calyntro

WHITEPAPER · REPOSITORY INTELLIGENCE

Predicting Defects Before They Happen

A prospective validation of Git-history risk signals across four production codebases — MongoDB Server, uv, Qt Base, and Apache Flink — measured without lookahead against real bug-fix activity.

0.71–0.84

AUC across four codebases (6-month window), strongest on Qt Base

72%

Of future bug-affected files concentrated in the top quartile of a churn-based risk ranking

4

Codebases, three languages, one consistent structural signal

Calyntro — Self-hosted Repository Intelligence
For CTOs, VPs of Engineering, and Engineering Managers

calyntro.com
demo.calyntro.com

Overview

At a glance

WHY THIS STUDY EXISTS

Most engineering organisations assess code risk by looking backward — at test coverage, past incidents, or a senior engineer's gut sense of where things are fragile. Calyntro's core claim is different: that the shape of a team's commit history already contains a forward-looking signal about where the next defect is likely to land. This whitepaper puts that claim to the test. We computed risk scores from Git history at a fixed cutoff date, then checked — months later, on data the scoring never saw — whether those scores actually predicted which files went on to receive bug fixes. They did, consistently, across four codebases in three languages.

0.71–0.84

AUC across the four codebases (6-month window), strongest on Qt Base

72 / 26

% of bug-affected files captured by the top-risk 26% of files, Qt Base (3-month window)

4 / 4

Codebases where the signal held up above random assignment, across three languages

CONTENTS

01 **The Problem — Where Will the Next Bug Surface?**
Why code review, coverage, and incident history all miss the same thing — and what a Git history actually encodes.

02 **Methodology — How We Measured It**
The time-lag design that rules out lookahead, how defects were labelled, and how accuracy was scored.

03 **The Ownership Paradox — Quality Now vs. Fragility Tomorrow**
Why concentrated ownership can go along with fewer bugs today, and why Calyntro still flags it as a risk.

04 **Results Across Four Codebases**
MongoDB Server, uv, Qt Base, and Apache Flink.

05 **What This Means for Engineering Leaders**
Four practical consequences for review policy, refactoring cases, org design, and incident accountability.

A **Appendix — Dataset Details and Methodology Notes**
Candidate selection, label extraction, predictor definitions, and study limitations.

01 · The Problem

SECTION 01

The Problem: Where Will the Next Bug Surface?

A production incident is underway. Somewhere between the payment edge case, the race condition in the auth layer, and the memory leak that only shows up under load, someone on the team is already drafting the post-mortem. One line tends to show up in nearly every version of that document: "We had no indication this module was at risk."

In most cases, that line isn't quite true. The file had been touched repeatedly for months. The person who understood it best had left the team eight weeks earlier. Complexity had been climbing for a year. No single fact was alarming by itself, but together they pointed straight at the module that failed. The signals existed. Nobody had aggregated them into something anyone was looking at.

How risk gets assessed today

Engineering organisations typically lean on four sources to judge where quality problems are likely to show up next, and each one has a real but limited job to do.

Code review and static analysis catch problems at the point of introduction — the diff in front of the reviewer. What they can't see is the structural history behind that diff. A file that has been fragile for two years looks exactly like a stable one the moment a new commit lands on top of it.

Test coverage tells you which lines executed during a test run, not whether the behaviour is correct or whether the module will hold up under future change. Plenty of high-coverage modules fail in production; plenty of low-coverage ones sit untouched for years.

Incident history is useful but reactive by construction. It tells a team where the last fire started, and it quietly underweights modules that have been fragile but simply hadn't broken yet.

Engineering intuition is often remarkably accurate — senior engineers usually know where the bodies are buried. It just doesn't scale, doesn't survive when that engineer leaves, and doesn't hold up when a decision needs to be explained to a board.

None of these four are wrong. They just don't answer the question an engineering leader actually needs answered before the incident, not after it: which files are most likely to need a bug fix in the next three to six months?

The record is already sitting in the commit history

> Every change a team makes leaves a trace. Over enough time, those traces form a structural record of how the software has actually evolved — not how the architecture diagram says it should.

