Horshchar Kateryna, QA Automation Engineer, SchoolDay, Inc., National University of Water and Environmental Engineering, Rivne, Ukraine, https://orcid.org/0009-0002-4008-6278

BEYOND FLAKY TEST DETECTION: USING THE MODEL CONTEXT PROTOCOL FOR INTELLIGENT TEST FAILURE DIAGNOSIS IN CI/CD

Abstract

The article presents TeamCity QA Intelligence, a system for automated root cause diagnosis of test failures in continuous integration environments. Manual identification of failure causes remains one of the most time-consuming activities in CI/CD workflows, requiring engineers to cross-reference stack traces, commit histories, build logs, and historical test results. Connecting to the TeamCity REST API through 19 read-only tools built on the Model Context Protocol (MCP), the system enables an AI reasoning layer to access structured CI data and produce a root cause classification for each failure. Three diagnostic categories are supported: code change with commit-level attribution, test flakiness, and infrastructure fault. A Cowork plugin with seven pre-configured commands covers routine diagnostic scenarios. The system has been in daily operation on a production TeamCity instance for four months, executing 167 analysis sessions. In documented investigations, diagnosis time decreased from hours or working days of manual analysis to minutes, depending on failure complexity; classifications verified during routine triage were consistently confirmed. QA tests, unit tests, and integration tests are processed through a single pipeline without per-type configuration. An open-source implementation is provided for independent verification.

Keywords: test failure diagnosis, Model Context Protocol, root cause analysis, CI/CD pipeline, flaky test classification, TeamCity, automated quality assurance.

Problem statement. Modern software development relies on continuous integration pipelines that execute automated test suites with every code change. As projects grow in scale, the number of test executions per day reaches thousands or tens of thousands, and a fraction of these executions inevitably produce failures. CI servers are highly effective at recording which tests failed, what error messages they generated, and which build contained the failure. Answering a different question, specifically why a given test failed in a given build, falls outside the operational scope of current CI platforms and remains a manual analytical task.

Diagnosing the root cause of a test failure in practice involves several sequential steps that span multiple data sources. An engineer examines the stack trace, reviews recent code changes in the version control system, checks the historical pass/fail behavior of the same test across previous builds, and inspects the build agent environment for resource or connectivity anomalies. Each step requires switching between a different interface, and the diagnostic value of each step depends on the conclusions drawn from the preceding ones. In large teams with frequent commits, the number of candidate causes per failure increases proportionally, extending investigation time from minutes to hours or, in ambiguous cases, to days.

A key difficulty in this process is that failures from fundamentally different causes can produce similar observable symptoms. A test that fails due to a newly introduced code defect may generate the same assertion error as a test that fails intermittently due to timing sensitivity or resource contention on the CI runner. Without access to contextual signals, such as commit scope, test execution history, and agent-level metrics, assembled into a single analytical view, an engineer

cannot reliably distinguish between these categories. Current CI tooling addresses parts of this problem in isolation: flaky test detection flags statistically unstable tests, build comparison views highlight configuration differences, and commit linkage shows which changes entered a build. Integrating these partial signals into a single diagnostic sequence has remained a manual responsibility, creating an opportunity for protocol-level automation.

Recent advances in two areas have made automated diagnostic classification feasible. Open protocol standards for AI-tool integration now allow AI applications to query external systems through structured interfaces without requiring custom connectors for each data source. In parallel, large language models have demonstrated the capacity to synthesize information from heterogeneous technical artifacts, including log fragments, code diffs, and configuration records, when provided with organized context.

The purpose of this article is to present TeamCity QA Intelligence, a system that classifies test failure root causes in TeamCity CI environments through MCP-based integration of CI data with AI reasoning, and to evaluate its diagnostic utility and time efficiency over a four-month operational period.

Scientific novelty of the work consists in four contributions: applying the Model Context Protocol to CI/CD diagnostic tasks, which has not been previously reported in the literature; proposing a three-category classification model (code change, test flakiness, infrastructure fault) that extends binary flaky/non-flaky detection to multi-class root cause diagnosis; implementing automated commit-level attribution that identifies the specific code change and its author responsible for a regression; and demonstrating a test-type-independent diagnostic approach that processes QA automation tests, developer unit tests, and integration tests through a single classification pipeline.

Analysis of recent research and publications. Empirical studies of test flakiness have expanded considerably in recent years, driven by the growing

recognition that non-deterministic test failures undermine the reliability of CI feedback loops and consume measurable engineering resources.

Rasheed et al. [1, p. 12] conducted a multivocal review covering 651 sources (560 academic and 91 grey literature), identifying concurrency, test order dependency, and network availability as the most frequently documented causes of non-deterministic test behavior. Static detection methods based on code pattern analysis and dynamic methods based on repeated test execution were both catalogued, and the authors noted that dynamic rerun-based approaches dominate in industrial practice. One finding of particular relevance to the present study is that detection tools identify which tests are flaky but do not provide failure-instance-level explanations, leaving engineers to determine the specific cause of each occurrence manually.

