Telling Failures Apart: Fault Injection, Failure Diagnosis and AI Code Changes

Shirshak Jung Thapa

Florida Atlantic University

November 2026. DOI: https://doi.org/10.5281/zenodo.24429391. *License: CC BY 4.0.*

Abstract

When something a system depends on fails, the people who run it work out what failed from what it reports: error codes, log lines, stored state, HTTP status. Error-handling code produces that evidence and changes like any other code. On Kill Bill, an open-source billing platform, we measured how well failures can be told apart from this evidence and what code changes do to it. We injected 5 to 12 kinds of failure at six code paths through eleven entry points, each case twice, and counted the pairs of failure kinds that leave different evidence. In release 0.24.18, outside one entry whose log is too unstable to measure, 56 of 235 pairs cannot be told apart from what the system states. Of 27 careless edits we planted, eight made failures harder to tell apart; five of these kept the business outcome. The existing tests caught one of the eight and none of those five. Across eleven consecutive releases, we could trace a change in this evidence to the code once; it moved evidence between channels and merged no failures. Two AI models (Claude Sonnet 5.5 and Opus 5.5) made 144 maintenance changes, half with an instruction to keep failures distinguishable. None of the 141 that compiled made any pair harder to tell apart: 0 of 141 (95% interval 0 to 3%). In this system the risk to diagnosis lies in the code as it stands and in tests that do not check evidence, more than in the changes measured.

Keywords: failure diagnosis; observability; fault injection; error handling; logging; regression testing; AI code changes; large language models; empirical software engineering

1. Introduction

Suppose a payment has been stuck in an unknown state since the morning, and an engineer opens the logs of the billing system to find out why. The background task that retries unresolved payments, called the payment janitor, has written one line:

> Unable to repair paymentId='...', paymentTransactionId='...',
> currentTransactionStatus='UNKNOWN', newTransactionStatus='UNKNOWN'

The engineer needs to know what went wrong. Kill Bill, the open-source billing platform this paper studies, reaches the payment provider through a connecting program called a plugin, and the plugin can fail in different ways. It may have reported an error of its own. It may have taken too long to answer, failed to get a database connection, or answered with nothing at all. Each calls for a different response. But the janitor writes the same line in all four cases, and nothing else the system records differs between them. We caused each of the four failures ourselves, twice each, and compared everything the system reported. The four records were identical.

This paper is about such evidence: how much of it a real system gives, and what happens to it when the code changes. Systems depend on other things: a database, a payment provider, a lock that another process holds. When one of them fails, the people who run the system diagnose the failure from what the system states about it: the error code it returns, the lines it writes to its own log, what it stores, and the HTTP status it answers with. A timeout and a refused connection need different fixes, so it matters that they look different.

The code that handles errors decides all of this. It catches the failure, chooses a code and a message, logs the exception or does not, and decides what to store. That code is maintained like any other. It gets reorganised, cleaned up and given a consistent logging style. More and more often this is done by AI coding tools, which already open pull requests in open-source projects by the thousands [Ouatiti et al. 2026]. A change of this kind can keep the business result and pass every test while two failures that used to leave different evidence start to leave the same evidence.

Tests are not built to notice this. A test usually checks that the right thing happens: the payment is marked failed, the account is set aside, an error reaches the caller. It rarely checks that a timeout and a refused connection still look different. If a change makes them look the same, the tests can pass, and the loss shows up later, when someone is trying to diagnose an outage.

This matters to three groups: the engineers who are called when a system fails, the maintainers who review changes to error-handling code, and anyone deciding how far to trust such changes when an AI tool writes them. Earlier work has studied whether logs are good enough to diagnose failures and where log statements should go (Section 7). We found no study that measures, on a real system, whether different failures stay distinguishable from each other when the code changes, or whether the system's own tests would notice a loss.

We measured this on one real system, Kill Bill 0.24.18, and asked four questions:

1. **Today.** How well can different failures be told apart in the current code, from what the system itself states?
2. **Would a loss be noticed.** When a change makes failures harder to tell apart, do the project's existing tests fail?
3. **The project's own changes.** From one release to the next, how often did the project's own changes alter what can be told apart?
4. **AI maintenance.** When an AI model does ordinary maintenance on the code that handles these failures, how often does a pair of failures stop being distinguishable?

To answer them we built a measuring instrument inside Kill Bill's own test modules. We chose six code paths, each a call from one part of the system to something it depends on. At each path we injected between 5 and 12 kinds of failure. We reached each path through one or two of eleven entry points: a direct Java call, the event bus, a notification queue, or an HTTP request. Every case ran twice, each time in a fresh Java process. From each run we recorded seven to nine kinds of evidence, which we call channels, and counted the pairs of failure kinds whose evidence differs consistently. A change loses a pair when two failures that could be told apart before can no longer be told apart after.

We then measured three kinds of change against that baseline. First, we planted 27 small careless edits by hand, of the kinds that remove evidence, to see whether the measure and the existing tests react. Second, we

ran the same tests, unchanged, on other releases of Kill Bill, to see what the project's own changes did. Third, two AI models made 144 maintenance changes to the file on each path that holds most of its error handling. They worked on three tasks: reduce duplicated code, simplify error handling, and make the logging consistent. Each task came in two wordings, one of which asked the model to keep failures distinguishable. The plan, the failure kinds, the tasks and the planted edits were written down before the measurements they govern.

Much of what a change could lose is already gone. Leaving aside one entry whose log is too unstable to measure, 56 of the 235 pairs of failures cannot be told apart from what the system itself states. Kill Bill folds six different lookup failures into "No invoice to generate". It reports a refused connection during an overdue check as "Fatal NotificationQ dispatch error, data corruption". It turns a null answer from a plugin into an empty list.

The tests do not guard what remains. Of the 27 planted edits, 8 made failures harder to tell apart. Five of those kept the business outcome, and the project's existing tests failed for none of the five. They failed for one of the eight, an edit that also broke the outcome. In one, a database timeout during invoice generation now reaches the caller as error 4007, "No invoice to generate", the same code a missing account produces. All 548 tests of the invoice module still pass. The tests did fail for five planted edits, and in each case the edit changed something a test checks: an error code, an HTTP status or the business result. No edit that only removed evidence was caught.

The project's own changes rarely touched this evidence. We ran the same tests on eleven consecutive releases, published between October 2024 and September 2026. In nine of the ten steps from one release to the next, nothing changed that we could attribute to the code. In the tenth, release 0.24.18 fixed a fault in invoice generation and gave five log messages a single wording. Read from the log alone, 18 pairs of failures merged. Read from everything the system states, none did, because the error code still separated them. And the gaps are old: the pairs that cannot be told apart in 0.24.18 were already indistinguishable in the earliest release on which each path was measured.

The AI changes did no harm that we could measure. None of the 141 AI changes that compiled lost a distinguishable pair, on any entry and on either set of channels: 0 of 141, with a 95% interval of 0 to 3%. Only 16 of the 141 altered any evidence at all outside the noisy entry, so the chance to merge failures was smaller than 141 suggests. The same holds for each path, task, wording and model taken alone. The models reworded some log lines and restructured the error handling, but they reworded lines that several failures already shared, and they kept passing the caught error to the logger.

Taken together, in this system the risk to diagnosis lies less in the changes we measured than in the code as it stands and in tests that do not check evidence. The instrument is itself an ordinary test: kept with a project, it would let the build check that failures stay distinguishable. Section 2 defines what "distinguishable" means here and why some channels are kept apart. Section 3 describes the design, Section 4 the results, and Section 5 what they mean. Section 6 sets out the threats to validity, Section 7 the related work, and Section 8 concludes.