Which files accumulate changes faster than the rest. Which are touched by many hands and which by one. Which are growing more complex while fewer people understand them. None of this is noise — it reflects the real pressure points of the development process, the modules that sit at the intersection of too many competing concerns.

Software engineering research has documented for decades that these structural properties correlate with defect density. Files that change in frequent, concentrated bursts; files whose institutional knowledge has narrowed to one person; files that sit at the confluence of many dependencies — these are where bugs accumulate, and not at random.

The gap isn't in the evidence. It's between that evidence and the decisions engineering leaders make every week about where to point review capacity, where to invest in refactoring, and who needs to shadow whom before a key person leaves.

What changes once the signal is validated

Knowing a structural signal exists is different from knowing it holds up reliably in production codebases, across languages and team sizes, and — critically — that it works prospectively, without quietly using future information to make the prediction look better than it is.

A signal that retrospectively explains defects that already happened is an interesting research finding. A signal that identifies, ahead of time, which files will need bug fixes in the months to come is an operational tool. This whitepaper reports a validation study built to test the second kind: risk signals computed from Git history at a fixed point in time, checked against which files actually received bug-fix commits in the three and six months that followed, across four large production codebases, three languages, and four distinct project profiles.

The results hold up. The signal is real, it's consistent across all four codebases, and it's precise enough to change how an engineering leader allocates a finite amount of review and refactoring time.

02 · Methodology

SECTION 02

Methodology: How We Measured It

The one rule that makes or breaks a study like this

A method that uses future information to predict the future isn't a prediction — it's a description dressed up as one. That sounds obvious, but it's exactly where most defect-analysis approaches quietly fail: they compute risk scores from the full commit history, including commits that post-date the very bugs they claim to have predicted. The result looks convincing in a slide deck and is useless the moment a team tries to act on it before the fact. Our design rules this out entirely.

A strict before-and-after split

We picked a fixed cutoff date, T, and split each repository's history into two periods that never overlap.

Before T — the prediction window. Every risk signal was computed exclusively from commits made before T: how often each file changed, how authorship was distributed across contributors, which files pulled in a disproportionate share of activity relative to the rest of the codebase. Nothing from after T entered the calculation.

After T — the observation window. We then looked forward and tracked which files received bug-fix commits in the months that followed, using two horizons — three months and six months — to see whether the signal holds at both.

The score for a file comes entirely from data before T. The label — did this file get a bug fix afterward? — comes entirely from data after T. There is no lookahead by design, not by discipline.

Figure 1. The time-lag validation design: scores and labels are computed from strictly non-overlapping periods.

This reproduces exactly the situation a team is in when they run Calyntro in production: scores computed from history, applied to the present, and checked against what actually happens next.

What counted as a defect

We used the public ticket data each project already links to its commits — Jira for MongoDB, Qt and Flink, GitHub Issues for uv — and identified commits whose associated tickets were classified by the project's own maintainers as bugs, defects, or errors.

That matters for two reasons. First, it avoids defining a defect using the same signals we're trying to validate — a file changing frequently isn't automatically a defect, it has to be tied to an actual bug report. Second, it gives us labels grounded in maintainer judgment rather than in automated static-analysis output, which varies wildly in precision from one language and codebase to the next.

This dependency on maintainer-assigned ticket types is also a structural constraint of the method — see the note on tracker discipline in the Appendix (A.6), which explains why one additional candidate codebase was excluded from this study on exactly this basis.

Files with at least one bug-fix commit in the observation window were labelled positive. All other files with at least three commits before T were labelled negative, including files that saw no activity after T. This mirrors the question a team actually faces: out of all the files in my codebase, which ones will need a bug fix?

How we scored accuracy

We report two metrics, both standard in predictive analytics but worth a plain-language explanation for a reader more used to delivery metrics than statistical ones.

Discrimination (AUC) measures how well the risk scores separate files that go on to receive bug fixes from files that don't. A score of 0.5 means the signal carries no information — no better than a coin flip. A score of 1.0 means perfect separation, which no real predictor ever reaches. Defect-prediction studies commonly report AUC values of roughly 0.6–0.8 for history-based signals; what matters in practice is how far above 0.5 a signal sits, and how consistently it holds across different codebases.