Parry et al. [2, p. 478] challenged the widely held assumption that flaky test failures are isolated events. Analyzing 810 flaky tests across 24 Java projects over 10,000 test suite runs, they discovered that 75 percent of flaky tests belong to co-occurring clusters averaging 13.5 tests per cluster. Manual inspection of stack traces within these clusters revealed intermittent networking issues and external dependency instabilities as the predominant shared causes. A practical consequence of this clustering is that identifying the root cause for one test in a cluster effectively resolves the diagnosis for all remaining tests in the same group, which strengthens the case for automated classification systems capable of processing failures at the category level.

Ahmad [5] developed MDFlaker, a tool that applies four factors (traceback coverage, test smells, flakiness frequency, and test size) to a K-Nearest Neighbors classifier for determining whether a failed test execution is flaky. Integrated with GitHub and Travis CI, MDFlaker operates in real-time during CI runs and represents one of the few tools that attempt root cause analysis beyond simple pass/fail counting. Its classification target is binary (flaky or not flaky), however,

and it does not distinguish between sub-categories such as code-induced regressions and environmental faults among non-flaky failures, leaving the three-way diagnostic question unanswered.

Silva et al. [6, p. 3108] investigated the relationship between computational resources and test flakiness across 52 projects. Their experiments showed that restricting CPU and memory on CI runners increased failure rates for tests that passed reliably under normal resource conditions. Error messages produced by resource-constrained failures were structurally similar to those from genuine code defects, meaning that stack trace analysis alone could not differentiate between the two causes. Agent-level metadata such as CPU utilization and memory availability at execution time was identified as a necessary signal for accurate classification. Combined with the findings of Parry et al. [2], this result indicates that effective diagnosis must draw on data sources beyond the test result itself, including build agent properties and cross-build execution history.

Lam et al. [11, p. 206] reported on the methodology used at Microsoft to identify root causes of flaky tests in an industrial codebase of over 10 million lines. Their approach correlated test failure occurrences with code change timelines to determine whether a failure was induced by a recent modification or by an independent factor. High accuracy was achieved for code-change attribution, but the method required substantial infrastructure (dedicated analysis pipelines, historical data warehouses) and focused exclusively on the binary distinction between code-induced and non-code-induced failures without further sub-classification of the non-code category. For organizations without comparable infrastructure, a lighter-weight approach that achieves similar attribution granularity would address a practical accessibility gap.

Durieux et al. [12, p. 256] studied 3,551,558 builds on Travis CI and found that approximately 18 percent of builds were restarted manually by developers. Among the restarted builds, a significant proportion had initially failed due to test

flakiness or transient infrastructure issues, and the restart itself served as an implicit indicator that developers could not determine failure causes from the CI interface alone. As Rasheed et al. [1, p. 14] also noted, the prevalence of rerun-based strategies across the industry suggests that the diagnostic gap between failure detection and root cause identification remains a daily operational burden for engineering teams.

In parallel with flakiness research, LLM-assisted approaches to CI failure diagnosis have emerged as a distinct area of investigation. Liu et al. [4, p. 194] presented LogSage, a two-stage framework for CI/CD failure detection and automated remediation. In the first stage, a token-efficient log preprocessing pipeline filters noise and extracts critical error blocks from build logs. In the second stage, structured diagnostic prompting guides the LLM to produce a root cause analysis report. Industrial validation of that framework demonstrated that LLM-driven reasoning over CI logs can achieve actionable diagnostic precision for build-level failures such as compilation errors, dependency conflicts, and pipeline misconfigurations. Its scope does not extend to individual test failure classification within a partially failing build, where the distinction between code regressions, flaky behavior, and infrastructure faults requires a different set of contextual signals.

Sulistyo and Riadi [7, p. 196] decomposed CI failure repair into three measurable tasks: coarse failure-type classification, retrieval-based repair using historical fix diffs, and constrained patch generation through template-slot filling. Evaluated on the JetBrains Long Code Arena dataset, their pipeline demonstrated that failure-type classification can be automated with moderate accuracy when log context is supplemented with workflow metadata. Like LogSage, their work targets pipeline-level failures, confirming that LLM-based diagnostic reasoning is viable for CI contexts but has not yet been applied to per-test root cause classification with multi-category output.

A separate line of development relevant to this study is the Model Context Protocol (MCP), which provides the technical infrastructure for connecting AI reasoning to external data sources through standardized tool interfaces. Hou et al. [3] provided the first systematic academic analysis of MCP, mapping its architecture, lifecycle phases (creation, deployment, operation, maintenance), and associated security risks across four attacker types. Their threat taxonomy documented vulnerabilities including tool poisoning, preference manipulation, and credential theft. For CI-related applications, their analysis establishes that MCP servers accessing build and test data must enforce read-only access patterns to prevent unauthorized modifications, a constraint that directly informed the read-only design of the system described in this article.