2. Background: telling failures apart

2.1 The system

Kill Bill is an open-source subscription billing and payment platform written in Java [Kill Bill repository]. It is organised in modules: accounts, catalog, subscriptions, invoices, payments, overdue handling and others. The modules talk to each other through internal Java interfaces, an event bus, and notification queues that hold work to be done later. Payments go through plugins, one per payment gateway. A REST interface sits on top. We use release 0.24.18 throughout and, for the third question, other releases of the same 0.24 line.

Kill Bill suits the question for three reasons. It handles money, so diagnosing a failure quickly matters. Its error handling is spread over many layers, each of which may log, wrap or swallow an exception. And its own test modules bind mock versions of the dependencies of each module, so failures can be injected without changing the code under study.

2.2 Failure kinds, entries and channels

A **path** is a call from an entry point of one module to something it depends on, together with the error handling between them. An **entry** is the way the call is started. There are four: a direct Java call to the module's public interface (what the REST layer calls), an event posted on the bus, a notification placed in a queue, and an HTTP request to Kill Bill's own test server. A **failure kind** is one way the dependency can fail. The kinds for each path follow a rule fixed before any run:

- every checked exception the dependency declares, that is, every error its interface announces it can raise (two kinds with different error codes where the exception carries a code);
- a database timeout;
- a refused connection;
- a null answer;
- a lock held by another thread, where the path takes a lock;
- for the payment plugin, its own outcomes, such as a declined or a pending payment.

A run with no failure serves as a control and is not counted.

Each run is reduced to one **signature** per **channel**:

- **T, result type:** what reaches the caller, either "returned" with the shape of the value, or the chain of exception classes.
- **C, error code:** Kill Bill's numeric error codes on that chain.
- **M, exception message:** the messages on that chain.
- **L, own log lines:** the level, logger and text of each line the system writes, without stack traces.
- **X, attached text:** the exception text printed under a log line.
- **S, stored state:** whether the account was parked (set aside from billing), which rows were added to the account's tables, which events and notifications were queued, payment and transaction states.
- **G, gateway text:** the gateway error message stored with a payment, where there is one.
- **H, HTTP status** and **B, response body**, on the HTTP entry only.

Before signatures are compared, we mask everything that changes from run to run for reasons unrelated to the failure. That covers timestamps, dates, identifiers, object hash codes and timing figures, and on the HTTP entry a request counter and a worker-thread name.

2.3 Distinguishable

In plain words, two failures are distinguishable when the system leaves different evidence for them, and does so reliably. Precisely, two failure kinds A and B are **distinguishable on a channel** when the signature on that channel is the same in both runs of A, the same in both runs of B, and different between A and B. They are distinguishable on a **set of channels** when this holds on at least one channel in the set. A cell is one failure kind on one channel. A cell whose two runs differ is called **unstable**, and a pair that depends on an unstable cell counts as not distinguishable. The score of an entry is the number of distinguishable pairs among its failure kinds. Kinds that cannot be separated from each other form a **group**.

A change **loses** a pair when the pair is distinguishable before the change and not after, on the same entry and set of channels. It **gains** a pair in the opposite case. The business **outcome is kept** when, for every failure kind and entry, the result (what was returned or raised, what HTTP status was sent) and the business data stored are the same as before. A raised exception counts as the same result whatever its type or code, so an edit can keep the outcome and still change which exception the caller receives.

We report two sets of channels. The primary set is **what the system itself states**, written SYS: the error code, its own log lines and stored state, plus the HTTP status on the HTTP entry. The secondary set is **all channels**, written ALL. The reason for the split is that the result type, the exception message and the attached text can simply repeat the class and message of the exception we injected, which we chose. Two injected exceptions with different text will always differ on those channels as long as the text is printed somewhere, whether or not the system classifies them. SYS counts only what the system decides to say.

A worked example shows the difference. On path 1, invoice generation calls a billing service to collect the account's billing events. That service can throw six checked exceptions: an invalid catalog for the tenant, a missing plan, an unknown account by identifier or by record number, and an invalid subscription or bundle. For any of the six, the caller receives error code 4007 with the message "No invoice to generate for account ... and date ...", and the log shows the same warning line. On SYS the six form one group. On ALL they are fully separated, because the warning carries the original exception's text underneath. An operator reading the log carefully can recover the cause. An alert, a dashboard or a support script that keys on the error code cannot.

3. Study design

We wrote the plan down on 19 November 2026, before any main run. By then we had done a feasibility test, built the six paths and run a pilot of 24 AI changes on path 1. The paths and failure kinds, the target files, the tasks and wordings, and the catalogue of planted edits were each written down before the measurements they govern. The comparison of releases was extended from one pair to ten on 20 November 2026, under a rule written before the added runs (Section 3.4).

3.1 Paths and entries

Table 1 in Section 4.1 lists the six paths with their entries and failure counts. In brief:

- **P1, invoice generation to billing events** (invoice module). Entries: the Java call that the REST layer uses to trigger an invoice, and a subscription event on the bus. Ten failure kinds: six checked exceptions with different codes, timeout, connection, null answer, account lock held.
- **P2, purchase to payment plugin** (payment module). Entries: the Java purchase call, and the same purchase as an HTTP request against Kill Bill's own test server, which runs all real modules. Eleven kinds on the Java entry and twelve on HTTP: a plugin exception, timeout, connection, null answer, account lock (HTTP only, see below), and seven outcomes of the mock gateway plugin (error, pending, unknown, canceled, its own exception, a delay longer than the plugin timeout, and a lock failure inside the plugin).
- **P3, invoice payment to invoice status** (payment module, through the invoice payment control plugin). Entries: the Java call to pay an invoice, and an invoice-creation event on the bus. Six kinds.
- **P4, overdue refresh to account lookup** (overdue module). Entries: an overdue-check notification in its queue, and a tag-removal event on the bus. Six kinds. No public call starts an overdue refresh.
- **P5, payment janitor to plugin payment information** (payment module). Entries: the Java call that reads a payment with plugin information, which runs the janitor on the spot, and a janitor notification in its queue. Five kinds.
- **P6, payment-method refresh to the plugin** (payment module). Entry: the Java refresh call. Five kinds.

Two details apply to the P2 Java entry. Its cases are run one at a time. And the lock kind is left out there, because it does not reach its injection point within the plugin timeout.

3.2 Measurement

Each case (one failure kind on one entry) runs in its own Maven invocation and so in its own Java process, and every case runs twice. Each test pins the system clock to 15 March 2026, 12:00 UTC, before creating any data. The injection point is the place in the code where the mock dependency raises the failure. The test checks only that this point was reached. Everything else is recorded, not judged. The capture window runs from arming the fault until the entry call returns, plus two seconds. For entries on the bus or a queue, the test first waits until no event or notification for the account is pending. The baseline used 186 runs, all of which ended normally and reached their injection point. The original plan took 5,888 injection runs, and the added releases took 1,674 more.

The environment was a single Windows machine with JDK 11, Kill Bill's embedded H2 database and its test profile, with up to four runs in parallel.

For each changed version of the code, which we call a unit, we record whether it compiles, whether the existing tests pass, whether the outcome is kept, and the pairs lost and gained on each entry and channel set. The existing tests are the whole test suite of the module that holds the changed file (548 tests for invoice, 186 for payment, 22 for overdue), run as the project runs them, without the injection tests.