Concentration (top-quartile) is the more intuitive number for a leader who needs to make a practical call. It answers a direct question: if you take the approximately 25% of files ranked highest risk by churn, what share of the bug-affected files that show up later do they actually account for? A random baseline concentrates roughly 25% of bug-affected files in 25% of files, by definition. Anything meaningfully above that means the signal is doing real work — narrowing a large codebase down to the slice where most of the trouble is about to occur. *(Note: the exact quartile boundary may cover slightly more than 25% of files when multiple files share the same churn score at the boundary — see A.2 for per-codebase figures.)*

Choosing the codebases

We selected candidates on three criteria: the repository and its issue tracker both had to be public, there had to be enough pre-T history to produce a meaningful signal, and the set as a whole needed to span different languages, team sizes, and maturity levels.

MongoDB Server (C++), uv (Rust), the Qt Base repository (C++), and Apache Flink (Java) span a mature maintenance codebase, a fast-growing open-source tool, an industrial-scale framework widely used in embedded and systems software, and a large Java platform for distributed data processing. Full dataset details and cutoff dates are in the Appendix.

We had no prior involvement with any of the four projects and no way to influence the results after the fact. The same cutoff date T was used for all four codebases.

03 · The Ownership Paradox

SECTION 03

The Ownership Paradox: Quality Now vs. Fragility Tomorrow

One finding from the validation study looks, at first glance, like it contradicts one of Calyntro's own metrics — and it's worth walking through carefully, because getting this wrong in front of a CTO is exactly the kind of thing that undermines an otherwise solid pitch.

In MongoDB Server — our largest, most mature codebase — files with a single dominant contributor went on to accumulate fewer bug fixes, not more. The pattern is not uniform across codebases (see the note below). Yet Calyntro's Silo Risk analysis flags exactly these files as high-risk. Two metrics, seemingly pointing in opposite directions, both computed from the same underlying data.

They aren't in conflict. They're measuring two different things.

What ownership concentration actually captures

Ownership concentration describes how commit authorship for a file is distributed up to a given point in time — 90% means one author wrote or touched nine out of every ten commits; 40% means authorship is spread more evenly. In MongoDB, high concentration went along with fewer defects, and a plausible reason is easy to understand: a single author who keeps coming back to the same file builds up deep contextual knowledge. They remember why the edge case on line 247 exists and what broke three years ago. That depth of context is what prevents mistakes.

This is a structural quality signal — a read on how well the code is understood right now, by the people currently maintaining it.

What Silo Risk actually captures

Silo Risk isn't measuring code quality at all. It's measuring organisational resilience.

A file with one dominant author tends to be high quality precisely because of that concentrated expertise — but the quality is contingent on that person staying available. When they leave, go on extended leave, change teams, or simply become the bottleneck for every question about the module, the accumulated context doesn't transfer on its own. The bus factor is one. A quality trajectory that was being protected by a single expert now faces an uncontrolled transition.

Silo Risk is a flag for that fragility. It answers a different question than the ownership-quality signal: not "how bug-prone is this file today?" but "how exposed are we if the person keeping this file healthy becomes unavailable?"

Two dimensions, not two opinions

These aren't competing metrics. They're orthogonal — and both need to be visible to an engineering leader at the same time.

	Low ownership concentration	High ownership concentration
Author still present	**Knowledge is distributed; some coordination overhead, but no single point of failure.**	**Deep expertise, fewer defects — a stable situation today.**
Author at risk of leaving	**Knowledge is already distributed — the org absorbs the departure without much disruption.**	**The blind spot: code quality has been depending on one person the whole time.**

The bottom-right cell is where the two signals converge into something urgent: high structural quality today, combined with high fragility tomorrow. These files aren't breaking. That's exactly why they tend to stay off everyone's radar until the person holding them together leaves — at which point they become the next post-mortem's opening line.

Why the distinction changes what a team does next

Most defect-prediction tools treat ownership concentration as simply good or simply bad. Our results suggest the truth is more contextual: in some codebases concentration goes along with fewer defects, in others with more — and it becomes a liability the moment the organisational context shifts.