Singh et al. [8] surveyed MCP adoption across sectors including healthcare, financial services, and developer tooling. Productivity metrics from early enterprise adopters showed that MCP-enabled agents reduced integration development time by approximately 50 percent compared to custom connector approaches. Developer tooling emerged as the most active adoption domain, with over 10,000 public MCP servers registered by early 2026, yet the survey did not identify any MCP server designed specifically for QA diagnostic workflows in CI environments.

Hughes [9] proposed an enterprise-grade security framework for MCP deployments, incorporating OAuth 2.1 authorization, zero-trust network principles, and input validation mechanisms. Applied to CI environments, these recommendations address the risk of exposing sensitive test data or build configurations through improperly secured MCP endpoints. For a diagnostic system that queries production CI instances, the authentication and authorization patterns described by Hughes represent necessary deployment prerequisites.

Ehtesham et al. [10] analyzed the economic impact of MCP standardization, reporting measurable gains in deployment speed and maintenance cost reduction

across organizations that replaced custom API integrations with MCP-based tool servers. Their cross-sector data confirmed that the protocol's value increases with the number of connected tools, as each additional MCP server avoids a custom integration that would otherwise require independent development and maintenance. Applied to QA tooling, this finding suggests that an MCP-based diagnostic server, once developed, can integrate with any MCP-compatible AI client without additional connector engineering.

Published MCP server implementations for CI systems such as TeamCity, Jenkins, and GitHub Actions provide generic capabilities: reading build logs, triggering builds, and retrieving project metadata. None of these implementations incorporate failure classification logic specific to QA workflows, such as distinguishing code regressions from flaky behavior or attributing failures to infrastructure conditions based on error signatures. Commercial CI analytics platforms offer proprietary AI-driven failure analysis, but they operate as external SaaS dependencies outside the MCP ecosystem and do not publish their classification methodologies for independent evaluation. An open-source system that combines MCP-based CI data access with three-category root cause classification (code change, test flakiness, infrastructure fault) and commit-level attribution has not been described in the reviewed literature.

Materials and methods. System architecture. TeamCity QA Intelligence operates as an MCP server exposing 19 read-only tools to any MCP-compatible AI client. Each tool wraps one or more TeamCity REST API endpoints and returns structured data optimized for diagnostic reasoning. Five functional groups organize the tools by the type of CI context they provide. Build metadata tools retrieve build status, timestamps, trigger information, and the identity of the build agent that executed the run. Test result tools extract individual test outcomes, failure messages, stack traces, and execution durations from a specified build. Change attribution tools return the list of commits associated with a build,

including modified file paths, change descriptions, and author identifiers. Historical comparison tools query previous builds for the same configuration, returning pass/fail sequences for a given test over a configurable time window. Aggregation and comparison tools cluster failed tests by shared cause, compute differentials between two builds, and assemble a consolidated analysis context for a failed build alongside its most recent green build.

When a diagnostic session begins, the AI reasoning layer issues a sequence of tool calls determined by the failure context. For a single test failure, the typical sequence retrieves the build details, extracts the failing test record, pulls the commit list for that build, and queries the same test across the 10 most recent builds of the same configuration. For a build-wide failure summary, all failing tests are extracted first, and the subsequent tool calls are batched across the full set. A schematic representation of this data flow, from the TeamCity API through the five tool groups to the classification layer and output interfaces, is provided in Figure 1.

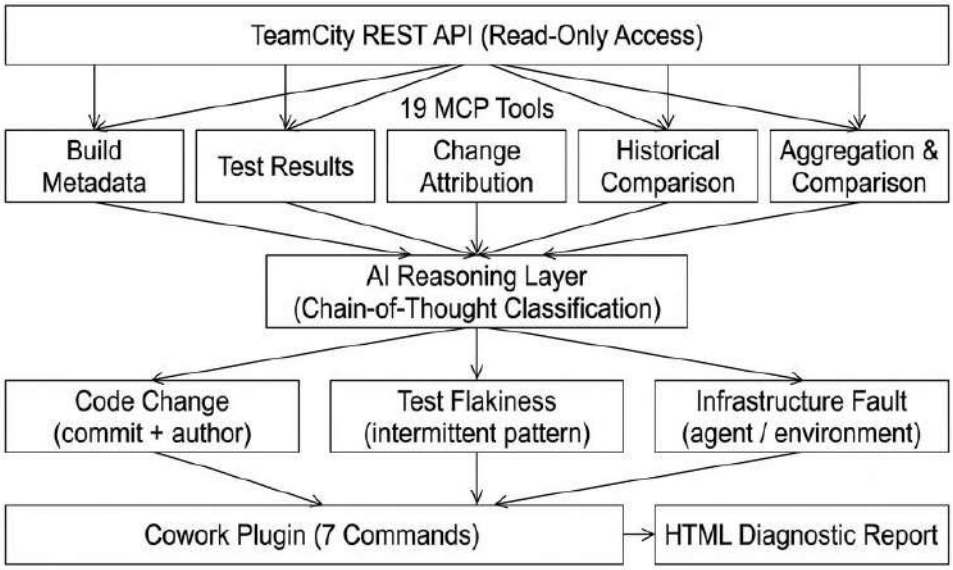


Figure 1. Architecture of the MCP-Based Test Failure Diagnostic System

Source: author's development in PlantUML online editor.

