calyntro

WHITEPAPER · REPOSITORY INTELLIGENCE

Predicting Defects Before They Happen

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

0.79
Peak AUC achieved on the Qt Framework, ahead of the published industry benchmark

61%
Of future bug fixes concentrated in Calyntro's top risk quartile

3
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 three codebases in three languages.

0.792

AUC on Qt Framework (6-month window) — above the 0.740 industry reference point

61 / 26

% of bug fixes captured by the top-risk 26% of files, Qt Framework

3 / 3

Codebases where the signal beat random assignment, with no exceptions

CONTENTS

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?**

> 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.

The record is already sitting in the commit history

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 three large production codebases, three languages, and two very different project profiles.

The results hold up. The signal is real, it's consistent across all three 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.

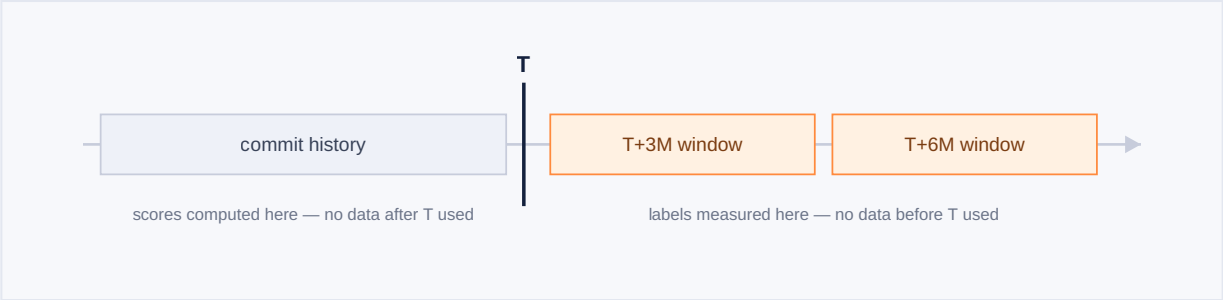

Figure 1. The time-lag validation design: scores and labels are computed from strictly non-overlapping periods.

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 and Qt, 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.

Files with at least one bug-fix commit in the observation window were labelled positive. Files touched after T with no bug-fix association were labelled negative. Files never touched after T were excluded from the label set altogether — a conservative choice that keeps inactive code from inflating the negative class.

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. What matters in practice is how far above 0.5 a signal sits, and whether that distance is competitive with other published results.

As an external reference point, we use Repowise — a commercial defect-prediction tool that has published validation results across 21 repositories and 9 programming languages, reporting an AUC of 0.740.

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 25% of files Calyntro rates as highest risk, what share of the bugs that show up later do they actually account for? A random baseline concentrates roughly 25% of bugs 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.

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), and the Qt Framework (C++) span a mature maintenance codebase, a fast-growing open-source tool, and an industrial-scale framework widely used in embedded and systems software. Full dataset details and cutoff dates are in the Appendix.

We had no prior involvement with any of the three projects and no way to influence the results after the fact. The cutoff date T was fixed before label extraction began and was never adjusted once results started coming in.

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.

Across all three codebases, files with a single dominant contributor — high ownership concentration — went on to accumulate **fewer** bug fixes, not more. 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 the validation study, high concentration predicted fewer defects, and the mechanism is easy to understand intuitively: 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 validation results say the truth is more contextual: concentration is a positive signal when the situation is stable, and 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.

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 inversion held consistently across MongoDB (C++, mature project), uv (Rust, active growth), and Qt Framework (C++, industrial-scale OSS) — three very different codebases producing the same pattern, which is a strong indication it reflects something structural rather than noise. A tool that naively treats "single author = risk" without this distinction will flag exactly the files that are, at this moment, the best-maintained parts of the codebase.

04 · Results
SECTION 04

Results Across Three Codebases

Three 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; and an industrial-scale C++ framework with decades of history and a broad contributor base. Each has a public issue tracker, which let us identify bug-fix commits entirely independently of the risk scores themselves.

All three used the same methodology: risk signals computed from commit history up to the cutoff date T (2025-08-15), labels drawn from the same issue trackers for the three- and six-month windows that followed. No parameters were tuned differently between runs.

MongoDB Server — the mature maintenance project

One of the most widely deployed open-source database engines in production. Our observation window covered more than 17,000 commits across over 40 modules in a C++ codebase exceeding a million lines.

