Performance Patterns for CI/CD Pipelines

Francesco Urdih[1,2][0009-0000-3507-5043], Theodoros Theodoropoulos[1][0000-0002-4618-4891], and Uwe Zdun[1][0000-0002-6233-2591]

[1] University of Vienna, Faculty of Computer Science, Software Architecture Research Group, Vienna, Austria
firstname.lastname@univie.ac.at
[2] University of Vienna, Faculty of Computer Science, UniVie Doctoral School Computer Science DoCS, Vienna, Austria

Abstract. Continuous Integration and Continuous Deployment (CI/CD) pipelines constitute an important aspect of modern software development, automating workflows to enable frequent integration, rapid feedback, and reliable software releases. The performance of these pipelines directly influences the speed and efficiency of the software delivery lifecycle, making optimization essential as development projects need to scale. This paper explores 9 foundational performance patterns that address key forces such as pipeline speed, resource efficiency, and scalability. The patterns deal with, among other things, reducing inefficiencies when running the pipeline and increasing the usage of available resources. One common strategy employed in the patterns to address inefficiency is reducing the number of tasks executed in the pipeline. Our pattern mining study draws upon a dataset from an empirical analysis of 31 grey literature sources, exploring practitioner perspectives on enhancing CI/CD pipeline performance. Furthermore, we analyze multiple mature GitLab and GitHub repositories in-depth to find known uses of the presented patterns.

Keywords: CI/CD · Continuous Integration · Continuous Delivery · Performance · Patterns · Grounded Theory.

1 Introduction

Continuous Integration and Continuous Deployment (CI/CD) pipelines [22] are automated workflows that streamline the building, testing, and deployment of software. By promoting frequent integration and reliable delivery, CI/CD enables development teams to receive faster feedback, improve software quality, and maintain consistent release cycles [25,23,3]. These pipelines typically involve several phases—including building, automated testing, artifact generation, and deployment—that ensure every code change is validated and production-ready. As such, CI/CD pipelines are a cornerstone of modern software engineering, closely aligned with agile methodologies and DevOps principles to boost productivity and system reliability.

In this context, pipeline performance becomes critical to accelerating the software delivery lifecycle. CI/CD pipelines automate repetitive tasks, facilitating rapid feedback and dependable releases [18]. To maintain these benefits at scale, performance considerations center on minimizing latency, maximizing throughput, and optimizing resource utilization across pipeline activities. Addressing performance is not merely a matter of speed. Inefficient pipelines can worsen system-related metrics, such as introducing serious bottlenecks, delaying feedback, inflating infrastructure costs. Furthermore, they can also diminish developer productivity [26,8], by, among other things, making developers switch context and activities while waiting for a pipeline run to complete. Ultimately, slow builds can postpone the identification of integration issues, reducing development velocity and slowing down time-to-market. Our proposed patterns help mitigate such issues, offering actionable solutions rooted in real-world usage while still grounded in fundamental CI/CD principles.

A growing body of practices and strategies has emerged to address performance challenges in CI/CD systems [24,27,7,1,15]. These strategies aim to enhance scalability and responsiveness by refining core aspects such as resource allocation, task orchestration, and failure recovery. By emphasizing modular design, parallel execution, and efficient dependency management, teams can build pipelines that are not only faster but also more resilient and adaptable.

In this paper, we organize these strategies into a set of performance patterns, providing a structured lens through which to understand and apply performance improvements in CI/CD workflows. While individual techniques may have been discussed in prior work, they are rarely synthesized into reusable, high-level abstractions. Our pattern-based approach bridges this gap by translating practical experiences and scattered best practices into cohesive design guidance that can be applied across various CI/CD tools and environments. We present a set of foundational performance patterns for CI/CD systems, exploring their role in improving pipeline efficiency and maintainability. We examine the underlying forces driving the adoption of these patterns and demonstrate their application through examples, primarily focused on GitLab CI/CD. Our goal is to provide practical guidance for DevOps practitioners, software engineers, and system architects aiming to enhance the performance of the leveraged CI/CD systems.

This article is structured to build a cohesive argument for the adoption of performance patterns in CI/CD systems. Section 2 reviews the relevant scientific literature, highlighting existing knowledge on CI/CD performance and identifying gaps that our pattern-based approach addresses. Building on this foundation, Section 3 outlines the methodology used to derive and validate the performance patterns presented in this work. Section 4 illustrates how they can be applied in practice to improve pipeline performance. Finally, Section 5 reflects on the broader implications of our findings and offers recommendations for both practitioners and future research.

2 Related Work

Several works have focused on CI/CD patterns to speed up pipelines. For instance, Gallaba et al. [7] proposed a tool that automatically leverages the caching of dependencies across pipeline executions. Abdalkareem et al. [1] introduced rule-based techniques for skipping pipeline runs. Machalica et al. [15] studied metrics and indicators to select which tests to run and which to skip. However, these studies focus on isolated patterns or techniques for speeding up pipelines rather than providing a comprehensive perspective.

Other works have aimed at cataloging CI/CD patterns and practices more broadly [6,13]. Giao et al. [5] have analyzed GitHub repositories to understand the CI/CD technologies developers use. By leveraging the GitHub search API, repositories using state-of-the-art CI/CD tools are identified and analyzed. Although insightful, these efforts do not emphasize performance optimization.

Performance-related studies have also been conducted. For instance, Ghaleb et al. [11] have studied several characteristics of CI builds that may be associated with the long duration of CI builds by performing an empirical study on 104,442 CI builds from 67 GitHub projects. Furthermore, Yin et al. [27] list performance-related CI patterns. Their study analyzed techniques developers use to speed up CI pipelines, based on 2,896 open-source `CircleCI`[3] jobs.

Our work focuses on CI/CD and is not limited to CI pipelines. Furthermore, our patterns catalog is based on the grey literature we studied. Despite the value offered by grey sources for software engineering investigations [9], none of the previous studies employ them. Few other studies use grey literature regarding CI/CD pipelines. Zampetti et al. [28] cataloged 79 CI bad smells relying on 2300 Stack Overflow posts. Contrary to them, we focused on CI/CD performance patterns rather than generic CI bad practices. To conclude, after exploring the corresponding scientific literature, it became apparent that there is a research gap in terms of the identification and analysis of the various CI/CD performance patterns. This study is geared towards mitigating this gap.

3 Research Method

This study aims to uncover and validate CI/CD performance-related patterns, relying on grey literature and analysis of real-world repositories. Patterns offer reusable solutions to common problems, enabling teams to implement best practices and avoid pitfalls.

The primary knowledge sources utilized in this study were derived from an extensive analysis of 31 grey literature sources, including materials such as blog posts and system documentation [9]. We have employed grey literature given its high value for research in software engineering [10]. These sources were examined in depth using practices and principles from Straussian Grounded Theory (GT) [4], a systematic approach to derive theory from data. We follow the GT

[3] https://circleci.com/docs/: accessed May 2025

process of open, axial, and selective coding. The constant comparison procedure complemented coding to identify recurring patterns and relationships. To gather sources, we relied on search engines (e.g., Google, DuckDuckGo) and used queries such as "making CI/CD pipelines faster," "making CI/CD pipelines more efficient," etc.

To augment our analysis, we also examined the 5 most popular (i.e., most starred) public repositories on GitLab. We selected repositories by popularity, given its high precision in finding mature candidates for software engineering investigations [17]. All 5 repositories, listed in Table 1 alongside some of their characteristics, presented a CI/CD workflow deemed mature for our investigation. We focused exclusively on repositories from the community, excluding the ones maintained by the GitLab team. We have included the specific commit hash we investigated to enable the replication of our analysis. All the commits were pushed in December 2024 to the repository default branch. Table 2 summarizes the employed patterns. Furthermore, when less than 3 of these repositories applied a presented pattern, we looked for additional known uses, employing both search engines (e.g., Google, DuckDuckGo) and GitHub search API[4].

Table 1: Most starred GitLab repositories from the community

Full Name	Commit Hash	Programming Languages	Stars	Repository Type
`inkscape/inkscape`	0be98cd8	C++	3.5 k	Image editor
`CalcProgrammer1/openrgb`	c2215cbe	C++	3.0 k	Universal light driver
`fdroid/fdroidclient`	140fdaa8	Java, Kotlin	2.3 k	Android app manager
`veloren/veloren`	11f84d12	Rust	2.2 k	Online game
`baserow/baserow`	727ab462	Python, Javascript	2.0 k	SaaS

We defined specific methods for certain patterns to identify their presence in a repository based on GitLab documentation. For example, while *Comprehensive Pipeline Automation* is a broad concept, we narrowed our focus to fully automatic pipelines. Similarly, we established rigorous criteria to determine whether a specific pattern was applied. These definitions ensure clarity and consistency in the methodology applied in this paper.

