Architectural Design Decisions and Best Practices for Fast and Efficient 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] Software Architecture Research Group, Faculty of Computer Science, University of Vienna, Vienna, Austria
firstname.lastname@univie.ac.at
[2] UniVie Doctoral School Computer Science DoCS, Faculty of Computer Science, University of Vienna, Vienna, Austria

Abstract. Continuous Integration/Deployment (CI/CD) pipelines are critical for integrating developer changes and maintaining high-quality software deployments. The increasing frequency of commits and deployments places significant demands on CI/CD systems, requiring improved speed and efficiency. While numerous tools and techniques have been proposed to increase the velocity of CI/CD pipelines, there is a notable gap in architectural guidance for developers on key design decisions and best practices. To address this, we conducted a grey literature review using Straussian Grounded Theory to develop a UML-based model to guide software architects and developers in their decision-making. Our research focuses on identifying architectural design decisions (ADDs) and best practices as decision options that improve the speed and efficiency of CI/CD pipelines. The study analyses 38 sources, building a formal model comprising 6 ADDs and 30 best practices. This work contributes a structured, architecturally guided approach to optimizing CI/CD systems.

Keywords: CI/CD · Software Architecture · Design Decisions · Speed · Efficiency

1 Introduction

Continuous Integration and Continuous Deployment (CI/CD) [7] pipelines enable seamless integration of code changes, enhancing feedback speed and ensuring high-quality releases [1]. Speed and efficiency are crucial in CI/CD systems, as slow pipelines delay the development of new features [10] and overall deployments. With frequent commits and rapid releases becoming standard, the performance of CI/CD pipelines faces increasing pressure.

Several studies have proposed tools and techniques to enhance CI/CD performance [4, 5, 11]. However, there is no comprehensive approach guiding developers through the key architectural design decisions, available design options, and their interrelations. This lack of guidance is problematic as CI/CD pipelines

and their surrounding systems have evolved into complex infrastructures critical for building, testing, and deploying software.

To address this, we conducted a grey literature (GL) study [6] using Straussian Grounded Theory (GT) [2] to formalize UML-based models guiding software architects in their decision-making. More formally, we investigated:

- **RQ1**: Which best practices are practitioners using to improve the speed and efficiency of CI/CD pipelines?
- **RQ2**: What are the relations between the identified best practices?
- **RQ3**: Which ADDs are related to improving performance in CI/CD systems, and how are the design options (i.e., the identified best practices) connected with the ADDs?

The primary contributions of this work are: **(1)** a grey literature study on CI/CD best practices for speed and efficiency, analyzing 38 sources in-depth; **(2)** a formal model containing 6 ADDs, and 30 best practices.

2 Methodology

Figure 1 summarizes the research approach used in this work. We applied Straussian Grounded Theory to analyze 38 grey literature[1] sources. GT is a systematic method that, through iterative analysis, enables theory discovery from empirical data. Stol et al. [8] discussed its application in Software Engineering and proposed domain-specific guidelines.

We used popular search engines (e.g., Google, Bing, DuckDuckGo) to collect sources, starting with queries like "speeding up CI/CD pipelines" and "CI/CD pipelines performance." Following GT's *theoretical sampling*, additional GL was added based on analysis of previous sources. We also employed *backward snowballing*—examining references in already selected sources. Literature collection ended upon reaching *theoretical saturation*, i.e., when new sources stopped contributing to the developed model. We excluded sources deemed irrelevant (e.g., only discussing CI/CD benefits/challenges), of poor quality (e.g., lacking discussion), or heavily promotional. Sources that only briefly mentioned their product were retained. All the sources can be found in the replication package [9].

In line with Straussian GT, our analysis involved *open*, *axial*, and *selective coding*, all performed manually. In open coding, data fragments (e.g., sentences) were assigned concepts. Axial coding linked these concepts. Selective coding then grouped them under a central category. While coding, we applied a key GT technique: *memoing*—writing notes/sketches to clarify concepts, categories, and relationships, and to support theory development. Memos aid in the GT practice of *constant comparison*—comparing existing and new data to refine theory. Since GT analysis begins before data collection ends, we analyzed sources as we found them. All memos can be found in the replication package [9].