3.3 Planted edits

For each path we wrote a fixed catalogue of small edits of the kinds a careless change makes, one edit per variant, before any was measured:

1. two catch blocks with different messages merged into one message;

2. the exception object dropped from a log call;
3. a specific error code replaced by a generic one;
4. an exception swallowed and a default value returned;
5. a distinguishing log line removed;
6. on the HTTP entry of P2, a specific HTTP status mapped to 500.

Each applicable kind was planted once per path: kinds 1 to 5 on P1, P3 and P5, all six on P2, kinds 2 to 5 on P4 and kinds 3 and 4 on P6, 27 variants in all. Each has an identifier of the form PB_P1_4: path 1, edit kind 4. Where a kind could not apply, the catalogue states why (for example, no log line on P6 lies in the project's source). A variant that loses no pair on either channel set is set aside as having no effect. For the others we ask whether the outcome is kept and whether the existing tests fail.

3.4 The project's own releases

Each path's test was compiled and run unchanged on other releases of Kill Bill. We measured eleven consecutive releases of the 0.24 line, 0.24.12 to 0.24.22, which gives ten pairs of consecutive releases. The comparison began with one pair, 0.24.17 and 0.24.18, and was extended on 20 November 2026 to the five releases before 0.24.17 and every release after 0.24.18. The rule (which releases, how they are measured, how they are compared) was written before the added runs and was not changed.

No test file was edited to make it compile or pass on another release. A path counts as measured on a release only if its test compiles unchanged, every case ends normally and every case reaches its injection point; otherwise it is reported as not measurable there. For each pair of consecutive releases we compare the counts of distinguishable pairs and trace every pair lost or gained to the source lines that changed between the two.

This part shows how often the project's real changes altered what can be told apart. Whether a test would notice such a change is the question of the planted edits.

3.5 AI maintenance changes

Target files. On each path, the change targets the single source file holding the most catch blocks and log statements for that path, counted before any generation. The rule chose `PaymentPluginDispatcher` (P2), `InvoicePaymentControlPluginApi` (P3), `OverdueAsyncBusNotifier` (P4), `IncompletePaymentTransactionTask` (P5) and `PaymentMethodProcessor` (P6). For P1 the target is `InvoiceDispatcher`, and its 24 changes are those of the pilot. The P4 target runs only on that path's bus entry.

Tasks. Three ordinary maintenance requests, worded exactly as follows, each ending with "Keep the behaviour the same.":

- R1: "Refactor this class to reduce duplicated code."
- R2: "Clean up and simplify the error handling in this class."
- R3: "Clean up the logging in this class so that the messages follow one consistent style."

Wordings. The plain wording is the task alone. The guarded wording adds: "Failures that can be told apart today must stay distinguishable: do not merge, drop or reword the error codes, exception types or log messages that separate one kind of failure from another."

Models and size. Two models, `claude-sonnet-5-5` and `claude-opus-5-5`, each asked twice for every combination: 6 paths, 3 tasks, 2 wordings, 2 models and 2 repetitions, 144 changes in all. Each change is one request with no tools, containing the task and the whole target file, with high reasoning effort. The model answers with search-and-replace blocks, which are applied mechanically: each search text must occur exactly once. An answer that does not apply or does not compile is recorded and not repaired.

3.6 Checks of the instrument

Before any change was measured, each path's target file received a no-op edit (a two-line comment, which shifts every later line number) and went through the same pipeline. A sound instrument should report zero pairs lost and zero gained. A second no-op on P2 was run later under full load, as a check of noise on its HTTP entry.

3.7 Analysis

We report counts and proportions with exact 95% intervals (Clopper-Pearson) [Clopper and Pearson 1934], for each path, task, wording and model and overall. An interval says how high the true rate could plausibly be, given what we counted. For example, 0 losses in 141 changes is compatible with a true rate of up to 3%. No other statistical test is used. Every change that loses or gains a pair is listed in the replication package with the lines that cause it.

Three practical details. One case crashed in its set-up before the fault was armed and left no capture, so it was run again in a fresh process. Every unit whose existing tests failed had its suite run a second time, and the first result is the one counted. And two checks of noise on P2 were added: a second no-op edit, and an analysis that ignores the order of log lines.

4. Results

4.1 Today: many failures already look alike

Table 1 gives the baseline, the unmodified 0.24.18 code.

Table 1. Distinguishable pairs in unmodified Kill Bill 0.24.18

Path	Entry	Failure kinds	Pairs	SYS	ALL	Groups on SYS	Unstable cells
P1 invoice generation -> billing events	Java API call	10	45	27	45	3	0
P1 invoice generation -> billing events	event bus	10	45	27	45	3	0
P2 purchase -> payment plugin	Java API call	11	55	55	55	11	0
P2 purchase -> payment plugin	HTTP request	12	66	57 (noisy)	66	7	11
P3 invoice payment -> invoice status	Java API call	6	15	15	15	6	0
P3 invoice payment -> invoice status	event bus	6	15	15	15	6	0
P4 overdue refresh -> account lookup	notification queue	6	15	11	15	3	0
P4 overdue refresh -> account lookup	event bus	6	15	11	15	3	0
P5 payment janitor -> plugin payment info	Java API call	5	10	7	10	3	0
P5 payment janitor -> plugin payment info	notification queue	5	10	4	4	2	0
P6 payment-method refresh -> plugin	Java API call	5	10	7	10	3	0
all entries			301	236	295		
all except P2 HTTP			235	179	229		

Groups on SYS (failure kinds that cannot be told apart from what the system itself states):

- P1 API: {ACC_ID,ACC_RECORD,CAT_PLAN,CAT_TENANT,SUB_BUNDLE,SUB_ID} {CONNECTION,NULLRESP,TIMEOUT} {LOCK}
- P1 BUS: {ACC_ID,ACC_RECORD,CAT_PLAN,CAT_TENANT,SUB_BUNDLE,SUB_ID} {CONNECTION,NULLRESP,TIMEOUT} {LOCK}
- P2 HTTP: {CANCELED} {CONNECTION,LOCK,NULLRESP,TIMEOUT} {DELAY} {ERROR} {EXCEPTION,PLUGIN_EXC,PLUGIN_LOCK} {PENDING} {UNKNOWN}
- P4 QUEUE: {ACC_ID,ACC_RECORD} {CONNECTION,NULLRESP,TIMEOUT} {LOCK}
- P4 BUS: {ACC_ID,ACC_RECORD} {CONNECTION,NULLRESP,TIMEOUT} {LOCK}
- P5 API: {CONNECTION,NULLRESP,TIMEOUT} {LOCK} {PLUGIN_EXC}
- P5 QUEUE: {CONNECTION,NULLRESP,PLUGIN_EXC,TIMEOUT} {LOCK}
- P6 API: {CONNECTION,NULLRESP,TIMEOUT} {LOCK} {PLUGIN_EXC}

Three entries keep every pair apart on SYS: the P2 Java entry and both P3 entries. Elsewhere failures fall into groups. Leaving aside the P2 HTTP entry, whose log channel is unstable (see 4.2), 179 of 235 pairs are distinguishable from what the system states, and 56 are not. No cell is unstable on those ten entries (Table 1), so all 56 are stable collisions: both runs of each failure left the same evidence as the other failure. None is counted only because a capture was unreliable. Nor does the count rest on text we supplied: with the injected messages masked in the log lines, 57 pairs cannot be told apart instead of 56 (Section 6.1). On all channels 229 of those 235 are distinguishable; the six that are not all lie on the P5 janitor queue, where four failures leave identical evidence on every channel, the case of the introduction.

The groups follow a small number of patterns in the code.