- **Ahead of a restructuring** — files with high concentration and high silo risk are the first candidates for knowledge-transfer sessions, not because they're producing bugs now, but because the restructuring is about to remove the stability that's been protecting them.
- **Ahead of a departure** — losing someone from a high-concentration module isn't just a staffing gap. It's a latent defect risk that will surface over the following months.
- **In review policy** — pairing a second reviewer onto high-concentration files isn't about catching bugs that exist today. It's about building the secondary coverage that keeps those files resilient once the primary owner is gone.

These are plausible, well-reasoned consequences; our study did not test them directly.

Calyntro surfaces both dimensions independently, side by side. Ownership concentration shows where quality is currently well-protected. Silo Risk shows how fragile that protection actually is. An engineering leader needs both to see the full picture — either one on its own tells only half the story.

A NOTE ON SIGNAL DIRECTION

The link between ownership concentration and bug fixes is not uniform. In MongoDB, files with a dominant author received fewer bug fixes. In uv and Apache Flink, we saw no clear difference. In Qt Base, the direction was reversed. We therefore do not treat concentrated ownership as a quality guarantee. Silo Risk answers a different question — how exposed a team is if a key person becomes unavailable — and that question stands regardless of whether concentrated ownership currently goes along with fewer or more defects.

04 · Results Across Four Codebases

SECTION 04

Results Across Four Codebases

Four repositories, chosen to sit at distinct points in the space of real-world software projects: a large, mature C++ server application; a fast-growing systems tool written in Rust; an industrial-scale C++ framework with decades of history and a broad contributor base; and a large Java platform for distributed data processing. Each has a public issue tracker, which let us identify bug-fix commits entirely independently of the risk scores themselves.

All four used the same methodology: risk signals computed from commit history up to the cutoff date T (2025-08-15), labels drawn from each project's public issue tracker for the three- and six-month windows that followed. The same method and parameters were applied to all four codebases.

MongoDB Server — the mature maintenance project

One of the most widely deployed open-source database engines in production. The repository spans over 100,000 commits from its origins in 2007 through the cutoff date, covering more than 40 modules in a C++ codebase exceeding a million lines.

Dataset: 37,173 files with sufficient pre-T activity (≥ 3 commits before T) · full repository history from 2007 · 95%-bootstrap CI n=1,000

Window	Discrimination (AUC)	95% CI	Top-quartile concentration
T+3M (3 months)	0.751	[0.734, 0.769]	60.9% of bug-affected files in the top-risk 26.2% of the analysed files
T+6M (6 months)	0.739	[0.725, 0.753]	59.4% of bug-affected files in the top-risk 26.2% of the analysed files

60.9% of subsequently bug-affected files concentrate in roughly the top quarter of the analysed files, more than doubling the density relative to random selection across a dataset of 37,000 files.

uv — the active growth project

uv is a Python package manager and resolver written in Rust, built by Astral (astral-sh/uv), and at the time of analysis was among the fastest-growing open-source developer tools by commit velocity. Bug classification comes from the maintainer team via GitHub Issues.

Dataset: 871 files with sufficient pre-T activity (≥ 3 commits before T) · labels extracted via GitHub Issues (type: Bug) · 95%-bootstrap CI n=1,000

Window	Discrimination (AUC)	95% CI	Top-quartile concentration
T+3M (3 months)	0.789	[0.735, 0.845]	65.8% of bug-affected files in the top-risk 26.1% of the analysed files
T+6M (6 months)	0.781	[0.737, 0.825]	60.9% of bug-affected files in the top-risk 26.1% of the analysed files

The three-month AUC of 0.789 is a strong result — well within the upper range published defect-prediction research reports for structural signals of this kind. The pattern fits the project's stage: uv is shipping substantial new functionality at high velocity, which concentrates structural risk around the specific areas where new subsystems are introduced and stabilised.

Qt Base — the industrial-scale benchmark

Qt underpins a large share of embedded, industrial, and desktop software worldwide. The Qt Base repository we analysed spans over 80,000 commits across its full pre-cutoff history dating back to 2009, with bug tickets tracked through the project's public Jira instance at bugreports.qt.io — migrated mid-study to Atlassian Cloud, which meant handling a breaking API change in the data pipeline along the way.