Our analysis primarily concentrated on the `.gitlab-ci.yml` file and related YAML template files and scripts (e.g., Bash, Gradle) referenced within these files. We thoroughly reviewed these files. We analyzed the remaining ones (e.g., source files, documentation, etc.) only with keyword searches. Additionally, we analyzed pipeline job logs using keyword searches.

[4] https://github.com/search/advanced: accessed May 2025

Table 2: Usage of patterns in the analyzed GitLab repositories

Pattern	inkscape	openrgb	fdroidclient	veloren	baserow
Task Parallelization	X	X	X	X	X
CI/CD Architecture Scaling Strategy					
Pipeline Asset Caching	X	X	X	X	X
Incremental Build	X		X		
Pipeline Test Ordering			X		
Selective Testing			X		
Conditional Pipeline and Job Triggering	X	X	X	X	X
Comprehensive Pipeline Automation			X	X	
Mocked External Services			X		X

Besides analyzing these 5 repositories, we provide some source code examples to apply the patterns proposed. All the examples[5] have been tested on a public GitLab repository[6].

Towards contextualizing our approach, we leveraged a plethora of processes. Hentrich et al. [12] provide a detailed account of how GT's coding process can be mapped to pattern mining. Riehle et al. [21] elaborate on various systematic pattern mining methods, outlining key steps such as discovery, codification, evaluation, and validation of patterns [2]. These steps were integral to our methodology, with GT-based pattern mining incorporating them directly into the coding and constant comparison processes. Adopting these procedures ensures a rigorous and systematic approach to identifying and validating CI/CD performance patterns, enhancing the reliability and applicability of our findings.

All the grey literature sources, alongside some notes of our sources analysis and repositories investigations, can be found in the replication package[7]. Employing the notes we elaborated after the three coding phases, we propose a catalog of 9 patterns.

4 Performance Patterns

This section describes in depth the 9 patterns proposed in our work, which focus on optimizing the speed and efficiency of CI/CD pipelines. An overview of these performance optimization patterns is depicted in Table 3.

[5] Note that the presented examples may partially differ from the ones tested (e.g., distinct job names, shell arguments) as the former are often simplified for higher understandability.

[6] https://gitlab.com/random_researcher/cd-performance-patterns

[7] https://doi.org/10.5281/zenodo.14747351

Table 3: Overview of CI/CD Performance Optimization Patterns

Pattern	Purpose
Task Parallelization	Splits tasks into smaller, independent units that can run concurrently, reducing total pipeline time.
CI/CD Architecture Scaling Strategy	Dynamically adjusts resource allocation based on workload to improve performance and avoid over-provisioning.
Pipeline Asset Caching	Reuses previously built assets (e.g., dependencies, intermediate results) to reduce redundant work.
Incremental Build	Builds only modified parts of the system, reducing build time and avoiding unnecessary computation.
Pipeline Test Ordering	Prioritizes tests by importance or cost to deliver faster feedback with reduced resource consumption.
Selective Testing	Executes only tests relevant to the changes made, minimizing time without sacrificing coverage.
Conditional Pipeline and Job Triggering	Triggers only necessary pipeline phases or tasks based on the nature of the change.
Comprehensive Pipeline Automation	Fully automates the pipeline for reliability, consistency, and reduced manual intervention.
Mocked External Services	Replaces real external services with lightweight mocks during testing to increase speed and reduce dependency overhead.

4.1 Pattern: Task Parallelization

Context CI/CD pipelines typically may execute tasks sequentially unless explicitly configured for concurrency. However, many tasks—such as independent test suites, static analysis, or security scans—can be executed concurrently without introducing conflicts. This pattern applies when the pipeline contains tasks that are logically independent and can be safely executed in parallel. It also assumes the existence of unused infrastructure (e.g., idle runners or unused CPU cores) and a need for improved execution speed or feedback cycles.

Problem Many CI/CD jobs are executed sequentially, leading to inefficient use of infrastructure. As a result, some runners may be idling even though other tasks could already be executed in parallel, delaying feedback and increasing overall pipeline duration.

Forces

- **Effective Resource Utilization vs Dependency Management:** Employing all machines managed by the CI/CD orchestration tool to run as many CI/CD jobs as possible ensures effective execution of pipeline tasks, but this can increase the complexity of task coordination, particularly when dependencies and shared resources are involved.
- **Parallel Computations vs Data Consistency and Reliability:** Ensuring a consistent state of data —such as files and environment variables—is important during build, test, and deployment phases. Failing to do so can worsen the repeatability of pipeline executions, ultimately delaying feedback when bugs appear. Nevertheless, the consistency requirement poses challenges when tasks are executed in parallel.
- **Error Handling:** Prompt termination of jobs upon detecting errors or failing to meet predefined quality thresholds helps conserve resources and maintain pipeline integrity. However, determining appropriate termination strategies is difficult when task dependencies must be accounted for, as failure in one task does not always justify halting the entire workflow.

Solution Enable concurrent execution of independent CI/CD tasks by leveraging both inter-job and intra-job parallelism. Use dependency analysis to identify which tasks can safely execute in parallel, and configure the pipeline to run them concurrently, thereby reducing overall execution time and improving feedback cycles.

Solution Details Begin by identifying all tasks in the pipeline and analyzing their dependencies. Determine which jobs are independent and can be safely executed in parallel without introducing conflicts. Explicitly define dependencies where required, using orchestration features such as `needs` in GitLab or equivalent mechanisms in other CI/CD tools.

Configure inter-job parallelism by placing independent tasks in the same pipeline phase or explicitly marking them as non-dependent. For tasks that are internally parallelizable (e.g., test runners or build tools), configure the tool to utilize multiple threads or CPU cores using flags or environment variables.

Although infrastructure underutilization is assumed, it is important to assess whether the available capacity can meet the increased demand of parallel execution. Monitoring tools can help identify idle runners, underused cores, or bottlenecks.

To reduce risks from concurrency, tasks should be designed to be stateless and avoid reliance on shared data or global state. Where shared state is unavoidable, synchronization mechanisms must be put in place to ensure consistency.

Example Figure 1 shows an example of a CI/CD pipeline in which the unit tests are isolated and independent. Thus, parallel unit testing, as provided by JUnit, can be enabled. Integration testing is then sequentially run after the unit

tests. The pipeline contains static code analysis and security scans as quality controls. Both run in parallel to the basic tests and are invoked concurrently. Performance and user acceptance testing are parallel pipeline tasks, as they have no overlap. However, performing both in parallel may be a waste of resources if one fails. Further, an analysis showed that users could perform manual User Acceptance Testing (UAT) in parallel per system feature, with little overlapping concerns that would require coordination among testers. Also, the usability tests are performed in parallel. The UAT parallelization is performed in the UAT coordination component (outside the pipeline).

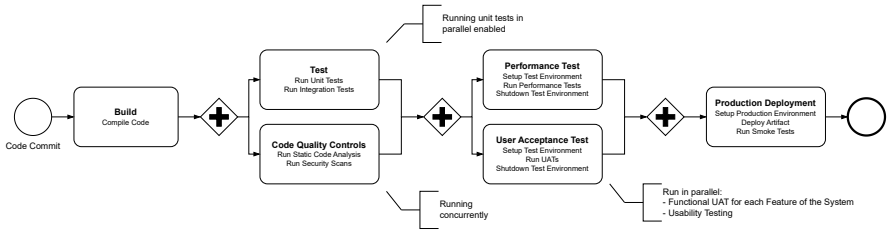

Fig. 1: Test Parallelization to Support Task Parallelization and Pipeline Test Ordering

In the following listing, part of the pipeline shown in Figure 1 has been implemented with the GitLab CI tool. While these examples are GitLab-specific, the concepts are generalizable and applicable to other CI/CD tools such as GitHub Actions, CircleCI, and Jenkins. We have defined 4 phases and 7 jobs, 4 of which are implemented. The `basic_tests`, `static_analysis`, and `security_scans` jobs run all in parallel, while the unit tests are executed 4 per time.