- **Checked exceptions with different codes collapse where the caller wraps them in its own code.** On P1, six lookup failures become error 4007 "No invoice to generate", a message that describes a normal outcome. On P4, two different account errors produce one log line.
- **Runtime failures share a generic route.** On P1, P5 and P6, a timeout, a refused connection and a null answer form one group. On P4 the three end in the queue's "Fatal NotificationQ dispatch error, data corruption...", which names a cause that did not occur.
- **Null answers look like normal results.** On P6 a null answer from the plugin comes back as an empty list of payment methods. On P5 it leaves the payment in its unknown state, exactly as if the plugin had reported nothing new. On P3 it leaves the same trace as an invoice that is legitimately not yet committed, a normal case outside our set of failure kinds.
- **Locks mostly stay apart.** A held lock forms its own group wherever it is injected, except on the P2 HTTP entry, where it is answered with the same status and code as a timeout.

Nearly every gap closes on all channels. Kill Bill usually passes the exception to the logger or lets it reach the caller, and the attached text or the exception repeats whatever the failing dependency said. But that rescue depends on the text of the exception, which on a real system is written by someone else's code. And it is lost to anything that reads codes and statuses and not stack traces.

4.2 The instrument is quiet, except on one entry

The six no-op edits lost and gained nothing, on every entry and both channel sets, and kept every outcome and every existing test. On five paths not a single cell changed. On the P2 HTTP entry, however, the log lines of several threads (the HTTP worker, the bus and a callback servlet in the test server) interleave in a different order from run to run. In the baseline 11 cells of that entry are unstable, and the second P2 no-op, a seventh edit run later under full load, gained 3 SYS pairs by itself. On that entry the SYS count moves by about three pairs without any code change, so **P2 HTTP results on SYS are within noise and support no claim**. P2 conclusions rest on its Java entry and on its HTTP entry with all channels. A later reanalysis that ignores the order of log lines points the same way. In that reanalysis, both no-ops and 20 of the 24 P2 changes show the same three gained pairs on that entry, and in the other four every difference involves a failure kind whose evidence was unstable in that unit.

Outside that entry, stored state was occasionally unstable: across the measured units of the original plan, 33 cells in 30 units, all on stored state (26 cells in 23 of the AI changes). They come from a bus record that moves from the live table to the history table inside the two-second window when the machine is under load. None changed a pair count.

4.3 Planted edits: the tests did not notice lost evidence

The planted edits show the measure at work on changes that are meant to hurt. Table 2 lists the 8 of 27 planted edits that lost at least one pair.

Table 2. The planted edits that lost a pair

Variant	Edit	Pairs lost, SYS (per entry)	Pairs lost, ALL (per entry)	Outcome kept	Existing tests
PB_P1_4	exception swallowed, default returned	API 18, BUS 18	API 0, BUS 0	yes	pass (548)
PB_P2_4	exception swallowed, default returned	API 15, HTTP 9	API 15, HTTP 15	no	fail (2 of 187)
PB_P3_4	exception swallowed, default returned	API 1, BUS 1	API 1, BUS 1	yes	pass (186)
PB_P4_2	exception dropped from a log call	QUEUE 0, BUS 0	QUEUE 1, BUS 1	yes	pass (22)
PB_P4_4	exception swallowed, default returned	QUEUE 0, BUS 0	QUEUE 0, BUS 3	yes	pass (22)
PB_P4_5	distinguishing log line removed	QUEUE 0, BUS 0	QUEUE 1, BUS 1	yes	pass (22)
PB_P5_4	exception swallowed, default returned	API 3, QUEUE 0	API 1, QUEUE 0	no	pass (186)
PB_P6_4	exception swallowed, default returned	API 3	API 1	no	pass (186)

Variants 27; with an effect 8; of these outcome kept 5. Existing tests failed: 1 of 8 (12%; 0-53%) of those with an effect, 0 of 5 (0%; 0-52%) of those that kept the outcome.

The clearest case is PB_P1_4. In invoice generation, the catch block for runtime exceptions returned an empty list instead of rethrowing. On the Java entry a database timeout now reaches the caller as error 4007, "No invoice to generate for account ... and date ...", the same code and message an unknown account produces. The timeout, the refused connection and the null answer each merge with the six lookup failures: 18 pairs lost on each entry. By this study's definition the outcome is kept: the call still raises an error, and the same data is stored. All 548 tests of the invoice module pass. The attached text in the log still names the timeout, so on all channels nothing is lost, but anything that acts on the error code now reads a timeout as a missing account.

Four more variants kept the outcome and lost pairs. In PB_P3_4, an invoice error is swallowed and the default "abort the payment" result is returned, so an invoice that does not exist and an invoice in the wrong state can no longer be told apart. On P4, dropping the exception from the overdue dispatcher's warning (PB_P4_2), removing that warning (PB_P4_5), and swallowing runtime exceptions in the bus handler (PB_P4_4) each merged account failures or runtime failures on all channels. These three lost nothing on SYS only because, on SYS, the failures they merged were already merged in the unmodified code; the edits removed the attached text that was the last thing keeping them apart. The tests of the modules passed for all five variants that kept the outcome: **0 of 5 (95% interval 0 to 52%)**.

The one variant with an effect that the tests caught, PB_P2_4, also broke the outcome: a swallowed plugin failure left payments without a state. Over all eight with an effect, the tests failed for 1 (12%; 0 to 53%). The two variants on P5 and P6 that swallowed a plugin exception changed what the caller receives (a payment or an empty list instead of an error) and passed the tests as well.

Table 3 sorts the catalogue by kind of edit.

Table 3. The planted edits by kind

Kind of edit	Variants	Lost a pair	Lost a pair, outcome kept	Existing tests failed
1. messages of two catch blocks merged	4	0	0	1 (PB_P3_1)
2. exception dropped from a log call	5	1	1	0
3. specific error code made generic	6	0	0	2 (PB_P2_3, PB_P3_3)
4. exception swallowed, default returned	6	6	3	1 (PB_P2_4)
5. distinguishing log line removed	5	1	1	0
6. specific HTTP status mapped to 500	1	0	0	1 (PB_P2_6)
all	27	8	5	5

Nineteen variants had no effect, and the captures show why. Often the removed text was repeated elsewhere. Kill Bill's payment interface logs an "ENTERING" and an "EXITING" line that echo the exception message, and the lock library writes its own line next to the one we removed. In other cases the edited line was already shared by the kinds it might have separated, which, as Section 4.5 shows, is also why the AI changes lost nothing.

Four of the no-effect variants were caught by the tests, and in each the edit changed a value a test pins. Making the plugin timeout code generic (PB_P2_3, code 7100 to 7019) and answering a timeout with 500 instead of 504 (PB_P2_6) each failed a timeout test. Merging the abort catch block (PB_P3_1) and making the abort code generic (PB_P3_3) each failed a test of the aborted state. All four failures reproduced on the re-run. PB_P2_3 and PB_P2_6 also changed the outcome by our definition, since the HTTP status of the delay kind moved from 504 to 500. They lost no pair, because on the HTTP entry the delay stays apart from the other 500 answers through its log lines.

Taken together: in this study the existing tests caught degradations that changed a code, a status or an outcome they check, and did not catch any degradation that only removed evidence.

4.4 The project's own releases

We start with one pair of releases, 0.24.17 and 0.24.18. All six path tests compiled unchanged on 0.24.17 and reached every injection point. Between the two releases the source changes in the modules of these paths were in the invoice module, plus one test-support class of the HTTP layer and the version of a shared parent library. Table 4 compares the counts.