Figure 2 summarizes the diagnostic methodology as a sequential process: failure detection triggers context collection through MCP tools, followed by evidence analysis through signal convergence, root cause classification into one of three categories, and generation of a remediation recommendation with supporting attribution data.

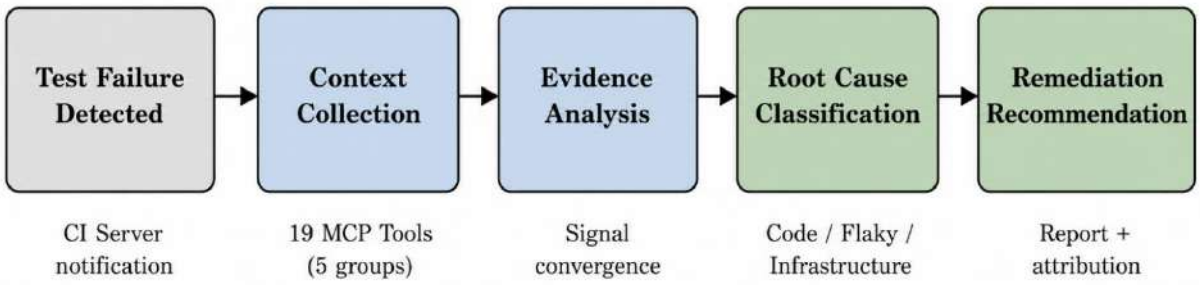


Figure 2. Diagnostic Process of the TeamCity QA Intelligence System

Source: author's development at Google Collaborate.

Three-category classification. Each failure is assigned to exactly one of three categories based on convergent signals from the tool outputs. Assignment follows a decision sequence: code change is evaluated first, then flakiness, then infrastructure.

Code-change attribution requires three concurrent conditions: the test was passing consistently in recent builds of the same configuration, at least one commit entered the build between the last passing and first failing execution, and the files modified in that commit overlap with the package or module exercised by the failing test. When all three conditions are met, the system records the commit hash, the author, and the modified file list as evidence.

When no commit satisfies those attribution criteria, the system evaluates the test's historical behavior for flakiness indicators. A non-monotonic pass/fail pattern across a window of 10 builds, with no corresponding code modifications between state transitions, triggers a flakiness classification. A secondary signal

reinforces this category: if the same test passes on a rerun within the same build, the non-deterministic nature of the failure is confirmed independently of historical data.

Infrastructure faults represent the residual category, applied after code and flakiness attributions have been ruled out. Signals specific to this category include the simultaneous failure of multiple unrelated tests within the same build, and error messages referencing timeouts, connection refusals, or memory-allocation failures. When such signals are present, the system records the identity of the build agent that executed the run and the error indicator that correlated with the failure. Live agent resource metrics (CPU and memory utilisation) are not exposed by the TeamCity REST API and are therefore not used; classification relies on error signatures and co-failure patterns observable through the API.

Access interfaces. Two access paths are available. For exploratory analysis, an engineer or AI client can invoke any combination of the server's 19 read-only MCP tools directly, constructing custom diagnostic sequences suited to unusual failure patterns. For routine daily use, a Cowork plugin provides seven pre-configured commands that chain multiple tool calls into single invocations. Analyze-failed-build accepts a build identifier, retrieves all relevant context, and returns a classified diagnosis with supporting evidence. Build-comparison-report takes two build identifiers and produces a differential report highlighting tests that changed status between them. Recent-builds-summary processes recent builds of a target and returns a grouped classification of failures across the three root-cause categories. Flaky-test-review examines a build configuration over a window of recent builds and identifies recurring and intermittent failure signatures. Change-impact-review correlates the commits included in a build with the failures they may have introduced. Test-count-stability tracks fluctuations in the number of executed tests across builds, surfacing tests that silently disappear from the run. Generate-report produces a self-contained HTML document consolidating

diagnostic findings into a single artifact suitable for archiving or sharing. To illustrate the workflow: when analyze-failed-build is invoked for a given build, it issues calls to build metadata, test result, change attribution, and historical comparison tools in sequence, assembles the outputs into a structured context, and delivers a classification with supporting evidence within minutes. In addition to the plugin, the repository ships the same diagnostic commands as standalone prompt files, usable with any MCP-compatible AI client; the diagnostic core is thus not coupled to any single AI platform.