For this scenario, it makes sense to design the pipeline using phases. Nevertheless, in other scenarios, explicit job dependencies using the `needs`[8] keyword may be more appropriate. With `needs`, jobs from one phase do not need to wait for all the jobs from the previous phase to complete, but only specific ones.

```
image: gradle:8.1.1-jdk11-alpine

phases:
  - build
  - test_first_phase
  - test_second_phase
  - deploy

build_job:
  phase: build
  script:
    - ./gradlew assemble
```

[8] https://docs.gitlab.com/ee/ci/yaml/index.html#needs: accessed May 2025

```
basic_tests:
  phase: test_first_phase
  script:
    - ./gradlew test --tests "tests.unit"
    -DmaxParallelForks=4
    - ./gradlew test --tests "tests.integration"

static_analysis:
  phase: test_first_phase
  script:
    - ./gradlew staticAnalyzer

security_scans:
  phase: test_first_phase
  script:
    - ./gradlew securityAnalyzer

# ...remaining jobs...
```

Consequences The application of the pattern *Task Parallelization* presents the following consequences:

(+) The pipeline becomes faster thanks to multiple jobs running at the same time, as well as jobs employing multithreading.
(+) The jobs can scale better when the workload is increased.
(+) Fewer machines will be idling, waiting to execute a job.
(+) When executing a job, this will employ more of the available resources (i.e., processor and memory).
(−) The tasks have to be initially adapted and continuously maintained to employ parallel computations. Special attention is required to identify dependencies and handle errors while multiple tasks are executed at the same time.
(−) The costs of operating more machines, as well as employing more resources within a machine, increase.

Related Patterns *Task Parallelization* can be particularly effective when combined with *Pipeline Test Ordering*. By categorizing and prioritizing tests, you can parallelize the execution of critical and short-running tests, reduce bottlenecks, and improve feedback times. Tests can be parallelized in the test framework or via parallel pipeline phases.

Known Uses In GitLab, all jobs in the same phase run in parallel[9] unless dependencies are defined. For this reason, all the 5 repositories do apply inter-job parallelism. Nevertheless, only 3 repositories (`inkscape`, `openrgb`, `baserow`)

[9] Whether two jobs actually run in parallel also depends on the number of available runners.

configure explicit job dependencies with `needs`. In the other two projects, a job in stage B must wait for each job in stage A before starting, even if it doesn't depend on all of them.

In addition, we have found evidence for intra-job parallelism for 3 repositories:

- `inkscape` runs multiple `make`[10] commands specifying to use 3 parallel threads. Furthermore, the project employs `ninja`[11], which, by default, attempts to use as many threads as available cores.
- `veloren` uses `cargo`[12] to build and test the Rust application. This tool uses all available cores unless specified otherwise.
- `baserow` employs the `pytest-split`[13] package to split Python tests based on their expected execution time and then executes them in separate threads.

4.2 Pattern: CI/CD Architecture Scaling Strategy

Context Development teams encounter fluctuating workloads in their CI/CD pipelines, with some phases experiencing surges in demand during peak development periods.

Problem Fixed infrastructure allocation for CI/CD pipelines can lead to inefficient resource utilization, where resources are either underused during low demand or insufficient during high demand. This imbalance impacts execution times and increases costs.

Forces

- **Performance Stability vs Operational Costs:** The resource utilization of CI/CD pipelines fluctuates over time—daily (day vs. night), weekly (weekday vs. weekend), and monthly [20]—and the infrastructure must be capable of sustaining baseline performance under these varying loads. Nevertheless, ensuring such stability often requires overprovisioning or dynamic scaling strategies, which introduce additional operational complexity.
- **Efficiency vs Cost and Availability:** Fleet configuration must minimize wasted computational resources (e.g., during idle periods) and control overall expenditures, but optimizing for cost efficiency can conflict with the need to maintain performance headroom for peak workloads.
- **Effectiveness vs Maintenance:** A functional and reliable runner fleet supports CI/CD operations effectively. However, it requires constant maintenance overhead and increases infrastructure complexity, especially when tailored strategies are used to manage scaling, fault tolerance, and software updates.

[10] https://www.gnu.org/software/make/manual/make.html: accessed May 2025
[11] https://ninja-build.org/manual.html: accessed May 2025
[12] https://doc.rust-lang.org/cargo/: accessed May 2025
[13] https://jerry-git.github.io/pytest-split/: accessed May 2025

Solution Implement dynamic scaling for pipeline resources based on workload demand. Use cloud-native solutions, container orchestration platforms, or CI/CD tools supporting auto-scaling capabilities to adjust resources dynamically.

Solution Details CI/CD Architecture Scaling Strategy involves real-time monitoring of pipeline workloads and using scaling triggers to allocate or release resources. For example:

- Use cloud-based runners or Kubernetes clusters with auto-scaling capabilities.
- Implement thresholds for resource allocation based on pipeline metrics such as task queue length, CPU, or memory utilization.
- Define policies to scale down resources during idle times to reduce costs.
- Ensure scaling mechanisms are robust to avoid delays during scaling up. Use predictive scaling where historical data informs future demand patterns, optimizing resource availability.

Example Figure 1 illustrates a CI/CD pipeline running on a Kubernetes cluster with auto-scaling enabled. The cluster consists of three nodes, each one dedicated to hosting a distinct part of the pipeline. Additional pods are spawned during peak hours to handle concurrent tasks like builds, tests, and deployments. Off-peak hours trigger resource downscaling. In the case of the corresponding figure, this process is showcased using the testing part of the pipeline as an example.

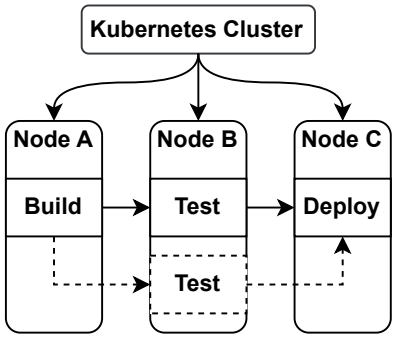

Fig. 2: A Kubernetes Cluster, Hosting a CI/CD Pipeline, Exhibiting Dynamic Autoscaling.

Consequences The application of the pattern *CI/CD Architecture Scaling Strategy* presents the following consequences:

(+) The pipeline becomes faster thanks to the architecture scaling to the resources needed.

(+) Operational costs decrease when the scaling strategy focuses on efficiency, keeping active only the machines currently executing tasks and shutting down or hibernating the others.
(−) An architecture scaling to the pipelines' needs adds maintenance overhead and increases overall infrastructure complexity.
(−) Operational costs increase when the scaling strategy focuses on performance, trying to match as much as possible the resource demands of the CI/CD workflows.

Related Patterns *CI/CD Architecture Scaling Strategy* can be combined with *Task Parallelization* to keep up with resource demands during parallel task execution without over-provisioning resources. This ensures that, if the tasks are designed to be run in parallel, they are actually run in parallel and they are not queued because of a lack of machines.

Known Uses We could not determine whether the pattern is applied in the investigated repositories since GitLab autoscaling capabilities[14] are accessible only with high-level repository roles. While searching for possible autoscaling signatures (e.g., in the documentation and/or source files), we did not find any evidence. Nevertheless, the pattern may still be implemented in each of them.

To provide examples of CI/CD Architecture Scaling Strategy, we focused on `ARC`[15], a GitHub action capable of autoscaling self-hosted machines for CI/CD pipelines. We identified three popular repository where this technology is applied in one or more pipelines: `mudler/LocalAI`[16], `0chain/zwalletcli`[17], `ifooth/devcontainer`[18].

4.3 Pattern: Pipeline Asset Caching

Context Development teams encounter fluctuating workloads in their CI/CD pipelines, with some phases experiencing surges in demand during peak development periods. This pattern is particularly relevant in environments where scalable infrastructure (such as cloud platforms or container orchestration systems like Kubernetes) is available to dynamically provision and de-provision resources.

In contrast, teams operating on fixed, self-managed infrastructure may face limitations in applying this strategy. Without access to elastic resources, the ability to respond to workload spikes is constrained, requiring careful provisioning and potentially leading to trade-offs between performance and resource cost. Therefore, while the strategy is applicable in various deployment contexts, it is most effective when some level of infrastructure scalability is available.

[14] https://docs.gitlab.com/runner/runner_autoscale/: accessed May 2025
[15] https://github.com/actions/actions-runner-controller: accessed May 2025
[16] https://github.com/mudler/LocalAI/tree/e81ceff: accessed May 2025
[17] https://github.com/0chain/zwalletcli/tree/0f11d2e: accessed May 2025
[18] https://github.com/ifooth/devcontainer/tree/44d39d1: accessed May 2025

Problem Constantly re-creating pipeline assets during different phases and runs, including re-downloading dependencies, re-generating artifacts, and reconfiguring environments, can significantly slow down the CI/CD process and increase resource consumption.