Table 4. The same tests on releases 0.24.17 and 0.24.18

Path	Entry	SYS 0.24.17 -> 0.24.18	ALL 0.24.17 -> 0.24.18	Channels whose count moved	Cells that differ	Unstable cells
P1	API	27 -> 27	45 -> 45	L 27->9, X 39->45	12	0
P1	BUS	27 -> 27	45 -> 45	none	24	0
P2	API	55 -> 55	55 -> 55	none	0	0
P2	HTTP	58 -> 57	66 -> 66	L 10->3, X 66->55	10	7
P3	API	15 -> 15	15 -> 15	none	0	0
P3	BUS	15 -> 15	15 -> 15	S 4->5	0	1
P4	QUEUE	11 -> 11	15 -> 15	none	0	0
P4	BUS	11 -> 11	15 -> 15	none	0	0
P5	API	7 -> 7	10 -> 10	none	0	0
P5	QUEUE	4 -> 4	4 -> 4	none	0	0
P6	API	7 -> 7	10 -> 10	none	0	0

Cells that differ = (failure kind, channel) signatures that are not identical in the two releases. On the P2 HTTP entry they are all on the log channel, whose line order varies from run to run in either release; its SYS count supports no claim.

Paths P3 to P6 are identical on every stable cell, and the one P3 bus difference comes from an unstable stored-state cell. On P2, all ten differing cells hold the same log lines in a different order, the noise of Section 4.2. P1 differs in 36 cells and in its outcome, and every difference traces to 69 changed lines of `InvoiceDispatcher`.

- In 0.24.17 the billing-lookup catch blocks log "Failed to retrieve BillingEvents for accountId=..", and the runtime failures log nothing at this layer. Release 0.24.18 logs "Failed to generate invoice for accountId=.." in all five catch blocks. On the Java entry this merges the checked failures with the runtime failures in the log: 27 distinguishable pairs on that channel become 9. SYS stays at 27, because the error code (4007 against none) still separates them.
- Release 0.24.18 logs runtime exceptions with their text before rethrowing them, so the attached text separates more: 39 pairs become 45.
- Release 0.24.18 parks the account when the billing lookup fails on the bus entry, a fix for a failure the earlier release left unhandled. Stored state and the outcome differ for every failure kind, but the parked and not-parked split separates the same groups as before.
- A held lock on the bus is rescheduled through the invoice notification queue in 0.24.17. In 0.24.18 it is retried by the bus until the queue logs "Fatal NotificationQ dispatch error, data corruption". Different content, still its own group.

So the project's own change, which fixed a real outcome, moved evidence between channels (less in the log, more in the attached text) without changing which failures the system as a whole can separate. Read channel by channel, it looks like a loss; read as a whole, it is not.

Table 5. Ten pairs of consecutive releases, 0.24.12 to 0.24.22

Release pair	Source lines changed	Paths compared	Pairs lost / gained, SYS	Pairs lost / gained, ALL	Channel counts that moved	Cells that differ	P2 HTTP on SYS: lost / gained
0.24.12 -> 0.24.13	320	5 (not P2)	0 / 0	0 / 0	none	0	not compared
0.24.13 -> 0.24.14	107	5 (not P2)	0 / 0	0 / 0	P3 BUS S 4->5	1	not compared
0.24.14 -> 0.24.15	0	6	0 / 0	0 / 0	none	0	2 / 5
0.24.15 -> 0.24.16	22	6	0 / 0	0 / 0	none	0	6 / 0
0.24.16 -> 0.24.17	8	6	0 / 0	0 / 0	P3 BUS S 5->4	0	0 / 1
0.24.17 -> 0.24.18	291	6	0 / 0	0 / 0	P1 API L 27->9; P1 API X 39->45; P3 BUS S 4->5	36	1 / 0
0.24.18 -> 0.24.19	0	6	0 / 0	0 / 0	P2 API S 34->24	0	0 / 4
0.24.19 -> 0.24.20	93	6	0 / 0	0 / 0	P2 API S 24->34	0	4 / 1
0.24.20 -> 0.24.21	84	6	0 / 0	0 / 0	P2 API S 34->24	2	0 / 2
0.24.21 -> 0.24.22	0	6	0 / 0	0 / 0	P2 API S 24->30	2	2 / 3

Source lines changed = lines added plus lines removed between the two releases in the main sources of the five modules the paths run through (invoice, payment, overdue, the HTTP layer and the server profile), with comments and blank lines ignored. Pairs lost / gained on SYS leave out the P2 HTTP entry, which has its own column and supports no claim. Channel counts and cells leave out that entry's log and attached-text channels. A channel is written path, entry, channel, count on the earlier release -> count on the later one.

The wider comparison shows how typical that pair is. Table 5 lists the ten pairs of consecutive releases, which the project published between October 2024 and September 2026. Nine of the eleven releases were measured on all six paths. On 0.24.12 and 0.24.13 the P2 test does not compile, because its HTTP part uses a configuration class that is not available to Kill Bill before 0.24.14, so P2 is compared from 0.24.14 on. Table 5 rests on 58 comparisons of a path on two consecutive releases. That is five paths on the first two release pairs and six paths on the other eight.

No pair of failures was lost or gained in any of the ten pairs, on SYS outside the noisy entry and on every entry with all channels: 0 of 10 (95% interval 0 to 31%). Since no step lost or gained a pair, the groups of Table 1 are the same on every release measured. The pairs that cannot be told apart in 0.24.18 were already indistinguishable in the earliest release on which each path was measured.

Apart from the release pair already described, the only cells that differ are five stored-state cells. Each differs only in whether one bus record had been counted when the capture ended. The only channel counts that moved are stored-state counts on an entry where a stored-state cell was unstable in one of the two releases. Most of this is the load effect of Section 4.2. Six stored-state cells were unstable on the added releases. A failure kind with an unstable cell is left out of the count for that channel, so the count drops for that release and returns in the next. One case may be more than load. On the P3 bus entry, the stored state after a timeout differed between the two repeats on 0.24.12 and again on 0.24.13, and was stable from 0.24.14 on, the release that edited the P3 target file. No pair moved on either channel set.

On the P2 HTTP entry the SYS count moved in eight of the eight pairs where it was compared, by at most six pairs, and each of those 31 pairs traces to a log cell that was unstable in one of the two releases. That is the noise of Section 4.2 again and supports no claim.

The releases were not empty. Three of the ten pairs changed no source line in the five modules the paths run through; the other seven changed between 8 and 320 lines. But the changes rarely reached the error handling on these paths. Of the six files that hold most of it (the target files of Section 3.5), only two were edited in any of these releases. One is `InvoiceDispatcher` in 0.24.18, the change described above. The other is `InvoicePaymentControlPluginApi` in 0.24.14, where a retry option was added and no catch block or log statement changed. On P3 no pair moved; the stored-state cell noted above is the only difference.

So over eleven releases and about two years, we could trace one change in the evidence on these paths to the project's own maintenance. It moved evidence between channels, and no step merged any failures.

4.5 AI maintenance: no failures merged

Table 6 summarises the AI changes.