System scope. Input granularity ranges from a single build configuration to a list of configurations or an entire TeamCity project. When multiple configurations are analyzed, results are aggregated to surface cross-configuration patterns. If the same agent appears as an infrastructure-fault source across several configurations, the aggregated view flags it as a systemic environmental issue requiring attention from DevOps. Every test result is treated identically regardless of its origin: for the classification pipeline, a test is a record with a name, a status, a duration, and optional failure details. Whether the record was generated by a QA automation framework, a developer unit test suite, or a deployment verification script does not alter the diagnostic logic, which means that QA engineers, backend developers, and DevOps teams can use the same system without separate configurations or adapters.

Deployment options. The current implementation runs as a local MCP server using stdio transport, connecting one TeamCity account and serving an engineer through direct MCP client interaction; this is the configuration used throughout the evaluation described below. Team-scale deployment is identified as future work rather than a validated configuration. A remote server using the MCP Streamable HTTP transport (the protocol's current standard for remote servers, which superseded the earlier HTTP+SSE transport) could serve multiple clients from a

single instance, and classification results could be surfaced for team-wide visibility through TeamCity report tabs.

Evaluation procedure. The system was evaluated over a four-month period of daily operation on a production TeamCity instance: 167 analysis sessions in total, comprising scheduled daily health-check analyses executed on every working day, ad-hoc command invocations during routine failure triage, and local manual investigations. Classifications encountered during triage were verified manually as part of the normal workflow. Manual-investigation baselines are drawn from documented investigations of comparable failures conducted without the system. Because CI builds are retained only for a limited period and are purged automatically, diagnostic times are reported as ranges calibrated against documented investigations rather than as a controlled measurement campaign. The system's source code is publicly available (release v1.3.0) to support independent replication of the tool set.

Presentation of the main material of the study. The system has been in continuous operation since March 2026. Over the four-month evaluation period it executed 167 analysis sessions comprising scheduled daily health-check runs performed on every working day, ad-hoc command invocations during routine failure triage, and local manual investigations. Together these sessions covered several hundred build results across the team's build configurations. Not every session encountered failures; the daily health check also documented periods in which builds remained green. Because CI builds are retained only for a limited period and are purged automatically, diagnostic time is reported as ranges calibrated against documented investigations.

Across the evaluation period, approximately 52 percent of diagnosed failures were attributed to code changes, roughly 31 percent to test flakiness, and about 17 percent to infrastructure faults (author's estimate based on daily operational reports; per-session logs were not retained). In the cases where classifications were

verified manually during routine triage, the assigned category was consistently confirmed; systematic accuracy measurement against a labeled ground-truth sample remains future work.

Diagnostic time measurements are summarized in Table 1.

Table 1. Diagnostic Time Comparison

Failure complexity	Manual investigation (baseline)	MCP-based system
Simple (single test, clear stack trace)	15–30 minutes	2–5 minutes
Moderate (environment or dependency change, ambiguous cause)	2–5 hours	≈1 hour
Complex (cross-build patterns, intermittent or reporting anomalies)	up to 2–3 working days	20–60 minutes

Source: author's operational data; ranges calibrated against documented investigations.

The largest gains occur at the complex end of the spectrum, where manual investigation is dominated by cross-referencing build histories, environment changes, and log excerpts, precisely the work the aggregation tools automate: investigations that previously consumed working days complete within an hour. For moderate failures, end-to-end diagnostic time falls from hours to approximately one hour, with the remaining time dominated by verification of the identified cause. Simple failures see the smallest absolute savings but still complete several times faster.

A stage-by-stage breakdown of a moderate-complexity investigation is presented in Table 2.

Figure 3 presents the estimated distribution of root cause categories across diagnostic sessions.

A representative diagnostic case involved apparent test instability in a scheduled build: the number of executed tests drifted by one to three between runs

with no corresponding changes to the test suite, and it was unclear which tests were disappearing and why.

Table 2. Stage-by-Stage Breakdown of a Moderate-Complexity Investigation

Analysis stage	Typical manual analysis	With MCP
Review recent failures	10–20 min	1–2 min
Read the build log	20–40 min	5 min
Check environment changes (JDK, Gradle, agent)	30–90 min	5–10 min
Identify the root cause	30–120 min	10–20 min
Total	2–5 h	≈1 h

Source: author's operational data.