Dataset: 3,322 files with sufficient pre-T activity · 1,151 positive labels at T+3M · 2,034 at T+6M

Window	Discrimination (AUC)	Top-quartile concentration
T+3M (3 months)	0.615	42% of bug-affected files in the top-risk 29% of the codebase
T+6M (6 months)	0.619	—

The signal here is above chance and directionally consistent, but weaker than in the other two candidates — which fits what MongoDB Server actually is: a mature codebase where the high-churn, high-risk areas have already been identified and incrementally stabilised over years of active development. Even so, 42% of subsequent bug fixes land in the top-risk quarter of the codebase, which is a meaningful reduction in the search space for a team deciding where to focus. The top-quartile number is the more actionable read here — even with a narrower AUC margin, the signal reliably concentrates future bug activity into a defined, smaller slice of the code.

uv — the active growth project

uv is a Python package manager and resolver written in Rust, built by Astral, 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: 2,847 files with sufficient pre-T activity · labels extracted via GitHub Issues (type: Bug)

Window	Discrimination (AUC)	Top-quartile concentration
T+3M (3 months)	0.717	40% of bug-affected files in the top-risk 25% of the codebase
T+6M (6 months)	0.730	—

The six-month AUC of 0.730 sits within one point of the 0.740 Repowise reference — a strong result for a single codebase in a single language, where the benchmark was built by averaging across 21 repositories and 9 languages. 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. Files that pulled in disproportionate churn before T were significantly more likely to need a bug fix afterward.

Qt Framework — the industrial-scale benchmark

Qt underpins a large share of embedded, industrial, and desktop software worldwide. The Qt Base repository we analysed spans over 36,000 commits across a 5.5-year observation window, 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: 6,675 files with sufficient pre-T activity · 10,318 Bug-type tickets resolved

Window	Discrimination (AUC)	Top-quartile concentration
T+3M (3 months)	**0.770**	**61% of bug-affected files in the top-risk 26% of the codebase**
T+6M (6 months)	**0.792**	**60% of bug-affected files in the top-risk 25% of the codebase**

The strongest results across all three candidates, and the only ones that clear the Repowise reference outright. 61% of all files that received a bug fix in the three months after the cutoff were already sitting in Calyntro's top-risk quartile — a quartile covering just 26% of the active codebase.

Qt combines everything that makes the churn signal most effective: a long, stable history that gives the signal room to accumulate; active ongoing development that concentrates structural risk in identifiable places; and a large, well-structured issue tracker producing high-quality labels. It also sits squarely in the C++ embedded and industrial software domain — where defect cost is highest, and where a forward-looking risk signal has the most to offer an engineering leader.

Cross-codebase summary

Codebase	Language	Profile	AUC (T+6M)	Top-quartile (T+3M)	vs. Repowise
MongoDB Server	C++	Mature maintenance	0.619	42% in 29%	−0.121
uv	Rust	Active growth	0.730	40% in 25%	−0.010
Qt Framework	**C++**	**Industrial scale**	**0.792**	**61% in 26%**	**+0.052**
Repowise (reference)	9 languages	21-repo average	0.740	—	—

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

The signal holds up across languages and project types. MongoDB (C++), uv (Rust), and Qt (C++) are three technically distinct environments, and in every one the churn-based risk signal beat random assignment. 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.

The signal gets stronger as project activity increases. The weakest result shows up in the most mature, most deliberately stabilised codebase; the strongest in the most active, highest-velocity one. That's interpretable: in a codebase under heavy active development, structural risk signals carry more forward-looking information simply because there's more structural change happening to predict. Teams running Calyntro on actively evolving codebases — which is where defect prediction earns its keep in practice — should expect signal strength toward the upper end of this range.

The top-quartile metric is the number that actually drives a decision. A team using Calyntro's risk ranking to prioritise review capacity, refactoring investment, or knowledge-transfer sessions isn't working from an abstraction. In the Qt analysis, 61% of the defect-fix activity over the following three months was already concentrated in a quarter of the codebase that Calyntro had ranked highest-risk ahead of time. For a team that can't review everything with equal intensity, that concentration is where the decision actually lives.

A note on complexity as an additional predictor

We also tested cyclomatic complexity and a combined hotspot score (complexity × churn, normalised 0–100) across both Qt and MongoDB. The results surface a history-length effect worth knowing about.