Dataset: 16,804 files with sufficient pre-T activity (≥ 3 commits before T) · 10,318 Bug-type tickets total in tracker · 95%-bootstrap CI n=1,000

Window	Discrimination (AUC)	95% CI	Top-quartile concentration
T+3M (3 months)	0.817	[0.792, 0.839]	71.6% of bug-affected files in the top-risk 25.7% of the analysed files
T+6M (6 months)	**0.839**	**[0.820, 0.857]**	**75.5% of bug-affected files in the top-risk 25.7% of the analysed files**

The strongest results across all four codebases, and the signal strengthens over time: 72% of all files that received a bug fix in the three months after the cutoff were already in the top-risk quartile of the churn ranking. By six months, 75.5% of bug-affected files were in that same quarter — covering just 26% of the analysed files.

Apache Flink — the large Java platform

Apache Flink is a distributed stream- and batch-processing framework written in Java, developed in the open under the Apache Software Foundation. The repository spans more than 38,000 commits since 2010, and bug tickets are tracked in the project's public Jira. It broadens the study beyond C++ and Rust to a mainstream enterprise language.

Dataset: 23,056 files with sufficient pre-T activity (≥ 3 commits before T) · 95%-bootstrap CI n=1,000

Window	Discrimination (AUC)	95% CI	Top-quartile concentration
T+3M (3 months)	0.682	[0.632, 0.730]	50.0% of bug-affected files in the top-risk 25.0% of the analysed files
T+6M (6 months)	0.710	[0.669, 0.749]	57.5% of bug-affected files in the top-risk 25.0% of the analysed files

Flink is the weakest of the four results, and it is still clearly above chance: the top quarter of files by churn captured twice as many bug-affected files as a random selection at three months, and 2.3 times as many at six months. As in the other codebases, the signal strengthens between the three- and six-month window.

Cross-codebase summary

Codebase	Language	Profile	AUC (T+6M)	95% CI	Top-quartile (T+3M)
MongoDB Server	C++	Mature maintenance	0.739	[0.725, 0.753]	60.9% of bug-affected files in 26.2%
uv	Rust	Active growth	0.781	[0.737, 0.825]	65.8% of bug-affected files in 26.1%
Qt Base	**C++**	**Industrial scale**	**0.839**	**[0.820, 0.857]**	**71.6% of bug-affected files in 25.7%**
Apache Flink	Java	Large data platform	0.710	[0.669, 0.749]	50.0% of bug-affected files in 25.0%

Top-quartile boundaries sit slightly above 25% due to tied churn values at the boundary — files with identical churn counts land on the same side. Effective thresholds: 25.0%–26.2% across codebases.

Three things stand out once the four codebases sit side by side.

The signal holds up across languages and project types. MongoDB (C++), uv (Rust), Qt (C++), and Flink (Java) are four technically distinct environments, and in every one the churn-based risk signal beat random assignment clearly — AUC ranging from 0.68 to 0.84 across both windows, with 95% confidence intervals entirely above 0.5. The consistency matters more than any single number — it suggests the signal reflects something structural about how software actually evolves, rather than an artefact of one language or toolchain.

Signal strength varies between projects. The weakest result appears in Apache Flink, the strongest in Qt Base. Four codebases are a first sample; teams should expect results in a similar range, not an exact figure.

The top-quartile metric is the number that actually drives a decision. In the Qt analysis, 72% of the bug-affected files in the three months following the cutoff were already concentrated in the 26% of files that ranked highest by churn ahead of time. In uv, the same top quartile captured 66% of subsequently bug-affected files, and in Flink 50%. For a team that can't review everything with equal intensity, that concentration is where the decision actually lives.

05 · What This Means for Engineering Leaders

SECTION 05

What This Means for Engineering Leaders

A validated signal only matters if it changes a decision. Here are four places where it does.

1 · Review capacity is finite — allocate it by risk, not by habit

Code review is the most widely used quality gate in software development, and also the one most uniformly spread across a codebase regardless of actual risk. A commit to a stable module with five active contributors usually gets roughly the same review attention as a commit to a module that's been touched 80 times in six months by a single author who's about to leave.