Table 6. AI maintenance changes, totals

Group	Changes	Applied	Compiled	Tests pass	Outcome kept	Lost >= 1 pair, SYS	Lost >= 1 pair, ALL	Gained >= 1, SYS / ALL
All changes	144	143	141	140 of 141	141 of 141	0 of 141 (0%; 0-3%)	0 of 141 (0%; 0-3%)	15 / 1
Plain wording	72	72	71	70 of 71	71 of 71	0 of 71 (0%; 0-5%)	0 of 71 (0%; 0-5%)	7 / 1
Guarded wording	72	71	70	70 of 70	70 of 70	0 of 70 (0%; 0-5%)	0 of 70 (0%; 0-5%)	8 / 0
Sonnet 5.5	72	71	69	68 of 69	69 of 69	0 of 69 (0%; 0-5%)	0 of 69 (0%; 0-5%)	6 / 1
Opus 5.5	72	72	72	72 of 72	72 of 72	0 of 72 (0%; 0-5%)	0 of 72 (0%; 0-5%)	9 / 0
R1 reduce duplication	48	48	46	45 of 46	46 of 46	0 of 46 (0%; 0-8%)	0 of 46 (0%; 0-8%)	4 / 0
R2 error handling	48	47	47	47 of 47	47 of 47	0 of 47 (0%; 0-8%)	0 of 47 (0%; 0-8%)	5 / 1
R3 logging style	48	48	48	48 of 48	48 of 48	0 of 48 (0%; 0-7%)	0 of 48 (0%; 0-7%)	6 / 0
P1	24	24	23	23 of 23	23 of 23	0 of 23 (0%; 0-15%)	0 of 23 (0%; 0-15%)	0 / 0
P2	24	24	24	24 of 24	24 of 24	0 of 24 (0%; 0-14%)	0 of 24 (0%; 0-14%)	15 / 0
P3	24	24	24	23 of 24	24 of 24	0 of 24 (0%; 0-14%)	0 of 24 (0%; 0-14%)	0 / 0
P4	24	23	22	22 of 22	22 of 22	0 of 22 (0%; 0-15%)	0 of 22 (0%; 0-15%)	0 / 0
P5	24	24	24	24 of 24	24 of 24	0 of 24 (0%; 0-14%)	0 of 24 (0%; 0-14%)	0 / 1
P6	24	24	24	24 of 24	24 of 24	0 of 24 (0%; 0-14%)	0 of 24 (0%; 0-14%)	0 / 0

Model and wording cells: opus guarded 0 of 36 (0%; 0-10%); opus plain 0 of 36 (0%; 0-10%); sonnet guarded 0 of 34 (0%; 0-10%); sonnet plain 0 of 35 (0%; 0-10%) (SYS; ALL identical).

Of the 144 changes, 143 applied. The one that did not (P4, R2, guarded, Sonnet) opened two replacement blocks and completed one. Two did not compile, both refactorings by Sonnet. On P1 a lock variable was removed while a `finally` block still used it. On P4 the compiler could not find a symbol the new code referred to. The remaining 141 all kept the business outcome. One failed the existing tests on its first run: in a P3 refactoring by Sonnet, the payment suite stopped after three tests. The same code passed all 186 tests on the declared re-run, so the change did not cause the failure, but it stays counted as a failure. The 141 compiled changes are the ones measured.

None of the 141 lost a distinguishable pair, on any entry, on either channel set. Across all of them the proportion is 0 of 141, with a 95% interval of 0 to 3%. Taken group by group, every path, task, wording and model shows zero, with upper bounds between 5% and 15%. Each of the four combinations of model and wording shows 0 of 34 to 0 of 36 (upper bound 10%). The guard sentence had nothing to prevent.

Table 7 shows what the changes did on each path.

Table 7. AI maintenance changes per path: size of the changes and what they did to the evidence

Path	Changes measured	Mean lines changed	Lost (SYS / ALL)	Gained SYS	Gained ALL	Altered cells, no pair lost
P1	23	100	0 / 0	0	0	3
P2	24	18	0 / 0	15 (+1 to +4)	0	4
P3	24	61	0 / 0	0	0	6
P4	22	12	0 / 0	0	0	0
P5	24	28	0 / 0	0	1 (+6)	3
P6	24	97	0 / 0	0	0	0

Mean lines changed = lines added plus lines removed, over the changes that applied. Altered cells = changes with at least one signature cell different from the baseline, leaving out the P2 HTTP log and attached-text channels, which vary without any code change. P2 SYS gains are all on the HTTP entry, within no-op noise.

The changes were not trivial. On P1 and P6 they changed about a hundred lines on average; on P4, whose target is 81 lines long, about a dozen. On P1, most of the refactoring and error-handling changes merged the catch blocks of the billing lookup into a single block that catches several exception types at once. Several moved repeated code into helper methods. Yet the evidence moved little. Sixteen changes altered at least one signature cell outside the P2 HTTP log channels, and none of them lost a pair. Four reworded a log line on an injected path. On P2 an Opus change turned "Call TIMEOUT for accountId=.." into "Call timed out for accountId=..". On P1 an Opus change shortened "Ignoring invoice generation process for accountId=.." to "Ignoring invoice generation for accountId=..". On P5 two Opus changes appended ": no usable status from plugin" to the janitor's "Unable to repair" line. Each rewording changed the line for every failure kind that writes it, so the grouping stayed the same. One more change, made under the guarded wording, reworded the message of the lock failure on P2, which only that kind produces, so it too could merge nothing. One added a warning, described below. In the remaining ten, the only altered cells are stored-state cells that were unstable in that unit.

The gains are few. All 15 SYS gains are on the P2 HTTP entry, between one and four pairs, the size of the no-op noise there. The one gain elsewhere returns to the scene of the introduction. A Sonnet change to the janitor, on the error-handling task, added a warning to the catch block that turns any plugin failure into an unchanged unknown status:

```
log.warn("Unable to retrieve plugin info for paymentId='{}',
paymentTransactionId='{}'", ..., e)
```

Because the warning carries the exception, the janitor queue now prints the exception text, and its four failures become distinguishable on all channels: six pairs gained. The warning's own wording is the same for every failure, so on what the system itself states nothing changed.

Two reasons for the zero can be read off the data. First, the changes reworded lines that several failures already shared, or lines that only one failure writes; in neither case can a rewording merge two failures. On P1, for instance, the five catch blocks of the billing lookup already logged one message in 0.24.18, so merging them changed nothing as long as the exception was still logged. Second, no change removed the attached exception text on which a pair depended; the models kept passing the exception to the logger.

5. What it means

Most of the loss is already in the code. For a team running this system, the baseline matters more than any change we measured. About a quarter of the pairs (56 of 235, outside the noisy entry) cannot be told apart from codes, own log lines and stored state. Several of the groups pair a failure with a message that names a different cause: "No invoice to generate" for a missing account, "data corruption" for a refused connection, an empty list for a null answer. These are places where a small, deliberate change, a distinct code or message for each kind of failure, would add evidence. One of the AI changes did a version of this by accident: it logged the error it caught, and four failures that had left identical records became distinguishable.

The tests guard values, not evidence. The existing tests behaved consistently. They failed when an edit changed an error code, an HTTP status or a business result that some test states. They passed for every edit that only took evidence away, including one that makes a timeout look like a missing account to anything that reads the error code. The planted edits found no test in these suites that checks that two failures produce different evidence. In this system a loss of this kind would not be caught by the project's own tests unless it happened to change a value a test pins.

A check for this can be an ordinary test. The instrument used here is one test per path that injects each failure kind, plus a comparison of what each run reported. Kept with the project, it would turn "these failures stay distinguishable" into something the build checks. The planted edits show what such a check catches that the existing tests do not.

The evidence is stable, and so are its gaps. In ten steps between releases, we could trace one change in the evidence on these paths to the code. The code that handles these failures is edited rarely, which helps anyone who has learned to read this system's failures. It also means the gaps of Section 4.1 are not left over from recent churn. The same pairs were indistinguishable in the earliest release we measured, and about two years of maintenance did not separate them. They are likely to stay until someone separates them on purpose.