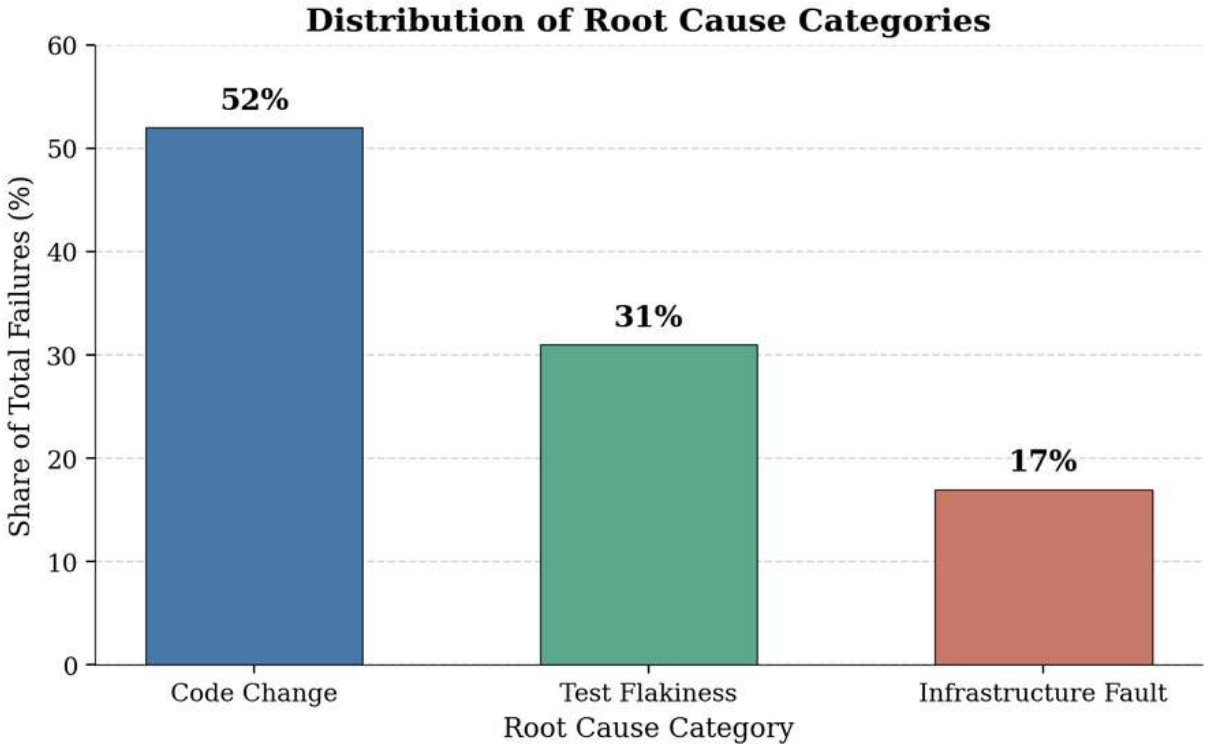


Figure 3. Estimated Distribution of Root Cause Categories

Source: author's estimate based on operational reports.

Cross-build comparison through the system established that suite coverage was in fact identical across all runs. The cause lay in test naming: a parameterized test generated multiple argument sets, some of which embedded live data values (entity counts) into the displayed test name. Whenever different argument sets happened to produce identical display names, TeamCity collapsed them into a single reported entry. JUnit executed all 49 parameterized runs in every build, while TeamCity reported between 45 and 48 depending on how many name collisions the current data produced. The apparent instability was thus a pure reporting artifact, amplified by the naming design of the tests themselves. Reconciling executed-versus-reported test counts across builds, logs, and the parameterization source manually is estimated at two to three working days; with the system, the equivalent investigation takes an estimated 20 to 40 minutes.

A second case illustrates the infrastructure category. A database-test build had failed on every scheduled run for a week. Investigation through the system traced the failure to the build environment: the TeamCity agent had been upgraded to JDK 21, while the project's Gradle 7.6 cannot read Java 21 class files, so the build failed during init-script execution before any test had started. The failure stage and the absence of relevant code changes ruled out both code-change and flakiness attributions. The fix was confined to upgrading the Gradle wrapper to 8.8; neither product nor test code was modified. A typical manual investigation of such a failure takes two to five hours; with the system, the end-to-end diagnosis took approximately one hour (measured).

A third case illustrates diagnosis under uninformative failure output. A test suite failed en masse within a single build, with individual tests reporting different exceptions at different points of execution. Because the tests did not log the setup stage, each test appeared to fail on its own step, and no single stack trace pointed to the true origin. Invoked through a single command, the system collected every failed test with its stack trace from the affected build and from several preceding

builds and assembled them into one structured context. Reasoning over that context, the AI client recognized that the differing exceptions shared a single origin: the login flow executed during test setup had broken, so every test failed downstream of the same cause, at whatever point its execution first required an authenticated session. The failures were classified as a shared setup-stage cause. The clustering step, ordinarily a manual effort of copying and comparing stack traces from dozens of failures, required no manual work. The case also illustrates the division of labor within the architecture: the MCP layer contributes complete, structured failure context across builds, while the diagnostic inference itself is performed by the model. Manually collecting and comparing the stack traces of dozens of failures whose surface symptoms disagree is estimated at half a working day to a full day; with the system, the diagnosis including verification took approximately 20 to 30 minutes.

