International Journal of Computer (IJC)
ISSN 2307-4523 (Print & Online)
https://ijcjournal.org/index.php/InternationalJournalOfComputer/index

A Gate-Model Approach to Test Layer Scheduling in CI/CD Pipelines and its Effect on Deployment Frequency

Skadorva Vitali*

Test Automation Developer Specialist, Intact Financial Corporation, Gatineau, Canada
Email: vitali.skadorva@qaapproved.ca

Abstract

The article examines a gate model for scheduling test layers in CI/CD pipelines and its effect on deployment frequency, feedback speed, and the resilience of software delivery processes. The relevance of the study is determined by the growing complexity of microservice systems and by the limitations of monolithic E2E runs, which slow down change validation, increase infrastructure load, and reduce confidence in automated testing results. The aim of the work is to develop and empirically verify an architecture that links test layers with discrete quality gates and software development lifecycle events. The article's scientific novelty lies in formalizing the relationship among test granularity, trigger-based execution schedules, and CI/CD pipeline throughput. It is shown that moving lightweight checks to the commit and Pull Request stages, while retaining E2E tests in the schedule, enables reducing total test execution time by 90%, decreasing flakiness from 25% to 2%, and increasing deployment frequency to multiple deployments per day. The article will be useful for DevOps engineers, system architects, QA leads, and software engineering researchers.

Keywords: CI/CD; gate model; trigger model; DORA metrics; deployment frequency; test decomposition; Shift-Left.

Received: 3/24/2026
Accepted: 5/24/2026
Published: 6/2/2026

---

* *Corresponding author.*

1. Introduction

In the context of digital transformation, migration to cloud-based microservice architectures, and the increasing complexity of user interaction patterns, the ability of organizations to deliver software to end users rapidly, safely, and continuously has become a key factor in market survival [1]. At the core of this operational flexibility lie the methodologies of continuous integration and continuous deployment.

The industry standard for measuring the effectiveness and maturity of these processes currently relies on DORA metrics [2]. Among these metrics, four fundamental indicators stand out: deployment frequency, change lead time, mean time to restore service, and change failure rate. Empirical studies confirm that high deployment frequency and minimal development cycle time are inseparably correlated with elite engineering team performance and the broader commercial success of technology products [3].

Nevertheless, achieving CI/CD performance indicators is widely hindered by entrenched architectural antipatterns in quality assurance [4]. The classical approach to automated testing of complex enterprise web applications has historically relied on heavyweight, monolithic end-to-end (E2E) tests that simulate user actions in a real browser, with access to real databases and third-party services [5]. In the context of modern component-based interfaces, the exclusive use of E2E tests causes an inevitable combinatorial explosion of checks. This turns the upper E2E testing layer into a cumbersome distributed monolith. Such test suites run for hours, consume enormous amounts of cluster resources, and yield a high percentage of false negatives due to network fluctuations, interface desynchronization, or temporary unavailability of test environments [6]. As a result, the CI/CD pipeline ceases to serve as a tool for rapid delivery and becomes a critical bottleneck. This delays feedback to developers, forces teams to adopt infrequent batch releases, and reduces deployment frequency, returning the process to waterfall logic within nominally Agile-oriented sprints.

The **relevance** of this study stems from the industrial need to overcome the architectural crisis posed by monolithic testing pipelines. The transition from monolithic E2E testing to a multilayer decomposed architecture is a necessary condition for DevOps process optimization. Yet it remains insufficient on its own. Tests extracted to lower levels must also be orchestrated correctly within the CI/CD pipeline itself. The academic literature of recent years has actively investigated predictive test selection [7], the integration of machine learning for build failure prediction, and the analysis of pipeline security [8]. However, the scientific discourse reveals a shortage of applied, architecturally grounded frameworks for precisely scheduling test layers and integrating them as discrete quality gates within a unified pipeline backbone [9].

The research **problem** lies in the contradiction between the aspiration of modern engineering teams toward high deployment frequency, ideally continuous delivery of every commit, and the structural inability of traditional pipelines to provide rapid, deterministic, and reliable validation of changes without excessive consumption of time and resources.

The **purpose** of this study is to develop, theoretically substantiate, and empirically validate a gate model for scheduling test layers in CI/CD pipelines, and to conduct a quantitative assessment of its impact on key pipeline

performance metrics, including deployment frequency and feedback speed.

To achieve this purpose, the following **objectives** were formulated within the study. First, to conceptualize the architecture of a gate model in which each test layer acts as an isolated, discrete pipeline stage with its own blocking gate. Second, to develop and describe a trigger model that logically links the execution of specific testing layers to particular events in the software development lifecycle. Third, to conduct an empirical analysis of the effect of the implemented architecture on the temporal characteristics of the pipeline using an array of logs and metrics collected in a corporate environment. Fourth, to extrapolate the obtained technical metrics to the organizational level by assessing the influence of the proposed model on the macro-level DORA metrics. Fifth, to analyze and identify the technological barriers, design risks, and structural limitations accompanying the transition of large businesses to the proposed architecture. The study is guided by two research **hypotheses**.

H1: The implementation of a gate model that aligns test layers with software development lifecycle events reduces the total execution time of the CI/CD pipeline and accelerates feedback on code changes.

