Approximated Computing for Build Jobs in Continuous Integration

A Catchy Optional Subtitle that Grabs the Attention

MSc Thesis

Natália Struharová

Delft University of Technology

TU Delft

Approximated Computing for Build Jobs in Continuous Integration

A Catchy Optional Subtitle that Grabs the Attention

by

Natália Struharová

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Thursday 23rd of May, 2024 at 9:30 AM.

Student number:	4935519	
Project duration:	September 15, 2023 – May 23, 2024	
Thesis committee:	Prof. dr. A. van Deursen,	TU Delft
	L. Miranda da Cruz	TU Delft
	June Sallou	TU Delft
	Jana Webber	TU Delft
Supervision:	Prof. dr. A. van Deursen,	TU Delft
	L. Miranda da Cruz	TU Delft
	June Sallou	TU Delft

This thesis is confidential and cannot be made public until December 31, 2024.

Cover: Canadarm 2 Robotic Arm Grapples SpaceX Dragon by NASA under CC BY-NC 2.0 (Modified)

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

Continuous Integration (CI) has become a cornerstone of modern software development, gaining widespread adoption due to its ability to facilitate frequent and dependable code integration. However, its benefits are offset by high computational costs and energy consumption, particularly in the build phase. With its growing popularity, it is crucial to reflect on the efficiency of the CI process. This thesis proposes a novel framework to optimise energy consumption in the build jobs of CI pipelines, with primary focus on minimising compilation workload. Leveraging static dependency analysis and commit information, the framework introduces guided partial compilation, targeting only files affected by changes. The results demonstrate its ability to maintain CI reliability while significantly reducing energy consumption in real-world projects, with a 22% reduction of energy consumption in compilation-only experiments, and up to 63% energy savings in experiments that extrapolate the effects of partial compilation across the rest of the build job. The contributions in this research offer a stepping stone toward the imperative establishment of sustainable standards within the CI practice.

keywords: Sustainable Continuous Integration (CI), Build Job, Guided Partial Compilation

Preface

I would like to start by expressing my gratitude to Dagmar and Vlado, my parents, who not only made it possible for me to pursue higher education, but also gave me the space to explore new things and make my own decisions along the way. It gave me a sense of empowerment that I hope to keep throughout the next stages of my life, and for which I will be eternally grateful to both of you. I also want to thank my sisters, Naďa, Saša and Hana, who have been my cheerleaders and emotional support ever since my first years at TU Delft.

Next I want to thank my friends, Dan Andrescu, Ion Babalau, Orestis Kanaris, Marko Matušovič, Laura Muntenaar, Dan Plamadau, Radu Rebeja, Mariana Samardzic, Ioana Savu and Tamara Trubačová. All of you have made it easier for me to achieve this with your constant support, which I am immensely grateful for, just as I am grateful to have you all in my life.

I would like to thank June Sallou for being an excellent supervisor to me. Besides being a great research mentor, she taught me to be more confident in the decisions I make and have a healthier relationship with the work I do. Both on a personal and a professional level, I am very grateful for being able to work with her on this project. I would also like to thank Luís Cruz for introducing me to sustainable software engineering through his great Master course, and for continually helping to shape this Master thesis with his knowledge and insights. Finally, I would like to thank Arie van Deursen for advising on this thesis and making the time to be a part of this process.

Natália Struharová
Delft, May 2024

Contents

1

Introduction

This chapter lays the groundwork for the thesis, establishing the context by introducing the background of continuous integration (CI) practices and the associated challenges. It then presents the problem statement, articulates the research questions, outlines the methodology, and elucidates the contributions of the thesis. Finally, the chapter concludes by providing an overview of the subsequent chapters, setting the stage for the exploration of CI build job optimisation.

1.1. Context

Continuous integration (CI) is a modern software development practice that allows frequent yet reliable code contributions to be merged into the code base. Through applying developer-defined checks to new commits, CI ensures that a predetermined minimum quality standard is always maintained in the latest version of application code. Techniques used in the practice of CI, such as self-testing and general automation of builds have been shown to directly improve the overall level of software quality [8]. One of the greatest additional benefits of CI to developers come from the efficiency it brings to development by enabling teams to release twice as fast on average [16]. These features together with the ability to automate these checks make CI an increasingly popular practice in the software development community, proved by the wide-spread adoption of the practice by major tech companies such as Google or Mozilla [20].

However, the benefits of CI are currently being counteracted by significant drawbacks. Particularly, these drawbacks most often manifest as high cost of computational resources and long waiting time for the CI pipeline feedback.
As the code base grows, so does the number of tasks in building and testing in CI pipelines, making the execution more computationally expensive [8]. For example, Google's TAP (Tools for Developers), which is the CI system used by all Google sub-projects, costs millions of dollars for computation only, which excludes the cost of developers who work on and maintain TAP [16]. Another commercial example is Mozilla, which estimates the cost of CI to be greater than $200,000 per month [26].
From the practical standpoint, another major problem is the time-consuming nature of the CI pipeline. An extensive study that reviewed over a one and a half million of builds has found that a build job can take up to 83 minutes to run on a new commit [16]. Developers often have to wait for the result of the CI pipeline executed on their new code contribution to be able to work with the most up-to-date code version for their next task. In case the commit does not pass the CI pipeline, they must also introduce new fixes and rerun the pipeline for the amended code. For many of them, the lengthy feedback loop presents a substantial limitation for using CI in their everyday work [28].

Both of these drawbacks, the monetary costs and the long execution times, together called for ideas on how to improve the efficiency of CI. In response to this demand, research has produced multiple different strategies of minimising the workload that had to be executed in the pipeline. Some existing research has focused on identifying the patterns of commits where developers manually skipped the CI pipeline execution, and used them to create heuristics that determine commits for which the CI

execution can be safely skipped [2][1]. Other research has been exploring the dependencies between different tasks of the CI pipeline and introducing parallel computing to situations in which it can be applied without losing the quality of the result [9]. The existing studies have addressed the costs and the time-consuming nature of CI. However, the solutions also come with limitations. One of them is the coarse granularity of operation, where the solutions either skip the CI execution or run it in its entirety, without attempting workload reduction in each commit. Another one is that while the costs and duration may be reduced by approaches such as parallelisation, such solutions do not target the reduction of another strongly correlated aspect - the energy consumption.