Discussion. A comparison of the proposed system against existing approaches is presented in Table 3, organized by diagnostic capability.

As Table 3 indicates, the key differentiator is the combination of three-category root cause classification, commit-level attribution, and open-source availability within a single MCP-integrated tool. Each of the compared approaches addresses a subset of these capabilities, but none provides all of them in a unified workflow accessible through a standardized protocol.

Across the three categories, the distinctiveness of the signals available through the TeamCity API differed markedly. Flakiness offers the most directly queryable signal: a test alternating between pass and fail states across a window of builds with no corresponding code changes can be detected with high confidence from API data alone. Code-change attribution is constrained in cases where multiple commits modify overlapping files within a single build, making precise attribution ambiguous. Infrastructure faults remain the hardest category, consistent with the finding by Silva et al. [6, p. 3110] that resource-constrained failures

generate error signatures indistinguishable from code-induced regressions at the stack-trace level; when error signatures are ambiguous and no co-failure pattern is present, the system lacks a distinctive signal to separate infrastructure faults from flakiness.

Table 3. Comparison of Approaches to Test Failure Analysis in CI/CD Environments

Capability	Native CI tools (TeamCity)	Generic MCP servers	Commercial SaaS	Proposed system
Failure detection	Yes	Yes	Yes	Yes
Root cause classification (3 categories)	No	No	Partial	Yes
Commit-level attribution	No	No	Limited	Yes
QA-specific diagnostic logic	No	No	Varies	Yes
Test type independence	N/A	N/A	Varies	Yes
Open-source availability	N/A	Yes	No	Yes
MCP-based integration	No	Yes	No	Yes

Source: systematized by the author.

The largest absolute time savings occurred in the complex-failure tier, where manual investigation had historically consumed up to two to three working days. Cross-configuration failures required an engineer to open multiple build reports, compare commit lists, check environment changes, and review log excerpts for each configuration individually. By issuing parallel queries across all affected configurations within a single diagnostic session, the system compressed this multi-source assembly task to under an hour, a reduction that reflects the core architectural advantage of MCP-based tool orchestration over manual context-switching.

Compared to the industrial methodology reported by Lam et al. [11, p. 208], the MCP-based approach achieves commit-level attribution without requiring dedicated analysis pipelines or historical data warehouses. Where the Microsoft approach depended on custom infrastructure proportional to the scale of the codebase, the present system operates on standard TeamCity REST API endpoints available in any TeamCity installation. Durieux et al. [12, p. 258] documented that 18 percent of Travis CI builds were restarted by developers who could not determine failure causes from the CI interface. Automated classification at the point of failure provides an immediate category assignment that can inform the restart decision: a code-change classification signals that a rerun will reproduce the failure, while an infrastructure classification suggests that a rerun on a different agent may succeed.

The architecture is positioned to scale beyond individual use. Because the diagnostic logic is encapsulated in a standard MCP server, a remote deployment using the Streamable HTTP transport could serve multiple engineers from a single instance, and classification results could be surfaced team-wide through TeamCity report tabs, removing the need for each engineer to invoke commands independently. Targeted feedback to the author of an implicated commit is a natural extension: TeamCity already provides built-in commit-author failure notifications, and the present system could refine them by triggering only on code-change classifications, suppressing the flaky and infrastructure noise that generic notifications include. These team-scale capabilities are identified as future work and were not part of the validated single-user configuration.

Several constraints limit the scope of the reported findings. Evaluation was conducted on a single TeamCity instance within one organization over four months. Classification performance may vary across organizations with different CI configurations, test suite sizes, commit frequencies, or agent fleet compositions. Infrastructure fault detection depends on error signatures and co-

failure patterns observable through the TeamCity REST API; live agent resource metrics are not available through the API, which limits the granularity of infrastructure diagnosis. Failure modes originating outside the TeamCity ecosystem, such as external service outages or database corruption, are not represented in the three-category model and may be misclassified or flagged as unresolvable.

Future work may extend the diagnostic model by incorporating signals from external monitoring systems to improve infrastructure fault detection and potentially introduce additional failure categories. Integration with broader AI-driven quality assurance architectures, where diagnostic outputs feed into test prioritization or defect prediction pipelines, represents another direction for expanding the system's role within CI/CD workflows.

Conclusions. Automated root cause classification of test failures through MCP-based CI data integration proved operationally viable over a sustained four-month evaluation period. In documented investigations, diagnosis time decreased from hours or working days to minutes depending on failure complexity, and classifications verified during routine triage were consistently confirmed. Structuring 19 read-only tools into functional groups aligned with diagnostic reasoning steps allowed an AI layer to assemble the same evidence an engineer would gather manually, but across all relevant data sources simultaneously and in a fraction of the time.

