one of k samples passes hidden tests [15], and iterative self-repair lifts that pass rate with more attempts [6]; issue-resolution agents inherit it: SWE-bench [32] and SWE-agent [70] report single-shot resolution on frozen issues, and surveys of the area catalogue dozens of such benchmarks and agents [22, 65]. Even hardened-test variants such as EvalPlus [40] and multi-dimensional suites that look "beyond correctness" [52, 74] strengthen the *test*, not the *time* axis. Recent work begins to probe what happens *after* the patch: characterisations of where issue-resolution agents fail [31], the observation that AI-agent pull requests often go unmerged [3], and calls for reproducible agent evaluation [39, 51]. durability@N is the deployment-time dual these efforts lack: a fix that passed once and recurred in front of a customer scores 1 at pass@1 and 0 at durability@N.

Patch overfitting and correctness. Automated program repair long ago found that plausible (test-passing) patches are frequently *incorrect* [54, 59], the central risk in generate-and-validate systems [10, 36] and a still-open problem: patch-correctness classifiers remain brittle [68, 71, 72]. Curated fault sets such as Defects4J [33] and maps of repair effectiveness [4] enable controlled study but not longitudinal recurrence. We observe overfitting *in the wild*, over deploy time, with user-reported ground truth, and quantify how often a regression test fails to prevent it.

Hallucinated and unpinned contracts. A dominant recurrence mode we find, relying on an API/schema shape that later shifts, connects to code *hallucination*: LLMs invent or mis-bind APIs [2, 37, 73], and mitigations retrieve real API context at generation time [18, 21, 48]. Our "contract-boundary-unpinned" mode is the durability consequence of the same gap: the contract is not hallucinated at write time but is left *unpinned*, so a later upstream change re-breaks the fix.

Test effectiveness and flakiness. That 94% of our fixes shipped a regression test yet 54% of recurrences also had one echoes work on tests that are *present but ineffective*: flaky and order-dependent tests [11, 12, 29, 43, 55], mutation testing as an effectiveness probe [9, 53], and execution-aware regression selection [35]. Self-debugging loops that generate their own tests [17] improve single-shot pass rates but do not certify durability. Fault-localization quality bounds repair quality [42, 56, 69]; our "site-over-pattern" mode is localisation that stops at the reported site.

Retrieval-augmented generation. Our prevention mechanism applies RAG [38] to an organisation's *own* distilled robustness principles rather than to documentation or code [61, 67]. The novelty is the retrieval target: institutional, recurrence-derived lessons used to inoculate the next fix.

Multi-agent fleets, demand mining, and cost. Our setting, a fleet of LLM agents on one codebase, is the subject of a fast-growing multi-agent-SE literature [24, 50, 57, 60, 64], with large-scale industrial deployments now reporting on LLM code review [63] and developer-productivity gains [25]; but those works study task-solving and acceptance, not the durability of what the fleet ships. The user-side signal we mine (re-asked features) connects to reopened-bug studies [62] and app-review / feature-request mining [1, 26, 30, 46, 47]; we add the *rectification* axis: was the repeated ask ever durably delivered. Finally, the recurrence tax relates to maintenance-cost, technical-debt, and defect-prediction economics [5, 7, 13, 27, 41, 44, 58], here measured for an agent fleet rather than human developers.

Method. For RQ2 we use LLM-assisted qualitative coding validated against an author-adjudicated gold standard, an approach now studied directly in software-engineering research [45]; §4.7 details the codebook and reliability protocol, and §5.2 reports where the labels are reliable (concrete domains, ask-type) and where they carry boundary noise (cross-cutting themes).

3 CONCEPTUAL FRAMEWORK: FROM PATCH CORRECTNESS TO DELIVERY DURABILITY

The central claim of this paper is that AI coding teams are optimizing the wrong terminal unit. A patch is an implementation event: some files changed, tests ran, CI passed, and the branch merged. That event matters, but it is not the user outcome. The outcome is delivered intent: the original workflow, defect, or operator need remains solved after the change reaches production and the surrounding system keeps moving, a shift toward intent-centric rather than code-centric accountability that the broader literature is also beginning to argue for [19]. A green patch can therefore be locally correct and operationally non-durable. It may fix the named screen but miss the shared component, pass a mocked serializer shape but drift from the deployed API, or satisfy a narrow reproduction while leaving the business invariant untested.

We model agent work as a transition from reported intent to deployed state, not from prompt to patch. The report starts as an intent claim: a user cannot complete a workflow, a feature is missing, a prior fix appears broken, or a production invariant has been violated. The agent then proposes a code transition. CI and review observe whether the transition is syntactically and locally plausible. Delivery durability observes whether the intended state remains

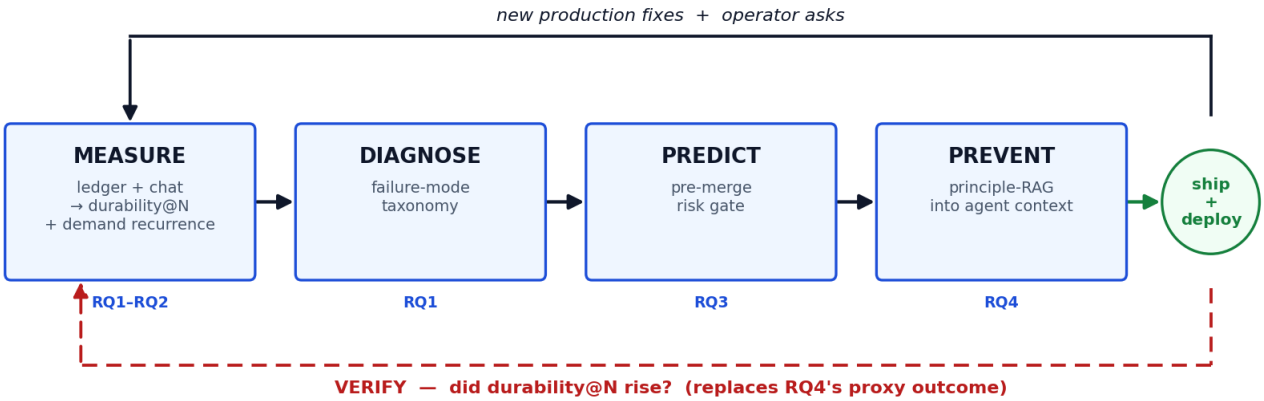

Figure 1: The DuraLoop framework. Each stage is an independent contribution (RQ tags) and feeds the next; shipped fixes and new operator asks re-enter Measure, and a Verify pass closes the loop by checking whether durability@N actually rose.

true after deployment. This reframing is small but consequential: it separates code production from delivery closure, and it lets a team ask whether repeated work is actually a regression or merely unfinished scope being reported again.

Repeated work has five states in this framework. **Never delivered** means the team closed or deferred the request without any deployed behavior that satisfies the intent. **Partial delivery** means a narrow slice shipped, but the acceptance surface was broader than the patch. **Delivered then regressed** is the classic recurrence: the desired behavior was live and later broke again. **Clarification** means the second request changes or sharpens the original intent rather than proving failure. **Durable delivery** means the state remains solved through the observation window. These states matter because they require different controls. Treating all repeated asks as regressions causes wasted testing effort; treating true regressions as vague feature churn hides preventable engineering failure.

The dominant technical mechanism we observed is incomplete generalization. The agent solves the visible site of the complaint but does not lift the fix to the invariant that makes the complaint possible. Four variants recur. In a site-only fix, the reported component or endpoint is patched while sibling sites keep the same bug. In an unpinned contract, a consumer assumes a producer shape, permission rule, path, or external payload without asserting it at the boundary. In a missed invariant, the regression test replays the symptom but does not encode the rule that must remain true across future refactors. In missing fleet memory, the next agent starts cold and repeats a lesson already learned by another session. These are not primarily model mysteries. They are failures of state, context, and control flow around the model.

The control logic follows directly from the state model. False closure needs a delivery-state ledger, because a request should not become done merely because code was written. Partial delivery needs explicit acceptance criteria, because the agent needs the boundary of the user intent before it can preserve it. Delivered then regressed needs invariant tests and sibling-site search, because a symptom replay can pass while the class of bug survives. Contract drift needs pinned producer and consumer boundaries, because downstream code is only durable if the upstream shape is asserted where it changes. Missing memory needs principle retrieval, because a fleet cannot learn if every session starts from isolated context. High blast radius needs a gate, because shared packages, auth, migrations, tenant flags, schedulers, and full-stack contracts create larger recurrence surfaces than leaf edits.

The concept map is therefore: a user report expresses intent; the delivery-state ledger classifies the current state; the risk gate estimates how widely the patch can propagate; the memory layer retrieves similar prior failures and robustness principles; the agent edits under invariant and contract constraints; CI and review check local correctness; deployment evidence decides whether the intent is live; and durability observation asks whether the intent stayed live at 7 and 14 days after deployment. Patch correctness sits inside this map as one checkpoint. It is not the finish line.

This framework also explains why an operations dashboard matters. The model is not asked to remember everything. Instead, an external control plane stores states, evidence, summaries, prior lessons, release facts, and post-deploy observations. In our implementation, 0x3admin and its DeepSeek-backed services provide that control plane, and the design choice that matters most is enforced in code rather than left to the agent: a feature ledger advances through an explicit state machine (`REQUESTED` → `IN_PROGRESS` → `CLAIMED_SHIPPED` → `VERIFIED`, with `BROKEN` and `STALE` branches), and the `VERIFIED` transition is **mechanically gated** so that only a smoke, CI, or webhook actor may certify a fix as delivered. An agent cannot mark its own work verified. A `VERIFIED` capability whose smoke evidence ages past a freshness window decays to `STALE` and loses its do-not-redo protection, which is the delivery-state expression of durability decay. Around this ledger, chat and context retrieval prepare the agent, transcript ingestion records what happened, a file-level risk scorer ranks edits by blast radius before work starts, learning tables distil session patterns, release sync connects commits to deployed changes, and recurrence/demand observation closes the loop. DuraLoop is the operational form of this conceptual model: move

success from patch written to intent still delivered, and make the "still delivered" judgement mechanical, not self-reported.

4 STUDY DESIGN

4.1 Setting

A live multi-tenant B2B SaaS maintained by a small team and a fleet of LLM coding agents under an explicit ship protocol (branch → MR → CI → merge → deploy → verify). We intentionally avoid stack, tenant, module, and vendor specifics in the released manuscript and artifacts because this is a named industrial case study rather than a double-blind anonymous benchmark. Process rules require that every shipped fix append a ledger row (root cause, fix, regression test, status), that every repeat report append a recurrence-log entry with a first-principles cause analysis, and that recurrences distil a generalised principle.

The agent fleet is not a controlled comparison of model families. Operational logs indicate heavy use of Opus-family models via Cursor, with model labels taken directly from IDE/runtime logs, and with other model families used for specific replay/judging and coding tasks. We therefore treat the model mix as part of the production setting, not as an experimental factor; per-model durability is deferred because the released fix-level dataset does not robustly attribute every shipped patch to a model.

4.2 Data and extraction

A stdlib parser converts the three markdown ledgers into structured records and joins `recurrence` → `prior fix` → `principle`. Tenant, operator, vendor, path, and workspace identifiers are pseudonymised or generalised at release time; a leakage scan checks known private tokens across released artifacts. Chat effort is measured by scanning transcript JSONL for tokens, tool-use blocks, and turns.