While it is directly linked to high costs and slow feedback loops, there is little attention dedicated to measuring the energy output of CI pipelines or efforts to investigate possible improvements in energy-related terms. Besides the time- and cost-efficiency of CI, a major external motivation for examining the energy consumption is the high and rapidly-growing demand for energy in data centres, on which tech companies often delegate responsibilities such as running of their CI pipelines. Based on the IEA report from 2023 on the worldwide energy consumption, the energy demand of data centres and transmission networks accounted for up to 1.5% of the electricity used globally [17]. According to the report, despite efficiency improvements in the recent years, the workloads handled by data centres have been growing rapidly over the past years, ranging between 20% to 40% of growth annually. The organisation itself has attributed most of this growth to tech companies, as it has recorded that the combined energy demand of Amazon, Microsoft, Google and Meta has more than doubled between 2017 and 2021. In terms of the goals to get on track to establish net zero CO_2 for the global energy sector by 2050, the IEA itself marked the data centre sector as one that requires more efforts in improving its energy efficiency, further highlighting the necessity of investigating the possible improvements in the workload handled by these centres, such as CI.

One of the few works related to energy consumption in CI pipelines published by Limbrunner in 2023 provides a comprehensive analysis of usage of different types of jobs in the CI pipeline as well as their respective energy consumption figures [22].

The study has found that out of all job categories, the build job, which is responsible for compiling and subsequent construction of application code, consumes 37184 J (joules) per job on average. Out of all the job categories included in this study, the build jobs consume the most energy during pipeline execution. The individual steps of the build job, which are smaller units of workload the job is comprised of, were observed to be the most energetically demanding out of all other steps. The energy required to execute a step of a build job amounts to 3903.80 J on average. For perspective, this accounts for over 45% of the energy consumption of all categories of steps, with the second highest step energy requirement, that of the test job, being measured at a drastically lower average of 657.43 J. The findings are visually represented in a bar chart shown in Figure 1.1, highlighting the substantially higher energy consumption associated with build job steps compared to other job categories.

In a separate study, an analysis of repositories using TravisCI showed that build jobs are the second most commonly used within their respective CI pipelines [7]. Given the widespread adoption of build jobs in CI pipelines alongside their notable relative energy consumption, the category of build jobs makes for a compelling field for research aimed at enhancing its efficiency. Targeting the build phase of the pipeline for optimisations has the potential to bring significant reductions to the entire pipeline, all the while forming the path to a more sustainable standard of CI.

1.2. Problem Statement

Given the current state of research and existing data on energy efficiency of CI pipelines, the underlying problem can be extracted into a more concise formulation:

1. **Popularity of CI utilisation is high and rising, however, it is a highly energetically demanding process**: CI is already a popular tool in development, and it is projected to be adopted even more often in the future [16]. The rising trend of adoption further exacerbates the situation in terms of energy consumption, as the current power demands of data centres are already problematically high and therefore in an urgent need of efficiency improvements.
2. **The CI optimisation research field is lacking in energy-focused improvements**: While there are some recent works that have focused on measuring the energy consumption of jobs and steps

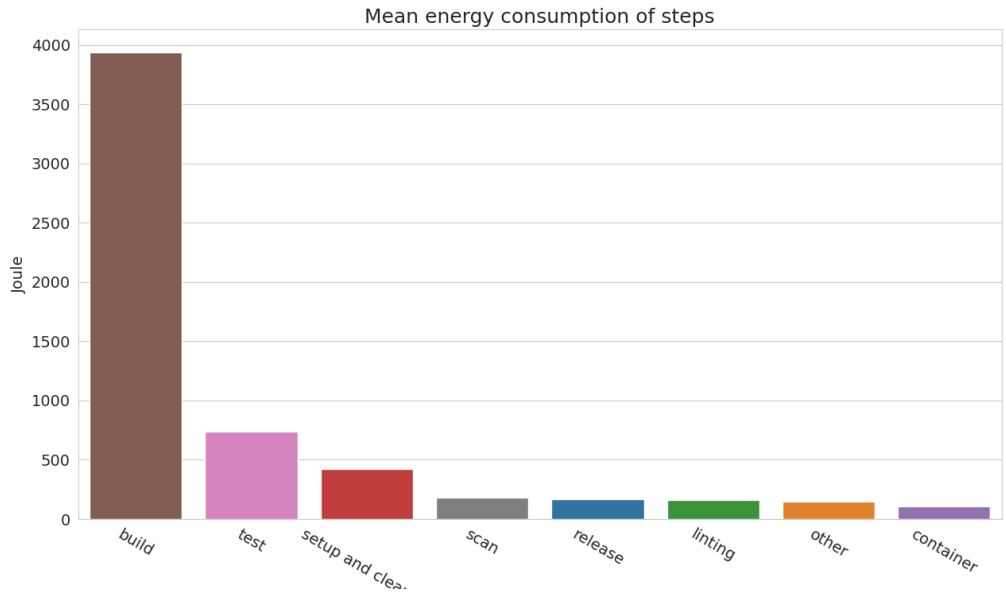


Figure 1.1: The mean energy consumption for steps in each CI job category (based on Limbrunner's research [22])

involved in CI, there is limited investigation into possible improvements of its energy efficiency. One problem is that most existing research has focused on efficiency improvements, but rather than investigating the energy-related optimisation, the work has focused on improving the speed reducing the workload done in the pipeline. However, as has been proven by existing research, the correlation between execution time is not directly correlated with energy consumption [3]. Therefore, to understand the energy output and efficiency of the CI pipeline, we must directly measure how much energy does the process consume.

3. **Most existing CI-optimising tools operate with coarse granularity**: Research into optimisations of CI has given rise to some tools that reduce the workload of CI pipelines, however, majority of past solutions are built to it either skips the execution of the pipeline all-together or execute it in its entirety depending on the nature of the associated commit. Considering that there are ways to optimise jobs and steps of the pipeline, the solutions that run the whole pipeline instead of skipping it may still be executing redundant workload. Therefore, we propose that the opportunities of optimising different jobs of the pipeline are explored, such that every CI pipeline is executed with higher efficiency.
4. **One of the most substantial contributors of energy consumption within the CI pipeline is the build job category**: The existing literature shows that the build jobs are a significant contributor to the overall energy requirements of the CI process overall. A highly energy-demanding yet commonly used category of jobs such as the build category presents a problem which must be addressed in order to bring efficiency improvements into the CI process. Given that it is one of the most energy-consuming jobs, optimising builds is likely to show greater improvements in the overall energy consumption compared to other jobs in the pipeline.