H2: The relocation of component and UI integration checks to earlier pipeline stages, while preserving E2E tests within scheduled runs, increases the resilience of the delivery process and supports higher deployment frequency.

The scientific **novelty** of the study lies in formalizing the relationship between the granularity of test layers, their trigger-based scheduling, and the throughput of CI/CD pipelines. The article proposes a trigger architecture that mathematically and infrastructurally substantiates the practical implementation of the Shift-Left paradigm by moving computationally lightweight yet massive checks to ultra-early stages of the pipeline. In this way, a reliable mechanism is established to prevent cascading failures in distributed microservice systems. The study complements the existing theoretical foundation of software engineering by shifting the academic focus from the analysis of test internal semantics to the systemic mechanisms of test orchestration, parallelization, and scheduling in modern cloud pipelines.

2. Materials and Methodology

The methodological basis of this study is formed on the principles of software engineering, systems analysis, and empirical measurement of computing system performance. To ensure high validity and reproducibility of results, the study relies on a comprehensive design that combines quantitative and qualitative approaches. The principal method selected was the industrial case study method, with comparative analysis before and after the architectural intervention. The choice of this method was driven by the need to study complex engineering processes in their natural corporate context, where isolated laboratory experiments cannot reproduce the scale of real workloads, network delays, and team interactions.

The data for the empirical section were gathered from the corporate sector in 2023, 2024, and 2025, while the main object of the quantitative analysis was Organization A, a large global corporation from the insurance sector. The main reason why this organization was selected as the primary study object was the highly representative character of its existing technology stack. Organization A had the most wide-ranging software complexity: multiple brands, localizations, legacy systems, security and quality gate requirements. The CI/CD pipelines were

classical pipelines that had been developed for many years, which were completely blocked by monolithic multi-hour end-to-end runs. Additional cross-industry validation and calibration of the proposed methodology were conducted using a pool of anonymized companies from related sectors: a global e-commerce platform (Organization B), a developer of enterprise artificial intelligence platforms (Organization C), and a provider of digital security solutions (Organization D). In order to comply with confidentiality policies and non-disclosure agreements, all company names, specific brands, and proprietary products in the text of the article have been anonymized. At the same time, indications of industry affiliation have been preserved, allowing assessment of the scalability context.

For the collection and structuring of empirical data, direct exports of performance metrics from CI/CD platforms were used. The data collection process covered the analysis of 26 complete release cycles. The following parameters were measured and recorded: absolute execution times of test suites, with millisecond-level detail, across individual business scenarios and architectural layers. The instability index is calculated as the ratio of false-negative failures to the total number of runs. The time spent by engineers on manual failure analysis and pipeline reruns. The aggregated macro-indicators for labor costs for regression testing and infrastructure support are annual.

In addition to empirical measurements, the study used methods of content analysis of technical documentation and architectural patterns, as well as system modeling of pipeline configurations. For visual interpretation of processes, user journey flowcharts, state graphs of Agile cycles, and fragments of configuration code reflecting the logic of pipeline branching were analyzed and included in the study.

3. Results and Discussion

In the classical, historically established software development architecture, automated testing functions as a unified, monolithic, and often indivisible stage located at the very end of the code integration process [10]. For many years, engineering teams relied on mimicking end-user actions in browser-based E2E tests as the ultimate criterion for quality. However, such an approach leads to serious infrastructural and logical congestion in delivery pipelines. Under a monolithic E2E run, the CI/CD server is forced to initialize a full operating system environment, launch a headless browser, load megabytes of client scripts, send real network requests to transactional databases, and wait for asynchronous rendering of the entire DOM tree for every check, including the smallest one.

The consequence of this excess is that the CI/CD pipeline remains blocked for hours. Feedback reaches the software engineer with a substantial delay, rendering the idea of continuity moot. To visualize the root of the problem, it is necessary to analyze a typical user journey in a modern application, as shown in Figure 1.