Qt Framework (5.5-year pre-T history):

Predictor	AUC T+3M	AUC T+6M
Churn Score	**0.770**	**0.792**
Hotspot Score (complexity × churn)	0.741	0.770
Complexity alone	0.684	0.723

MongoDB Server (8.5-month pre-T history, T = 2026-01-15):

Predictor	AUC T+3M	AUC T+6M
Hotspot Score (complexity × churn)	**0.625**	**0.610**
Complexity alone	0.579	**0.621**
Churn Score	0.585	0.594

With deep history, churn wins outright — complexity and churn are correlated (complex files tend to accumulate more changes), so combining them adds noise rather than signal. With short history, the ranking flips: churn hasn't had time to accumulate meaningful variance across files, while complexity, as a static property independent of observation window length, keeps its discriminative power — and the hotspot score, blending both, becomes the strongest short-horizon predictor.

In practice: churn is the better predictor once a codebase has years of history behind it; complexity and the hotspot score are the more robust choice on a freshly imported repository or a short observation window. The two signals aren't redundant, they're context-dependent

— a team analysing a recent import should weight the hotspot view more heavily, while a team with years of history can lean primarily on churn. Full numbers for both codebases are in Appendix A.7.

05 · Implications
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. Sixty-one percent of the Qt codebase's bug-fix activity concentrated in a quarter of its files; the remaining three-quarters accounted for under forty percent. That doesn't mean the rest of the codebase deserves zero attention — it means that when review capacity is constrained, which it always is, the allocation should follow where risk is structurally concentrated rather than an unspoken assumption that every file carries equal odds of producing the next defect. 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. More practically, the risk signal should move after a genuine refactoring investment: a module that was accumulating churn from many fragmented contributors, and is then stabilised under clearer ownership, should see its structural risk profile improve — measurably, with a before and an after.

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 — the ones whose concentrated authorship was, per Section 3, actively suppressing defect rates — lose the contextual protection that kept them stable. The code hasn't changed, but the organisation around it has, and the risk profile moves with it.

The same applies to team restructurings, onboarding periods when new contributors start working in modules they don't yet understand, and ownership handovers between teams. These events are almost always predictable well in advance — the restructuring is planned months out, the departure has a notice period, the onboarding schedule is known. What's usually missing is specificity: which parts of the codebase are most exposed to each of these events. A risk signal built on ownership concentration makes that exposure visible before it turns into an incident, and points directly at where structured pairing or documentation work is most urgently needed.

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 — static analysis hadn't flagged anything, coverage was adequate, there'd been no recent incidents nearby.

What that sentence actually describes, in most cases, is the absence of a prospective, quantified risk signal — not the absence of the underlying evidence. The retrospective signals were there: rising churn, narrowing authorship, climbing complexity. They just weren't surfaced, aggregated, or turned into anything forward-looking. Organisations running a validated risk signal aren't only better positioned to catch individual incidents before they happen. They're in a different position with respect to their own codebase — they 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. A risk-ranked codebase tells a team where to point their existing quality practices most intensively — it doesn't stand in for those practices. What it 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.

Appendix · Dataset & Methodology
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 — MongoDB Jira, GitHub Issues for uv, and Qt Jira all met this bar; and (4) no prior involvement or ability to influence how bugs were classified or commits structured.

Two additional candidates were evaluated and excluded. CPython was dropped for a label taxonomy mismatch — 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. A second, internal candidate — an embedded C++ project with roughly 60 developers — is currently in preparation, with results to follow once the engagement permits disclosure.

A.2 — Dataset summary

Codebase	Language	Tracker	History window	Files (churn ≥ 3)	Bug tickets linked
MongoDB Server	C++	Jira (jira.mongodb.org)	2017 – 2025-08-15	3,322	~1,150 (T+3M)
uv	Rust	GitHub Issues (astral-sh/uv)	2023 – 2025-08-15	2,847	linked via #NNN refs
Qt Framework (Base)	C++	Jira (bugreports.qt.io → atlassian.net)	2020 – 2025-08-15	6,675	10,318 Bug-type

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

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 — Task, Feature, Improvement, and so on — were not classified as bug-fix commits.