1.3. Solution Proposal

To fill the existing gaps, it is crucial to seek ways of keeping the benefits of CI while reducing its energy consumption, and in doing so, reducing its impact on the environment. To address the urgent problems presented above, this thesis proposes a solution in form of a framework that optimises the energy consumption of the build phase of CI pipelines, particularly the compilation. The central idea of the solution is applying approximate computing in the form of partial compilation. Partial compilation refers to the act of compiling a subset of the entire set of source files in the project. As a result, when applied in the pipeline, the compilation of some files is skipped, and thus the workload done is reduced,

leading to potential energy efficiency improvements. Every individual commit is a candidate for partial compilation. Therefore, the approach can potentially bringing improvement in energy efficiency in every individual CI pipeline run, increasing the granularity in workload selection compared to existing solutions. Finally, the framework focuses on reduction of workload in the build phase by employing partial compilation in the compiling stage. By targeting one of the primary contributors to overall CI energy consumption, it paves the way for substantial energy savings. Through this approach, we aim not only to enhance the efficiency of CI processes but also to contribute towards a more sustainable and environmentally conscious software development paradigm.

With the aim of maintaining reliability of the pipeline's results, the solution employs two main sources information to construct the partial compilation sequence. Inspired by an existing, proven solution on test selection in CI pipelines, it uses statically-extracted dependency relations between different parts of the code base together with the information on changes made within the related commit. Having shown effectiveness in test selection in choosing a relevant subset of tests to be executed, this thesis adopts the underlying mechanism to guide the partial compilation by selecting relevant files in a similar manner.

1.4. Research Questions

In formal terms, the core focus of this thesis is summarised in the following research question and its sub-questions:

1. Can static dependency analysis and commit information be leveraged to produce guided partial compilation in a CI pipeline build job in order to improve its energy efficiency?
    a) Does the produced partial compilation sequence catch the breaks in the code as reliably as the full compilation sequence?
    b) To what extent can the guided partial compilation improve the energy efficiency of the CI pipeline in real-world projects?

The main objective of the research is to study the partial compilation guided by commit changes and static dependency analysis, specifically applied in the context of build jobs in CI. There are two attributes that this thesis explores in detail, targeted by the two sub-questions.

The first sub-question investigates the ability of this approach in maintaining reliability of the CI results. For developers to be able to eventually use this approach in practice, it is important to know if the validity of CI checks can be maintained despite conducting only the partial compilation guided by dependencies and changes.

The second sub-question explores the capacity of the approach to bring tangible improvements in the energy efficiency of build jobs. By studying the effects this approach has on reducing the energetic impact of the pipelines utilised in real-world projects, we illuminate the potential power of this partial compilation approach in improving the general sustainability of the CI process.

1.5. Contributions

This thesis report encapsulates the exploratory study of CI build job optimisations using guided partial compilation. With that, during this study, several contributions have been produced. We use this section to briefly explain each major contribution.

1.5.1. Build-optimising Framework for CI

The main contribution of this thesis is the CI build-optimising framework that aims to improve the energy efficiency of the build job in the pipeline. The framework uses the commit information and dependency analysis to produce a partial compilation sequence, targeting only the files that could have been affected by the changes made in the commit. The framework is a proof of concept for this specific approach rather than being a guaranteed optimal solution. With our experiments, we show that the technique used by the framework captures the build breaks in all use cases in the defined scope. The framework has been also been shown to successfully reduce the total energy consumption of the build job, proving that the approach has the potential to increase the energy efficiency of its execution.

As has been mentioned earlier in the introduction, existing solutions aim to optimise the CI pipeline

execution in different manners. The framework, which is our proposed solution that addresses the same problems, features its own approach, different in several ways. One of the significant contributions is its **fine granularity** in operation - by targeting the build job within each CI pipeline, the framework can potentially bring an energy efficiency improvement to each CI run. Another distinct feature of the framework is its **focus on the nature of changes** made within a commit. The framework is sensitive to the changes made in the code, basing its partial compilation decisions on the type and location of the change made. Different code changes can cause different effects, therefore understanding the changes made in context of the code structure helps the framework identify the relevant files that could be potentially affected by a given change. Finally, the framework is aware of dependencies on a method- and field-level, such that if a method is changed, only the files dependent on that particular method will be marked for compilation rather than all files that are dependent on the changed file. The **high specificity of dependencies** helps narrow down the extent of the changes made in the commit compared to a coarser granularity which may cause unnecessary compilation files that are loosely related, but could not have been affected by the change.

The framework is written in Python, and has been made open-source such that it can be used, tested and enhanced further by the online software developer community. The future vision for the framework is that the proof of concept will be further studied and developed such that it can be used in practice by developers in their projects' respective CI to improve its energy efficiency.

1.5.2. Energy-measuring Tool for CI execution

Another contribution of this thesis is the tool we created to conduct experiments in which we measure the energy consumption and efficiency of the CI pipeline execution. While the tool has been developed primarily for experimentation, it can be used in practice by developers to gain insights into the energy efficiency of CI used in their own projects. Given the commit unique IDs (SHA), the framework automatically identifies the commits in the chosen repository and runs the CI pipeline for all the commits. The tool can be configured to conduct a custom number of reruns of the pipelines in a pseudo-random order. After each pipeline run, the build- and energy-related data. These results can then be reviewed by developers for better insights on the energy efficiency of their custom CI pipelines.

Similarly to the build-optimising framework, this tool is written in Python and is an open-source project as well, accessible to the public for utilisation and enhancement.

1.5.3. Enhancements to Energy-measuring Plugin

During our experiments, we used an open-source energy-estimating tool called EcoCI [15]. However, to ensure that the tool fulfils the needs of our study to the greatest extent possible, we made the following amendments to the EcoCI code base.

- **Adding a new CPU model**: The central processing unit is a major determinant of the estimation. To ensure that we get the most accurate estimation of energy consumption, we extended the list of supported CPU models added the CPU model of the hardware used to run the experiments in this study. Particularly, we added support for AMD Ryzen 9 7900X.
- **Adding duration into reporting**: The EcoCI estimation tool also measures the duration of the measured job or step execution in the CI pipeline. However, during preliminary research, we have noticed that the duration is not included in the JSON report on energy-related metrics. To ensure we could also collect the time data, we included the duration in seconds to be extracted and stored in the file with the other metrics.