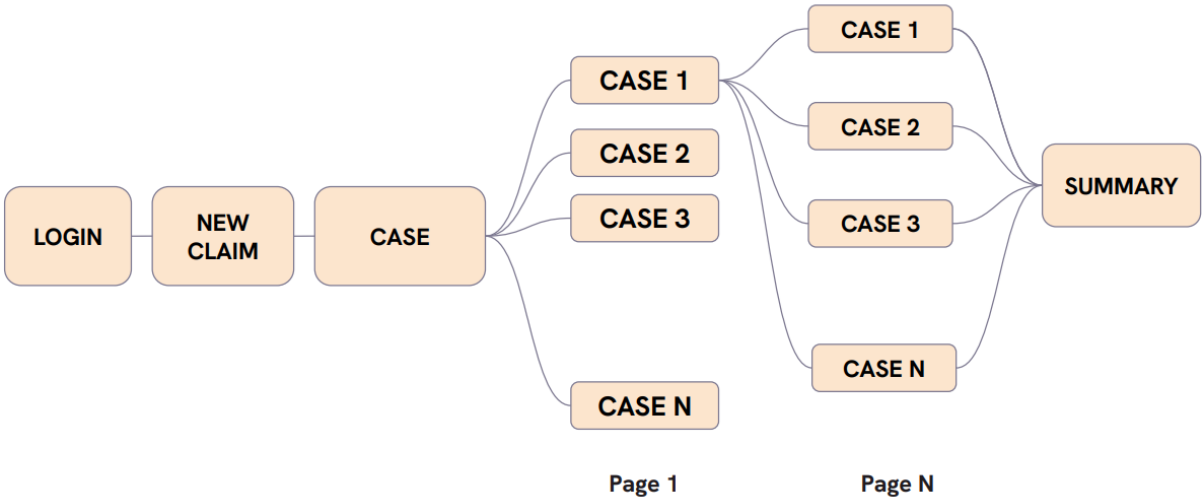


Figure 1: User flow chart from request creation to final case summary

This diagram illustrates the multi-step transition of a user through the system. The journey begins with the basic steps of authorization (LOGIN), process initiation (NEW CLAIM), and transition to an entity (CASE). The system then branches into multiple nested pages (Page 1, Page N), each containing uniform, reusable blocks (CASE 1, CASE 2, and so forth) that are ultimately aggregated into a summary (SUMMARY). Within the monolithic E2E testing paradigm, the pipeline is forced to traverse this long, resource-intensive path hundreds of times to verify each button in the CASE 1 block on each page. If the error is hidden in a basic element, for example, in input field validation at the NEW CLAIM stage, the pipeline will still attempt to execute all subsequent tests, which will fail because the user path is blocked. This generates colossal amounts of informational noise in CI/CD reports and wastes server resources.

To overcome this dysfunction, the industry conceptualized the Shift-Left paradigm. Yet in practice, the leftward shift is often interpreted superficially as a mere requirement to write more unit tests. In the context of CI/CD, a genuine leftward shift means a radical restructuring of the pipeline's routing. This idea is clearly demonstrated by the analytical diagram in Figure 2.

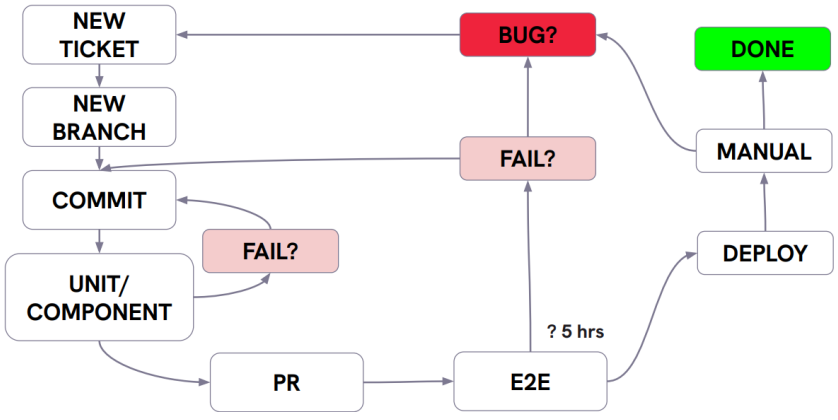


Figure 2: Shift left algorithm

The presented diagram visualizes in detail the contrast between early and late defect detection. On the left, the main initial chain is constructed: task creation (NEW TICKET), branch creation (NEW BRANCH), commit (COMMIT), and immediate launch of unit and component checks (UNIT/COMPONENT). The key element here is the pink FAIL? block, which forms a closed loop of immediate feedback. If the test at this stage fails, the process returns to code writing (COMMIT) within minutes. The developer remains within the task context and fixes the defect at once.

The right side of the diagram shows the traditional pattern. The code passes through Pull Request creation and proceeds to the E2E end-to-end testing stage, which, as emphasized by the label 5 hrs, takes hours. Detection of a FAIL at this late stage returns the process far back to COMMIT or even NEW TICKET. The cost of fixing such a defect increases because the developer has already moved on to another task, and the pipeline has remained blocked for a long time, preventing other team members from merging code. In this way, the diagram visually demonstrates that the earlier an error is detected, the fewer excessive transitions are made along the chain. The practical implementation of this cycle, however, requires the introduction of strict pass-through mechanisms.

The fundamental engineering solution to pipeline blocking was the development and implementation of the gate model. In traditional, non-optimized pipelines, tests are often grouped into a single large logical pool, launched in parallel or sequentially, without strict interdependence between layer success. The gate model assumes the pipeline is decomposed into discrete, independent stages, each ending with a logical barrier, a quality gate.

The mathematical and logical basis of the gate model rests on the principles of preventing cascading failures and minimizing computational costs within clusters. If an introduced code change breaks a basic, isolated UI component, it will, with mathematical inevitability, lead to the failure of dozens of integration tests and hundreds of E2E tests involving this component. If all these tests are launched simultaneously or without strict intermediate gates, the computational cluster, for example, Kubernetes nodes or cloud runners, will spend hours of valuable processor time executing E2E scenarios that are doomed to fail.

The gate model establishes a strict rule: failure at a lower, faster, and cheaper layer immediately closes the gate and physically prevents the launch of more expensive upper layers. If a component test returns an error status, the entire pipeline is terminated immediately. The engineer receives notification of the error within 1–2 minutes. Expensive server resources for UI integration and E2E testing are not reserved or consumed. This increases the efficiency of cloud infrastructure utilization, reduces operational costs, and maintains high CI/CD throughput for other parallel development branches.