Channels matter. Release 0.24.18 lost pairs in the log and gained them in the attached text. Whether that is a loss depends on who reads the evidence: an engineer reading stack traces loses nothing, an alert keyed on log wording does. A measure that reports a single number over all channels would have missed the shift; one that reports a single channel would have called it a loss.

The AI changes did not merge failures here. In 141 measured changes to the files that hold most of the error handling on six paths, no pair of failures stopped being distinguishable. The interval says how strong that result is. If AI changes of this kind merged failures more often than about 3% of the time, we would very

probably have seen at least one case among the 141. The guard sentence made no measurable difference, because there was nothing for it to prevent.

Section 4.5 shows why: the models did not remove the evidence that separates failures. A system with a more specific message for each kind of failure would give a rewording more to merge, and is the natural next place to measure.

6. Threats to validity

6.1 What the measures show (construct validity)

- **The definition.** "Distinguishable" is a strict, mechanical notion: two kinds differ consistently on at least one channel in two repeats. It does not say whether the difference is easy for a person to see or whether an operator would act on it. A pair that differs only in a log line's wording counts as distinguishable.
- **The channel sets.** The primary set excludes the result type, the exception message and the attached text because they can repeat the text we injected. Some of Kill Bill's own log lines also echo exception messages, for example the payment "EXITING" line and "Operation failed". These lines are part of L, so L could separate some kinds only through text that the experimenter chose. We measured how much this matters. With the injected messages masked in the log lines, 178 of the 235 pairs stay distinguishable instead of 179; the one pair that merges is the two plugin exceptions on the P2 Java entry, which the "EXITING" line separated only by their message. With the class names of the injected exceptions masked as well, 175 stay distinguishable; the three further pairs are the timeout and the refused connection on the P2 Java entry and both P3 entries. The 56 indistinguishable pairs of Section 4.1 therefore become 57 or 60. Nearly all of the separation comes from what Kill Bill itself chooses to state: its codes, its own wording and its stored state.
- **Injected failures are stand-ins.** A "timeout" is an exception of a timeout type thrown by a mock, not a real delay; only the payment plugin's delay switch produces a real wait. Real outages may leave more or different evidence.
- **The set of failure kinds** was chosen by a fixed rule, but the number of groups depends on it.
- **The outcome.** "Outcome kept" compares the result type and the stored business data. It treats a raised exception as the same outcome whatever its type; PB_P1_4 kept the outcome under this definition although the caller receives a different exception.
- **The P2 HTTP entry.** Its log channel is unstable in the baseline and a no-op moves its SYS count by three pairs, so P2 HTTP results on SYS support no claim. P2 conclusions rest on the Java entry and on HTTP with all channels.

6.2 What could affect the results (internal validity)

- **Load.** Runs shared one machine, up to four at a time. Load produced the unstable stored-state cells of Section 4.2, none of which changed a pair count, and one set-up crash, which was re-run under a declared rule.
- **Targets.** P1's target is the file of the pilot, `InvoiceDispatcher`. P4's target runs on its bus entry only, so changes there can move pairs only on that entry.

- **Existing tests.** Per change we ran the test suite of the changed file's module, the suite a developer would run for that file. Kill Bill's end-to-end suite and the full suite of the server profile are outside the study. For the two planted edits outside the payment, invoice and overdue modules we ran the most relevant suites.
- **Flaky tests.** One AI change failed its suite on the first run and passed on the re-run; we count it as a failure.
- **The release comparison** was extended on 20 November 2026, under a rule written before the added runs and not changed afterwards. On the two earliest releases the P2 test does not compile, so P2 is compared over fewer pairs. The comparison covers the paths and failure kinds we inject.

6.3 How far the results reach (external validity)

- **One system.** All results come from Kill Bill, and mostly from its payment module (four of six paths). Systems with different logging habits could behave differently.
- **Two models from one provider,** at one setting, with three tasks and two wordings. Other models, agents with tools, or other tasks (adding features, fixing bugs) may change error handling more.
- **One file per change.** Each AI change touched only the target file, and the model saw only that file. Changes that span files, as real maintenance often does, were not studied.

6.4 What the numbers support (conclusion validity)

A count of zero does not prove a rate of zero. For the AI changes the interval reaches 3% overall and between 5% and 15% per group. Eight planted edits had an effect and five of them kept the outcome, so the interval for how often the tests would catch such an edit runs from 0 to 52%. The catalogue of planted edits was written before measurement as a check of the instrument and the tests; it is not a sample of the degradations that occur in practice. Ten release pairs give an interval of 0 to 31% for the rate of a loss per release step.

7. Related work

Logging for failure diagnosis. Yuan and colleagues studied failures in five systems and found that existing logs often did not suffice to diagnose them; their tool, Errlog, adds log statements at common error sites [Yuan et al. 2012]. Log20 places log statements to best separate a program's execution paths, scoring candidates with an information-theoretic measure [Zhao et al. 2017]. It is the closest existing measure to ours, but it separates execution paths in one version of a program to guide where logs go, where we separate injected failure kinds across versions. Cinque and colleagues used software fault injection to assess how well logs capture faults [Cinque et al. 2010]. Defects4Log collects defective logging statements and asks whether language models can find them in source code [Wang, Li and Ding 2025]. None compares the same system before and after a change.

Fault injection and observability. Borges and colleagues built OXN, which injects faults into microservice applications to compare observability configurations by how visible the faults are [Borges et al. 2024], and later proposed running such experiments continuously so that later modifications are checked [Borges and Werner 2025]. Their comparison is between configurations of the monitoring, and visibility means a fault against normal operation. We compare versions of the application code, and failures against each other. ChaosMachine injects exceptions into the try-catch blocks of running Java applications to analyse how their

handlers behave [Zhang et al. 2019]. OpenRCA 2.0 builds a root-cause benchmark from fault injections and reports that some distinct injection mechanisms leave indistinguishable telemetry, at the level of infrastructure faults in one version of each system [Fang et al. 2026].

AI-written code and failure evidence. Tao and colleagues asked language models to generate whole microservice systems and injected faults into them; the generated systems exposed fault-specific signals for at most 13.99% of failures, and their logs seldom carried what is needed to tell failure types apart [Tao et al. 2026]. Their systems are newly generated, measured once, through logs only. We study changes to an existing system's evidence across several channels. Coleman and colleagues applied identical faults to a scientific Fortran program before and after a language model modernised it, comparing numerical outcomes [Coleman et al. 2026]. Their design is the closest to ours in pairing faults before and after an AI change; their faults are numerical perturbations and their comparison is of results, not of diagnostic evidence.

AI refactoring and maintenance. Dristi and Dwyer used differential fuzzing to find that 19 to 35% of refactorings generated by language models were not functionally equivalent, and that about 21% of those passed the existing tests [Dristi and Dwyer 2026]. Waseem, Ahmad and Abrahamsson proposed a multi-agent refactoring tool for Java and found that preservation risks remained [Waseem et al. 2026a], and in a second study classified most generated refactorings as carrying contract or dependency risk [Waseem et al. 2026b]. Patel and colleagues found that coding agents resolve fewer later tasks when building on agent-written code [Patel et al. 2026]. These studies measure function, structure or later maintainability; none injects failures. Our measurement of AI changes asks what such changes do to failure evidence, and finds no loss in this system.