To model the ADDs, we used the meta-model in the replication package [9]. Each design decision can involve multiple design option combinations. We highlighted especially beneficial ones using the *«can be combined with»* relation.

[1] Grey literature [6] includes practitioner books, videos, blogs, presentations, etc.

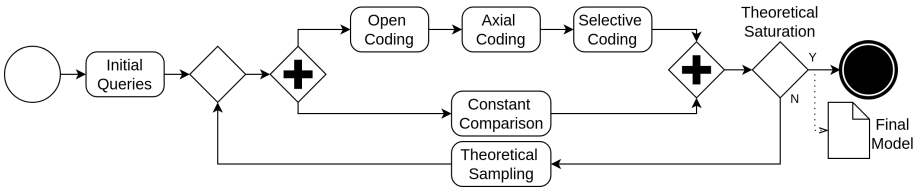

Fig. 1. The methodology applied in our paper.

Best Practice	Number of Mentions	Addresses Speed	Addresses Efficiency
Shared Across ADDs			
Pipeline Observability	20	✔	✔
ADD: Tasks Reduction			
Pipeline Asset Caching	22	✔	✔
Conditional Pipeline and Job Triggering	17	✔	✔
Interruptible Pipelines	9	✔	✔
Selective Testing	8	✔	✔
Incremental Build	5	✔	✔
Minimal Image Design	4	✔	✔
Mocked External Services	4	✔	✔
Asset Retention Policy	4		✔
Single Build	3		✔
ADD: Tasks and Resources Organization			
Task Parallelization	28	✔	
Task Splitting	16	✔	
Custom Runner Classes	13	✔	✔
CI/CD Architecture Scaling Strategy	11	✔	✔
Automatic Task Splitting	9	✔	
Manual Task Splitting	8	✔	
Pipeline Test Ordering	8		✔
Task Merging	3	✔	✔
ADDs: Conditional Triggering			
File Changes Condition	10	✔	✔
Job-Level Condition	9	✔	✔
Pipeline-Level Condition	9	✔	✔
Branch Condition	8	✔	✔
Trigger Type Condition	6	✔	✔
Commit Message Condition	3	✔	✔
ADD: Task Parallelization			
Intra-Pipeline Parallelism	24	✔	
Inter-Jobs Parallelism	19	✔	
Independent Jobs	13	✔	
Job Matrix	10	✔	
Inter-Pipelines Parallelism	5	✔	
Intra-Job Parallelism	4	✔	

Table 1. The best practices presented in this work, grouped by ADD and sorted by the number of mentions across grey sources.

3 Architectural Design Decisions

This section presents the study results as an Architectural Design Decision model. Table 1 overviews all the decision options, as well as how many sources reference them. A detailed mapping (e.g., the degree to which a source mentions a practice) is included in the replication package [9]. Given the space restrictions, we included only one model's view in Figure 2, leaving all the other views in the replication package [9].

ADD: Tasks Reduction. Improving the speed and efficiency of a CI/CD pipeline can be achieved by reducing the number of tasks, without compromising its core purpose, such as building, testing, and deploying applications. Figure 2 shows all related options. Notably, monitoring the pipeline, as promoted by the *Pipeline Observability* best practice, supports task reduction by revealing performance issues and bottlenecks.