For the demand side (RQ2) we extract every `<user_query>` turn from the transcripts, drop pasted stack-traces and agent-template boilerplate, and cluster near-duplicate asks (Jaccard + sequence similarity) into **request clusters**, recording how many times each was asked (`ask_count`). Each cluster's representative text is **redacted first, then theme-classified** into one of 17 curated themes plus an ask-type, under a written codebook and **validated against an author-adjudicated gold standard** (§4.7), replacing the brittle keyword taxonomy that left 52% of clusters uncategorised. **Rectification** is inferred conservatively: a cluster is `done` only when ≥3 of its distinctive tokens overlap a work-notebook line within two lines of `MERGED`/`deployed`/`live`; otherwise `partial`, `open`, or `recurring_open` (asked 2+ times, still no completion match). This is a lower bound on delivery, deliberately mirroring the conservative `recurred_forward` defect label.

We also code a separate lifecycle flag, `reask_justified`, for repeated demand in the author-adjudicated gold sample. It distinguishes **never delivered**, **partial delivery**, **delivered then regressed**, and **clarification / scope refinement**. We report this flag on the gold sample only: it is the right conceptual split for the paper, but we do not scale it to all 600 clusters until a dedicated classifier for this lifecycle label is validated. That guardrail avoids turning a useful distinction into an over-claimed population estimate.

4.3 The durability@N metric

For a fix `f` shipped at `t0`, `durability@N(f)=1` if the shipped fix does not recur within N days after deployment, else 0. Fleet durability is the Kaplan–Meier [34] survival function over fixes, right-censoring fixes shipped too recently to observe for N days. It is the dual of pass@1: pass@1 asks "did it pass once?"; durability@N asks "did it stay fixed?". The five DuraLoop stages (Fig. 1) operationalise this end to end; §7 details each.

4.4 Labels, features, and taxonomy coding

The primary label is `recurred_forward = 1` iff a later recurrence-log entry cites the fix's bug ID (this is the Kaplan–Meier event). We also report `recurred_any` (forward or self-flagged) as a sensitivity. **Fix-time features** carry no hindsight: stream (FE/BE/full-stack), file count, regression-test present, root-cause and fix text length, blast-radius annotated, and keyword flags for *defensive catch*, *mock*, *shared package*, *schema/field*, and *deploy/pipeline*. Recurrence and principle fields are excluded. Each recurrence is assigned a primary **failure mode** by a documented keyword-scored rubric over its title, "why-back" analysis, family tag, and cited principle, and is reported at two granularities: a fine **9-way** sub-type and a coarse **3-way** "kind of gap", both validated against an author-adjudicated gold over all 92 recurrences (§4.7, §5.1).

4.5 Statistical analysis

Kaplan–Meier for durability@N; Fisher's exact test for 2×2 feature/recurrence associations, each reported with a **Wilson 95% CI** on the rates and a **Woolf 95% CI** on the odds ratio, and **Benjamini–Hochberg FDR** correction (q-values) across the eight-feature family to control multiple comparisons; logistic regression with repeated stratified 5-fold CV (×10) for prediction, reporting ROC-AUC with a 95% CI. Because every forward recurrence occurs within 13 days, a fix observed ≥14 days is **fully observed**; **87 of 300 fixes (29%) have <14 days of follow-up and are right-censored**, so durability@14 reads on the fully-observed set and we re-run key associations on it to remove censoring confounds.

4.6 RQ4 intervention protocol

We index the 169 principles with TF-IDF and, for each of **25** recurrence cases (the full eligible set, 25 of the 26 recurrences with both a reconstructable naive report and a known durability invariant), build a *naive bug report*: title and symptom only, no post-hoc root cause. The agent then proposes a fix under three arms: **control** (report only), **placebo** (report + 3 *random* principles), and **treatment** (report + the top-3 principles retrieved from that report). The placebo arm controls for the obvious confound: that *any* extra context makes the agent more thorough. Each fix is scored by an **independent, arm-blind** judge (one fix at a time, avoiding paired-contrast bias) under **two judge families**: the same DeepSeek model that produced the fix, and a **independent-model** OpenAI `gpt-4.1` judge that cannot share the fixer's blind spots. All runs use repeats=3 with per-case majority vote to damp generation stochasticity.

Artifact	Count
Production bug fixes	300
Logged recurrences	92
Distilled robustness principles	169
Fixes shipping a regression test	282 (94%)
Agent chat sessions	935
Est. chat tokens / tool calls	63.2M / 201k
Mined feature-request turns	924
Distinct request clusters (RQ2)	600

4.7 Qualitative coding and label validation

RQ2's themes are not trusted from a single model pass. We follow LLM-assisted qualitative coding with explicit reliability checks [45]:

1. **Codebook.** A written manual (released) defines the 17 themes, 5 ask-types, the delivery-lifecycle `reask_justified` flag, and the 9 RQ1 failure modes, each with a definition, decision rule, and examples, plus a subject-over-verb tie-break.
2. **Gold sample.** A **stratified sample of 150 clusters** is drawn across predicted theme × ask-count × delivery status, balanced ~50/50 re-asked vs once-asked so the estimate is not dominated by either; only redacted text is exposed.
3. **Coders.** The **authors adjudicate a gold standard** for all 150 items directly from the redacted text and the codebook, drawing on full knowledge of the system's modules; this reference is independent of the production classifier's model and prompt. Two further **codebook-prompted LLM coders from different model families** (OpenAI `gpt-4.1`, DeepSeek) label the same items blind to each other and to the gold. The author coding is itself LLM-assisted but author-adjudicated; a fully independent multi-human panel is the stronger future validation (§8).
4. **Reliability.** We report pairwise **Cohen's** κ and percent agreement among all coders, and, against the author gold, per-theme precision/recall/F1, accuracy, and a confusion matrix for the production classifier (the one scaled to all 600 clusters) and each LLM coder. The classifier's **measured accuracy vs the gold** is the stated confidence bound on the RQ2 distribution, reported separately for theme (noisy) and ask-type (robust).

This makes the RQ2 distribution auditable: a reader can re-run the coders, re-derive κ, and see exactly which theme boundaries the classifier and the gold disagree on (§5.2, Fig. 7).

5 RESULTS

5.1 RQ1: Recurrence is common, fast, customer-caught, and survives tests

One in seven fixes recurs. 44 of 300 fixes (**15%**) have a logged forward recurrence, a lower bound because it requires the recurrence log to cite the ID; **37%** of fixes are involved in a recurrence *family*. The Kaplan–Meier curve (Fig. 2) gives durability at 1 day = **0.88**, at 7 days = **0.84**, and at 14 days = **0.82**, and is flat thereafter: **every recurrence in the corpus occurred within 13 days.** Fixes that fail, fail fast.

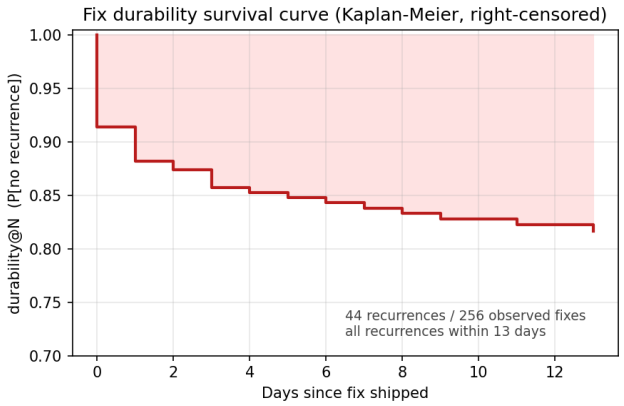

Figure 2: Fix-durability survival curve (durability@N), Kaplan-Meier with right-censoring.

This is an operational truth from an instrumented workflow, not a perfect population estimate. A recurrence must be noticed, recorded, and linked back to the earlier fix. The point is that the lower bound is already large enough to make merge-time pass rates inadequate as the sole success metric.

The safety net is the customer. 75 of 92 recurrences (82%) were caught by an end user, not by tests, telemetry, or CI (Fig. 3).

Tests do not confer durability. 94% of fixes shipped a regression test, yet **15.6%** of *tested* fixes recurred and **54%** of recurrences shipped with one. The only fixes that never recurred were the 18 trivial, untested ones (0%); in the fully-observed subset the tested-fix recurrence rate rises to **27.5%**. A test pins the symptom that was understood, not the invariant that was missed.

Most recurrences are incomplete generalization (Fig. 4). We validate the failure-mode coding against an author-adjudicated gold over all **92** recurrences (§4.7) and report only what survives that check. The reliable signal is the *coarse* mechanism: **incomplete generalization**, the agent fixed the reported *site or instance* without lifting the fix to the shared root (a pattern across sibling call-sites, or an unpinned contract/boundary), accounts for **70%** of recurrences by the author gold and **66%** by the keyword rubric, an agreement that is robust to coder (coarse three-way agreement **63%**, this bucket's F1 = **0.75**). The remainder splits into **diagnosis/intent gaps** (layered symptom-vs-root-cause, intent mismatch; **18%**) and **verification/deploy gaps** (test-in-isolation,

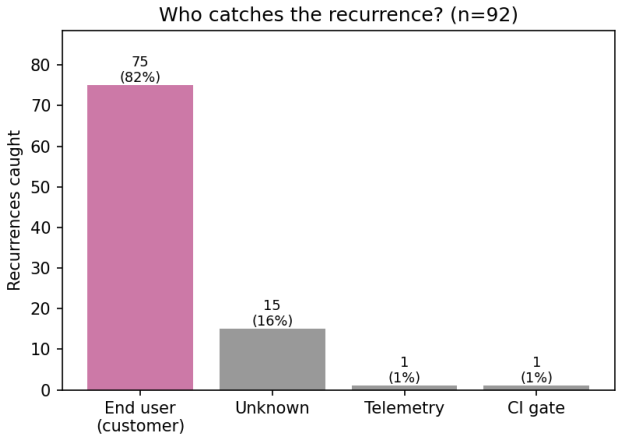

Figure 3: Who catches the recurrence: end users, not tests, telemetry, or CI.

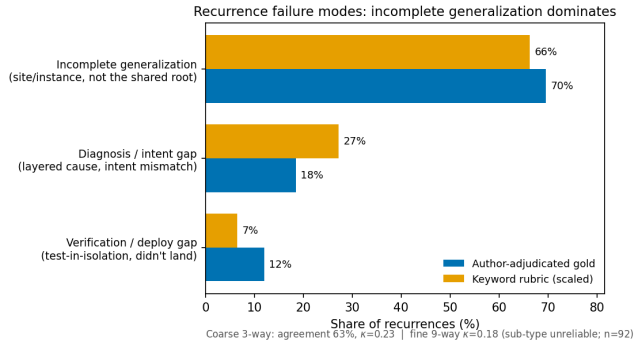

Figure 4: Recurrence failure modes, author-validated. The coarse "kind of gap" axis is reliable: incomplete generalization (fixing the site, not the shared root) dominates at 70% (author) / 66% (rubric); the finer 9-way sub-type is not separable (kappa=0.18). n=92.

deploy-didn't-land; **12%**). The *finer* question, whether a given case was specifically an *unpinned contract* or a *one-site patch*, is **not reliably separable** (fine 9-way κ = **0.18**, accuracy 0.34): rubric and author agree the case belongs in the incomplete-generalization bucket but redistribute it between the two sub-labels. We therefore anchor the claim on the dominant coarse mechanism and do **not** lean on per-subtype rates.