The implementation of the gate model cannot function in a vacuum. It requires linkage to version control lifecycle events. During the study, a multilayer trigger model was designed and implemented. It determines a schedule according to which a specific testing layer is launched only when a defined event occurs. This model is built on a fine balance between the representativeness of quality validation and the speed of feedback delivery.Positioning the proposed trigger architecture against earlier work clarifies where its contribution sits. Predictive test selection approaches reduce pipeline duration by skipping checks deemed unlikely to fail [7], a route that depends on the stability of historical signal and offers no structural guarantee against cascading failures across architectural

layers. Twin-pipeline and AI-assisted orchestration proposals [8] target runtime decisions at a higher level of abstraction and assume that an underlying layered topology already exists. Studies of E2E parallelization [6] and feature-based test generation [9] address the cost of monolithic suites from within the suite itself, leaving the question of when each layer should fire outside their scope. The gate model formalises that missing scheduling layer and attaches it to lifecycle events, which makes it complementary to the predictive and parallelization tracks rather than a substitute for them.

The On Commit trigger, which fires on code commit, opens the first gate and launches the Unit and Component layers. These are the most atomic and fastest tests in the system. They function in full isolation from the network, routing, and the real database. Component tests mount interface elements directly in a virtual DOM. This makes it possible to verify internal business logic and the correctness of rendering in different states. The execution time of this gate usually ranges from a few seconds to 3–4 minutes.

The On PR trigger, which is activated when a Pull Request is created, launches the second gate and corresponds to the UI-Integration layer. This stage occurs when the developer records their readiness to merge code into the main branch. At this stage, the application page is fully assembled. A substantial architectural decision involves intercepting all real API calls to the server and replacing them with deterministic stubs or mocks. This approach enables verifying component connectivity, client-side routing, and state management mechanisms at high speed. At the same time, the influence of network timeouts and other sources of instability is excluded. The execution of this gate usually takes up to 10 minutes.

The On Schedule trigger, used for nightly or release runs, launches the final gate, represented by the heavyweight E2E layer. At this level, only full and critically significant business journeys are verified. Validation is performed through real database integrations and external microservices. These runs do not create obstacles for daily development. To understand the architectural separation of these flows, consider the graphical scheme shown in Figure 3.

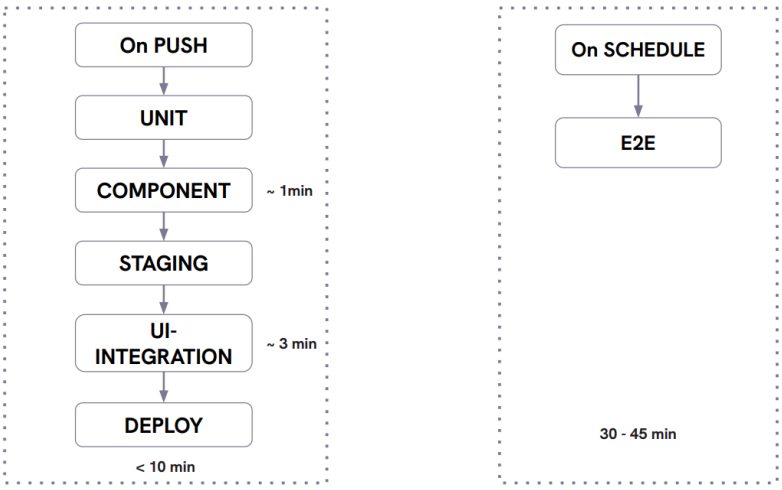


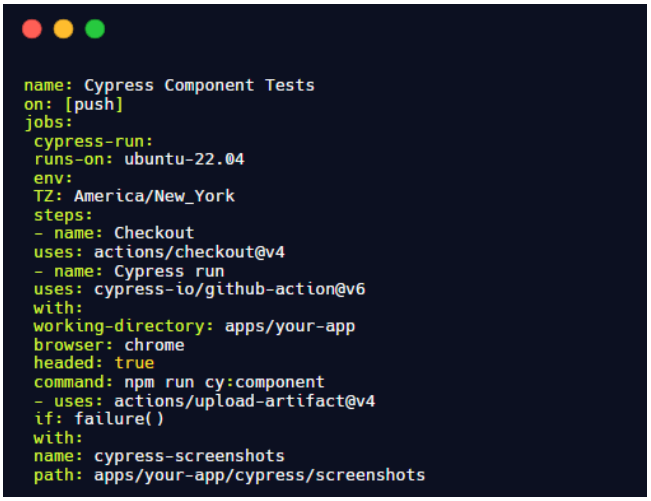
Figure 3: Splitting CI/CD checks into quick and scheduled stages

This illustration clearly demarcates two independent execution loops in CI/CD. The left loop on PUSH represents the backbone of the rapid feedback cycle, in which the UNIT, COMPONENT, STAGING, UI-INTEGRATION, and DEPLOY blocks execute sequentially and depend on one another. The most important aspect is the indicated total duration of the entire loop, which is less than 10 minutes. This allows the entire chain to be launched on every code push without interrupting engineers' flow state.