Files were labelled positive if they appeared in at least one bug-fix commit within the observation window, negative if touched after T with no bug-fix association, and excluded from the label set entirely if never touched after T — a conservative choice that keeps dormant code from inflating the negative class.

For uv, ticket references follow the GitHub Issues convention (`#NNN` in the commit message). For MongoDB and Qt, Jira project prefixes (`SERVER-` , `QTBUG-`) 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 (qt-project.atlassian.net). The Jira REST API v2 endpoint used for ticket-type lookup was removed on the new platform (HTTP 410). Enrichment for Qt tickets required detecting the redirect and switching to the v3 endpoint (`/rest/api/3/search/jql`). This was handled inside the data pipeline; no Qt tickets were lost to the migration.

A.4 — Predictor computation

Three predictors were computed per file from commit history up to T. **Churn score** is the count of distinct commits touching a file before T — the primary signal reported throughout this whitepaper, measuring how often a file has been the site of change activity rather than how many lines changed. **Ownership concentration** is the share of pre-T commits attributed to a file's single most active author — 100% means one author made every commit, 30% means the most active author accounted for less than a third. **Complexity (last known)** is the cyclomatic complexity of the file at its most recent pre-T state, as recorded by the import pipeline's static analysis component, used both standalone and as a factor in the hotspot score.

The specific scoring functions built on top of these base signals are proprietary and are not disclosed in this document. The predictors above are the inputs; the validation results reflect the outputs.

A.5 — Benchmark reference

The external benchmark used throughout this study — AUC 0.740 — comes from Repowise's published validation results (as of August 2025), covering 21 repositories across 9 programming languages. The methodology behind that figure isn't publicly documented to the level of detail provided in this whitepaper; the comparison is made against the reported headline AUC. We include it as a reference point for what a "good" AUC looks like in this domain, not as a head-to-head competitive claim.

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. Some bug-fix commits will be missing from the label set because they were filed under a non-bug type; some non-bug commits may reference tickets reclassified later. These errors are expected to be roughly symmetric, and to weaken rather than inflate the measured signal.

History length affects signal quality. MongoDB's weaker result relative to uv and Qt is at least partly explained by the shorter effective history window available in the study period compared to Qt's 5.5-year span — longer pre-T history generally produces more stable churn and ownership signals.

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. The signal may behave differently where commit discipline is lower or bug tracking less systematic — the forthcoming embedded C++ customer analysis is intended to address this directly.

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.

A.7 — Complexity and hotspot score, extended results

The hotspot score combines cyclomatic complexity and churn count, normalised 0–100. All four predictors were tested across both codebases at different pre-T history lengths.

Qt Framework — cutoff T = 2025-08-15, pre-T history: 5.5 years, 6,675 files

Predictor	AUC T+3M	AUC T+6M	Top-quartile T+3M
Churn Score	**0.770**	**0.792**	**61% of bugs in 26% of files**
Hotspot Score (complexity × churn)	0.741	0.770	55% of bugs in 25% of files
Complexity alone (cyclomatic)	0.684	0.723	48% of bugs in 25% of files
Ownership Concentration	0.421	0.409	inverted signal — see Section 3

MongoDB Server — cutoff T = 2026-01-15, pre-T history: 8.5 months, 7,142 files

Predictor	AUC T+3M	AUC T+6M	Top-quartile T+3M
Hotspot Score (complexity × churn)	**0.625**	0.610	**39% of bugs in 25% of files**
Churn Score	0.585	0.594	35% of bugs in 26% of files
Complexity alone (cyclomatic)	0.579	**0.621**	36% of bugs in 25% of files
Ownership Concentration	0.429	0.436	inverted signal — see Section 3

Finding. The relative strength of churn versus complexity depends on history length. With deep pre-T history (Qt, 5.5 years), churn dominates — complexity and churn are correlated, and combining them adds noise. With short pre-T history (MongoDB, 8.5 months), churn variance is limited and complexity, as a static structural property, retains independent predictive power. The hotspot score is the most robust choice across both conditions.

Implication for Calyntro. For fresh imports or short observation windows, the hotspot score is the recommended primary signal. As commit history accumulates beyond 12–18 months, churn progressively takes over. Complexity remains a useful maintainability signal independent of history length. This is broadly consistent with D'Ambros et al. (2012), who report similar AUC ranges for static complexity metrics used as standalone predictors.

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