Forces

- **Pipeline Performance:** CI/CD pipeline jobs often require either fetching artifacts from external services or computing them locally. Caches remove the need by saving the results of a pipeline run. Nevertheless, excessive usage of caches can decrease pipeline speed when less time is required for re-processing the artifacts than using the cache. This risk is present especially for heavy artifacts in distributed cache systems, where network speed affects the time for using artifacts.
- **Efficiency vs Complexity and Consistency:** Redundant computations or repeated artifact downloads should be minimized to conserve resources, but implementing effective caching and reuse mechanisms adds complexity to pipeline design and management. Special attention should be put into the consistency of the cached artifacts. These must accurately reflect the current state of the codebase to ensure correct builds and tests.
- **Reliability:** When the result of a complex task (e.g., distributed computations) is cached, future pipelines have higher chances of success. However, the pipeline must re-run all tasks if the cache is not consistent anymore with the codebase state. Failing to do so can make the pipeline less predictable and ultimately less reliable.
- **Resource Usage:** Caches reduce the number of computations required to accomplish a task. Nevertheless, cached assets must be stored on disks managed by the orchestration tool, which needs an adequately provisioned capacity.

Solution Implement pipeline asset caching for storing and reusing assets such as dependencies, build artifacts, configurations, and environments. Configure the CI/CD system to determine when cached assets can be reused and when they need to be updated to ensure consistency.

Solution Details A caching strategy and realization should be derived for each artifact that benefits from being reused across pipeline phases or runs. For instance, consider typical solutions for the following types of assets:

- *Dependencies*: Use the cache functions of package managers, such as `npm`, `maven`, and `pip`, to store locally the downloaded dependencies. Ensure that the CI/CD pipeline reuses these caches unless there are changes in the dependency list or version updates.
- *Build Artifacts*: Cache the build outputs to avoid redundant compilations. Caching these artifacts can significantly reduce build times, especially for large projects.

- *Environments*: Cache environments such as Docker images and layers or VM snapshots. Use these cached setups to speed up environment provisioning.
- *Other Artifacts*: Include caching of additional artifacts such as configurations, security scan reports, performance metrics, accuracy metrics, or generated documentation to avoid recreating these files when unnecessary.

In addition, developers should consider implementing *cache invalidation* strategies to maintain the accuracy of assets. This can include tracking changes in the code base, dependency updates, or configuration changes to decide when the cache needs to be updated. Note that most tools can recognize whether a cache is still valid or if it should be processed/downloaded again.

Other aspects of the caching pattern are essential for its effectiveness, such as the *assets availability:* not all cached assets can or should be used in every pipeline. Specifically, some assets can be cached only for the duration of a specific pipeline and deleted as soon as this terminates, while others can be used across multiple pipeline runs. Furthermore, CI/CD tools can be configured to have separate caches for protected and not-protected branches. This functionality is useful to address security concerns but may limit the benefits of the pattern by doubling the number of times certain assets are built or downloaded.

Besides availability, the cache *location* determines the speed to access it. For dynamic runners, a distributed cache can be more effective than a runner's local cache by reducing the time of the *warming* phase.

Example In GitLab CI/CD, two caching strategies are available. Assets can be saved to be used across multiple pipeline runs or only across jobs of the same pipeline run. The former strategy is available with the `cache`[19] keyword, while the latter with `artifacts`[20].

In the GitLab CI/CD pipeline code excerpt below, the cache settings are defined under the clause `cache`. When a job is completed, GitLab saves the specified cache paths in its caching system. In subsequent pipeline runs, GitLab attempts to restore the cache with the specified key. If a cache matching the key is found, the cache files are downloaded and extracted to the job's working directory before the job scripts are executed.

```
build-job:
  phase: build
  script:
    - gradle clean build --build-cache
  cache:
    key: "$CI_COMMIT_REF_SLUG"
    paths:
      - .gradle/
      - build/
    policy: pull-push
```

[19] https://docs.gitlab.com/ee/ci/yaml/index.html#cache: accessed May 2025
[20] https://docs.gitlab.com/ee/ci/yaml/index.html#artifacts: accessed May 2025

Consequences The application of the pattern *Pipeline Asset Caching* presents the following consequences:
(+) The pipeline becomes faster and more efficient thanks to fewer repeated tasks across multiple runs.
(+) The jobs can scale better when the workload is increased.
(−) Cached assets have to be consistent with the state of the codebase. Invalidation strategies must be implemented to avoid issues with inconsistency.

Related Patterns *Pipeline Asset Caching* complements *Incremental Builds* by storing build outputs and dependencies. Caching test results or setups can complement *Selective Testing* by storing information on code changes required to execute the tests selectively. It also complements *Pipeline Test Ordering*, as necessary artifacts, test configurations, and test environments can be cached for later phases or future pipeline runs.

Known Uses All 5 investigated repositories apply caches for at least one type of asset. `inkscape` and `fdroidclient` configure a cache to use across multiple pipeline runs with the `cache` clause. The remaining 3 projects only apply the `artifacts` functionality, caching assets only for the duration of a pipeline. Table 4 illustrates in detail the type of cached artifacts for each repository.

Table 4: Summary of Cached Artifacts by Repository

Project Name	Build Results	Documentation Pages	Docker Layers
`inkscape`	X	X	
`openrgb`	X		
`fdroidclient`	X		
`veloren`	X		
`baserow`			X

4.4 Pattern: Incremental Build

Context Many CI/CD projects commit often and do not apply significant changes in a single commit.

Problem Building the entire codebase from scratch for each pipeline run, regardless of the actual scope of changes, leads to unnecessary time and resource consumption. This is particularly inefficient for projects where only small portions of the code change between commits. Development teams need strategies that limit the build scope to changed components to maintain fast feedback loops.

Forces Although the forces for the *Incremental Build* pattern are the same as the *Pipeline Asset Caching* pattern, they differ by focusing on the build process and its dependency analysis:

- Accurate change detection and dependency tracking are required to determine which components need rebuilding, which can introduce computational overhead.
- There is a tension between the need for fast builds and the potentially costly analysis needed to detect the minimal required rebuild scope.

Solution Implement incremental builds by configuring the build system to recognize changes in the code base and recompile only the changed components and their dependencies. This approach uses tools and scripts to analyze changes and determine the minimum necessary rebuilding components.

Solution Details Incremental builds require the setup of a build system, such as `make` or `gradle`, that supports change detection and dependency management. These systems track file changes and dependency graphs to identify only the affected components.

Developers must ensure proper configuration and maintenance of build scripts to capture dependencies accurately and employ caching mechanisms between CI/CD pipeline runs to store build artifacts for reuse.

Example `gradle` is a popular technology supporting *Incremental Builds* per default. The command `gradle jar` in the example below specifies that `gradle` should build a jar file of the application. `gradle` tracks changes in dependencies and input files and only rebuilds targets that have changed since the last build. This means that if a file has not changed, `gradle` will reuse the output of previous builds, speeding up the build process.

The example below uses the cache key `APPLICATION_VERSION`. This ensures that the cache is used consistently for all builds of a certain version of the application and that the integrity of the incremental build process is maintained.

```
build-job:
  phase: build
  script:
    - gradle jar
  cache:
    key: "$APPLICATION_VERSION"
    paths:
      - .gradle/
      - build/
    policy: pull-push
```

Consequences The application of the pattern *Incremental Build* presents the following consequences:

(+) The pipeline becomes faster and more efficient thanks to fewer computations for build jobs across multiple runs.
(+) The build jobs can scale better when the workload is increased.
(–) Incremental builds must correctly reflect the current state of the codebase. This adds operational complexity because stale or inconsistent outputs can arise if dependency tracking or change detection fails. Thus, invalidation strategies must be implemented to ensure correctness.

Related Patterns *Incremental Build* benefits from *Pipeline Asset Caching*, which can be used to store build outputs and dependencies.

Incremental Build can be combined with *Selective Testing*, as the impact and dependency analyses performed for the build can be reused to select the impacted tests.

Task Parallelization can be supported in the *Incremental Build* when builds are run in parallel. *Conditional Pipeline and Job Triggering* complements *Incremental Build* as it can decide whether to launch the build at all.