The right loop, on SCHEDULE, is visually contrasted with the left one. It moves the 30–45 minute E2E checks into a separate background process that starts according to a predefined schedule. Such parallelization is the architectural key to ensuring high pipeline throughput without compromising deep quality control.

The theoretical propositions of the gate model require precise engineering implementation on continuous integration servers. In modern automation practice, pipeline configuration is implemented through the Infrastructure as Code paradigm, in which launch logic, container configuration, and failure handling are declaratively defined in configuration scripts stored alongside the source code [11].

For a detailed illustration of how the component layer is functionally integrated as the first blocking gate into a real pipeline, consider the configuration shown in Figure 4.



```
name: Cypress Component Tests
on: [push]
jobs:
 cypress-run:
 runs-on: ubuntu-22.04
 env:
 TZ: America/New_York
 steps:
 - name: Checkout
 uses: actions/checkout@v4
 - name: Cypress run
 uses: cypress-io/github-action@v6
 with:
 working-directory: apps/your-app
 browser: chrome
 headed: true
 command: npm run cy:component
 - uses: actions/upload-artifact@v4
 if: failure()
 with:
 name: cypress-screenshots
 path: apps/your-app/cypress/screenshots
```

Figure 4: Example of tests in CI/CD

The image shows a fragment of a YAML configuration for the GitHub Actions platform that translates the theoretical trigger model into executable code.

The key architectural decisions are distinctly visible in the syntax of this file. The [push] declaration directly implements the trigger model. It instructs the orchestrator to initialize the process whenever new code is added to the repository. In this way, the launch mechanism becomes embedded in the configuration's structure and acquires a formally specified character.

The runs-on: ubuntu-22.04 block defines a rigid, yet ephemeral, operating environment. This solution is fundamental to ensuring execution consistency. It reduces the probability of discrepancies between the

developer's local environment and the pipeline environment. As a result, the influence of environmental variability on test results is reduced, and fluctuations that could distort the interpretation of failures are eliminated.

In the steps section, the npm run cy: component command starts the component testing layer directly. This execution is carried out in strict isolation from the rest of the system. As a result, the check focuses on the local properties of components and their internal behavior. This enables a cleaner diagnostic signal and faster defect detection at the corresponding architectural level.

The presence of a specialized step with the condition if: failure() for uploading artifacts, in this case, cypress-screenshots, demonstrates a high-level approach to handling gate failures. In the event of gate failure, the system automatically collects and stores diagnostic data. Such a mechanism reduces the time the developer spends investigating the causes of the failure, that is, triage. As a consequence, the pipeline ceases to be perceived as an opaque technical procedure and becomes an instrument of intelligent feedback.

Such declarative pipeline orchestration ensures that the gate model is strictly enforced by the server: failure at the Cypress run step automatically terminates the process, blocking subsequent deployment steps, thereby implementing the fail-fast methodological principle.

For empirical confirmation of the effectiveness of the gate model and trigger schedule, it is necessary to draw on primary data collected during the multi-month practical implementation of the methodology in the corporate environment of Organization A, a major insurance provider. Initially, the organization's CI/CD pipeline was in a deep state of stagnation, fully blocked by a monolithic E2E run comprising 621 scenarios. After the coverage audit, architectural decomposition, and configuration of the new pipeline topology, the test coverage structure and temporal metrics changed radically. The macro-aggregated data are presented in Table 1.

The CI/CD pipeline initially relied on 621 automated tests. After implementation of the gate model, the architecture expanded to 1,697 tests, including 1,257 component tests, 360 UI integration tests, and 80 end-to-end tests. This change increased coverage granularity and validation detail by a factor of 2.7. At the same time, the total execution time fell from 3 hours 41 minutes 57 seconds to about 20 minutes 30 seconds for the full sequential run across all layers. This reduction reached 90% and released more than three hours of machine time.

The flakiness rate decreased from 0.25, or 25%, to 0.02, or 2%, which corresponds to a 92% reduction in false-negative failures. Failure analysis time per cycle declined from 8 hours 30 minutes to 1 hour 30 minutes, which reduced manual work by 7 hours per release cycle. Regression cost across 26 yearly cycles dropped from 96 hours 10 minutes 48 seconds to 8 hours 53 minutes, yielding savings of 87 hours 17 minutes of engineers' working time. Maintenance effort per year also fell from 221 hours to 39 hours, which produced savings of 182 hours of engineers' working time.

Analysis of this data reveals a fundamental engineering phenomenon, the decomposition paradox. Despite the total number of executed automated checks increasing by almost 3 times, the cumulative execution time in the pipeline decreased by a striking 90%. This paradox is resolved by the mathematics of isolated execution: the 1,257 component tests shifted to the on-commit stage execute in just 59 seconds, thanks to the absence of overhead from

browser rendering and network I/O.