The recurrences are real, not a logging habit. A fair objection is that the recurrence log and this analysis are one artifact, so "recurrence" might be a bookkeeping reflex rather than a repeated edit to the same code. We test the ledger against an independent source, the actual git history of the two product repositories, in two ways. *Provenance:* of the commit identifiers the ledger claims it merged, **195/218 commit SHAs (89%)** and **374/375 merge-request references (99%)** resolve to real commits in git, so the entries describe changes that genuinely shipped. *Re-touch:* for bug pairs the ledger groups as one recurring issue, we ask git, not the ledger, whether their fix commits touch overlapping files. They do in **37 of 77 evaluable pairs (48%)** versus **2.5%** for random distinct-date bug pairs. This is a **19× enrichment** (Fisher odds ratio 36, $p<10^{-6}$; median gap 1 day). Fixes linked by recurrence independently land on the same files far more than chance, which a fabricated or over-eager log would not produce (`git_anchor.py`; counts are a snapshot of the live ledger and drift by a few as the log grows, but the magnitudes are stable).

5.2 RQ2: Re-asked work is a delivery-state problem, not just recurrence

The defect log is only one channel. The prompts themselves show what operators keep coming back for. Mining every request turn yields **600 request clusters from 924 ask turns** (Fig. 5a). **238 clusters (40%)** are asked two or more times, up to **8×**. That is **2.6× larger** than the 92 logged defect recurrences, and it comes directly from user demand rather than from an agent filing the right ledger row.

The important correction is that a re-ask is not automatically a regression. In the author-adjudicated gold sample, **77 re-asked clusters** have a lifecycle label (Fig. 6): **32 (42%) were never delivered**, **38 (49%) were partial delivery**, and only **7 (9%) were delivered and then regressed** (no cluster in this sample coded as pure clarification). This changes how we should code. For never-delivered work, the fix is scoping and backlog closure. For partial delivery, the fix is an acceptance checklist. For delivered-then-regressed work, the fix is invariant testing and durability tracking. Treating all three as "recurrence" would prescribe the wrong control.

What this means for coding. Every repeated report should be triaged into a delivery state before an agent edits code:

The scaled demand corpus still matters, but now we read it correctly. Only **42% of all asks** reach a done state; **441 clusters (48%)** sit `open` or `recurring_open`. Among the 238 re-asked clusters, **56% are still unrectified**. Persistence does not buy delivery.

The misses concentrate where the specification is vague. We report this only as a directional signal because theme labels have moderate reliability (§4.7, Fig. 7). Well-scoped, schema-backed asks are more often rectified; open-ended judgement work is less often rectified. By ask-type, only **65 of 600** clusters are net-new features; the rest are **enhancements (236)** and **defect/complaints (204)** about work already attempted. The problem is not that operators only ask for new things; it is that attempted work often lacks a durable definition of done.

Rather than reproduce domain-specific prompts, Table 2 shows the pattern after operational details are generalised.

How trustworthy are these labels? The lifecycle split above is reported on the author gold only; we do not scale it to all 600 clusters until that specific flag is validated. Theme and ask-type are validated separately (§4.7, Fig. 7). The production classifier and two independent codebook-prompted coders agree substantially with one another on 17-way theme (κ **0.60–0.75**), but only moderately with the author-adjudicated gold: the scaled classifier reaches **58% accuracy** (κ **0.55**) and the best codebook coder **63%**. Ask-type is stronger: **77%** accuracy, κ **0.67**. The safe reading is that theme

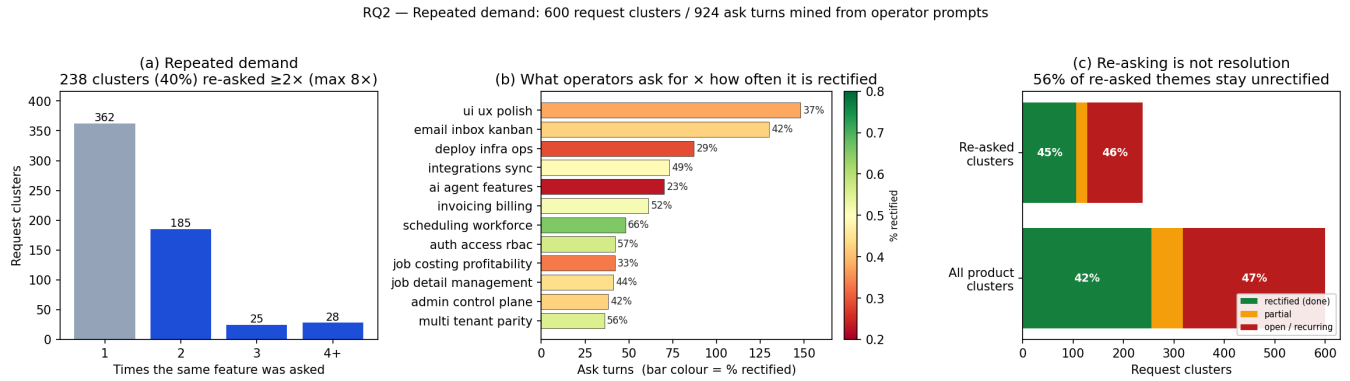

Figure 5: RQ2 repeated demand. (a) how often each ask is repeated; (b) demand volume per theme coloured by rectification rate; (c) re-asked themes stay unrectified.

Repeated-report state	What it usually means	Coding control
Never delivered	The ask never landed despite being discussed	make a crisp acceptance checklist; close it only on deployed evidence
Partial delivery	A visible slice shipped but the real intent is still missing	list the missing sub-cases before patching; test the complete workflow
Delivered then regressed	A shipped behavior broke later	write an invariant test and search for the refactor/contract that dropped it
Clarification / refinement	The target evolved	split a new requirement; do not count it as a failed fix

Repeated-ask pattern	Typical status	What the agent missed
"You implemented the visible slice, but not the full calculation / workflow"	partial delivery	acceptance criteria and end-to-end invariant
"Again, use the first-principles rule we already wrote for this family"	recurring_open	retrieved organisational memory
"Match this existing screen / behaviour exactly"	recurring_open	sibling-site search and canonical UI primitive
"This query/sync/process keeps failing at production scale"	recurring_open	production-shaped verification, not local happy-path testing
"The data is technically present but surfaced in the wrong state or grouping"	done or partial	state model / classification boundary

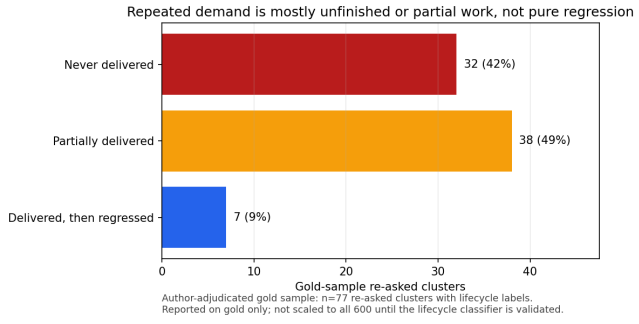

Figure 6: Delivery lifecycle of repeated demand in the author-adjudicated gold sample. Most repeated asks are unfinished or partially delivered work, not pure delivered-then-regressed recurrence.

counts are approximate, while `ask_count`, rectification status, and the gold lifecycle split are the safer operational signals.

5.3 RQ3: Recurrence is predictable at fix time

The useful question is not "can we predict every recurrence?" It is "which patches deserve deeper review before merge?" A logistic model over fix-time features predicts forward recurrence with **ROC-AUC 0.72 (95% CI 0.59–0.84)** under repeated stratified CV (44 positives / 300). That is enough for routing, not enough for automation. Univariate associations are reported with Wilson rate CIs, Woolf odds-ratio CIs, and Benjamini–Hochberg q-values across the eight-feature family (Fig. 8).

After multiple-comparison correction, only blast-radius survives ($q<0.01$); shared-package, regression-test, and defensive-catch do not ($q\approx0.30$), so we treat them as descriptive rather than established markers. The regression-test "no" cell is a small-n comparison (n=18, wide CI), not a zero-risk claim. This sharpens the practical takeaway: the one robust, *pre-merge-available* signal is

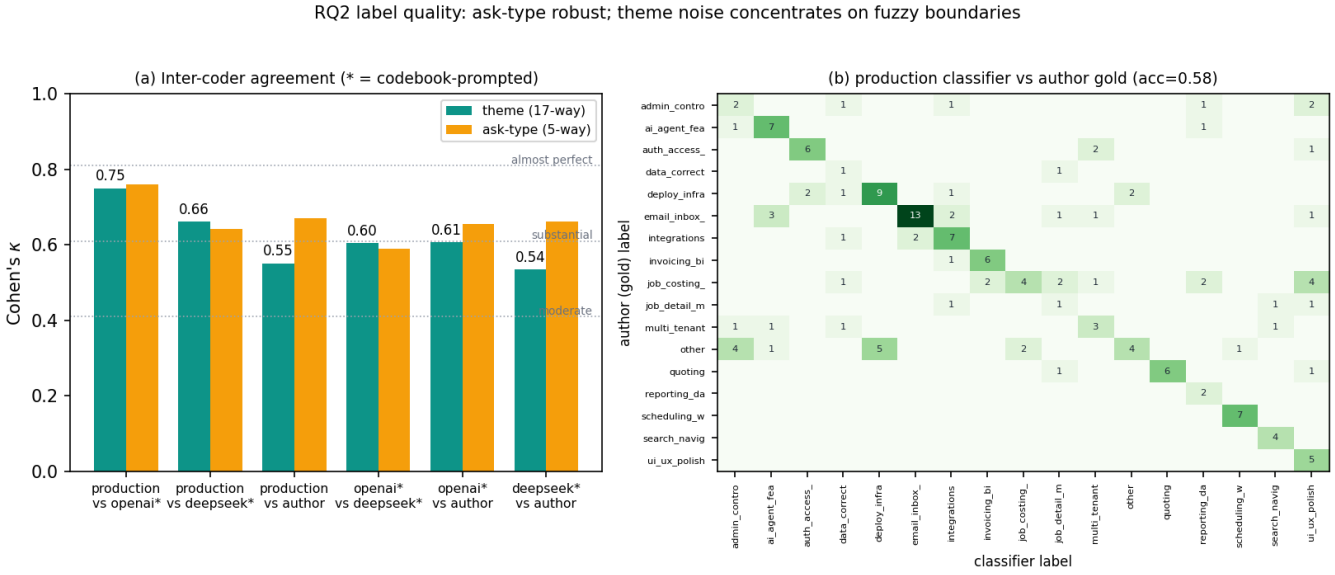

Figure 7: RQ2 label quality. (a) pairwise Cohen's kappa for theme (17-way) and ask-type (5-way): the model coders agree substantially with each other but only moderately with the author-adjudicated gold on theme, while ask-type stays higher. (b) confusion of the scaled production classifier vs the author gold (accuracy 0.58); the strong diagonal marks reliably-labelled concrete domains, the off-diagonal cells locate fuzzy boundaries.

Fix-time feature	Recurs (present)	OR [95% CI]	p	q (BH)
Blast-radius annotated	46.7% [30–64]	**7.0 [3.1–15.4]**	<0.001	**<0.01**
Touches a shared package	22.9% [13–37]	1.97 [0.95–4.3]	0.12	0.30
Has a regression test	15.6% vs 0% (n=18)	not estimated [0.4 to 117]	0.086	0.30
Defensive catch present	9.5% [5–18]	0.53 [0.24–1.3]	0.19	0.30

the agent's **own blast-radius annotation**, exactly what a durability gate should act on, and the logistic model's modest lift is largely carried by it. That the strongest marker is the agent's self-declared risk is convenient, not circular: it is recorded before the recurrence exists. It is also not evidence that a gate failed: during the observation window the durability gate was not yet enforced, so blast-radius annotation marks the *intrinsic* difficulty of a change rather than the effect of a control. An annotated change recurs more because cross-cutting work is genuinely harder to make durable, which is exactly why it is a useful signal for *routing* the gate we propose in §7.