Known Uses Only 2 repositories build their application incrementally: `inkscape` and `fdroidclient`. The former saves the `ccache`[21] directory across multiple pipeline runs, while the latter saves `gradle`'s cache.

Another example of the pattern is `wireshark/wireshark`[22], where the `ccache` folder is saved at the end of a pipeline.

4.5 Pattern: Pipeline Test Ordering

Context CI/CD requires frequent and automated testing[23] to ensure the functionality and quality of the committed code. In addition, some tests, such as performance or user acceptance tests, are inherently longer running than other tests, such as unit tests.

Problem As projects increase in complexity and team size, the number of tests and quality controls also increases, leading to longer execution times for the CI/CD pipelines. Thus, testing can become a bottleneck in the CI/CD pipeline and seriously affect the overall performance and scalability of the CI/CD architecture.

Forces

[21] https://ccache.dev/: accessed May 2025
[22] https://gitlab.com/wireshark/wireshark/-/tree/12a9e2f2: accessed May 2025
[23] The term testing is used broadly in this context, including all kinds of tests and quality controls such as code linters, static code analysis, or security scans

- **Feedback Speed vs Feedback Type:** Tests that provide the most valuable and actionable feedback should be prioritized to enable rapid detection and correction of errors, but focusing solely on speed may lead to deprioritizing broader test coverage, potentially allowing certain issues to go unnoticed until later stages.
- **Feedback Speed vs Efficiency:** Limited computing resources necessitate efficient utilization of infrastructure, particularly by minimizing resources spent on failing pipelines. Nevertheless, aggressive optimization may compromise test completeness or delay the detection of non-critical but relevant defects.
- **Risk Management:** Tests vary in their criticality and impact; prioritizing high-risk areas helps ensure that failures with significant consequences are caught early. However, this approach may result in delayed feedback on lower-risk components that can still affect overall software quality.

Solution Execute first the shorter-running and most critical tests and quality controls to provide fast and relevant feedback. Categorize tests according to runtime and criticality to optimize their ordering in the CI/CD pipeline.

Solution Details Tests and quality controls in CI/CD pipelines are categorized into different groups, such as unit tests, integration tests, system tests, end-to-end tests, security checks, code linting, security scans, static code analysis, other code and design quality checks, performance tests, load tests, and user acceptance tests. Each group serves a specific purpose and has different execution times and resource requirements. Categorization helps to structure the pipeline phases effectively according to some essential criteria.

From a performance and scalability point of view, running shorter-running tests, such as unit tests, early in the pipeline avoids waiting a long time for basic tests that may fail. This also gives developers quick feedback. These tests are usually not resource-intensive and can quickly validate individual components or functions. Early detection of problems enables faster troubleshooting and resolution, reducing the overall cycle time of the pipeline.

A second important criterion is related to importance and risks. Running critical tests early, ensures the application's most important aspects are checked before moving on to less critical tests. Critical tests often include those related to core functionality, security scans, and high-risk areas that could halt the pipeline in the event of failure to prevent further development of faulty software. This is not necessarily the case for all tests or quality controls. For instance, a performance or user acceptance test can have a non-optimal result but still lead to production deployment – with improvements in later development iterations. In contrast, a failed test of a core functionality or security aspect is critical and thus should halt the pipeline.

When combining this pattern with *Task Parallelization*, several tests, quality controls, and other pipeline phases are executed simultaneously and not one after the other. This technique utilizes the available computing resources to

reduce the time required to execute the tests. For instance, independent tests in one category, like different unit tests, can be parallelized. Usually, this happens outside the pipeline, e.g., in the test framework. In addition, independent tests with similar running-time length, importance, or risks can be run in parallel, too. This usually requires parallel pipeline steps. For instance, performance tests can be run in parallel to load tests. Please note that there is a risk of wasting resources due to parallelization, as the tests in a parallel branch may fail, meaning that the other tests have run unnecessarily.

Selective Testing can help execute only tests related to the latest code changes. While this is a complementary pattern, it is not the focus of *Pipeline Test Ordering*, which centers on structuring the execution order rather than deciding test inclusion.

Example The CI/CD pipeline in Figure 1 is an example for *Pipeline Test Ordering*. First, the unit tests are run, followed by the longer-running integration tests. They are the two most critical kinds of tests in this pipeline. It usually makes sense to run unit tests before integration tests, as a failed unit test would report a localized problem quickly back to developers. In contrast, integration tests might take longer to report back and provide less information on where a problem might be. Furthermore, a failed unit test will likely cause an integration test to fail, but the reverse is unlikely.

The quality controls also report back quickly and are critical, too. Thus, they can be run in parallel with unit and integration tests. Some pipelines would run security scans after unit tests to ensure the essential functionality works before performing these scans.

The performance and user acceptance tests have a reasonably long runtime and are likely similarly critical, so it is acceptable to run them in parallel. A typical alternative is to run performance tests before user acceptance tests, as the latter usually take longer.

Consequences The application of the pattern *Pipeline Test Ordering* presents the following consequences:

(+) More relevant feedback becomes faster, thanks to job ordering based on relevance.

(+) Less computations are spent on failing pipelines.

(−) Feedback for less critical components may be delayed, especially when the ordering strategy is aggressive.

Related Patterns *Pipeline Test Ordering* often involves *Conditional Pipeline and Job Triggering* to execute only the most relevant tests and potentially reduce the total number of tests. Optimizing tests often means running them in parallel, i.e., using *Task Parallelization*. Tests can be parallelized in the test framework or via parallel pipeline phases.

Known Uses Across the 5 GitLab repositories we focused on, we have found only 1 (`fdroidclient`) to be applying this pattern. In `fdroidclient`, a linting phase is defined before the actual tests, while in all the other projects, there is no prioritization between types of tests.

Other examples of the pattern are `wireshark`, where fuzzy testing is applied after unit tests, and `antora/antora`[24], where linting is done before testing.

4.6 Pattern: Selective Testing

Context Many CI/CD projects commit often and do not apply major changes in a single commit. Therefore, the outcome of many tests does not change compared to previous executions.

Problem Modern CI/CD pipelines must provide timely feedback to developers to sustain high development velocity. However, executing the full test suite for every code change leads to long feedback cycles, increased resource consumption, and potential bottlenecks. The challenge lies in maintaining testing efficiency and feedback speed as the codebase and number of tests grow.

Forces

- **Feedback speed vs Complexity:** Developers need rapid feedback to iterate quickly and catch errors early. Nevertheless, determining which tests are affected by a commit requires complex analysis or accurate dependency tracking, which is hard to implement.
- **Efficiency vs Reliability:** Developers must trust that test results reflect the actual state of the system, but skipping relevant tests due to incomplete impact analysis may cause bugs to slip through and reduce trust in the CI pipeline.
- **Scalability vs Maintenance:** As tests are added to a growing repository, selective testing can avoid execution times from considerably growing. This requires constant maintenance of the tracking system.

Solution Implement *Selective Testing* by performing a Test Impact Analysis [14] in your CI/CD pipeline to identify and execute only those tests relevant to recent code changes. Integrate tools that automatically track code changes and map them to specific tests. You can also prioritize tests based on risk and historical data to optimize testing efforts further.

Solution Details Test impact analysis is a technique that identifies and executes only tests relevant to the latest code changes. It helps to determine which parts of the codebase are affected by the changes and selects the tests accordingly. The analysis aims to minimize the number of tests performed by concentrating

[24] https://gitlab.com/antora/antora/-/tree/12c615e6: accessed May 2025

on the relevant tests, saving time and resources. As a downside, realizing a test impact analysis can be complex.

There are various tools for test impact analysis. A simple option is to base the analysis on an existing code coverage tool, such as `JaCoCo`, `SonarQube`, `NCover`, or `JSCoverage`. We can assign code changes to specific tests by letting the coverage tool capture metadata about code segments executed during the tests and log this information for analysis. Using this data, the CI/CD pipeline can determine which tests are affected by the changes in a pull request, enabling targeted test execution. This approach mirrors the selective testing procedures that developers often use locally.

This technique is based on static code analysis, analyzing the code structure and dependencies to identify affected areas. A general alternative is dynamic code analysis [19]. Dynamic code analysis uses runtime information to monitor the actual execution of the program, track dependencies, and execution paths, and determine which tests are affected by code changes. This approach can provide more accurate insights as it observes the code's behavior during execution and captures dynamic interactions that static analysis may miss. However, it can also be less precise, as any paths that are not executed will be missed by dynamic analysis, and dynamic analysis is resource-intensive. Especially for the latter reason, dynamic analysis or hybrid approaches are often not chosen over static analyses if pipeline performance and fast feedback should be improved.