1.6. Thesis Overview

The remainder of this thesis work is structured in the following way: Chapter 2 explains the fundamental concepts that the proposed solution relies on. Chapter 3 follows with an in-depth analysis of the related work in terms of existing efforts in optimising CI. The subsequent Chapter 4 explains the approach which this thesis employs in order to address the identified challenges and answer the research questions. Chapter 5 follows the methodology by explaining the solution from a technical standpoint. In Chapter 6, the results of experiments that validate the proposed solution are presented and analysed. Chapter 7 discusses the findings and their implications. Finally, Chapter 8 raises ideas for future refinement work

and concludes the research covered by this thesis.

2
Background

This chapter explains the concepts essential to understand the remainder of this thesis. All concepts and tools are presented in detail to help the reader understand the design of the solution.

2.1. Fundamental concepts

To enable to readers to better comprehend the ideas behind the solution proposed by this thesis, we use this section to elaborate on the conceptual ideas related the the context of the problem and the construction of the solution.

2.1.1. Continuous Software Engineering

Continuous software engineering is the practice which organisations employ to rapidly develop and deploy new versions of software applications. While it is a wide field subsuming several different processes and operations, the technical part of continuous software engineering revolves around three concepts: continuous integration (CI), continuous delivery (CDE) and continuous deployment (CD). CI is an often automated process which merges code contributions from multiple developers, often several times a day. In this process, it is ensured that the incoming changes undergo a code quality change consisting of building and subsequently testing the application in its current state. CDE is employed to ensure that the latest application version is always in a production-ready state ready to be deployed, given that all builds and tests are successful. CD is an extension of CDE, in which, provided that the building and testing phases pass, the new version of the application is deployed automatically. With that, CD is meant to be a fully automated process of delivering up-to-date applications directly to its users. In practice, developers assemble pipelines in which the primary step is the CI. While this step can also be used on its own, it is commonly followed by the secondary step of either CDE or CD. Whether an organisation employs CDE or CD, conducting code quality checks on the most recent software version within the CI process is essential in advancing to the next step of the continuous development pipeline. This pipeline is often referred to as the CI/CD pipeline. A scheme of such pipeline is illustrated in Figure 2.1 [27].

As can be seen in Figure 2.1, the CI/CD pipeline starts with the CI as the first phase. This initial step, and with it the pipeline, is triggered when a developer commits code changes to the source repository which stores the application's code base. As most organisations use version or source control for development, such pipeline is usually integrated within source control tools, such as Git [REF to git]. As the pipeline is triggered, a feedback loop is created in which developers receive a passing or failing result from the different steps in the pipeline. If they receive a passing result from the last step, the CDE or the CD, the commit changes are merged, and the updated application is marked as production-ready or deployed respectively. In this case, the feedback loop is closed. However, if they receive a failing result from either CI or CDE/CD, they must locate the source of the problem and update their changes to fix the break [27]. When they upload these changes, developers again wait for the results of the pipeline. This feedback loop continues until the pipeline eventually passes and the code changes are accepted and merged into the application's code base.

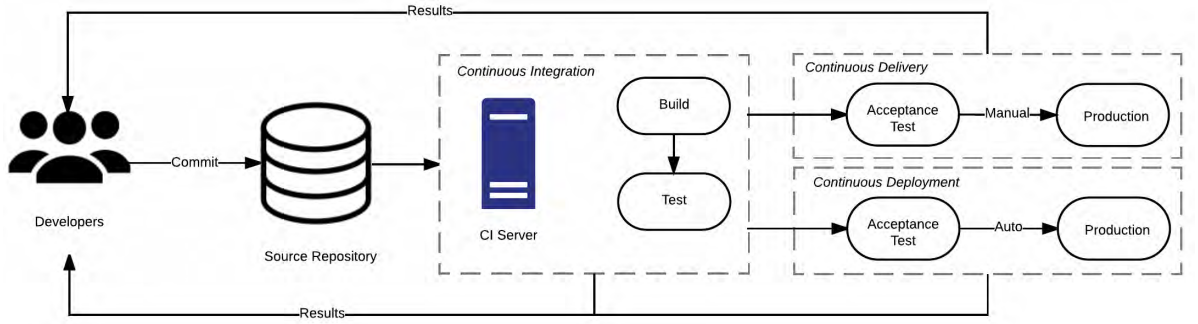


Figure 2.1: The relations between continuous integration, delivery and deployment [27]

Continuous Integration
Continuous integration (CI) is an automated pipeline which first builds the code, and often follows this step by testing it to ensure a certain code quality standard. As can be seen in Figure 2.1, CI pipeline starts when a developer commits a code change to the source repository. To verify these changes, the pipeline takes as input the updated application, created by applying these changes to the code base stored in the repository. This updated version then undergoes the first phase of the pipeline: the build phase. In this phase, the necessary dependencies are downloaded and installed, after which the code is compiled [16]. While this phase also serves as a validity check, it also ensures that the code can be executed. This is the essential reason for taking precedence in order before the test phase, in which the code is run against a series of tests. For both the build and the test phase, the specific steps and tests to be run are predefined by the CI operator who assembles the pipeline.

2.1.2. Approximate Computing
Approximate computing (AC) is a computational paradigm rooted in the recognition that exact computation or unwavering adherence to peak-level service demands typically entail significant resource consumption. Instead, AC allows for selective approximation or occasional deviation from specifications, aiming to achieve efficiency gains disproportionate to the resources expended. Its strategy lies in balancing trade-off between accuracy and resource efficiency, enabling significant energy savings [24].

Partial compilation is a technique that falls under the approximate computing category. In particular, it is the decision to compile only a subset of a given set of files rather than the whole set. With regards to continuous integration, specifically the build jobs in which the source code files are compiled, this thesis employs partial compilation to reduce the number of files that are compiled, and through this, reduce the energetic output of the build jobs.

2.2. Static Dependency Analysis
Static dependency analysis examines the relations between different parts of the code base without compiling or running the code. Dynamic analysis, in which the code is compiled and run, is not always a viable option. For example testing, which is a practice of dynamic analysis, can become very time-consuming and energetically expensive to conduct for larger code bases or extensive test suites. In case of non-deterministic behaviour in concurrent systems, collecting dependencies dynamically may not cover all program traces. Given the limited application opportunities and the possibly time-consuming nature of dynamically obtaining dependencies, static analysis is a cheaper and more versatile way of finding dependencies [21].