The drop in the instability index from 25% to 2% has the most critical psychological and operational significance for CI/CD processes. At an instability level of 25%, the pipeline loses its primary function: providing reliable information. Engineers stop trusting red statuses and treat them as another network flicker, leading to the ignoring of real defects and requiring constant manual intervention to rerun failing jobs. Rigid isolation of checks within the gate model and exclusion of the network layer at the UI integration stage restored 98% determinism to the pipeline. For a clearer understanding of the mechanics of how exactly the redistribution of tests across triggers changed feedback speed, it is useful to consider the breakdown of metrics for specific critical business scenarios in Table 1.

Table 1: Impact of multi-layer planning on the speed and stability of business scenarios in CI/CD

Business Functional Block	Scenario, Time Before	Time After	Remaining E2E Status, Executed Nightly	Change in Flakiness Index
Claim Insurance Details	Details, 28 min 57 sec	10.5 sec, Component + 30 sec, Integration + UI	Fully moved to lower layers	Completely eliminated, 0
Dashboard, Customer Dashboard	Main 15 min 33 sec	712 ms, Component + 13 sec, Integration + UI	Fully moved to lower layers	Completely eliminated, 0
File Upload Document Module	Upload, 26 min 18 sec	20 sec, Component + 16 sec, Integration + UI	Retained as part of E2E, simplified path	Significantly reduced
Car Accident Claim Submission	Claim, New Car 44 min 09 sec	Basic field-validation checks extracted	4 min 48 sec, only the critical business path retained	Localized to a narrow segment
Analytics, Financial Analytics Module	9 min 49 sec	841 ms, Component + 28 sec, Integration + UI	Fully moved to lower layers	Completely eliminated, 0

Reading the per-scenario data in Table 1 alongside the aggregate figures sharpens the interpretation of where the speedup originates. The Dashboard and Analytics scenarios, which dropped from minutes to sub-second component checks plus tens of seconds at the UI integration layer, share a structural property in that their failure

modes are local to a single page and require no cross-service state. The File Upload and Car Claim scenarios behave differently, retaining a residual E2E footprint because the underlying business journey crosses service boundaries that mocks cannot represent without contract guarantees.

Treating these two clusters as architecturally distinct rather than as a single test population is what allows the trigger schedule to deliver uniform latency gains. The flakiness collapse from 25% to 2% also tracks this split, with the largest variance reduction observed in the cluster fully relocated to lower layers and a smaller yet meaningful improvement in the residual E2E cluster owing to its narrowed surface.

This data demonstrate the micro-level of pipeline optimization and prove the effectiveness of the concept. The Dashboard scenario, whose display logic validation previously took more than 15 minutes in a real browser, is now validated at the Pull Request stage in less than 14 seconds. The balanced routing approach shows complex transaction-heavy business processes, such as Car Claim, that is, car insurance processing with calls to dozens of external systems, were not fully excluded from the E2E layer because they represent genuine user journeys. Nevertheless, owing to the aggressive extraction of local UI logic checks from them, for example, validation of number input masks or tab switching, the execution time of the remaining Car Claim E2E test decreased from 44 minutes to 4 minutes 48 seconds. These remaining 5-minute tests were moved to a separate on-schedule trigger, which finally unloaded the main daytime pipeline of developers.

The described technical optimization of the pipeline has a macroeconomic impact that is directly reflected in DORA metrics, recognized as the industry benchmark for assessing the effectiveness of software development teams.

First, the most radical impact was exerted on deployment frequency. Before implementing the gate model, the integration pipeline remained permanently blocked for 3.5 hours during each build's validation. Under such infrastructural constraints, engineering teams were physically unable to deploy code more than once or twice per day. This stimulated the vicious practice of batching, the artificial grouping of numerous heterogeneous commits from different developers into one heavy, high-risk release. After the pipeline was converted to the decomposed trigger model with a guaranteed feedback cycle of less than 10 minutes, the technical limitation on the number of deployments was completely removed. Developers gained the ability to merge code safely into the main branch and deploy incremental changes to production dozens of times per day, thereby moving the organization into the High Performers category on the DORA scale.

Second, the lead time for changes measurably decreased. This metric measures the time interval from the moment the first line of code is written to the successful launch of this functionality in the production environment. Eliminating nearly 4 hours of waiting in the CI/CD queue directly reduced lead time. Yet an even more important effect lies in preventing hidden losses associated with cognitive context switching. In the specific case of Organization A, the demonstrated savings totaled 269 working hours per team per year. This is a colossal hidden resource extracted from routine waiting and reinvested in developing new product features.

Third, substantial improvements are observed in stability metrics, such as mean time to restore service and change

failure rate. Reducing testing instability from 25% to 2% means red pipeline signals now have exceptionally high reliability. The gate model serves as a filter: it physically prevents isolated defects from leaking from the component to the page level and from the page level to production. The closed controlled cycle of quality check integration does more than stop the process. It directs its algorithm into the channel of early correction. If a defect is detected at an early stage via the On Commit or On PR triggers, the process is returned to the NEW TICKET block, or corrections are introduced into the current PR, well before the defect reaches the DEPLOY phase. This prevents the emergence of critical incidents in the working environment, thereby naturally reducing Change Failure Rate and bringing the need for emergency service restoration to an absolute minimum, since a failure affecting the end user is avoided in advance.