Integrating test impact analysis tools into existing CI/CD pipelines involves configuring these tools to automatically detect code changes and trigger the appropriate tests, ensuring seamless operations and reducing manual intervention. Automation within the pipeline enables consistent and efficient execution of test impact analysis with minimal disruption to the development workflow.

In addition to identifying affected tests, test impact analysis can increase efficiency by prioritizing tests based on risk factors such as historical failure rates, the susceptibility of tests to failure, or the criticality of the code being tested. This approach helps to focus resources on the most important tests, optimize testing efforts, and improve software reliability.

As the codebase grows, the test impact analysis needs to scale to handle the increasing complexity and volume, which requires a robust infrastructure and efficient algorithms. Regular maintenance of the test selection logic is essential to adapting to changes in the code base and ensuring continuous accuracy. This requires regular updates and validation to keep the system effective.

Example Figure 3 illustrates how the results of a test impact analysis are stored in an artifact of a CI/CD pipeline so that subsequent phases can access this information. This phase analyzes code changes, and the affected tests are recorded in a file, which is then saved as an artifact. Both the Unit Tests and Integration Tests phases retrieve this artifact to execute only the relevant tests.

Besides storing artifacts, the CI/CD cache can also support *Selective Testing*. `Bazel`[25] is an example of a technology supporting *Incremental Builds* per default,

[25] https://bazel.build/: accessed May 2025

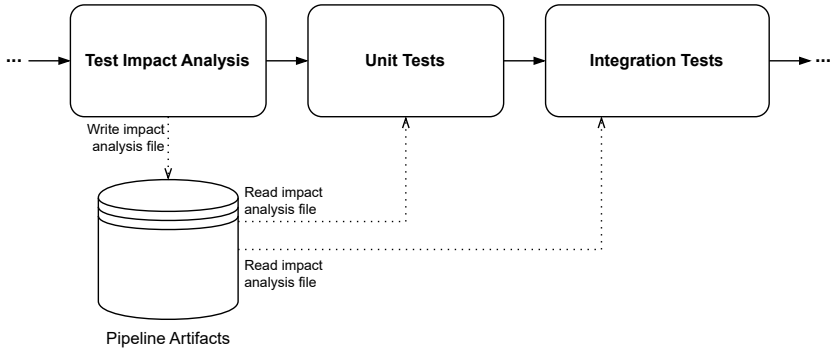

Fig. 3: Pipeline Excerpt Showing a Test Impact Analysis Stored in an Artifact and Used by Two Tests

as well as *Selective Testing* and also the integration of the two. For instance, the command `bazel build //:TestGroup -disk_cache=bzcache` in the example below specifies that `Bazel` should build all targets in the `bzcache` directory and only run tests for the code parts that have changed or are affected by changes to the code base. This approach leverages `Bazel`'s ability to track dependencies and changes and ensures that only relevant tests are executed, optimizing both build and test times. Using an explicit `-disk_cache` argument, `Bazel` can maintain its cache in a specific folder to persist across CI pipeline runs.

```
test:
  phase: test
  script:
    - bazel build //:BazelProject --disk_cache=bzcache
    - bazel test //:Tests1 --disk_cache=bzcache
    - bazel test //:Tests2 --disk_cache=bzcache
  cache:
    key: "$BAZEL_VERSION$"
    paths:
      - bzcache
```

Consequences The application of the pattern *Selective Testing* presents the following consequences:

(+) The pipeline becomes faster and more efficient, especially in pipelines with large test suites, as only impacted tests are executed.
(+) Improved scalability, as the number of tests executed can grow proportionally to the scope of each change rather than the overall size of the codebase.
(−) Increased pipeline complexity due to the integration and maintenance of test impact analysis mechanisms.
(−) Risk of false negatives if the analysis fails to identify all relevant tests, potentially decreasing reliability.

(−) Limited tool support or inconsistent results depending on the programming language, framework, or analysis approach.

Related Patterns *Selective Testing* benefits from *Pipeline Asset Caching*, which can store information on code changes required to execute the tests selectively.

Selective Testing can be combined with a prior *Incremental Build*, as the impact and dependency analyses performed for the build can be reused to select the impacted tests.

Selective Testing complements *Pipeline Test Ordering*, as only relevant tests are executed in each optimized phase.

Conditional Pipeline and Job Triggering complements *Selective Testing* as it can select pipeline triggers that decide whether to launch the test phase at all.

Known Uses When investigating the usage of the pattern, we have looked for specific job triggers based on file changes[26]. Although multiple repositories applied such rules, only `fdroidclient` uses them for selective testing.

Furthermore, we have investigated the usage of programming-language-specific tools[27] that run tests based on source code changes and searched their *signatures*. We have found no evidence in any repository.

Additional known uses of the pattern are `google/startup-os`[28], a tutorial system using `bazel` to automatically detect test changes, and `berty/berty`[29], a monorepo defining file changes as rules to conditionally run the tests.

4.7 Pattern: Conditional Pipeline and Job Triggering

Context As projects become more complex in CI/CD environments, commits are made for very distinct operations, such as adding comments, updating documentation, and testing new functionalities. Each of these operations requires different tasks to be run when the commit is pushed.

Problem Triggering the whole CI/CD pipeline for every change can lead to unnecessary pipeline runs or pipeline phase executions, increased resource consumption, and longer feedback times. This inefficiency can overwhelm shared resources and slow the development cycle.

Forces The forces related to this pattern are the same as the *Selective Testing* pattern, but are related to the whole pipeline and not only the testing phase.

[26] https://docs.gitlab.com/ee/ci/yaml/index.html#ruleschanges: accessed May 2025
[27] As an example, `pytest-testmon` is a Python library which can run only tests related to changes.
[28] https://github.com/google/startup-os/tree/5f30a62: accessed May 2025
[29] https://github.com/berty/berty/tree/e11e95a: accessed May 2025

Solution Configure pipeline and pipeline phase triggers so that the pipeline and its phases run only under specific conditions, such as when changes are made to specific files or directories or when changes are made to specific branches. This selective triggering ensures that the pipeline and its phases are only executed when required.

Solution Details Configure the CI/CD system to evaluate each commit or merge request and determine if it meets the predefined pipeline triggering conditions. For example, developers can use path-based rules to trigger pipelines only when changes to files or directories are essential to the next production build. Also, developers can restrict the execution of pipelines to specific branches, such as development or release branches, where changes are more likely to impact the system significantly.

Each pipeline phase can have a different optimization of its triggers, i.e., we can optimize at the phase level. For instance, we can run basic building and testing all the time, but later phases are only executed when committing to the main branch.

Before considering *Conditional Pipeline and Job Triggering*, at the pipeline level, encourage developers to use local scripts and tools to perform initial tests and checks prior to committing changes to the version control repository.

Example The following example uses different trigger conditions in a build, test, and deploy phase, expressed using Gitlab CI/CD rules. The build is triggered for changes only in the `config` and `src/core` directories. The tests are executed for the same directories, or if a commit to a main or release branch has been made. Deployment is only triggered for a commit to the main branch.

```
build:
  phase: build
  script:
    - echo "Building project..."
  rules:
    - changes:
      - config/**/*
      - src/core/**/*

test:
  phase: test
  script:
    - echo "Running tests..."
  rules:
    - changes:
      - config/**/*
      - src/core/**/*
    - if: '$CI_COMMIT_BRANCH == "main" ||
          $CI_COMMIT_BRANCH =~ /^release\/.*/'
```

```
deploy:
  phase: deploy
  script:
    - echo "Deploying..."
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
```

Consequences The application of the pattern *Conditional Pipeline and Job Triggering* presents the following consequences:

(+) Improved speed and efficiency by running only the necessary jobs, if any.

(−) Added complexity to the pipeline configuration, requiring careful rule management. Risk of misconfiguration may lead to critical jobs or phases being unintentionally skipped.

Related Patterns *Incremental Build* and *Selective Testing* complement *Conditional Pipeline and Job Triggering* as they perform further selections of builds and tests, respectively. Thus, these patterns can be applied in combination.

Known Uses Each repository we investigated applies this pattern, although the trigger rules they apply differ. We have found logic based on trigger types (e.g., commit push, merge request), file changes, branch name, commit message, and runners available. Table 5 shows in detail which project applies which rules.