What this means for coding. We should not add heavyweight review to every agent patch. We should add a mandatory durability gate when the change is any of: shared-package, full-stack contract, auth/RBAC, tenant-specific configuration, migration/data-shape, scheduler/celery, or cross-module calculation. The gate asks for exactly three checks: sibling call-site search, contract/invariant test, and principle retrieval. This is cheaper and more targeted than treating all fixes as equally risky.

Positioning. AUC 0.72 is broadly comparable to reopened-bug and software defect-prediction models, which commonly report AUCs around 0.7–0.85 [27, 44, 62], but those use richer *post-hoc* process and code metrics, whereas ours uses only coarse metadata

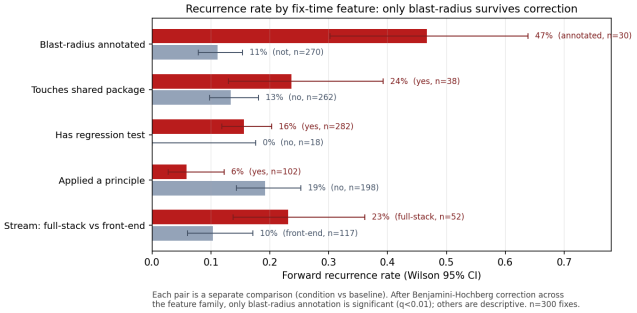

Figure 8: Recurrence rate by fix-time feature (RQ3), each a condition-vs-baseline pair with Wilson 95% CIs. Only blast-radius annotation survives Benjamini-Hochberg correction.

available *before* merge, and we apply Benjamini–Hochberg correction, still uncommon in a literature where evaluation consistency is itself a recognised problem [41]. The contribution is therefore not a stronger classifier but a pre-merge, durability-targeted one that a gate can act on.

5.4 RQ4: Principle retrieval is a proxy feasibility result

Observational. Fixes whose ledger row records an *applied* robustness principle recurred far less: **5.9% vs 19.2%** (OR 0.26, p=0.002). This is confounded: principle-logging is newer, so such fixes have less follow-up, and agents cite a principle when they already recognise a risky pattern. Restricting to fully-observed fixes the effect attenuates to **14.6% vs 29.7%** (OR 0.41, p=0.066). We treat the observational result as a hypothesis, not a result, and test it directly.

Interventional (three-arm replay). This is a feasibility probe, not live deployment validation. The retrieval premise holds first: the applicable principle is in the **top-3 retrieved 75%** of the time (top-5: 85%) from the bug report alone (Fig. 9, right). At this corpus scale (169 principles), lexical TF-IDF is already sufficient to seed the loop and keeps retrieval transparent and cheap; dense or hybrid retrieval [21, 61] is the natural scaling path as the principle base grows, and we leave that comparison to future work. On the replay (n=25, repeats=3):

Treatment beats the no-principle control under **both** judges; the independent-model judge, which removes the shared-model blind spot, makes it **significant** (**40% vs 16%, all 6/6 discordant pairs favour treatment, p=0.031**). The same-model judge agrees in direction but is underpowered (it marks most fixes non-durable in *both* arms). The placebo result marks the boundary of the claim: random principles do not beat control under DeepSeek (0% vs 8%) but sit slightly above it under gpt-4.1 (24% vs 16%), so at n=25 we cannot prove the gain is strictly *relevance*-driven rather than "principle-shaped context helps." We claim only what the data supports: **structured principle context improves judged invariant quality over no-principle context in this replay; relevance-specificity and live durability remain open.** The outcome is a proxy: an arm-blind judge of invariant satisfaction, not live recurrence (§8).

What this means for coding. The intervention suggested by the replay is practical: before the agent edits, retrieve a few prior lessons for the same bug shape and put them in the prompt. The principle does not need to be long. It needs to remind the agent of the invariant it usually forgets: search the sibling sites, pin the contract, verify the real deploy path, or avoid swallowing the failure. That is the fleet's missing shared memory. The live claim is deliberately deferred: the next validation step is to run this gate prospectively and measure whether durability at 7 and 14 days improves.

5.5 RQ5: The illustrative recurrence tax is human time and trust

A transparent illustrative cost model (12× billed-token multiplier, $5/1M tokens, $80/hr eng, 0.75 hr triage per recurrence, $200 per user-caught incident) puts the observed-window recurrence tax at **~$20,600**: AI tokens are **$36 (0.2%)**, engineer time **~$5,600 (27%)**, and customer-visible incidents **~$15,000 (73%)** (Fig. 11). The lever is not cheaper models; it is fewer human re-fixes and fewer customer-visible repeats. The dollar amount should be read as a tunable business-case calculation, not measured finance data.

This is a one-sided ledger by construction: it counts the cost of recurrence but not the cost of running DuraLoop. The framework has its own overhead, extra review on gated changes, principle upkeep, and the durability-observation infrastructure, and a poorly-calibrated gate can spend more senior review than it saves by routing low-risk patches for deep review. Because the gate is currently heuristic (§7.3), we do not claim a net saving; we claim that the recurrence tax is large enough, and concentrated enough in human time and customer trust, to justify building and then measuring the controls. The net-ROI number requires the calibrated gate and the closed-loop durability@N deployment, which is why we frame it as the next study (§8) rather than reporting a savings figure here.

6 WHAT ACTUALLY WORKED, AND WHY

The results point to a small number of coding controls, and they are useful because each maps to a failure mode we observed, not because they are generic "best practice" advice. A single-shot objective rewards locally plausible code: make the failing path pass, satisfy the visible test, and move on. Production durability needs something stronger, because the patch must survive future states of an evolving repository. This is the deployment-time version of patch overfitting [54, 59], which frozen benchmarks cannot express [32]. The dominant mechanism is incomplete generalization, sometimes an unpinned contract (an API shape, schema, mock, or tenant setting the fix used but did not assert) and sometimes a site-only patch that misses the sibling call-sites; both are the durability siblings of API hallucination and shallow fault localisation [42, 73]. The fix is information flow, not model capability: give the agent the missing production context, the shared pattern, the contract boundary, the acceptance criterion, and the organisation's prior lessons, before it edits. We formalize that into the operational framework of §7.

7 OPERATIONAL ADOPTION FRAMEWORK FOR AGENTIC CODING TEAMS

For a coding team using Cursor-style agents, or a research team building its own autonomous repair system, the highest operational leverage is not another model swap. It is changing what counts as "done" and giving the agent the missing production context before it edits. DuraLoop should be adopted as a waterfall of controls, from cheapest and highest-confidence to heavier instrumentation. We describe each stage as a small input-to-output transform with an explicit artifact and signal, and we mark whether the stage is **implemented** in our control plane (§7.1) or **proposed** as the next step, so the reader can separate what we run from what we recommend:

The stages run cheapest-first, and each is an input-to-output transform with a durable artifact and an emitted signal so a team can adopt it incrementally and audit it later. "Implemented" marks what runs in our control plane today (§7.1); "partial" and "proposed" mark the implementation upgrades tracked in §7.3. The ROI logic is sequential. A team that adopts only the first two steps already reduces wasted agent cycles and review effort. Steps three and four reduce repeated defects. Step five improves agent context quality. Step six turns the loop into a measured operating system rather than a checklist. The scope of the evidence is that the *gate* (RQ3) and the *prevent* step (RQ4, proxy) are tested; the loop as a

Arm	Same-model judge (DeepSeek)	Cross-model judge (gpt-4.1)
Control (report only)	8%	16%
Placebo (random principles)	0%	24%
Treatment (RAG principles)	**16%**	**40%**
Treatment vs control (McNemar)	3–1, p=0.63	**6–0, p=0.031**
Treatment vs placebo	4–0, p=0.13	5–1, p=0.22

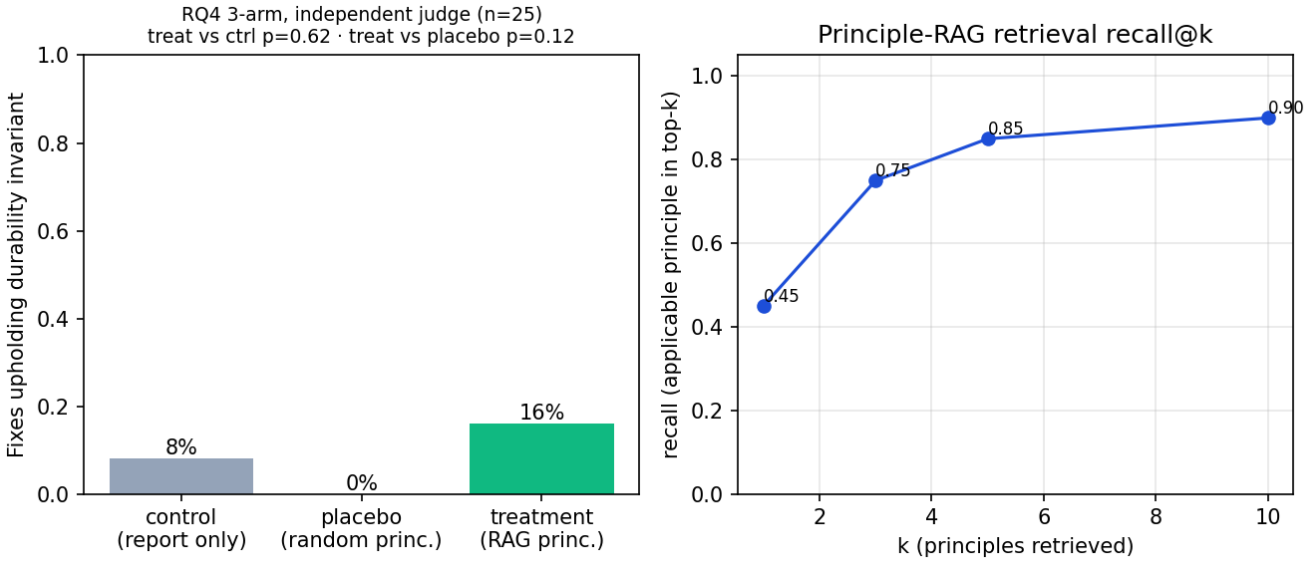

Figure 9: RQ4 three-arm replay (same-model judge) and retrieval recall@k.

Stage	Agent action	Artifact	Signal	Status
Stop false closure	Classify delivery state	feature ledger row	never/partial/regressed/clarify/durable	implemented
Gate blast radius	Escalate shared/high-risk edits	preflight record	blast-radius score, GO/BLOCKED	heuristic
Test invariant	Write the invariant test	test / acceptance check	pass/fail across siblings	partial
Pin contract	Assert producer/consumer boundary	fixture / type / API test	contract-drift flag	proposed
Retrieve lessons	Inject top-k principles	prompt context	retrieval recall@k	implemented
Measure durability	Re-observe at 7/14 days	durability row	durable/regressed, durability@N	implemented

whole is operated but not yet validated end-to-end by a controlled durability@N deployment (§7.3, §8).

7.1 Reference architecture: an implemented control plane

The architecture is a production control plane around the existing agent loop (Fig. 13), not only a paper design: most of it runs today in the 0x3admin operations dashboard. It works whether the team uses Cursor agents, a custom multi-agent orchestrator, or an internal repair bot. The key design choice is to make hidden state explicit and to move the one judgement agents are worst at, "is this actually done," out of the model: delivery state, risk state, retrieved memory, verification evidence, deploy evidence, and post-deploy durability all become records outside the agent.