Although the quantitative analysis in this work was based on data arrays from the large insurance company Organization A, the developed gate model was tested and implemented in a number of other corporate domains using completely different technology stacks. Implementation of the model in the global B2C e-commerce company Organization B confirmed its high effectiveness for high-load platforms, where the cost of downtime caused by a broken E2E pipeline is measured in hundreds of thousands of dollars in lost revenue per hour.

In the enterprise artificial intelligence sector of Organization C and in the digital security domain of Organization D, the multilayer trigger architecture made it possible to integrate highly specific checks without friction, for example, validation of ML model weights for data drift or static vulnerability scanning SAST, as additional independent gates into the overall CI/CD process. In all documented cases, a stable 70% to 90% reduction in pipeline execution time and a significant decline in false positives were observed. This proves that the proposed architectural pattern is independent of a specific framework and is universally applicable to the overwhelming majority of modern software products built on the principles of component architecture and microservice interaction.

Despite the proven effectiveness and economic indicators, scaling the gate model and the Shift-Left concept at the enterprise level is associated with objective technical limitations and organizational risks that must be critically assessed during architectural planning.

The most critical technological risk in the On PR trigger at the UI integration layer is the use of mocks. If the server team changes the structure of a real API response, for example, removes an obsolete field or changes a data type, and the frontend team does not update its static stubs in time, the tests in the CI/CD pipeline will continue to pass successfully, closing the gate with a green signal. The application itself, however, will break when deployed in the real environment. To mitigate this vulnerability, the architecture requires the mandatory implementation of advanced patterns such as contract-based testing, which objectively complicates the overall pipeline infrastructure and requires synchronization of versioning between teams.

An effective gate model presupposes intensive use of containerization and parallelization of computational flows. The execution of more than 1,200 component tests in 59 seconds is possible only with an elastic cloud infrastructure that can instantly provision and scale dozens of CI/CD agents simultaneously. For companies with limited on-premises computing resources or outdated build servers, this may become a serious financial and

technical barrier that prevents achieving the speeds stated in the article.

Changing pipeline topology and moving checks to the left requires a deep transformation of established zones of responsibility within IT divisions. In the traditional paradigm, tests are written by dedicated automation engineers at a late stage. In the trigger-based gate model, component and UI integration tests must be developed by the authors of the product code, frontend, and backend, within a unified technology stack. Acceptance of this philosophy of shared responsibility for quality often leads to resistance and takes much longer than the technical reconfiguration of YAML pipeline files themselves.

The future development of CI/CD pipelines will focus on overcoming the described barriers by integrating artificial intelligence and machine learning algorithms. Innovative concepts such as the AI Twin Pipeline propose using deep predictive modeling to continuously analyze the history of previous runs and automatically identify potential failure points even before the actual trigger fires. This means that, in the near future, AI agents will be able to dynamically and autonomously reconfigure gate architectures for a specific commit.

For example, if a neural network determines with high probability that a code change affects only visual CSS styles, it may algorithmically disable the validation gate for the transactional database, thereby saving even more expensive cloud computing resources and reducing execution time. The decomposed gate model developed and verified in this study, with its isolated layers, constitutes the necessary architectural basis for later organically superimposing machine learning agents to ensure genuinely autonomous orchestration of development.

The empirical claims of this study hold within a defined scope of applicability that frames how the results transfer to adjacent settings. The quantitative core rests on 26 release cycles of Organization A in the insurance sector, with cross-industry calibration drawn from Organizations B, C, and D, and the reported reduction band of 70% to 90% reflects this mix of stacks rather than a universal constant. Pipelines dominated by data-intensive integration paths, by long-running ML training stages, or by hardware-in-the-loop validation will require the trigger schedule to be re-derived from their own layer cost profiles before the headline figures can be expected to recur.

The mock-based UI integration gate inherits the contract-drift risk discussed earlier in this section, and projects without consumer-driven contract tooling should treat the 10-minute PR figure as a target conditioned on that tooling being in place. Three follow-on directions flow from these boundaries, namely longitudinal re-measurement across additional release cycles to test the stability of the decomposition paradox, extension of the trigger taxonomy to data and model artifacts in ML-heavy pipelines, and the development of contract-synchronisation mechanisms that close the residual gap between the UI integration gate and production behaviour.

4. Conclusion

The industrial crisis of scalability in modern CI/CD pipelines, driven by the growth of checks in monolithic end-to-end tests, poses a threat to the operational flexibility and economic efficiency of technology companies. The study, supported by empirical data analysis, shows that increasing automated testing volume alone is insufficient to ensure high quality and fast software delivery. The decisive factor lies in the architectural intelligence of their integration and scheduling within the continuous delivery process.

The developed, theoretically substantiated, and empirically validated gate model successfully addresses the infrastructural bottleneck by decomposing the monolithic pipeline into discrete, logically isolated gates, integrated with a multilayer trigger model. Redistribution of test loads on the basis of the ultra-early feedback principle made it possible to architecturally isolate and execute lightweight component checks on every developer commit with time costs of less than 1 minute, UI integration checks upon code merging in a Pull Request within 10 minutes, and to leave heavyweight resource-intensive E2E tests exclusively for background nightly scheduled runs.