Furthermore, each project uses job-level rules, which choose whether a job is executed or not, while 3 repositories (`inkscape`, `fdroidclient`, `veloren`) use workflow-level rules, which may prevent the entire pipeline from running.

Table 5: Summary of Rules Used by Repository

Project Name	Trigger Type	Branch Name	Files Changed	Runners Available	Commit Message
`inkscape`	X		X	X	
`openrgb`	X				
`fdroidclient`	X	X	X	X	
`veloren`	X	X	X		
`baserow`	X	X	X		X

4.8 Pattern: Comprehensive Pipeline Automation

Context Your teams run CI/CD pipelines, and humans perform some pipeline tasks.

Problem Manual processes in CI/CD pipelines are prone to human error, are time-consuming, and are inconsistent. These issues can lead to delays, failed deployments, and lower software quality. There is a need for a method to streamline and standardize these processes to improve the efficiency and reliability of the pipeline.

Forces

- **Performance vs Feasibility:** Human intervention in CI/CD pipelines can introduce significant delays, as it is neither immediate nor guaranteed to occur promptly, but eliminating manual steps entirely may not be feasible when human oversight is required for quality or compliance purposes.
- **Efficiency and Scalability vs Effectiveness:** Manual tasks depend on specific resources such as interactive environments and notification mechanisms, which complicate scaling as the number of projects and pipeline executions increases. Nevertheless, automating all such tasks may not be cost-effective for every scenario, particularly those requiring human validation.
- **Repeatability and Consistency vs Quality:** Automated processes typically provide more consistent and repeatable results compared to manual ones, but full automation may overlook context-sensitive issues or nuanced decisions that human operators are better equipped to handle.
- **Maintenance:** Automated tasks require ongoing maintenance of scripts and orchestration logic. However, retaining manual steps also entails maintaining manual testing environments and tools, adding complexity and redundancy, especially as automation remains an integral part of most pipelines.

Solution Aim to automate as many steps as possible in your CI/CD pipeline. This includes automating the build, test, deployment, and monitoring processes. Use tools and scripts to handle repetitive tasks, ensure consistency, and minimize human involvement in routine operations. Integrate checkpoints or human testing where human intervention is required for quality assurance.

Solution Details When implementing comprehensive automation within CI/CD pipelines, companies often use various tools and frameworks to optimize and improve their processes. Continuous integration tools such as `Jenkins` and `GitLab CI/CD` are often used to automate software development's build and test phases. With these tools, teams can automatically trigger builds and run tests when changes are made to the code base.

Infrastructure automation is another important component of the solution. Infrastructure-as-Code (IaC) tools [16] such as `Terraform` and `Ansible` facilitate the automated setup and management of deployment environments. By representing the infrastructure as code, these tools enable consistent and repeatable environment configurations at different deployment pipeline phases, reducing the risk of configuration mismatches and deployment errors.

Automated testing frameworks such as `JUnit` for unit testing and `Selenium` for UI testing ensure the software is tested consistently across different environments. These frameworks enable the execution of a wide range of tests, from unit tests to integration and end-to-end tests, without manual intervention, speeding up the testing process and improving reliability. However, not all kinds of user acceptance testing are easy to automate.

Despite the many benefits of extensive automation, it is crucial to recognize that human oversight is required at certain phases. Quality assurance reviews and user acceptance testing are examples of processes that require human judgment to ensure that the software not only functions correctly but also meets business requirements and user expectations. These human-led assessments can serve as critical checkpoints in the pipeline and ensure that the automated processes comply with company standards and regulatory requirements.

Example The pipeline shown in Figure 4 is an end-to-end fully automated pipeline. Contrary to that, the one depicted in Figure 1 contains a User Acceptance Test phase, which likely uses manual user acceptance testing steps. Developers could consider replacing this with automated UI tests using tools such as `Selenium`. They must consider how much effort would be required for this and whether the required quality level of human testing can be achieved by an automated solution. They could consider using *Conditional Pipeline and Job Triggering* or decision gates not to run the manual test phase for each pipeline run.

Consequences The application of the pattern *Comprehensive Pipeline Automation* presents the following consequences:

(+) Reduced human error and increased consistency and repeatability across pipeline executions.
(+) Accelerated delivery speed by minimizing delays introduced by manual tasks.
(+) Enhanced scalability by lowering the need for human intervention in routine processes.
(+) Improved traceability and compliance through codified and reproducible procedures.
(−) Automation logic requires ongoing maintenance and may become complex over time.
(−) Risk of over-reliance on automation can lead to blind spots in quality assurance if human checkpoints are underused or removed.

Related Patterns *Conditional Pipeline and Job Triggering* can help to achieve a compromise in *Comprehensive Pipeline Automation* by triggering human phases only for selected pipeline runs.

Most of the other patterns in this paper complement *Comprehensive Pipeline Automation* as they play a role in automation, for instance, by speeding up the automated solution, enhancing its consistency, or improving its reliability.

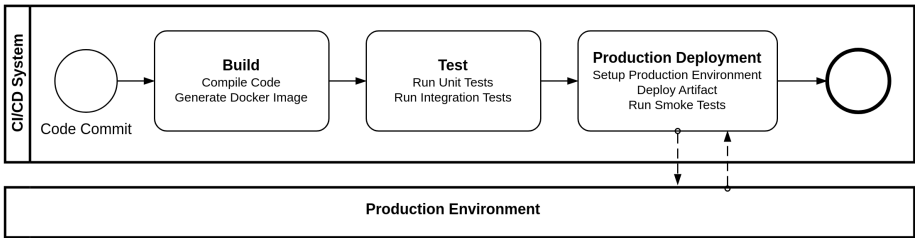

Fig. 4: Fully Automatic Pipeline Building, Testing and Deploying an Application

Known Uses We have chosen to look for full pipeline automation (i.e., each job runs automatically) to classify a repository as applying this pattern. The `when: manual`[30] option on GitLab determines whether a job runs manually. We have not excluded jobs that are optionally manually, i.e., jobs that run automatically but that can also be manually triggered. With this classification criteria, only `fdroidclient` and `veloren` apply the pattern.

A third example is `graphviz/graphviz`[31], a graph visualization tool.

4.9 Pattern: Mocked External Services

Context Connection with external services may be required to test an application thoroughly.

Problem Configuring external services (e.g., HTTP servers, databases, etc.) to be available while testing an application increases setup time, resource usage (memory and network), and overall execution time.

Forces

- **Performance vs Reliability and Maintenance:** The setup time for external services —such as downloading and installing dependencies—can constitute a significant portion of overall execution time during testing, but minimizing this setup may compromise the fidelity of the test environment or delay service availability. Furthermore, mockups add maintenance overhead as they must keep mimicking the services when these are updated.
- **Reliability:** Mockups typically require only source code and can eliminate the need for installing operating system-level libraries or full applications. Nevertheless, over-reliance on mockups may reduce test coverage for integration scenarios, potentially leading to undetected issues in real-world deployments.

[30] https://docs.gitlab.com/ee/ci/yaml/index.html#when: accessed May 2025
[31] https://gitlab.com/graphviz/graphviz/-/tree/90d36ca2: accessed May 2025

Solution Popular programming languages offer libraries to stub objects while testing. For each test that requires external services, create a mockup and configure it to behave as expected (e.g., returning a specific HTTP code to a request). Then, inject the mockup into the tested classes.

Solution Details Write the application source code to allow mockups to be introduced from tests. Depending on the programming language, this may be done through IoC (Inversion of Control) or with class constructors. Furthermore, define one single client class for the service, and connect only through this class. In the tests, define stubs that replace the service client objects. A mockup class inheriting from the client class may be a solution for simpler scenarios. A mocking library is preferable for more complex scenarios (e.g., an HTTP server) as it comes with many testing functionalities provided out of the box. Once the stubs are created, inject them into the application source code and run the tests.

Most often mockup tests are fully contained in the application source code, therefore in the pipeline it is enough to build the application and the tests, and run a test command. Nevertheless, it may be necessary to define environment variables in the pipeline job, to be used by the tests executed.

Although applying the pattern can improve the pipeline performance, making the mockup behavior as close as possible to the real service is important. Oversimplifications may increase the chances of undetected bugs in the application code. An example could be a database that returns a maximum of 100 rows per time while the configured mockups do not implement this limit.