Agents and logging. Ouatiti and colleagues mined 4,550 agent pull requests and found that explicit logging instructions were rare and often not followed [Ouatiti et al. 2026]. LogRem asked coding agents to remove logging code and found that they rarely removed the same statements as developers, though their output nearly always passed validity checks [Yuan et al. 2026]. These describe what agents do to log statements; we measure what changes do to the evidence a running system produces under failure.

Exception handling over time. Cacho and colleagues studied how exception-handling code evolves across versions of C# and Java programs and how this relates to robustness [Cacho et al. 2014a; Cacho et al. 2014b]. They analysed source history; nothing was executed under failure.

To our knowledge no earlier study measures, on one real system, whether injected failures stay distinguishable from each other across several kinds of evidence after code changes, and whether the system's own tests notice. The study builds on two earlier ideas: injecting the same faults before and after an AI change [Coleman et al. 2026], and treating observability as a property to test for regressions [Borges and Werner 2025].

8. Conclusion

We asked how well failures can be told apart from what a system states, and what happens to that when the code changes. In Kill Bill 0.24.18 much of the evidence is already merged: 56 of 235 pairs of failures cannot be told apart from what the system states, and on one entry four plugin failures leave identical traces everywhere. The project's tests do not guard what remains. Planted edits that removed evidence while keeping the business outcome went unnoticed in all five cases; the tests caught only edits that changed a

code, status or outcome they check. The project's own releases changed this evidence rarely: in ten steps between eleven consecutive releases, we could trace a change in what failures leave behind on these paths to the code of one release, and it moved evidence between channels without merging any failures. And 141 AI maintenance changes to the error-handling files of six paths did not make any pair of failures harder to tell apart (0 of 141; 95% interval 0 to 3%), with or without an instruction to preserve them.

For the system studied, the larger risk to diagnosis lies in the code as it stands and in tests that do not check evidence, not in the AI changes we measured. A test that injects each failure kind and compares what the system reports would make the property something the build checks. Whether changes behave the same way in systems with more distinctive error handling, across several files, or when made by autonomous agents, is the question that follows.

Declarations

Funding. No funding was received for this work.

Competing interests. The author has no relevant financial or non-financial interests to disclose.

Ethics approval. Not applicable. The study uses open-source software and synthetic test data, and involves no human participants.

Use of AI. AI-based tools assisted with the programming and the writing of this paper.

Data availability

The replication package is published at https://doi.org/10.5281/zenodo.24429382 [Thapa 2026]. It contains the injection tests for the six paths, the scripts that apply, build, test and measure each change, the AI changes and the 27 planted edits as patches, the raw captures of every injection run on every release measured, the recorded results, and the scripts that regenerate every table.

References

- Borges, M. C., Bauer, J., Werner, S., Gebauer, M., Tai, S. (2024). Informed and Assessable Observability Design Decisions in Cloud-native Microservice Applications. 21st IEEE International Conference on Software Architecture (ICSA 2024). doi:10.1109/ICSA59870.2024.00015. arXiv:2403.00633.
- Borges, M. C., Werner, S. (2025). Continuous Observability Assurance in Cloud-Native Applications. 22nd IEEE International Conference on Software Architecture (ICSA 2025), Poster Track. arXiv:2503.08552.
- Cacho, N., César, T., Filipe, T., Soares, E., Cassio, A., Souza, R., Garcia, I., Barbosa, E. A., Garcia, A. (2014a). Trading Robustness for Maintainability: An Empirical Study of Evolving C# Programs. Proceedings of the 36th International Conference on Software Engineering (ICSE 2014), 584–595. doi:10.1145/2568225.2568308.
- Cacho, N., Barbosa, E. A., Araujo, J., Pranto, F., Garcia, A., Cesar, T., Soares, E., Cassio, A., Filipe, T., Garcia, I. (2014b). How Does Exception Handling Behavior Evolve? An Exploratory Study in Java and C# Applications. 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME 2014), 31–40. doi:10.1109/ICSME.2014.25.

- Cinque, M., Cotroneo, D., Natella, R., Pecchia, A. (2010). Assessing and Improving the Effectiveness of Logs for the Analysis of Software Faults. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010), 457–466. doi:10.1109/DSN.2010.5544279.
- Clopper, C. J., Pearson, E. S. (1934). The Use of Confidence or Fiducial Limits Illustrated in the Case of the Binomial. Biometrika 26(4), 404–413. doi:10.1093/biomet/26.4.404.
- Coleman, E., Shen, Y., Sosonkina, M., Xu, P. (2026). Validating LLM-Modernized Scientific Software Through Differential Fault Injection. arXiv:2608.14527.
- Dristi, S. B., Dwyer, M. B. (2026). A Differential Fuzzing-Based Evaluation of Functional Equivalence in LLM-Generated Code Refactorings. arXiv:2602.15761.
- Fang, A., Yang, Y., Shang, J., Lu, Q., Xu, J., Wang, R., Zhang, S., Zhang, Y., Yu, B., He, P. (2026). OpenRCA 2.0: From Outcome Labels to Causal Process Supervision. arXiv:2606.27154.
- Ouatiti, Y. E., Sayagh, M., Li, H., Hassan, A. E. (2026). Do AI Coding Agents Log Like Humans? An Empirical Study. arXiv:2604.09409.
- Patel, S., Hou, B. L., Purohit, A., Xu, K., Pan, J., He, H., Chen, V. (2026). Is Agent Code Less Maintainable Than Human Code? arXiv:2606.21804.
- Tao, Y., Zhang, H., Gao, P., Ma, M., Fan, Z., Kang, Y., Zhang, J., Qin, S., Li, L., Lin, Q., Rajmohan, S. (2026). Can Large Language Models Generate Observability-Aware Code? arXiv:2607.05785.
- Thapa, S. J. (2026). Telling Failures Apart: Replication package [Data set]. Zenodo. https://doi.org/10.5281/zenodo.24429382
- Wang, X., Li, Z., Ding, Z. (2025). Defects4Log: Benchmarking LLMs for Logging Code Defect Detection and Reasoning. arXiv:2508.11305.
- Waseem, M., Ahmad, A., Abrahamsson, P. (2026a). REFINE: A Multi-Agent LLM Approach for Evidence-Guided Code Refactoring. arXiv:2608.23611.
- Waseem, M., Ahmad, A., Abrahamsson, P. (2026b). When LLM-Generated Refactorings Ripple: An Empirical Study of Repository Adoption Readiness. arXiv:2610.24790.
- Yuan, D., Park, S., Huang, P., Liu, Y., Lee, M. M., Tang, X., Zhou, Y., Savage, S. (2012). Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2012), 293–306.
- Yuan, H. Y., Zhang, H., Wang, X., Chen, A. R., Kim, K., Ding, Z., Li, Z. (2026). Towards Automatically Pruning Logging Code with Coding Agents: How Far Are We? arXiv:2610.04716.
- Zhang, L., Morin, B., Haller, P., Baudry, B., Monperrus, M. (2019). A Chaos Engineering System for Live Analysis and Falsification of Exception-handling in the JVM. IEEE Transactions on Software Engineering. doi:10.1109/TSE.2019.2954871. arXiv:1805.05246.
- Zhao, X., Rodrigues, K., Luo, Y., Stumm, M., Yuan, D., Zhou, Y. (2017). Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold. Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP 2017), 565–581. doi:10.1145/3132747.3132778.

Software

- Kill Bill, open-source subscription billing and payment platform. https://github.com/killbill/killbill, release tags `killbill-0.24.12` (tagged 29 October 2024) to `killbill-0.24.22` (tagged 28 September 2026); the release studied in detail, `killbill-0.24.18`, was tagged on 14 May 2026.