Delivery state is a mechanically-gated state machine, not a label the agent sets. A feature ledger advances each capability through `REQUESTED` → `IN_PROGRESS` → `CLAIMED_SHIPPED` → `VERIFIED`, with `BROKEN` and `STALE` branches. The transition into `VERIFIED` is restricted to mechanical actors (a smoke test, CI, or a deployment webhook); an agent or a chat model is not an allowed actor for that edge. This is the architectural form of the paper's central finding: a green test is not durability, so "delivered" must be asserted by a deploy-time signal, not by the writer of the code. Two further rules encode durability directly: a `VERIFIED` capability whose smoke evidence ages past a freshness window decays to `STALE` and loses its do-not-redo protection, and for multi-tenant capabilities the aggregate state is worst-tenant-wins, so one broken tenant marks the whole capability broken. This

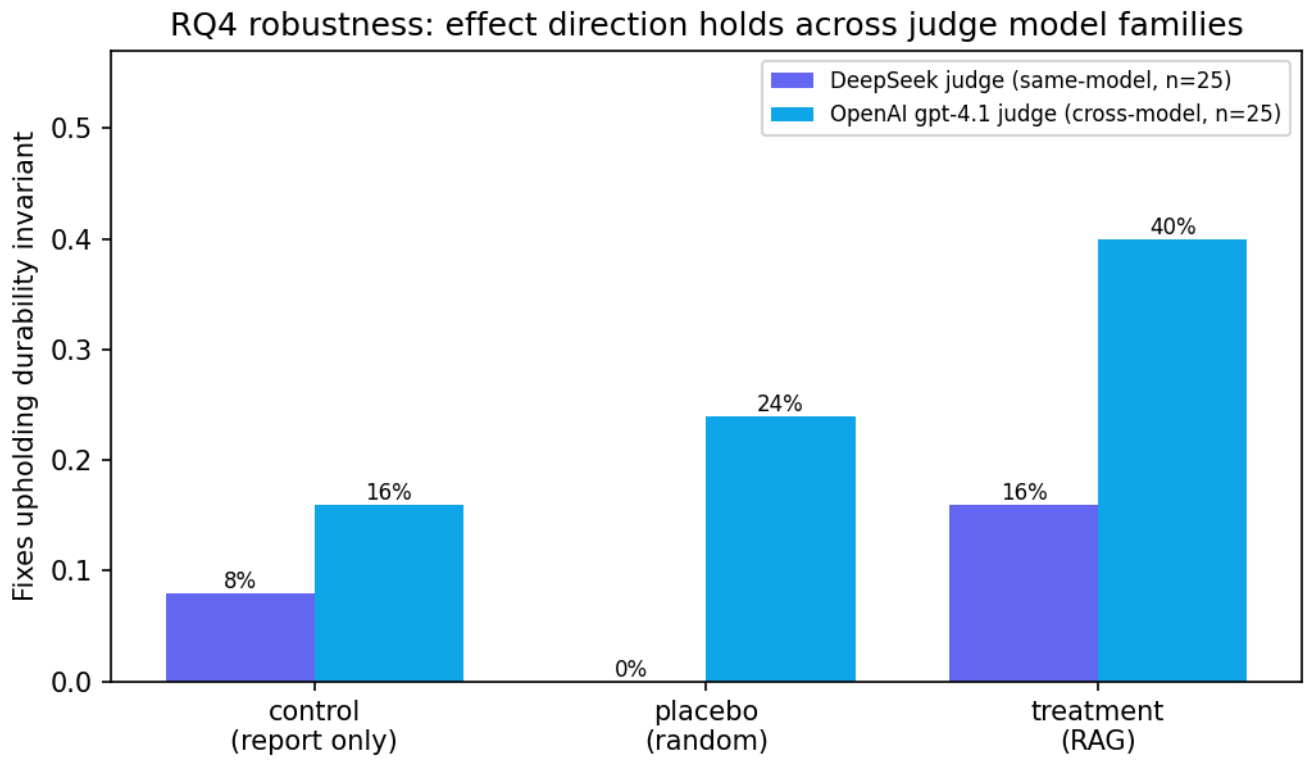

Figure 10: RQ4 independent-model robustness: treatment leads under both judge families.

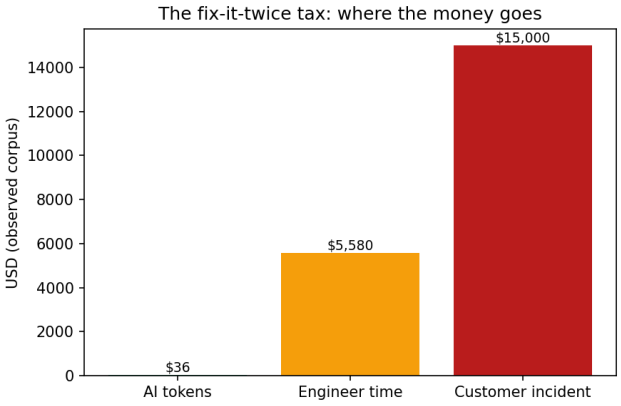

Figure 11: Where the fix-it-twice money goes (observed-window recurrence tax).

mechanical-verification stance is what the agent self-repair literature predicts is necessary: self-generated checks plateau because a model cannot test what it failed to conceive [16, 17, 49].

Pre-edit context and a risk score. An operator report enters AI Chat; the context layer scores the query, selects platform docs, API inventory, deployment guidance, code-health signals, and repo matches under a token budget, then sends an enhanced prompt through a DeepSeek chat proxy or the admin AI gateway. In parallel, a file-level intelligence layer scores every workspace file on deterministic axes (breakage risk, complexity, churn, centrality, contract drift, test coverage, god-file and line-limit flags, and bug/recurrence linkage) and ranks a refactor queue by `breakage × centrality × churn`. This gives the durability gate a quantitative blast-radius estimate, and a preflight check can return GO or BLOCKED before an agent claims a contested zone. The risk signal is real but heuristic; calibrating it against observed breakage is the open item in §7.3.

Multi-agent coordination. Because ten-plus agents edit one repository concurrently, the control plane includes a coordination ledger with zone claims that carry a time-to-live, a preflight gate, and a denormalized snapshot that tells the next agent what is already verified (do-not-redo), what needs attention, and what is blocked. This treats the fleet as a normative multi-agent system with shared state rather than independent sessions [14, 20, 24], which is exactly the regime where incomplete generalization across sibling sites is most likely.

Evidence capture and fleet memory. Cursor sessions and subagents ship redacted turn deltas into 0x3admin through an HMAC-protected ingest endpoint; each turn is keyed by session and turn index, content is hashed, retries are idempotent, and the session offset is monotonic. The backend stores raw turns, session

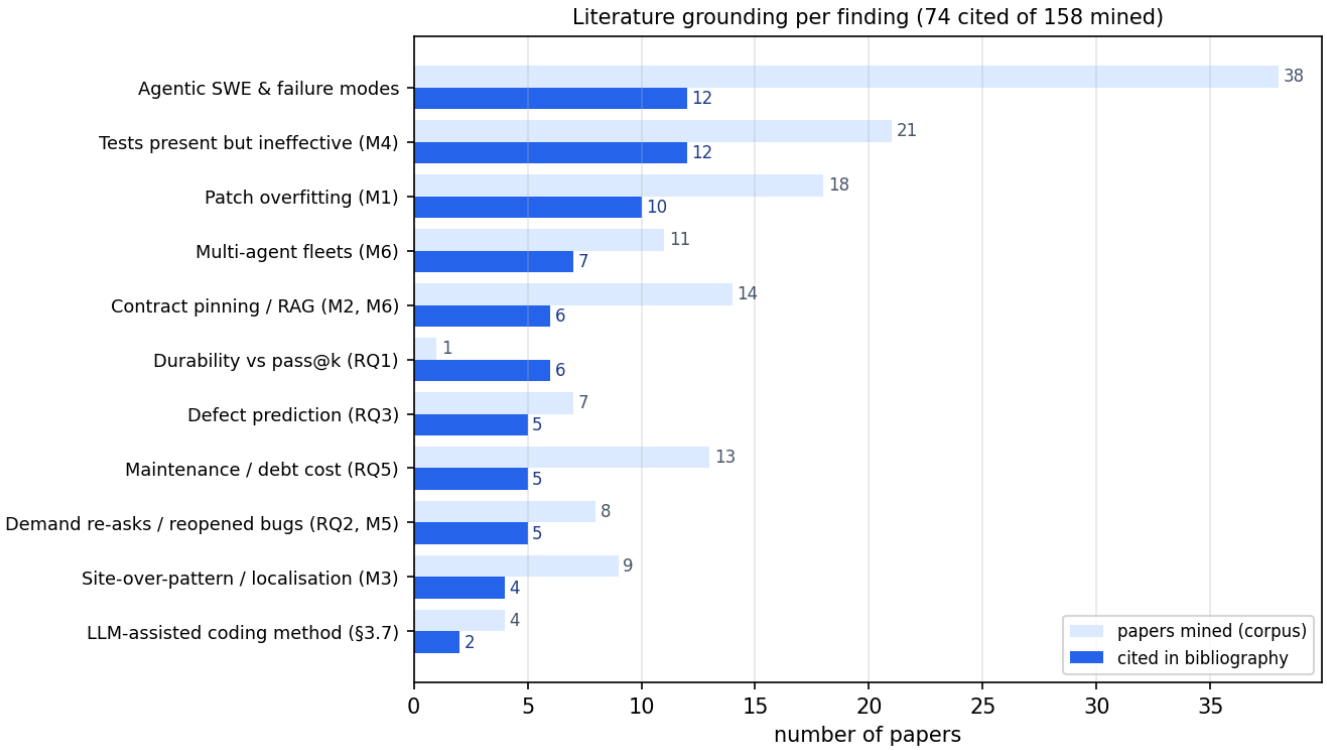

Figure 12: Literature map. Each DuraLoop finding (rows) is grounded in mined papers (bar = supporting papers in the curated bibliography); the corpus spans 158 downloaded papers across these lines of work.

summaries with an outcome label, module-level learnings, and promotable first principles in SQLite. This matters because fleet memory cannot live only in the next prompt; it needs a durable store that survives model changes, new agents, and concurrent sessions. Retrieval today is lexical TF-IDF over distilled principles [38, 61]; closing the loop so that a principle's retrieval weight rises or falls with whether it actually reduced recurrence is the memory open item in §7.3 [8].

Deployment closure and durability observation. GitLab release sync ingests production commits, deduplicates by SHA, writes product-release records, and asks DeepSeek for release summaries with a deterministic heuristic fallback. A feature-completion observer then recomputes, on a rolling 14-day window, whether each capability's acceptance criteria are still met, runs live HTTP probes against the deployed surface, and flags regressions (capabilities trending down). When a capability breaks or goes stale, the failure becomes a new principle for retrieval before the next risky patch. The verification evidence today is a deploy-time HTTP status probe plus the mechanical state machine; a deeper behavioural oracle is the verification open item in §7.3.

7.2 Implementation checklist

At the level of a single change, the table above becomes a per-edit checklist: classify the report before editing, write the invariant in one sentence, search the sibling sites, pin the contract at the boundary, gate blast-radius changes for deeper review, retrieve the top-k prior principles (the top-3 contains the applicable principle 75% of the time in this corpus), and close only on deployed evidence and a durability observation at 7 and 14 days, not when CI is green. The metric, harness, taxonomy rubric, principle-RAG retriever, lifecycle split, and git cross-check are released (Appendix A) so a team can adopt the pieces without rebuilding the research stack.

7.3 What is implemented, and what is still open

We separate the parts of DuraLoop that run today from the parts that are designed but not yet built or validated, because a framework presented as finished when it is not is exactly the kind of false closure the paper argues against. Each open item is a concrete next step with a literature direction.