Example A service is mocked and used in the application source code in the following listing. The service is configured to return true on the ping request. The test verifies that the application works without making HTTP ping requests.

```
public class TestWithMockups {
  @Test
  public void testUseServiceWithMockups() {
    Service service = mock(Service.class);
    when(service.ping()).thenReturn(true);
    Application app = new Application(service);
    assertTrue(app.run());
  }
}
```

Consequences The application of the pattern *Mocked External Services* presents the following consequences:

(+) Removal of the need for downloading, configuring, and launching external services, which can significantly reduce the total execution time of the test pipeline.

(+) Mockups reduce the variability associated with external dependencies (e.g., service downtime or network delays), leading to more stable and determin-

istic test executions. This can reduce test flakiness and improve confidence in automated pipelines.

(+) Developers can run the tests without the overhead of configuring external systems, making local development and debugging easier and more efficient.

(−) Mocked services may not fully replicate all the behaviors, limitations, or error conditions of real services, which can lead to gaps in test coverage and undetected integration issues in production.

(−) Mockups must be kept up to date as real service APIs evolve. Failure to update mocks may cause discrepancies between test and production environments, potentially introducing bugs that remain unnoticed until deployment.

Related Patterns Removing the configuration of services changes the nature of the tests, reducing possible dependencies between them. For this reason, *Mocked External Services* can often be combined with *Task Parallelization*.

Furthermore, since mockups may not behave exactly like the real services, it makes sense to use them in feature branches rather than in production ones. To achieve this functionality, the *Conditional Pipeline and Job Triggering* pattern can be used.

Known Uses We have found evidence of mocked objects for all 5 repositories; nevertheless, only 2 of them were mocking external services: `fdroidclient` mocked a database while `baserow` mocked a database, data providers, storages, and a mail server. In addition, we have found that `mudler/LocalAI` mocks an HTTP server and an ML model.

5 Conclusions

This paper has explored key performance patterns in CI/CD pipelines, focusing on their application to enhance modern software delivery systems' speed, efficiency and scalability. We demonstrated how these strategies can improve overall pipeline performance, reduce latency, and optimize resource utilization by identifying and analyzing fundamental patterns based on modularity, parallelism, and efficient dependency resolution. The case studies and examples from widely used CI/CD tools like `GitLab CI/CD` illustrated the practical impact of these patterns in real-world scenarios. Although our analysis mostly focused on `GitLab CI/CD`, the results are generalizable to other technologies such as `GitHub Actions`, `CircleCI`, etc.

We discussed the trade-offs in adopting these patterns, emphasizing the need to carefully consider project-specific requirements, infrastructure constraints, and team workflows. As the complexity of software development continues to grow, the demand for more efficient and resilient CI/CD pipelines will only increase. The patterns we have highlighted offer a robust foundation for addressing

performance challenges in this context, providing actionable insights for DevOps practitioners and software engineers seeking to optimize their pipelines.

Future research should focus on further refining these performance patterns and exploring their interaction with emerging technologies such as machine learning-driven optimization techniques. Moreover, investigating the long-term impacts of CI/CD performance improvements on software quality and team productivity could provide valuable insights for guiding future best practices and tool development. Ultimately, a deeper understanding of CI/CD performance patterns will help ensure that software systems continue to evolve in a fast, reliable, and sustainable manner.

Acknowledgments. This research was funded in whole or in part by the Austrian Science Fund (FWF) project CQ4CD, Grant-DOI: 10.55776/I6510. For open access purposes, the authors have applied a CC BY public copyright license to any author accepted manuscript version arising from this submission.

References

1. Abdalkareem, R., Mujahid, S., Shihab, E., Rilling, J.: Which commits can be ci skipped? IEEE Transactions on Software Engineering **47**(3), 448–463 (2019)
2. Almeida, F., Pinho, D., Aguiar, A.: Validating pattern languages: A systematic literature review. In: Proceedings of the 29th European Conference on Pattern Languages of Programs, People, and Practices. pp. 1–8 (2024)
3. Chen, L.: Continuous delivery: Huge benefits, but challenges too. IEEE software **32**(2), 50–54 (2015)
4. Corbin, J.M., Strauss, A.: Grounded theory research: Procedures, canons, and evaluative criteria. Qualitative sociology **13**(1), 3–21 (1990)
5. Da Gião, H., Flores, A., Pereira, R., Cunha, J.: Chronicles of ci/cd: A deep dive into its usage over time. arXiv preprint arXiv:2402.17588 (2024)
6. Duvall, P.M.: Continuous integration: Patterns and anti-patterns. DZone, Incorporated (2010)
7. Gallaba, K., Ewart, J., Junqueira, Y., Mcintosh, S.: Accelerating continuous integration by caching environments and inferring dependencies. IEEE Transactions on Software Engineering **48**(6), 2040–2052 (2020)
8. Gallaba, K., McIntosh, S.: Use and misuse of continuous integration features: An empirical study of projects that (mis) use travis ci. IEEE Transactions on Software Engineering **46**(1), 33–50 (2018)
9. Garousi, V., Felderer, M., Mäntylä, M.V.: Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. Information and software technology **106**, 101–121 (2019)
10. Garousi, V., Felderer, M., Mäntylä, M.V., Rainer, A.: Benefitting from the grey literature in software engineering research. In: Contemporary Empirical Methods in Software Engineering, pp. 385–413. Springer (2020)
11. Ghaleb, T.A., Da Costa, D.A., Zou, Y.: An empirical study of the long duration of continuous integration builds. Empirical Software Engineering **24**, 2102–2139 (2019)

12. Hentrich, C., Zdun, U., Hlupic, V., Dotsika, F.: An approach for pattern mining through grounded theory techniques and its applications to process-driven soa patterns. In: Proceedings of the 18th European Conference on Pattern Languages of Program. pp. 1–16 (2013)
13. Humble, J., Farley, D.: Continuous delivery: Reliable software releases through build. Test, and deployment automation. Pearson Education **1** (2010)
14. Jürgens, E., Pagano, D., Göb, A.: Test impact analysis: Detecting errors early despite large, long-running test suites. Whitepaper, CQSE GmbH (2018)
15. Machalica, M., Samylkin, A., Porth, M., Chandra, S.: Predictive test selection. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). pp. 91–100. IEEE (2019)
16. Morris, K.: Infrastructure as code. O'Reilly Media (2020)
17. Munaiah, N., Kroh, S., Cabrey, C., Nagappan, M.: Curating github for engineered software projects. Empirical Software Engineering **22**, 3219–3253 (2017)
18. MUSTYALA, A.: Ci/cd pipelines in kubernetes: Accelerating software development and deployment. EPH-International Journal of Science And Engineering **8**(3), 1–11 (2022)
19. Orso, A., Apiwattanapong, T., Law, J., Rothermel, G., Harrold, M.J.: An empirical comparison of dynamic impact analysis algorithms. In: Proceedings. 26th international conference on software engineering. pp. 491–500. IEEE (2004)
20. Parris, D.: How do the holidays impact engineering productivity? a statistical analysis. Blog (2022), https://www.statsignificant.com/p/how-do-the-holidays-impact-engineering
21. Riehle, D., Harutyunyan, N., Barcomb, A.: Pattern discovery and validation using scientific research methods. In: Transactions on Pattern Languages of Programming V, pp. 226–253. Springer (2025)
22. Shahin, M., Babar, M.A., Zhu, L.: Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. IEEE access **5**, 3909–3943 (2017)
23. Ståhl, D., Bosch, J.: Experienced benefits of continuous integration in industry software product development: A case study. In: The 12th iasted international conference on software engineering,(innsbruck, austria, 2013). pp. 736–743 (2013)
24. Thatikonda, V.K.: Beyond the buzz: A journey through ci/cd principles and best practices. European Journal of Theoretical and Applied Sciences **1**(5), 334–340 (2023)
25. Vasilescu, B., Yu, Y., Wang, H., Devanbu, P., Filkov, V.: Quality and productivity outcomes relating to continuous integration in github. In: Proceedings of the 2015 10th joint meeting on foundations of software engineering. pp. 805–816 (2015)
26. Vassallo, C., Proksch, S., Gall, H.C., Di Penta, M.: Automated reporting of anti-patterns and decay in continuous integration. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 105–115. IEEE (2019)
27. Yin, M., Kashiwa, Y., Gallaba, K., Alfadel, M., Kamei, Y., McIntosh, S.: Developer-applied accelerations in continuous integration: A detection approach and catalog of patterns. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. pp. 1655–1666 (2024)
28. Zampetti, F., Vassallo, C., Panichella, S., Canfora, G., Gall, H., Di Penta, M.: An empirical characterization of bad practices in continuous integration. Empirical Software Engineering **25**, 1095–1135 (2020)