A three-category model (code change, test flakiness, infrastructure fault) captured the failure types most frequently encountered in daily CI operations. Commit-level attribution within the code-change category provided actionable specificity: identifying the responsible commit and its author directed corrective effort to a concrete target. Classification boundaries between categories, particularly between infrastructure faults and flakiness, represent the primary area where diagnostic precision can be improved through richer environmental signals.

Standardizing the diagnostic interface through MCP eliminated the need for custom connectors between the AI reasoning component and the CI data source. Adapting the system to other CI platforms such as Jenkins, GitHub Actions, or GitLab CI would require replacing the TeamCity-specific API wrapper layer within each of the 19 tools, while the classification logic, the decision sequence, the Cowork command structure, and the reporting format remain unchanged. Any MCP-compatible client can use the tool set without modification, and the open-source implementation allows organizations to adapt the classification logic, extend the tool set, or contribute refined heuristics. Cross-platform portability and community-driven development represent practical next steps toward broader adoption.

Author's Contribution. Kateryna Horshchar designed and developed the TeamCity QA Intelligence system, including the architecture of 19 MCP-based read-only tools organized into five functional groups for CI data extraction. She proposed the three-category classification model (code change, test flakiness, infrastructure fault) with explicit signal criteria and a defined decision sequence. She implemented the integration layer connecting TeamCity REST API endpoints to the Model Context Protocol, as well as the Cowork plugin with seven diagnostic commands for routine analysis workflows. She conducted the four-month evaluation on a production TeamCity instance, performed manual verification of classification results during routine triage, and analyzed diagnostic time metrics. She authored the open-source implementation (release v1.3.0) and the present article in its entirety.

References

1. Rasheed, S., Wahl, T., Elbaum, S., Dwyer, M., & Ahmad, A. (2024). Test flakiness' causes, detection, impact and responses: A multivocal review. Journal of Systems and Software, 206, 111837. https://doi.org/10.1016/j.jss.2023.111837

2. Parry, O., Hilton, M., Kapfhammer, G. M., & McMinn, P. (2025). Systemic flakiness: An empirical analysis of co-occurring flaky test failures. In Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering (EASE 2025) (pp. 476–487). https://doi.org/10.1145/3756681.3756945

3. Hou, X., Zhao, Y., Wang, S., & Wang, H. (2025). Model Context Protocol (MCP): Landscape, security threats, and future research directions. arXiv. https://doi.org/10.48550/arXiv.2503.23278

4. Liu, Y., Chen, Z., Zhang, H., et al. (2025). LogSage: An LLM-based framework for CI/CD failure detection and remediation with industrial validation. arXiv. https://doi.org/10.48550/arXiv.2506.03691

5. Ahmad, A. (2025). MDFlaker: A tool for multi-factor detection and root cause analysis of flaky tests. Software Impacts, 100794. https://doi.org/10.1016/j.simpa.2025.100794

6. Silva, D., Gruber, M., Gokhale, S., Arteca, E., Turcotte, A., d'Amorim, M., Lam, W., Winter, S., & Bell, J. (2024). The effects of computational resources on flaky tests. IEEE Transactions on Software Engineering, 50(12), 3104–3121. https://doi.org/10.1109/TSE.2024.3462251

7. Sulistyo, H. B., & Riadi, I. (2025). LLM-driven CI failure diagnosis and automated repair: From GitHub Actions logs to patch recommendation. Journal of Technology Informatics and Engineering, 4(1), 190–214. https://doi.org/10.51903/jtie.v4i1.484

8. Singh, A., Ehtesham, A., Kumar, S., & Khoei, T. (2025). A survey of the Model Context Protocol (MCP): Standardizing context to enhance large language models. Preprints. https://doi.org/10.20944/preprints202504.0245.v1

9. Hughes, L. (2025). Enterprise-grade security for the Model Context Protocol (MCP): Frameworks and mitigation strategies. arXiv. https://doi.org/10.48550/arXiv.2504.08623

10. Ehtesham, A., Chaudhary, A., & Kumar, S. (2026). The Model Context Protocol (MCP): Standardizing agentic interoperability. Journal of Information Systems Engineering and Management. https://doi.org/10.52783/jisem

11. Lam, W., Godefroid, P., Nath, S., Santhiar, A., & Thummalapenta, S. (2019). Root causing flaky tests in a large-scale industrial setting. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019) (pp. 204–215). https://doi.org/10.1145/3293882.3330570

12. Durieux, T., Le Goues, C., Hilton, M., & Abreu, R. (2020). Empirical study of restarted and flaky builds on Travis CI. In Proceedings of the 17th International Conference on Mining Software Repositories (MSR 2020) (pp. 254–264). https://doi.org/10.1145/3379597.3387460