The single most important open item is the closed loop: we operate every stage, but we have not yet run the controlled experiment that shows applying the loop *raises* durability@N on shipped work. That is the difference between a control plane that records the right things and one that is proven to improve the outcome. The other four items are implementation upgrades, the gate from heuristic

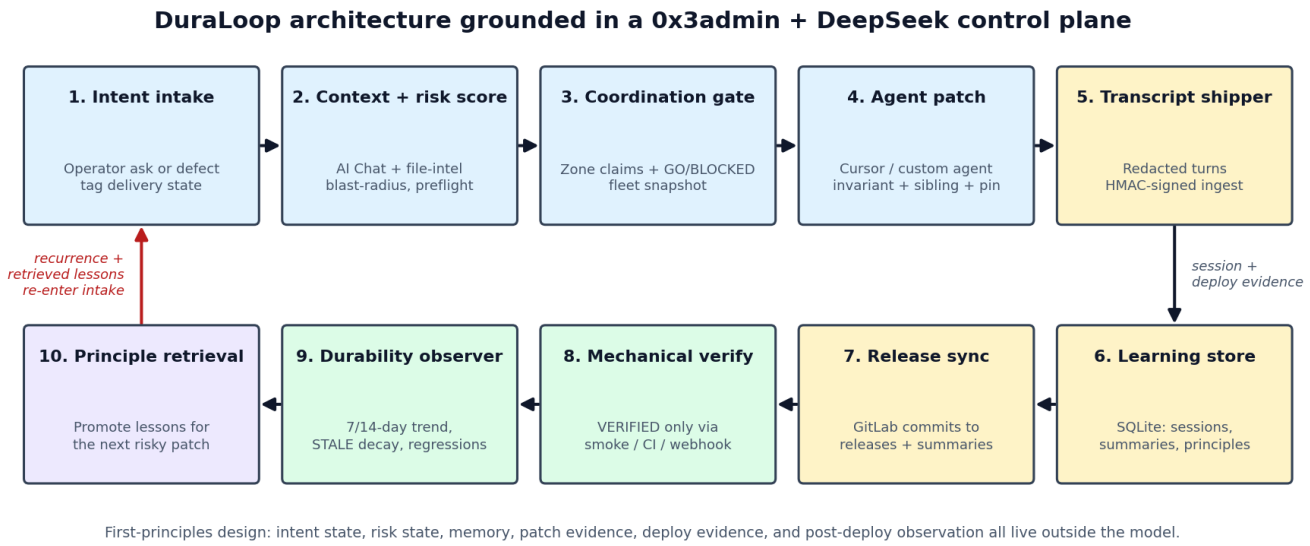

Figure 13: DuraLoop reference architecture. 0x3admin surrounds the agent loop with mechanical verification, risk scoring, transcript capture, principle memory, release sync, and durability observation.

Capability	Status	What is open	Direction
Mechanical verification (no agent self-certify)	**Implemented**	Probe is deploy-time HTTP status; not behavioural	property/metamorphic and mutation-adequacy oracles [9, 23, 35, 53]
Delivery-state ledger + 14-day durability observer	**Implemented**	Closed-loop A/B (did the loop raise durability@N?) not yet run	prospective durability@N deployment (§8)
Multi-agent coordination (claims, preflight, snapshot)	**Implemented**	Automated producer/consumer contract-sync is a heuristic flag, not a diff	normative-MAS contract synchronization [14, 20]
Risk gate (file-level blast-radius score)	**Heuristic**	Not calibrated against observed breakage; does not hard-block	calibrated, cost-aware automated review [28, 63, 66]
Principle memory (TF-IDF retrieval)	**Implemented (static)**	No reweighting by whether a lesson reduced recurrence	outcome-weighted / adaptive retrieval [8, 61]

to calibrated, the oracle from HTTP-status to behavioural, the contract pin from flag to diff, and memory from static to outcome-weighted, each of which the cited literature already shows is feasible. We also did not benchmark DuraLoop against simpler alternatives a team might reach for first, a stronger model, more self-repair iterations [6, 49], or an off-the-shelf automated reviewer [63]; that comparison is the next companion study.

8 THREATS TO VALIDITY

Construct. The RQ4 outcome is a **proxy**, a blind judge of invariant satisfaction, not live production recurrence. `recurred_forward` requires the log to cite the bug ID, so 15% is a lower bound. durability@N also does not separate two causes of recurrence: an *incomplete* fix (an agent-quality failure) and a later upstream change that breaks an otherwise-correct fix (codebase entropy that would also affect human-authored fixes). The unpinned-contract failure mode captures part of the latter, but the headline metric pools both, so durability@N is best read as a property of shipped maintenance work in this fleet, not purely as a measure of agent skill. We address **circularity**, that the log and the analysis share a source, with the ledger-independent git cross-check of §5.1: the re-touch overlap and timing are git-derived and independent of the log's narrative, while the provenance check still draws its identifiers from the ledger and so is only a consistency check. The failure-mode taxonomy is validated against an author-adjudicated gold over all 92 recurrences: the coarse "kind of gap" axis is reliable (incomplete generalization **66%** rubric vs **70%** author; coarse agreement 63%, bucket F1 = 0.75), but the fine 9-way sub-type is **not** (κ = 0.18, accuracy 0.34), so we report only the dominant coarse mechanism and avoid per-subtype rate claims; a multi-human panel remains the stronger validation. **Censoring:** 87 of 300 fixes (29%) have <14 days of follow-up, so durability@14 is computed on the fully-observed set; every observed recurrence fell within 13 days (hence the flat tail), but late recurrences beyond the window cannot be excluded. For RQ2, **rectification is a**

conservative notebook-match heuristic (it under-counts work shipped without a matching notebook line, so 42% is a floor), and the demand corpus is a **derived artifact**. The raw transcripts have since been rotated by the IDE, so the clustered/classified output is reproducible but the source turns are not re-extractable. The delivery-lifecycle split (`never_delivered`, `partial_delivery`, `delivered_then_regressed`) is therefore reported on the author-adjudicated gold sample only; scaling that flag to all 600 clusters requires a separate validated classifier.

Label quality and scaling. RQ2's theme labels are validated against an author-adjudicated gold: the model coders agree **substantially** with each other (κ 0.60–0.75) but only **moderately** with the gold (theme **58–63%** accuracy, $\kappa \approx 0.55$; §5.2), so per-theme counts are **approximate** and we treat them as such; ask-type is stronger (**77%**, κ 0.67). The gold is a **single author-adjudicated pass**, LLM-assisted but coded independently of the classifier's model and prompt and with full system knowledge, not an independent multi-human panel, so a full inter-human reliability study is the stronger future validation. We scale one classifier to all 600 clusters, so its **systematic** per-theme error (it over-labels UI/UX-polish and under-recalls job-costing) propagates to the distribution; we mitigate by releasing the labelled corpus and confusion matrix so any theme's reliability is inspectable, and by anchoring the headline on the robust ask-type and the theme-independent recurrence signal.

Literature coverage. Related work and the Discussion are grounded in a corpus of 158 open-access/arXiv PDFs mined programmatically across 21 topic seeds; this favours preprints and open-access venues and can miss paywalled or differently-phrased work, and arXiv versions may differ from published ones. We cite the 74 most relevant and disclose the seed-based sampling.

Internal. Univariate feature associations are corrected for multiplicity (Benjamini–Hochberg across eight features); only blast-radius survives ($q<0.01$), so we do not over-read the weaker associations (shared-package, test, defensive-catch; $q\approx0.30$). The observational principle effect is confounded by recency and selection (it weakens to $p=0.066$ fully observed). The replay addresses two threats: the shared-model-judge bias is mitigated by the **independent-model** judge (which *strengthens* treatment-vs-control to $p=0.031$), and the added-context confound is probed by the **placebo** arm. Generation is stochastic, so we use repeated sampling + majority vote rather than single runs.

External. Single product, one organisation, and one internal agent workflow; we therefore frame this as a grounded production case study / experience report, not a population estimate or universal benchmark. We make **no AI-versus-human durability claim**: we do not assert that agent-authored fixes recur more often than human-authored fixes would in the same system, because we have no human-authored control. The author-adjudicated coding is our only human reference, and it serves as a labelling proxy, not a comparison arm. The findings describe the durability of maintenance work in an AI-heavy fleet, not the relative quality of agents and people. The model mix is also contextual and **non-stationary** (§4.1): model versions shifted across the observation window, so the Kaplan–Meier curve pools fixes authored by an evolving system and time-since-fix cannot be cleanly separated from model-capability-at-fix-time. durability@N is therefore a property of the fleet-as-operated over the period, not of any single model version, and the released fix-level dataset does not support causal per-model comparisons.

Conclusion. At **n=25** the relevance-specific contrast (treatment vs placebo) is underpowered and the two judges disagree on how much random principles help. Crucially, n=25 is the *entire eligible universe* (25 of 26 recurrences with a reconstructable naive report and a known invariant), so power here is bounded by corpus size, not compute. It grows only as the live loop logs more recurrences. The next step is a **live durability@N deployment** (real outcome, larger N) and **human-rater** validation of both the judge and the taxonomy.

9 PROSPECTIVE VALIDATION: DURABILITY@N AS A LIVE BENCHMARK

The contributions above are descriptive: they measure how an AI-heavy fleet fails and show, by offline replay, that the proposed controls are feasible. The open causal question (§7.3, §8) is whether enforcing the durability protocol actually raises durability@N on shipped work? We frame the next phase as a pre-registered, before-and-after benchmark on the same fleet, with this paper supplying the baseline.

Pre-registered baseline (this paper). The control-period metrics reported in §5.1-5.5 are the fixed baseline, anchored by durability@7 = 0.84 and durability@14 = 0.82 (Fig. 2). Every recurrence occurred within 13 days, so a 14-day window is sufficient to score a cohort.

Treatment. The intervention is the operational adoption framework (§7) made mandatory rather than advisory, with one key constraint: agents may claim shipped but cannot self-certify verified. The mechanisms already run in the control plane (§7.1); the change under test is that agents are bound to them by rule, not by discretion.

Design and outcomes. Because the fleet has no human-authored control arm (§8), the design is an interrupted time series: cohorts of fixes shipped before and after the rule takes effect, scored on the same durability@N harness. The primary outcome is durability@14; secondary outcomes are durability@7, logged recurrence rate, the incomplete-generalization share, the never-delivered share, and human triage hours per shipped cluster. To counter non-stationarity (model-capability drift over the window, §8), cohorts are stratified by model version, and the analysis reports the within-version change, not a pooled before/after difference.

Measurement backbone. A benchmark is only as trustworthy as its capture. Every change in the rollout is registered and stored through the 0x3admin Story Board ("0x3scrum"): the chat-backlog ingest indexes each session, extracts the feature or bug it addressed, and links it to a delivery-state row, so the before/after corpus is complete, ID-linked, and auditable rather than reconstructed after the fact. This is the same instrumentation that produced the baseline, which keeps the two periods comparable and removes the survivorship gaps that make the current 15% recurrence figure a lower bound. The protocol that makes this capture non-optional is part of

the operational adoption framework (§7) and ships with the replication artifacts; the success criterion is a pre-registered, model-version-stratified rise in durability@14 beyond the baseline's Wilson interval.

10 CONCLUSION

Passing a test once is not the same as delivering durable work. In this production agent fleet, the failures that matter are concrete: a fix returns in front of a customer, a repeated ask was never actually delivered, a partial slice ships without the real intent, or a later refactor drops the invariant. The controls are concrete too: classify delivery state, gate high-blast-radius patches, search sibling call-sites, pin contracts, test invariants, retrieve prior robustness lessons, and measure durability after merge.