One key option for this ADD is *Pipeline Asset Caching* [4], where assets (e.g., dependencies, build artifacts) are cached for reuse within the current and future pipeline runs. This reduces the need to re-download assets from external services or re-compute them locally (e.g., binary packages), thus saving time and resources. However, cache management (e.g., restoring or validating cache data) has a performance overhead. Among other things, defining a *Retention Strategy* is crucial to choosing how long assets are kept before deletion. Three other options depend on cached assets: *Single Build*, *Incremental Build*, and *Selective Testing*. In the build phase, *Single Build* improves efficiency by building the application once, caching the result, and reusing it in subsequent jobs instead of rebuilding. *Incremental Build* further improves performance by saving build results and rebuilding only components (and their dependencies) affected by a commit—especially useful with frequent commits. Similarly, *Selective Testing* [5] reduces testing time by running tests only on files impacted by a commit, often leveraging cached code coverage data. Given the importance of testing for ensuring correctness, reliability, and performance, skipping tests may be an anti-pattern if the version reaches production. The best practice *Conditional Pipeline and Job Triggering* can help avoid this by running the pipeline or specific jobs based on custom conditions, adding flexibility. In addition to *Selective Testing*, *Interruptible Pipelines* can also benefit from *Conditional Triggering*. *Interruptible Pipelines* suggests stopping a pipeline if a new commit is pushed to the same branch, reducing workload. One example combines these: run all tests without interruption in the `main` or `master` branches, but run change-affected tests and allow interruptions on others.

The seven design options discussed so far can be applied to any existing pipeline, with many possible combinations. In contrast, two additional options may not always be applicable: *Mocked External Services* and *Minimal Image Design*. The former involves using mock-ups instead of real tools or services during tests, saving time and resources. The latter reduces container images to the essentials (e.g., via multi-stage builds), lowering the time and resources needed to build and pull images in subsequent jobs.

ADD: Tasks and Resources Organization. Organizing tasks and resources effectively can accelerate a CI/CD pipeline. This ADD includes eight design options: two focused on resources (*Custom Runner Classes* and *CI/CD Architecture Scaling Strategy*), five on tasks (*Pipeline Task Ordering*, *Task Merging*, *Task Splitting*, and its two variants), and one on both (*Task Parallelization*). *Custom Runner Classes* suggests using runner types with different specs (e.g., memory, CPU) for tasks with distinct resource needs. *CI/CD Architecture Scaling Strategy* adjusts system resources by scaling them based on workload.

Task Parallelization boosts throughput by increasing task concurrency. More on it is detailed in the corresponding ADD, presented below. *Task Parallelization* is especially effective when paired with *Task Splitting*, the practice of dividing large tasks—manually or automatically—into smaller jobs or workflows. This option depends on parallelization to enhance performance. Note that each executed job has a starting cost (e.g., loading a container image). For this reason,

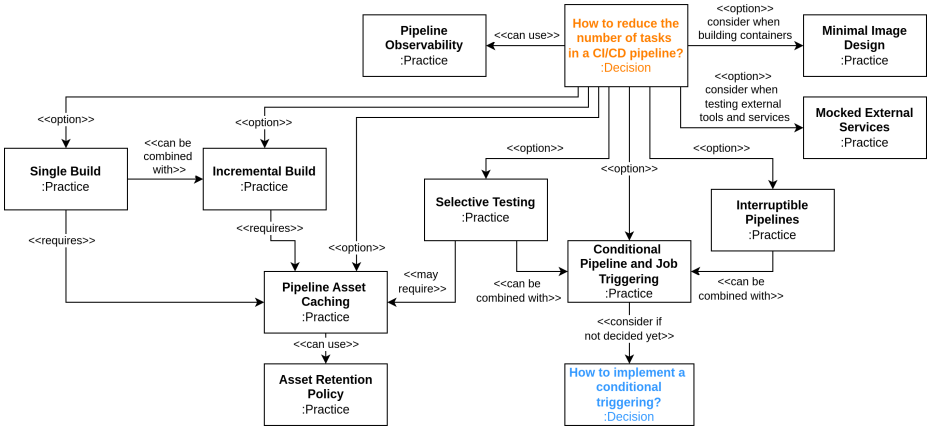

Fig. 2. Decision for reducing the number of tasks in a CI/CD pipeline.

when too many small jobs are present, *Task Merging* should be applied to combine them. Finally, *Pipeline Test Ordering* aims to reduce wasted computation (if the pipeline fails) by sequencing tests wisely: running resource-intensive ones (e.g., end-to-end tests) only after initial checks (e.g., linting, unit tests).