The validation results show that uniformity is a systematic misallocation. Seventy-two percent of Qt Base's bug-affected files were concentrated in roughly a quarter of the analysed files; the remaining three-quarters accounted for under thirty percent. A risk-ranked view makes that allocation explicit, and defensible in front of a team or a board.

2 · Refactoring investment gets a verifiable before-and-after

Refactoring is one of the hardest cases to make in engineering leadership: the cost is immediate and concrete, the benefit is future and probabilistic. "We refactored the auth module" is a difficult argument when nothing has broken yet.

The validation methodology hands over exactly the evidence structure that argument needs. A module sitting in the top risk quartile carries a measurably elevated probability of a future defect — the cost of leaving it alone is quantifiable in terms of incident probability, not vague references to technical debt.

3 · Organisational change creates predictable windows of risk

This is the implication that gets underestimated most consistently. When a key contributor leaves a team, their institutional knowledge doesn't transfer on its own. The files they owned lose the contextual knowledge that kept them maintainable.

The same applies to team restructurings, onboarding periods, and ownership handovers between teams. These events are almost always predictable well in advance. A silo-risk view makes that exposure visible before it turns into an incident, and points directly at where structured pairing or documentation work is most urgently needed. This is a plausible consequence, not one our validation measured directly.

4 · "No indication of risk" starts to mean something different

Back to the post-mortem line from Section 1. "We had no indication this module was at risk" is usually an accurate description of what information the team had in hand.

What that sentence actually describes, in most cases, is the absence of a prospective, quantified risk signal — not the absence of the underlying evidence. Organisations running a validated risk signal know, with quantified confidence, where the next problem is most likely to surface, rather than hoping review and testing will be enough.

What this doesn't replace

None of the above suggests that code review, testing, static analysis, or incident response should get less attention. They solve a different part of the problem. Review and testing catch issues at the point of introduction; a defect-prediction signal identifies where the structural conditions make an introduction more likely in the first place, before any specific defect exists to catch.

The two layers work together. What the risk signal does replace is the unspoken, unverifiable assumption that risk is spread evenly across a codebase — an assumption the validation results show to be reliably wrong, across languages, project sizes, and development velocities.

A · Appendix

APPENDIX

Dataset Details and Methodology Notes

A.1 — Candidate selection criteria

Codebases were included on four criteria: (1) a public repository and a public issue tracker, so both commit history and bug classification are independently verifiable — no proprietary or customer data was used; (2) sufficient pre-cutoff history, with a minimum of three commits before T required for a file to enter the predictor set; (3) reliable bug classification, meaning the tracker had to provide explicit type labels applied by the project's own maintainers rather than inferred by keyword matching; and (4) no prior involvement with any of the four projects and no way to influence how bugs were classified or commits structured.

One additional candidate, CPython, was evaluated and excluded on criterion (3): the project uses `type-bug` rather than bug as its issue type, which produced effectively zero usable labels under our classification query. A pipeline fix is noted for future runs, but the candidate was excluded rather than retrofitting the label extraction after the fact. This is not an isolated quirk of one project — see the general note on tracker discipline in Limitations (A.6), which this exclusion illustrates.

A.2 — Dataset summary

Codebase	Language	Tracker	Pre-T history	Files (churn ≥ 3)	Bug tickets (total in tracker)
MongoDB Server	C++	Jira (jira.mongodb.org)	2007-10-19 – 2025-08-15 (~18 years)	37,173	~1,150 linked (T+3M window)
uv (astral-sh/uv)	Rust	GitHub Issues	2023-10-04 – 2025-08-15 (~2 years)	871	~240 bug-type issues linked
Qt Base	C++	Jira → Atlassian Cloud	2009-08-04 – 2025-08-15 (~16 years)	16,804	10,318 Bug-type (total in tracker, not window-specific)

Codebase	Language	Tracker	Pre-T history	Files (churn ≥ 3)	Bug tickets (total in tracker)
Apache Flink	Java	Jira (Apache)	2010-12-15 – 2025-08-15 (~15 years)	23,056	—

All four analyses used cutoff T = 2025-08-15. Observation windows: T+3M = 2025-11-15, T+6M = 2026-02-15.