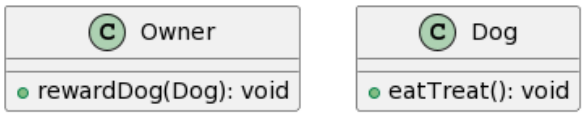
After the code is parsed and processed statically, different types of dependencies can be extracted. For this thesis, it is important to understand two different types of dependencies.

2.2.1. Call & Caller Graphs
A commonly used static dependency analysis technique used directly on methods of classes is a call graph. A call graph for a given method A is a directed graph in which the set of nodes represent all methods of the program, including method A. If method A calls any other method B, this relation

is represented by an edge directed from the node of method A to the node of method B. The set of equations 2.1 formally defines a call graph for method A represented by node A.

$$\begin{aligned} N &= \{A, B_1, B_2, \ldots, B_n\}, \\ E &= \{(A, B_i) | A \text{ calls method } B_i\}. \end{aligned} \tag{2.1}$$

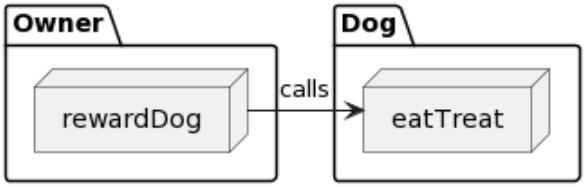
Call graph must be distinguished from a caller graph, which is also a directed graph that explains the relations between the program's methods However, while the node set is identical to the one in a call graph, the edges in a caller graph carry a different significance. In a caller graph for method A, there exists an edge from any method B that invokes method A. Figure 2.2 shows an example of a caller graph analysis. In the UML graph shown Figure 2.2a, it can be observed that the application has two classes, the Owner and the Dog class, each defining one public instance method. The connection between these two methods is depicted in Figure 2.2b, where a part of the $Owner.rewardDog$ method's implementation is shown. On line 4, the Dog object, passed as a parameter to this function, invokes the method $eatTreat$ defined in class Dog. The respective caller graph for function $Dog.eatTreat$ is shown in Figure 2.2c. Since the method $Dog.eatTreat$ is invoked by the method $Owner.rewardDog$, the $Owner.rewardDog$ becomes the caller of $Dog.eatTreat$, the caller graph, in this minimal example, consists of the caller and the called method nodes and an arrow pointing from the caller to the called method.



(a) The UML diagram for classes Owner and Dog

```
1  class Owner {
2      public void rewardDog(Dog dog) {
3          ...                          // Code local to Owner.rewardDog
4          dog.eatTreat()
5          ...                          // Code local to Owner.rewardDog
6      }
7  }
```

(b) The definition of the $rewardDog(Dog)$ method defined on class Owner



(c) The caller graph for the method $Dog.eatTreat$

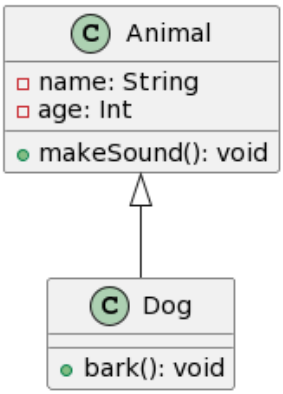
Figure 2.2: An example of a caller graph based on the method dependencies

2.2.2. Inheritance

Inheritance is a concept native to object oriented programming, which allows a class to inherit properties and methods from a different class. When class Dog inherits from class $Animal$, the Dog class can also be referred to as a subclass of class Animal. In the same scenario, class $Animal$ is the super-class of class Dog. The purpose of applying inheritance in object oriented programming is to create a hierarchical system for the classes, in which the sub-classes can reuse the behaviour defined in their respective super-class.

In Figure 2.3, such situation is depicted. The UML representation of the inheritance relationship mentioned above can be seen in Figure 2.3a, in which the relation of class Dog inheriting from class Animal is represented by a solid line pointing from the subclass to the super-class. As a result, Figure 2.3a

shows a valid example of a method declared in class *Dog*, which reuses a method defined on class Animal using the keyword *super*.



(a) A sample UML diagram displaying class *Dog* that inherits from class *Animal*

```
    public void bark() {
        super.makeSound() // Calls the implementation of Animal.makeSound
        System.out.println("Woof!")
    }
```

(b) A code snippet of class *Dog* defining its local method using a method of super-class Animal

Figure 2.3: A sample UML diagram and the associated code snippet

To construct the partial compilation sequence, the calls made by a subclass to a super-class through the use of the *super* keyword must also be detected, as they present a method-level dependency of the same significance as any dependency captured by a caller graph. However, since caller graphs do not always capture these dependencies, the proposed solution includes detection of inheritance and calls involving the *super* keyword to capture these dependencies as well Figure 2.3.

Both inheritance detection and caller graph construction are static analyses technique that capture connections between different parts of the code which hint at their possible interactions during runtime. This feature makes them an important concept for this thesis, as these two techniques are used to detect these interactions, which in turn guide the partial compilation used in the proposed solution.

2.3. Tools

Following the conceptual explanations for better understanding, this section lists the tools used in the rest of this study. By using the term .

2.3.1. Maven

Maven is a software project management tool, often also referred to as a package manager, made to build and manage Java-based projects. The main functions of Maven and other package managers for Java, such as Gradle, is to allow the developer to easily comprehend the state of development of a given application, as well as manage and configure it. Maven is based around a concept referred to as the build life cycles, which define the sequence of steps involved in building and distributing a particular project. Maven builds the Java by using the Project Object Model, or POM, which is an XML file written with a Maven-specific syntax to define the configuration of the project. The file, named within the project as **pom.xml**, encapsulates all project-related information, offering a comprehensive overview of the project to the developer, as well as the ability to quickly add dependencies, plugins, and other attributes to the project's configuration.

The Build Life Cycle

There are three built-in build life cycles defined in Maven: default, clean and site. The default life cycle executed the steps necessary to deploy the project, while clean and site handle cleaning the project and

deploying the project's website respectively. Each life cycle consists of an ordered sequence of phases, where each phase refers to a particular process. In this thesis, clean and site life cycles are not used, therefore, only the default life cycle will be explained in detail.