ADDs: Conditional Triggering. We previously introduced *Conditional Pipeline and Job Triggering* as a way to reduce tasks in a CI/CD pipeline. In the grey literature, we found several applications of this practice and identified three ADDs with six decision options. Architects must first decide where to apply conditions: at the *pipeline level* or *job level*. The former determines whether to trigger the entire pipeline, while the latter targets specific jobs, offering greater flexibility. Both approaches were common in practitioner sources and can be used together. After selecting the condition granularity, the next step is choosing the triggering rules. The most frequent is based on committed *File Changes*, but *Trigger Type* and *Branch* are also widely used. Finally, *Commit Message* content can serve as a trigger, though this rule is prone to human error (e.g., typos, forgetfulness) and should be applied with caution.

ADD: Task Parallelization. The most frequently mentioned practice in the examined sources is *Task Parallelization*. Thus, we modeled an ADD on how to apply it. We identified six design options, four of which are variants. Parallelism can be achieved by running multiple pipelines (*Inter-Pipeline Parallelism*) or within a single pipeline (*Intra-Pipeline Parallelism*). The latter is more widely cited in the grey literature, likely because not all CI/CD tools support defining multiple workflows per repository. Within a pipeline, parallelism can occur inside a job using multiple threads (*Intra-Job Parallelism*) or by executing several jobs concurrently (*Inter-Jobs Parallelism*). The latter includes two variants: *Independent Jobs* and *Job Matrix*. *Independent Jobs* are widely used, as CI/CD systems automatically run jobs in parallel if no explicit dependencies exist. This practice can also support *Manual Task Splitting*. *Job Matrixes* enable automatic

task distribution across workers and offer better maintainability than *Independent Jobs*, though they are unsuitable when tasks are highly interdependent or require strict sequencing.

4 Discussion

RQ1. Table 1 summarizes 30 best practices for improving CI/CD pipeline speed and efficiency, identified through analysis of 38 practitioner sources. These practices span various areas of CI/CD pipelines, including task parallelization, asset caching, and job/workflow triggering. Nine practices are particularly frequent, mentioned in at least one-third of the sources: *Task Parallelization*, *Pipeline Asset Caching*, *Pipeline Observability*, *Conditional Pipeline and Job Triggering*, *Task Splitting*, *Custom Runner Classes*, and three more related to *Task Parallelization*. Notably, several of the practices present multiple variants.

RQ2. We found several connections between best practices, including: *«require»* from *Incremental Build* to *Pipeline Asset Caching*, *«can use»* from *Pipeline Asset Caching* to *Asset Retention Policy*, *«has variant»* from *Inter-Jobs Parallelism* to *Independent Jobs*, and *«can be combined with»* from *Job Matrix* to *Automatic Task Splitting*. The *«can be combined with»* relation reflects only the most effective combinations, not all possibilities. A key insight is the central role of *Pipeline Observability*. While it does not directly improve speed or efficiency, it enables the adoption of many other practices. This was noted in over half (20 of 38) of the grey literature sources.

RQ3. We identified 6 architectural design decisions, each associated with multiple best practices. Not all options are universally applicable; for instance, *Minimal Image Design* can reduce enacted tasks only when containers are built in the pipeline. Furthermore, as highlighted for RQ2, some design options depend on others to function or are more effective when combined. Finally, we also discovered a *«can use»* relation linking the ADDs to *Pipeline Observability*.

5 Threats to Validity

Construct Validity. The construct validity of this study is influenced by the use of GL and GT, both of which pose potential risks. Grey literature varies in quality and credibility, but prior research supports its value in capturing practitioner perspectives [6]. We addressed issues by systematically selecting and analyzing 38 sources. For Grounded Theory, threats stem from the complexity and potential inconsistencies in its application. Stol et al. [8] identify essential steps—coding, memoing, and theoretical saturation—which we rigorously followed to ensure methodological rigor.