All four validation imports used `analysis_since: 2000-01-01` — effective start date is the earliest commit in each repository. Population B: all files with ≥ 3 commits before T; files not touched after T receive label 0.

A.3 — Label extraction

A commit was classified as a bug-fix commit if it referenced a ticket whose issuetype was Bug, Defect, or Error (case-insensitive). Commits with no ticket reference, or tickets of another type, were not classified as bug-fix commits.

Files with at least one bug-fix commit in the observation window were labelled positive. All other files with at least three commits before T were labelled negative, including files that saw no activity after T. This mirrors the question a team actually faces: out of all the files in my codebase, which ones will need a bug fix?

For uv, ticket references follow the GitHub Issues convention (`#NNN`). For MongoDB, Qt and Flink, Jira project prefixes (`SERVER-`, `QTBUG-`, `FLINK-`) were used to link commits back to tracker records.

Qt-specific note: bugreports.qt.io migrated mid-study from a self-hosted Jira Server instance to Atlassian Cloud. The Jira REST API v2 endpoint used for ticket-type lookup was removed on the new platform (HTTP 410). Enrichment required detecting the redirect and switching to the v3 endpoint. This was handled inside the data pipeline; no Qt tickets were lost to the migration.

A.4 — Predictor computation

The validated predictor is the churn score: the count of distinct commits touching a file before T. Ownership concentration — the share of pre-T commits by a file's most active author — was examined as a secondary signal (Section 3). Calyntro's further scoring functions are proprietary and were not part of this validation.

A.5 — Interpreting AUC in this domain

Defect-prediction studies commonly report AUC values of roughly 0.6–0.8 for history-based signals. We report both AUC and top-quartile concentration throughout this whitepaper because the two answer different questions for different readers: AUC is the standard statistical measure of discrimination; the top-quartile figure translates that discrimination into a concrete, actionable share of a codebase.

A.6 — Limitations

Ticket quality varies across projects. Bug classification in public trackers reflects maintainer judgment, which isn't perfectly consistent within or across projects. These errors are expected to be roughly symmetric, and to weaken rather than inflate the measured signal.

Bug identifiability depends on tracker discipline — this is a general constraint of the method, not a one-off issue. Our label extraction relies on maintainers consistently assigning a distinguishable issue type (Bug/Defect/Error) to tickets, and on commits referencing those tickets in a way the pipeline can parse. Where that discipline is inconsistent — a project uses a non-standard type taxonomy, bug fixes go unreferenced in commit messages, or ticket-type data is simply unavailable — the resulting label set is sparse or noisy, and any AUC or top-quartile figure computed on it is correspondingly less trustworthy. CPython (see A.1) is the clearest example encountered in this study: its `type-bug` taxonomy produced too few usable labels to include. It is possible to substitute heuristics in this situation — keyword matching in commit messages, for instance — but heuristic labels are noisier by construction, and any results built on them should be reported with visibly less confidence than the maintainer-labelled results in this whitepaper. We did not do this for any of the four codebases reported here; all four met the stricter, maintainer-label standard.

Signal strength differs between projects. AUC ranges from 0.68 to 0.84 and reflects project stage, activity and tracker quality. Four codebases are a first sample, not a general rule.

Open-source patterns may not transfer directly to proprietary codebases. Open-source projects tend to have more formalised contribution processes and more consistent commit and tracker conventions than internal enterprise codebases.

This study covers one class of risk signal. Churn and ownership concentration are structural signals derived from Git history. They don't capture runtime behaviour, dependency-chain risk, security surface, or the semantic complexity of individual changes — one input into a broader risk picture, not a substitute for it.

The validated ranking is churn-based. Silo Risk and Change Coupling are structural risk signals in Calyntro but were not tested as bug predictors here. The top-quartile figures in this whitepaper therefore describe a churn-based ranking.

SEE IT ON YOUR OWN CODEBASE

Run this analysis against your own repository

Calyntro is self-hosted repository intelligence for CTOs and Engineering Managers — knowledge silos, bus factor, change coupling, and now, validated forward-looking risk scoring. Your data never leaves your infrastructure.

The live demo runs the same analysis shown in this whitepaper against the public MongoDB repository, no login required.

calyntro.com · **demo.calyntro.com**