The main lesson from this case study is not "write more tests" or "use a bigger model." The lesson is to make the agent carry the missing context: the shared pattern, the contract boundary, the acceptance checklist, and the organisation's prior mistakes. DuraLoop is the closed loop for doing that, and the released artifacts let other teams test whether similar controls improve durability in their own fleets.

11 DATA AVAILABILITY AND SHARING

The anonymised artifacts accompany this manuscript: the structured `fix → recurrence → principle` dataset, the **600-cluster demand corpus** with theme, rectification, and gold-sample lifecycle labels, the durability@N harness, the failure-mode taxonomy, the cost model, the **coding manual** (`CODEBOOK.md`), the **150-item gold sample** with the **author-adjudicated gold labels** (`gold_author.csv`) and two independent LLM coders, the **reliability reports** (per-theme and per-failure-mode accuracy/F1 + confusion), the **git cross-check summary** (`git_anchor.json`), the mined **literature corpus metadata** with curated bibliography and litmap, and every figure-generating script. All quantitative claims regenerate from these via Appendix A and B. Raw transcripts, the tenant/operator redaction map, unredacted prompts, and the downloaded PDFs are withheld (tenant privacy / copyright); a leakage scan confirms zero known tenant/person tokens across all released files.

Other teams can build on this corpus. The datasets behind the study, longitudinal and ID-linked records of how a live agent fleet fixes, re-breaks, and re-fixes a production B2B SaaS, are not commonly available: they are expensive to generate (they presuppose a fleet running under an enforced process protocol on a real paying product) and organisations that hold comparable data treat it as proprietary. We share derived artifacts because the measurement method is more useful when other teams can calibrate it against their own fleets. Beyond the shipped artifacts, researchers can request additional anonymised aggregates, label exports, and replication guidance from the authors where privacy, contractual, and copyright constraints allow, so other teams can use this case study as a baseline and calibrate DuraLoop-style durability metrics on their own fleets.

12 ETHICS STATEMENT

The study is observational on one organisation's own engineering process; no experiment on human subjects was run, and the agent A/B (RQ4) is an offline replay on logged data. Chat transcripts and ledgers contain tenant and operator names and business detail: these are processed locally, pseudonymised at a letter boundary before any model call or release, and only the anonymised, redaction-scanned artifacts are shared (Data availability). Operators are colleagues acting in their normal roles; no private end-customer PII is analysed. The cost model uses labelled, adjustable assumptions, not real billing data, and is framed as illustrative. The literature corpus is downloaded for analysis only; full PDFs are not redistributed (copyright), and we release just mined metadata and the curated bibliography.

13 USE OF GENERATIVE AI

We state this plainly: the prose, much of the analysis code, and the editorial revisions of this manuscript were produced with heavy LLM assistance. The authors own the parts that matter for trust: they chose the research questions, supplied and stand behind the operational data, ran and reviewed the analyses, adjudicated the gold-standard labels, and remain responsible for every claim. Consistent with the paper's own thesis, the substantive numbers are mechanically reproducible from the released pipeline (Appendix A), so no claim rests on generated prose alone.

14 FUNDING

This work was supported by Bravada Group Pty Ltd (ABN **58 167 664 815**), Quotech Pty Ltd (ABN **14 694 316 482**), and Finsoeasy, a registered business name of Eye Dream Pty Ltd (ABN **95 650 714 060**). The sponsors provided the operational setting, engineering process data, and infrastructure on which the study is based.

15 COMPETING INTERESTS

The authors are directly involved in building and operating the product studied in this paper, and are affiliated with entities that sponsor or benefit from the Quotech platform. This creates a potential conflict of interest: the work evaluates a system the authors are helping create. To mitigate this, the paper reports the limitation explicitly, releases the anonymised derived data and analysis scripts, includes ledger-independent git corroboration for recurrence claims, and frames single-product results as a grounded experience report rather than a population estimate.

REFERENCES

[1] Yasaman Abedini and Abbas Heydarnoori. 2023. Can GitHub Issues Help in App Review Classifications? *arXiv preprint arXiv:2308.14211* (2023).
[2] Vibhor Agarwal, Yulong Pei, Salwa Alamir, and Xiaomo Liu. 2024. CodeMirage: Hallucinations in Code Generated by Large Language Models. *arXiv preprint arXiv:2408.08333* (2024).
[3] Khairul Alam, Saikat Mondal, and Banani Roy. 2026. Why Are AI Agent Involved Pull Requests (Fix-Related) Remain Unmerged? An Empirical Study. *arXiv preprint arXiv:2602.00164* (2026).
[4] Aldeida Aleti and Matias Martinez. 2020. E-APR: Mapping the Effectiveness of Automated Program Repair. *arXiv preprint arXiv:2002.03968* (2020).
[5] Rodrigo Rebouças de Almeida, Rafael do Nascimento Ribeiro, Christoph Treude, and Uirá Kulesza. 2020. Business-Driven Technical Debt Prioritization: An Industrial Case Study. *arXiv preprint arXiv:2010.09711* (2020).

[6] Johin Johny Arimbur. 2026. How Many Tries Does It Take? Iterative Self-Repair in LLM Code Generation Across Model Scales and Benchmarks. *arXiv preprint arXiv:2604.10508* (2026).

[7] Ali Asgari, Annibale Panichella, Pouria Derakhshanfar, and Mitchell Olsthoorn. 2025. What Challenges Do Developers Face in AI Agent Systems? An Empirical Study on Stack Overflow & GitHub Issues. *arXiv preprint arXiv:2510.25423* (2025).

[8] Mohammad Aghajani Asl, Majid Asgari-Bidhendi, and Behrooz Minaei-Bidgoli. 2025. FAIR-RAG: Faithful Adaptive Iterative Refinement for Retrieval-Augmented Generation. *arXiv preprint arXiv:2510.22344* (2025).

[9] Ezio Bartocci, Leonardo Mariani, Dejan Nickovic, and Drishti Yadav. 2023. Property-Based Mutation Testing. *arXiv preprint arXiv:2301.13615* (2023).

[10] Samuel Benton, Mengshi Zhang, Xia Li, and Lingming Zhang. 2021. Self-Boosted Automated Program Repair. *arXiv preprint arXiv:2104.04611* (2021).

[11] Alexander Berndt, Thomas Bach, and Sebastian Baltes. 2026. Flaky Tests in a Large Industrial Database Management System: An Empirical Study of Fixed Issue Reports for SAP HANA. *arXiv preprint arXiv:2602.03556* (2026).

[12] Alexander Berndt, Sebastian Baltes, and Thomas Bach. 2024. Taming Timeout Flakiness: An Empirical Study of SAP HANA. *arXiv preprint arXiv:2402.05223* (2024).

[13] João Paulo Biazotto, Daniel Feitosa, Paris Avgeriou, and Elisa Yumi Nakagawa. 2025. Automating Technical Debt Management: Insights from Practitioner Discussions in Stack Exchange. *arXiv preprint arXiv:2502.03153* (2025).

[14] Huaben Chen, Wenkang Ji, Lufeng Xu, and Shiyu Zhao. 2023. Multi-Agent Consensus Seeking via Large Language Models. *arXiv preprint arXiv:2310.20151* (2023).

[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

[16] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. *arXiv preprint arXiv:2304.05128* (2023).

[17] Xiancai Chen, Zhengwei Tao, Kechi Zhang, Changzhi Zhou, Wanli Gu, Yuanpeng He, Mengdi Zhang, Xunliang Cai, Haiyan Zhao, and Zhi Jin. 2025. Revisit Self-Debugging with Self-Generated Tests for Code Generation. *arXiv preprint arXiv:2501.12793* (2025).

[18] Yujia Chen, Mingyu Chen, Cuiyun Gao, Zhihan Jiang, Zhongqi Li, and Yuchi Ma. 2025. Towards Mitigating API Hallucination in Code Generated by LLMs with Hierarchical Dependency Aware. *arXiv preprint arXiv:2505.05057* (2025).

[19] Elyson De La Cruz. 2026. From Code-Centric to Intent-Centric Software Engineering: A Reflexive Thematic Analysis of Generative AI, Agentic Systems, and Engineering Accountability. *arXiv preprint arXiv:2605.11027* (2026).

[20] Hoa Khanh Dam, Geeta Mahala, Rashina Hoda, Xi Zheng, and Cristina Conati. 2025. Towards autonomous normative multi-agent systems for Human-AI software engineering teams. *arXiv preprint arXiv:2512.02329* (2025).

[21] Esakkivel Esakkiraja, Denis Akhiyarov, Aditya Shanmugham, and Chitra Ganapathy. 2025. DeepCodeSeek: Real-Time API Retrieval for Context-Aware Code Generation. *arXiv preprint arXiv:2509.25716* (2025).

[22] Jiale Guo, Suizhi Huang, Mei Li, Dong Huang, Xingsheng Chen, Regina Zhang, Zhijiang Guo, Han Yu, Siu-Ming Yiu, Pietro Lio, and Kwok-Yan Lam. 2025. A Comprehensive Survey on Benchmarks and Solutions in Software Engineering of LLM-Empowered Agentic System. *arXiv preprint arXiv:2510.09721* (2025).

[23] Mohammed Mehedi Hasan, Hao Li, Emad Fallahzadeh, Gopi Krishnan Rajbahadur, Bram Adams, and Ahmed E. Hassan. 2025. An Empirical Study of Testing Practices in Open Source AI Agent Frameworks and Agentic Applications. *arXiv preprint arXiv:2509.19185* (2025).

[24] Junda He, Christoph Treude, and David Lo. 2024. LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision and the Road Ahead. *arXiv preprint arXiv:2404.04834* (2024).

[25] Alex Heilman, Alex Kyllo, and Emerson Murphy-Hill. 2026. GitHub Copilot and Developer Productivity: An Observational Dose-Response Analysis. *arXiv preprint arXiv:2606.00438* (2026).

[26] Pablo Restrepo Henao, Jannik Fischbach, Dominik Spies, Julian Frattini, and Andreas Vogelsang. 2021. Transfer Learning for Mining Feature Requests and Bug Reports from Tweets and App Store Reviews. *arXiv preprint arXiv:2108.00663* (2021).

[27] Steffen Herbold. 2019. On the costs and profit of software defect prediction. *arXiv preprint arXiv:1911.04309* (2019).

[28] Busra Icoz and Goksel Biricik. 2025. Automated Code Review Using Large Language Models with Symbolic Reasoning. *arXiv preprint arXiv:2507.18476* (2025).

[29] Hasnain Iqbal, Zerina Begum, and Kazi Sakib. 2025. Reduction of Test Reruns by Prioritizing Potential Order Dependent Flaky Tests. *arXiv preprint arXiv:2510.26171* (2025).

[30] Mohsen Jafari, Forough Majidi, and Abbas Heydarnoori. 2025. Prioritizing App Reviews for Developer Responses on Google Play. *arXiv preprint arXiv:2502.01520* (2025).

[31] Yanjie Jiang, Yian Huang, Guancheng Wang, Junjie Chen, Hui Liu, and Lionel Briand. 2026. Characterizing the Failure Modes of LLMs in Resolving Real-World GitHub Issues. *arXiv preprint arXiv:2605.12270* (2026).

[32] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?. In *International Conference on Learning Representations (ICLR)*.

[33] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*. 437–440.

[34] E. L. Kaplan and Paul Meier. 1958. Nonparametric Estimation from Incomplete Observations. *J. Amer. Statist. Assoc.* 53, 282 (1958), 457–481.

[35] Cuong Chi Le, Cuong Duc Van, Tung Duy Vu, Thai Minh Pham Vu, Hoang Nhat Phan, Huy Nhat Phan, and Tien N. Nguyen. 2025. TestWeaver: Execution-aware, Feedback-driven Regression Testing Generation with Large Language Models. *arXiv preprint arXiv:2508.01255* (2025).