All research purposes and objectives were fully achieved. A multivector analysis of data from a large insurance corporation confirmed the significant practical impact of implementing the proposed architecture. The total absolute execution time of test suites in the pipeline decreased by 90%, and the destructive instability index decreased by 92%. The decomposition paradox was identified and described: a multiple increase in the total number of atomic tests under proper trigger scheduling leads to radical acceleration of the entire pipeline. Savings of 269 working hours per engineering team per year underscore the profitability of the business transition to the proposed model. The research hypotheses formulated in this study were confirmed by the results of the empirical analysis. The first hypothesis was supported by the observed reduction in overall pipeline execution time and by the acceleration of feedback cycles after the introduction of the gate model and trigger-based scheduling of test layers. The second hypothesis was supported by the decline in test instability, the growth of trust in automated validation results, and the increase in deployment frequency made possible by the redistribution of lightweight checks to the commit and Pull Request stages, while end-to-end scenarios remained within scheduled execution for critical business journeys.

From the perspective of macroeconomic development efficiency, the implementation of the gate model eliminated the infrastructural friction that historically blocked release engineering processes. The extreme acceleration of the feedback cycle directly improved all elite DORA metrics. It enabled organizations to increase deployment frequency to the working environment many times over, minimize lead time for introduced changes, and, owing to quality barriers, physically prevent critical defects from reaching users, thereby reducing system restoration time and the share of failed releases.

The practical significance of the presented work lies in providing DevOps engineers, system architects, and QA managers with a ready-made, universal, mathematically substantiated framework tested across various business domains for large-scale reorganization of continuous delivery processes. For further development and deepening of the proposed concept, future research by the scientific community should focus on the synergy of CI/CD and machine learning: integration of predictive AI algorithms for dynamic enabling and disabling of test gates in real time, as well as the development of frictionless mechanisms for automatic generation of API contracts for complete neutralization of the risks of desynchronization of isolated test environments in complex distributed microservice architectures.

References

[1] D. Barbosa, V. Santos, M. C. Silveira, A. Santos, and H. S. Mamede, “Highly Efficient Software Development Using DevOps and Microservices: A Comprehensive Framework,” Future Internet, vol.

18, no. 1, p. 50, Jan. 2026, doi: https://doi.org/10.3390/fi18010050.

[2] B. Wilkes, A. Maciel, and M.-A. Storey, "A Framework for Automating the Measurement of DevOps Research and Assessment (DORA) Metrics," Proceedings of 2023 IEEE International Conference on Software Maintenance and Evolution, Oct. 2023, doi: https://doi.org/10.1109/icsme58846.2023.00018.

[3] J. Rüegger, M. Kropp, S. Graf, and C. Anslow, "Fully Automated DORA Metrics Measurement for Continuous Improvement," Proceedings of ICSSP '24: International Conference on Software and Systems Processes, Jul. 2024, doi: https://doi.org/10.1145/3666015.3666020.

[4] A. Saxena, "Rethinking Software Testing for Modern Development," Computer, vol. 58, no. 6, pp. 49–58, May 2025, doi: https://doi.org/10.1109/mc.2025.3554094.

[5] B. Bylina and A. Antończak, "Analysis of end-to-end test automation tools based on the examples of Selenium WebDriver and Playwright," Annals of Computer Science and Information Systems, vol. 40, pp. 1–8, Nov. 2024, doi: https://doi.org/10.15439/2024f3747.

[6] D. Olianas, M. Leotta, F. Ricca, M. Biagiola, and P. Tonella, "STILE: A tool for optimizing E2E web test scripts parallelization," Journal of Systems and Software, vol. 222, p. 112304, Dec. 2024, doi: https://doi.org/10.1016/j.jss.2024.112304.

[7] K. Kodithyala, "Smart Test Selection in CI/CD: Optimizing Pipeline Efficiency," Journal of Computer Science and Technology Studies, vol. 7, no. 4, pp. 289–297, May 2025, doi: https://doi.org/10.32996/jcsts.2025.7.4.33.

[8] Y. Salvi and I. Kanade, "Twin Pipeline: AI Sibling of CI/CD," Proceedings of 2025 9th International Conference on Computing, Communication, Control and Automation (ICCCBEA), pp. 1–7, Aug. 2025, doi: https://doi.org/10.1109/iccubea65967.2025.11283703.

[9] P. Alian, N. Nashid, M. Shahbandeh, T. Shabani, and A. Mesbah, "A Feature-Based Approach to Generating Comprehensive End-to-End Tests," arXiv, 2024, doi: https://doi.org/10.48550/arXiv.2408.01894.

[10] M. I. Hossain, "Software Development Life Cycle Methodologies for Information Systems Project Management," International Journal For Multidisciplinary Research, vol. 5, no. 5, pp. 1–36, Sep. 2023, doi: https://doi.org/10.36948/ijfmr.2023.v05i05.6223.

[11] K. R. Mannem, "Demystifying Infrastructure as Code: A practical guide for DevOps professionals," World Journal of Advanced Engineering Technology and Sciences, vol. 15, no. 2, pp. 902–911, May 2025, doi: https://doi.org/10.30574/wjaets.2025.15.2.0580.