The default life cycle, which handles project deployment, comprises of several phases:

1. **validate**: checks that the project is correct and all necessary information is available
2. **compile**: compiles the source files in the project
3. **test**: runs the unit tests defined in the test suite of the project using the compiled sources
4. **package**: packages the compiled code in its respective distributable format, such as JAR
5. **verify**: runs integration tests against the existing code
6. **install**: install the packaged version of the project for other projects to use it as a dependency
7. **deploy**: copies the packaged version of the project to a remote repository to share it with other developers and projects

These phases are executed sequentially, as every phase in the life cycle depends on the phases executed before. For example, the package phase depends on the validate phase, which ensures the code is correct, then the compile phase, where the source code files are compiled, and finally the test phase, which runs the unit tests defined for the code. Only when all these phases are sequentially executed and successful, Maven can execute the package phase, which handles the packaging of the checked, compiled and tested code into JAR files.

Instead of running the default life cycle in its entirety, developers also have the option to calling only a specific phase of the life cycle. For example, if the developer only needs to see if the project's unit tests run without failure, the developer can call $mvntest$ command, which summons Maven to execute the phases from the default life cycle sequence up to the $test$ phase. That is, the $mvntest$ command would run the 'validate' and 'compile' phase before running the 'test' phase and output the results. For the purposes of these thesis and understanding the proposed solution, understanding the function and process of $mvn < phase >$ command is important. For Java-based projects that use Maven as its package manager, these commands are often part of the CI pipeline, especially for building and testing the project. As the solution targets the compilation process executed upon the source code files, this thesis will use and evaluate the proposed solution using the commands that run phases from the default cycle that include the 'compile' phase.

POM Structure & Syntax

As was mentioned before, POM file centralises project-related configuration in one place. As projects grow, however, several POM files may be necessary to define. Often, real-world projects are composed of several modules, or packages, which could be considered as sub-projects of the main project. They are often used to separate a sizeable code base into a sum of smaller structures, where each structure is a logical grouping of class files. When a project contains multiple modules, several POM files must be defined. Particularly, if there is N modules, the project normally contains $N + 1$ POM files. One POM is the project POM, also called the parent POM, and the rest are the child POM files. The parent POM resides at the root of the project and holds the common, global configuration which is automatically which the child POM files can read, given that they reference the parent POM. Each of the N modules contains exactly one child POM, and this child POM defines the configuration for the module it resides in. Usually, this POM either contains configuration that is not global, but only local to the related module, or alternatively, it can override the global configuration in the parent POM to fit the custom needs for that particular module. In this thesis, the configuration relevant to the solution will be applied globally, and therefore, it is only concerned with defining the configuration of the parent POM.

To explain the relevant XML tags in the POM syntax, it is best to put them in context. To illustrate this, a minimal POM sample file is shown in Figure 2.4.