[36] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72.

[37] Yunseo Lee, John Youngeun Song, Dongsun Kim, Jindae Kim, Mijung Kim, and Jaechang Nam. 2025. Hallucination by Code Generation LLMs: Taxonomy, Benchmarks, Mitigation, and Challenges. *arXiv preprint arXiv:2504.20799* (2025).

[38] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[39] Jingyue Li and André Storhaug. 2026. Reproducible, Explainable, and Effective Evaluations of Agentic AI for Software Engineering. *arXiv preprint arXiv:2604.01437* (2026).

[40] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[41] Xutong Liu, Shiran Liu, Zhaoqiang Guo, Peng Zhang, Yibiao Yang, Huihui Liu, Hongmin Lu, Yanhui Li, Lin Chen, and Yuming Zhou. 2023. Toward a consistent performance evaluation for defect prediction models. *arXiv preprint arXiv:2302.00394* (2023).

[42] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Can Automated Program Repair Refine Fault Localization? *arXiv preprint arXiv:1910.01270* (2019).

[43] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 643–653.

[44] Lech Madeyski and Marcin Kawalerowicz. 2017. Continuous Defect Prediction: The Idea and a Related Dataset. *arXiv preprint arXiv:1703.04142* (2017).

[45] Cristina Martinez Montes, Robert Feldt, Cristina Miguel Martos, Sofia Ouhbi, Shweta Premanandan, and Daniel Graziotin. 2025. Large Language Models in Thematic Analysis: Prompt Engineering, Evaluation, and Guidelines for Qualitative Software Engineering Research. *arXiv preprint arXiv:2510.18456* (2025).

[46] Quim Motger, Alessio Miaschi, Felice Dell'Orletta, Xavier Franch, and Jordi Marco. 2024. T-FREX: A Transformer-based Feature Extraction Method from Mobile App Reviews. *arXiv preprint arXiv:2401.03833* (2024).

[47] Abdullah Al Mujahid and Mia Mohammad Imran. 2026. "TODO: Fix the Mess Gemini Created": Towards Understanding GenAI-Induced Self-Admitted Technical Debt. *arXiv preprint arXiv:2601.07786* (2026).

[48] Sebastian Nielebock, Paul Blockhaus, Jacob Krüger, and Frank Ortmeier. 2024. ASAP-Repair: API-Specific Automated Program Repair Based on API Usage Graphs. *arXiv preprint arXiv:2402.07542* (2024).

[49] Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Is Self-Repair a Silver Bullet for Code Generation? *arXiv preprint arXiv:2306.09896* (2023).

[50] Abdelghny Orogat, Ana Rostam, and Essam Mansour. 2026. Understanding Multi-Agent LLM Frameworks: A Unified Benchmark and Experimental Analysis. *arXiv preprint arXiv:2602.03128* (2026).

[51] Ruwei Pan, Junlei Shen, Linhao Wu, Yueheng Zhu, Zixiong Yang, Yakun Zhang, Lu Zhang, and Hongyu Zhang. 2026. Toward Executable Repository-Level Code Generation via Environment Alignment. *arXiv preprint arXiv:2604.03622* (2026).

[52] Yun Peng, Jun Wan, Yichen Li, and Xiaoxue Ren. 2025. COFFE: A Code Efficiency Benchmark for Code Generation. *arXiv preprint arXiv:2502.02827* (2025).

[53] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Practical Mutation Testing at Scale. *arXiv preprint arXiv:2102.11378* (2021).

[54] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*. 24–36.

[55] Suzzana Rafi, Mahbub-Ul-Hoque Sumon, Md Erfan, Maruf Morshed Khan, August Shi, and Wing Lam. 2026. A Dataset of Reproducible Flaky-Test Failures. *arXiv preprint arXiv:2605.21677* (2026).
[56] Mohammad Rezaalipour and Carlo A. Furia. 2024. FauxPy: A Fault Localization Tool for Python. *arXiv preprint arXiv:2404.18596* (2024).
[57] Krishna Ronanki. 2025. Facilitating Trustworthy Human-Agent Collaboration in LLM-based Multi-Agent System oriented Software Engineering. *arXiv preprint arXiv:2505.04251* (2025).
[58] Mohammad Sadegh Sheikhaei and Yuan Tian. 2023. Automated Self-Admitted Technical Debt Tracking at Commit-Level: A Language-independent Approach. *arXiv preprint arXiv:2304.07829* (2023).
[59] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 532–543.
[60] Fangchen Song, Ashish Agarwal, and Wen Wen. 2024. The Impact of Generative AI on Collaborative Open-Source Software Development: Evidence from GitHub Copilot. *arXiv preprint arXiv:2410.02091* (2024).
[61] Hongjin Su, Shuyang Jiang, Yuhang Lai, Haoyuan Wu, Boao Shi, Che Liu, Qian Liu, and Tao Yu. 2024. EVOR: Evolving Retrieval for Code Generation. *arXiv preprint arXiv:2402.12317* (2024).
[62] Ankur Tagra, Haoxiang Zhang, Gopi Krishnan Rajbahadur, and Ahmed E. Hassan. 2022. Revisiting reopened bugs in open source software systems. *arXiv preprint arXiv:2202.08701* (2022).
[63] Kla Tantithamthavorn, Yaotian Zou, Andy Wong, Michael Gupta, Zhe Wang, Mike Buller, Ryan Jiang, Matthew Watson, Minwoo Jeong, Kun Chen, and Ming Wu. 2026. RovoDev Code Reviewer: A Large-Scale Online Evaluation of LLM-based Code Review Automation at Atlassian. *arXiv preprint arXiv:2601.01129* (2026).
[64] Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. 2024. MAGIS: LLM-Based Multi-Agent Framework for GitHub Issue Resolution. *arXiv preprint arXiv:2403.17927* (2024).
[65] Christoph Treude and Margaret-Anne Storey. 2025. Generative AI and Empirical Software Engineering: A Paradigm Shift. *arXiv preprint arXiv:2502.08108* (2025).
[66] Rosalia Tufano and Gabriele Bavota. 2025. Automating Code Review: A Systematic Literature Review. *arXiv preprint arXiv:2503.09510* (2025).
[67] Dean Wampler, Dave Nielson, and Alireza Seddighi. 2025. Engineering the RAG Stack: A Comprehensive Review of the Architecture and Trust Frameworks for Retrieval-Augmented Generation Systems. *arXiv preprint arXiv:2601.05264* (2025).
[68] David Williams, Ioakim Avraam, Aldeida Aleti, Matias Martinez, Justyna Petke, and Federica Sarro. 2026. Unveiling Practical Shortcomings of Patch Overfitting Detection Techniques. *arXiv preprint arXiv:2603.11262* (2026).
[69] Tongtong Xu, Liushan Chen, Yu Pei, Tian Zhang, Minxue Pan, and Carlo A. Furia. 2019. RESTORE: Retrospective Fault Localization Enhancing Automated Program Repair. *arXiv preprint arXiv:1906.01778* (2019).
[70] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*.
[71] Zhenyu Yang, Jingwen Wu, Zhen Yang, and Zhongxing Yu. 2025. Parameter-Efficient Fine-Tuning with Attributed Patch Semantic Graph for Automated Patch Correctness Assessment. *arXiv preprint arXiv:2505.02629* (2025).
[72] Quanjun Zhang, Ye Shang, Haichuan Hu, Chunrong Fang, Zhenyu Chen, and Liang Xiao. 2026. ComPass: Contrastive Learning for Automated Patch Correctness Assessment in Program Repair. *arXiv preprint arXiv:2602.07561* (2026).
[73] Ziyao Zhang, Yanlin Wang, Chong Wang, Jiachi Chen, and Zibin Zheng. 2024. LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation. *arXiv preprint arXiv:2409.20550* (2024).
[74] Jiasheng Zheng, Boxi Cao, Zhengzhao Ma, Ruotong Pan, Hongyu Lin, Yaojie Lu, Xianpei Han, and Le Sun. 2024. Beyond Correctness: Benchmarking Multi-dimensional Code Generation for Large Language Models. *arXiv preprint arXiv:2407.11470* (2024).

A REPRODUCIBILITY

The artifact ships every anonymised input the analysis needs, so all numbers and figures regenerate **without** raw data. One entry point runs the pipeline:

```
pip install -r requirements.txt
bash run_all.sh        # JSON + figures + PDF from data/
```

Two tiers. *Tier B* (default, what a reviewer runs) recomputes everything derivable from the frozen, anonymised JSON in `data/` and `lit/`: durability, taxonomy, reliability, prediction, cost, all figures, and the PDF, with no network or API keys. *Tier A* (authors) regenerates the source-derived datasets from the raw ledgers when `OPS_WORKSPACE_ROOT` points at the private operations workspace; the LLM-/network-dependent steps are run manually (they cost API calls and their outputs are shipped frozen).

```
# Tier B: deterministic, no LLM/network (run by run_all.sh)
python3 scripts/paper_analysis.py      # durability, CIs, BH-FDR
python3 scripts/value_model.py         # RQ5 cost model
python3 scripts/reliability.py         # RQ2 gold vs LLM coders
python3 scripts/reliability_modes.py   # RQ1 taxonomy reliability
python3 scripts/scale_labels.py        # labels -> 600 clusters
python3 scripts/lifecycle_analysis.py  # repeated-demand lifecycle split
python3 scripts/paper_figures.py       # core figures
python3 scripts/figure_lifecycle.py    # delivery lifecycle figure
python3 scripts/figure_architecture.py # proposed architecture figure
bash build/build_pdf.sh                # two-column ACM PDF

# Tier A: needs raw ledgers (OPS_WORKSPACE_ROOT) and/or API keys
export OPS_WORKSPACE_ROOT=/path/to/private-ops-workspace
python3 scripts/extract_fix_durability_dataset.py
python3 scripts/measure_chat_effort.py
python3 scripts/git_anchor.py          # git cross-check (§5.1)
python3 scripts/rq3_retrieval.py       # recall@k
OPENAI_API_KEY=... python3 scripts/feature_demand.py
python3 scripts/rq3_experiment.py --repeats 3
OPENAI_API_KEY=... python3 scripts/rq3_experiment.py \
    --repeats 3 --judge-provider openai \
    --out data/rq3-results-crossjudge.json
```

Every redaction runs **before** any LLM call; keys are read from the environment only and never persisted. The replay is seeded (`--seed`); `--limit`, `--k`, `--repeats`, `--no-llm` control cost. All outputs land in `data/` and `figures/`, from which this PDF is built. Exact per-case LLM verdicts may drift across model versions, but the deterministic descriptive pipeline and the aggregate effect direction/significance are stable.

B LITERATURE CORPUS

Related work and the Discussion (§6) are grounded in a programmatically mined corpus. `lit_fetch.py` queries the arXiv API across **21 topic seeds** (spanning LLM code generation and evaluation, agentic SWE, program repair, fault localization, test effectiveness, RAG, software maintenance, defect prediction, requirements mining, and related lines), de-duplicates, and downloads **158 full PDFs**. `lit_extract.py` extracts text (`pdftotext`, `pypdf` fallback) and, from each title+abstract, an LLM (`gpt-4.1`) distils a key claim, method, and the one DuraLoop finding it informs (the `relevance_tag` behind Fig. 12). `curate_refs.py` selects the **74 most relevant** into `references.bib` with real metadata and emits `lit/litmap.json`. PDFs are gitignored (copyright, ~180 MB); the mined metadata (`lit/corpus.jsonl`), notes (`lit/notes.jsonl`), bibliography, and litmap are released. Coverage is preprint-biased by construction (§8).