Internal Validity. This study faces two main threats: selection bias in GL and researcher bias in modeling ADDs. Selection bias could cause the omission of relevant sources. We mitigated this by using theoretical saturation as the stopping criterion, halting model development only when new sources added no design options. As shown in the replication package [9], no new options appeared

after S11, yet 27 additional sources were included for robustness. Design options were included only if mentioned in at least three sources—all but three options met this after analyzing half of the sources. More details are in the replication package. Researcher bias in modeling ADDs is another threat. While unavoidable, we minimized it through independent reviews of each ADD.

External Validity. A key threat to external validity is generalizability. To address this, we analyzed both technology-agnostic and technology-specific sources (spanning seven CI/CD tools), as detailed in the replication package [9]. This diversity enhances external validity, though applicability to less conventional setups may still be limited.

6 Related Work

Certain best practices presented in our work have already been analyzed in other studies. Gallaba et al. [4] proposed a framework to automatically cache dependencies across pipeline runs based on system calls. Memon et al. [5] presented a technique to apply *Selective Testing* by reversing the dependencies of the files modified in a commit. Although these works propose tools to help use the studied practice, their application is limited to that single practice.

Other works have studied more than one practice proposed in our study. Yin et al. [11] have cataloged practices to reduce tasks in pipelines, mining 7795 open-source repositories. Their findings present several practices related to *Conditional Pipeline and Job Triggering*. Contrary to our work, their paper focuses on one specific technology and provides only one class of practices. Additionally, Duvall et al. [3] mentioned in their book on Continuous Integration some techniques to improve the pipelines, such as: *Task Parallelization*, *Mocked External Services*, *Pipeline Observability*, *Pipeline Test Ordering*.

None of these works proposes formal models incorporating design decisions, decision options, and their relationships regarding speed and efficiency for CI/CD pipelines. To the best of our knowledge, no current work does that. In addition, despite the benefits of GL to analyze practitioners' views [6], none of the listed studies systematically utilized these knowledge sources.

7 Conclusions and Future Work

In this study, we used Straussian GT to analyze 38 GL sources focused on improving CI/CD pipeline speed and efficiency. Our findings are summarized in a model comprising ADDs, decision options, and their relationships. We aimed to answer three research questions. In RQ1, we identified 30 best practices for enhancing CI/CD workflows. For RQ2, we linked these practices, specifying possible, required, and variant combinations. In RQ3, we uncovered 6 ADDs along with their corresponding design options. Practitioners can apply the proposed model to better understand CI/CD pipeline best practices for performance or to revise existing architectures. Finally, our model can serve as a starting point for future extensions and validations of the proposed best practices.

Data Availability

All artifacts produced in this study are available in the replication package [9], including the memos from the coding phase and tables summarizing our findings. We used the *Wayback Machine* to archive each grey source indefinitely and included both the original and archived links.

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. Chen, L.: Continuous delivery: Huge benefits, but challenges too. IEEE software **32**(2), 50–54 (2015)
2. Corbin, J., Strauss, A.: Basics of qualitative research: Techniques and procedures for developing grounded theory. Sage publications (2014)
3. Duvall, P.M., Matyas, S., Glover, A.: Continuous integration: improving software quality and reducing risk. Pearson Education (2007)
4. 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)
5. Memon, A., Gao, Z., Nguyen, B., Dhanda, S., Nickell, E., Siemborski, R., Micco, J.: Taming google-scale continuous testing. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). pp. 233–242. IEEE (2017)
6. Rainer, A., Williams, A.: Using blog-like documents to investigate software practice: Benefits, challenges, and research directions. Journal of Software: Evolution and Process **31**(11), e2197 (2019)
7. 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)
8. Stol, K.J., Ralph, P., Fitzgerald, B.: Grounded theory in software engineering research: a critical review and guidelines. In: Proceedings of the 38th International conference on software engineering. pp. 120–131 (2016)
9. Urdih, F., Theodoropoulos, T., Zdun, U.: ADDs and best practices for fast and efficient CI/CD pipelines. https://doi.org/10.5281/zenodo.15639753 (2025)
10. Widder, D.G., Hilton, M., Kästner, C., Vasilescu, B.: A conceptual replication of continuous integration pain points in the context of travis ci. In: Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering. pp. 647–658 (2019)
11. 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)