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
       XMLSchema-instance"
2      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/
           xsd/maven-4.0.0.xsd">
3      <modelVersion>4.0.0</modelVersion>
4
5      <!-- The Basics -->
6      <groupId>org.samples</groupId>
7      <artifactId>sample-project-1</artifactId>
8      <version>1.0</version>
9      <properties>
10         <maven.compiler.source>1.7</maven.compiler.source>
11         <maven.compiler.target>1.7</maven.compiler.target>
12         <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
13         <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
14     </properties>
15
16     <!-- Build Settings -->
17     <build>
18         <pluginManagement>
19             <plugins>
20                 <plugin>
21                     <groupId>org.apache.maven.plugins</groupId>
22                     <artifactId>maven-compiler-plugin</artifactId>
23                     <version>3.12.1</version>
24                     <configuration>
25                         <source>17</source>
26                         <target>17</target>
27                         <useIncrementalCompilation>false</useIncrementalCompilation>
28                         <includes>
29                             <include>core/Dog.java</include>
30                         </includes>
31                     </configuration>
32                 </plugin>
33             </plugins>
34         </pluginManagement>
35     </build>
36 </project>
```

Figure 2.4: A sample pom.xml file containing the parent POM configuration

Starting at the beginning of the parent POM file, line 1 defines the $< project >$ element, inside of which the rest of the POM configuration lies. The configuration of the $< project >$ from line 1 through line 2 can be considered boiler-plate code necessary to define for the POM to be recognised by Maven. Similarly, the $< modelVersion >$ element on line 3, which defines the version for POM to be 4.0.0, is a necessary part of the global configuration. The customisation of the global configuration starts with the basic information defined between lines 6 and 8. The $groupId$ defines the name for the group of projects the current project belongs to, and the $artifactId$ defines the name for the current project, as well as the respective POM. The $version$ element defines the version of the project. These three elements must be defined in the parent POM, as this information is then used by child POM files to reference the parent. The next important category of configurable elements is the build settings category. The $< build >$ element generally handles the declaration of the project's directory structure as well as the management of plugins. In this case, the the directory structure is not explicitly defined, and therefore the default settings are used. The part important to understand the workings of the proposed solution is the $< pluginManagement >$ element. This element is only meant to be defined in the parent POM, as the $< pluginManagement >$ ensures that the plugins defined within its structure are inherited by the child POM files referencing this parent POM. Inside the element, $< plugins >$ element is defined, in which each plugin has its own individual definition and configuration within its respective $< plugin >$ element. While there are many plugins a developer could make use of in the build stage, there is one particular plugin, the Maven Compiler Plugin, which is a vital part of the proposed solution.

Maven Compiler Plugin

As the name suggests, the Maven Compiler Plugin is used to compile the sources of the project. It is implicitly included in the project with its default configuration, however, as for other plugins, these settings can be overridden by adding and configuring the plugin as it is shown in Figure 2.4 between lines 20 and 29. This configuration is the base for the one used in the proposed solution. In the configuration, the non-optional $< groupId >$, $< artifctId >$ and $< version >$ elements are defined to reference the compiler project and its specific version. The remainder of the plugin definition consists of an optional $< configuration >$ element [29].

This plugin is directly responsible for executing the partial compilation of source files, which can be defined within the $< configuration >$ element. First, the $< source >$ and $< target >$ elements are defined, where the former refers to the Java version of the source code and the latter defines the target compiled byte code version. The $< useIncrementalCompilation >$ element is an important part of this configuration. In this base configuration, its value is set to false, because it is not desirable for the incremental compilation mechanism to interfere with the workings of the proposed solution. By default, however, incremental compilation is enabled, and will serve as the baseline that the proposed solution is compared. Incremental compilation can be considered an alternative solution to the proposed solution, and will be further elaborated on in 3. To inform the plugin that some files will be compiled and some will not, the $< includes >$ element is created. Within this element, an $< include >$ element can be added. This element then holds each file name, or alternatively, a regex pattern to define a set of files included in the compilation. An example of this configuration which defines a partial compilation sequence can be seen between lines 28 and 30. Particularly, line 29 defines the inclusion of file $Dog.java$ from the $core$ package to be compiled. The use of the $< includes >$ element automatically configures the plugin to exclude all files that are not defined in this element.

Lastly, it is important to note that to use the configuration for the Maven Plugin Compiler, the Maven command must explicitly declare this to avoid falling back to the default setting. To do this, particularly for running the compile phase, we use the $mvncompile : compiler - fpom.xml$ instead of a simple $mvncompile$ command. Using this command, Maven will be instructed to use the compiler configuration from the $pom.xml$ file.

2.3.2. GitHub & GitHub Actions

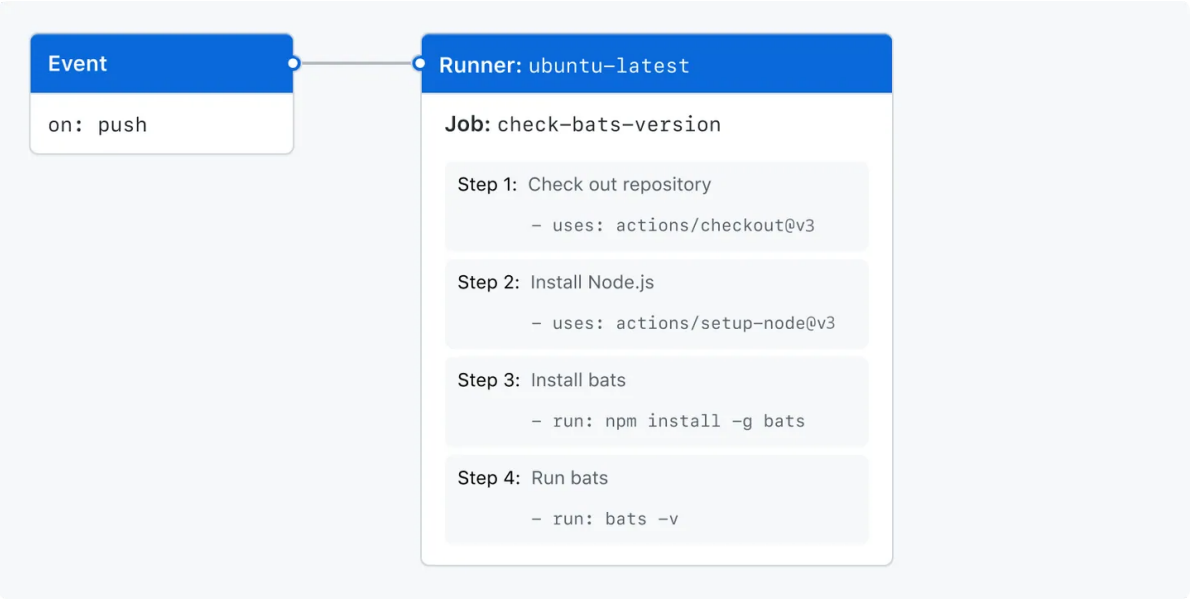
To implement the CI pipeline within a version control system as explained previously, this thesis makes use of Git as the version control system. In particular, GitHub is used as the platform where the repository is stored and operated upon [13]. GitHub also provides an integrated service called GitHub Actions that enables developers to define and run CI/CD pipelines. These pipelines are defined in files that are referred to as *workflows*. To execute a given workflow, GitHub Actions uses servers referred to as *runners* [14].

Workflows & Runners

To define the specific steps of the CI pipeline, developers must write the configuration and the sequence of steps into a YAML file, which is referred to as the *workflow*. To allow a workflow file to be used, it must be added to the ".GitHub/workflows" directory of a given repository.

```yaml
name: npm-example

on: [push]

jobs:
    check-bats-version:
        runs-on: ubuntu-latest

        steps:
        - uses: actions/checkout@v4

        - uses: actions/setup-node@v4
          with:
            node-version: '20'

        - run: npm install -g bats

        - run: bats -v
```

(a) A code snippet of class Dog defining its local method using a method of super-class Animal



(b) GitHub Actions visualisation of $npm - example$

Figure 2.5: A sample GitHub Actions workflow and its respective visualise execution sequence

An example of such workflow together with its visualisation in GitHub Actions can be seen in Figure 2.5. Before breaking down the workflow file in Figure 2.5a, it is important to understand the fundamental terminology used in GitHub Actions and its implementation of the CI pipeline:

- **Workflow** is an implementation of the CI pipeline process, which is composed of one or multiple jobs (for example, the build job, or the build and the test jobs). The workflow is implemented via a YAML file, in which the metadata of the workflow resides, as well as the jobs of the CI pipeline.
- **Job** is a sub-process of the CI pipeline process, such as the build job, which is composed of smaller sub-processes, or steps. Jobs represent the stages of the CI pipeline. When the workflow is executed, the result of each job is reported.
- **Step** is an individual task that can be defined within a job that exists in a workflow. It is the smallest building block of the CI pipeline process. It is responsible for running specific commands on the commit, such as the $mvncleancompile$ command.

In short, a *workflow* defines the CI pipeline, where different stages of the CI pipeline map to *jobs*. These jobs then contain specific, actionable *steps* that execute commands on the code. With that, we come back to the sample workflow in Figure 2.5a. Line 1 defines the name of the workflow, in this case, $npm - example$. In line 3, the configuration defines the set of events that trigger the execution of this workflow. In this case, the workflow would be triggered at every instance of a developer pushing their commits to the repository. Between lines 5 and 17, the list of jobs for the $npm - example$ is defined. Notice that this workflow only defines one job, the $check - bats - version$ job. It is mandatory for the developer to first define the runner on which a particular job is executed. Note that jobs can be executed on different runners, however, all steps within a job have to be executed on the job-assigned runner. In this case, the $check - bats - version$ job is set up to run on a GitHub-hosted runner which uses the latest version of the Linux Ubuntu operating system. GitHub Actions also offers different operating systems for the developers to run their workflow jobs on. Alternatively, developers can also use their own servers to run the CI pipeline. In this case, the configuration would be $runs - on : self - hosted$, possibly with more optional parameters in case the developers have more self-hosted runners available. When the runner environment is defined, steps can be defined for the particular job. The $check - bats - version$ job involves four distinct steps. Jobs always execute the steps sequentially, in the order they are defined in. The first step configured on line 10 is defined with the $uses$ keyword that runs $v4$ version of the $actions/checkout$ action. This step checks out the repository code such that the remainder of the steps can be run against the code. The second step defined on line 13 uses the $v4$ version of the $actions/setup - node$ action which installs the version of node specified in line 14. These preparation steps are then followed by two steps defined with the run keyword, which tells the job to execute a specific command on the runner. In this case, the step defined on line 16 runs the command that installs the $bats$ package using npm. The final step of the job, defined on line 18, runs the $bats$ command which outputs the version of the package.

The workflow workflow and its execution in GitHub Actions is visualised in Figure 2.5b. When the $Event$ happens, in this case, when code is pushed, GitHub Actions starts and attaches a new process which runs the workflow, and thus executed the CI pipeline. This can be seen on the right side of the visualisation. The workflow is attached to a runner of type $ubuntu - latest$, and the process defines the job to be run together with the associated steps. While the workflow is being executed, the runner outputs the logs generated by running the jobs and uploads them to the GitHub Actions process, together with the results for each job. The GitHub Actions platform provides these logs and results to the developer. In case of failure, the developers can find exactly the step of the job that has caused it to fail, enabling them to rapidly identify where the detected problem has occurred.

GitHub Actions offers developers the option of using a GitHub-hosted runner to complete the workflow tasks, and also provides the option to the developers to use their own servers as runners. This can be enabled by setting the value of $runs - on$ for a given job to $self - hosted$ instead of defining the operating system of a GitHub-hosted runner. In this case, the runner can be configured to simulate the environment necessary to run the workflow steps on a particular code repository, for example, the necessary JDK can be installed in the environment prior to running the workflow jobs [14].

2.3.3. Artifacts

In some cases, developers want to make use of the data generated while the workflow has been run. To provide such option, GitHub Actions allow the creation of storage of *artifacts*, which persist defined data after a job is completed. Artifact uploading has to be set up in the workflow file as a particular step within the target job. While artifacts can be uploaded in order to be used by subsequent jobs, they are often also uploaded for the developers to analyse when the workflow is executed. This often applies to artifact types such as log files or test results.

In case of the proposed solution, artifacts are used to retain the build metadata and energy consumption data, which are a crucial element in its validation and evaluation.

2.3.4. GitHub Actions API

To interact with the GitHub Actions tool, GitHub provides an API for developers to operate upon GitHub Actions set up from outside the platform. While there are many different API points and associated functions, for the purposes of this thesis, there are three functions necessary to know in order

to understand the evaluation process of the proposed solution:

1. **Collecting workflow runs**: Given the unique workflow ID, the collection of runs can be fetched by the API.
2. **Rerunning a given run**: Given the unique ID of a workflow run, the API can be used to trigger a rerun.
3. **Downloading artifacts of a run**: Given the unique ID of a workflow run, the API can download the associated artifacts of the run.

This set of functions of the GitHub Actions API is used to automate the validation and evaluation process of the proposed solution [12].

2.3.5. Git Hooks

Given an existing git repository, Git Hooks provide a mechanism for the developer to define specific behaviour which is meant to happen at different events related the usage of git. Such hook is essentially a script that the git system triggers when a specific event happens. Every git repository provides template files for hooks contained in $.git/hooks$ folder, where each template file is named based on the event that triggers the script contained inside the file. For example, developers can edit the $pre-commit$ template file and add a script to it which they want to be executed every time before a commit happens.

The pre-commit hook is especially important for the technical part of the proposed solution, as it is used to automatically apply the script that manages the construction of the guided partial compilation sequence.

2.3.6. EcoCI

EcoCI is a tool that aims to make CI pipelines more transparent in terms of their energy consumption. Using the information available on the hardware on which the CI pipeline is executed on, EcoCI estimates the energy consumption of the pipeline, or a specific part of the pipeline, and delivers the estimates to the developer. To give a well-rounded perspective on the energy consumption, it estimates four different values: the total energy output in joules, the average CPU utilisation in percentages, the average power output in watts, and the duration measured in seconds.

To allow for simple integration of EcoCI, it has been made available as a GitHub Actions plugin. This enables developers to make use of the tool by simply adding the EcoCI tool in their workflow file. A sample of such use is shown in the Figure 2.6, which is a workflow file derived from Figure 2.5a after integrating EcoCI in the pipeline.

To allow for simple integration of EcoCI, it has been made available as a GitHub Actions plugin. This enables developers to make use of the tool by simply adding the EcoCI tool in their workflow file. A sample of such use is shown in the Figure 2.6, which is a workflow file derived from Figure 2.5a after integrating EcoCI in the pipeline.

In this case, the EcoCI is integrated with the goal of measuring the energy consumption of a step in the job instead of the entire workflow job. Particularly, it estimates the energy consumption repository checkout step of the $check-bats-version$ job. In total, three steps that operate EcoCI have been added to the workflow: the EcoCI initialisation, the energy measurement execution, and the retrieval of the energy consumption data. The first step in the updated workflow is now the $start-measurement$ task, which sets up the EcoCI and starts the energy consumption estimation before the repository checkout step is executed. The repository check out step is then followed by the $get-measurement$ task of EcoCI, which logs the energy consumption estimation up until this point, with the starting point being the EcoCI initialisation. This measurement period is also referred to as a lap. The rest of the steps in the workflow are carried out as before, until the job reaches its last step. In this step defined on line 31, an artifact is uploaded to the GitHub Actions process. The if keyword set to $always()$ ensures that this step is executed regardless of the results of the previous job steps. This means that this artifact will be uploaded even despite potential prior failures within the pipeline. The remainder of the configuration instructs to runner to fetch the lap energy consumption estimation data, in this case, the energy consumption estimation of the repository checkout action. The developer can then

```yaml
name: custom-workflow

on: [push]

jobs:
    check-bats-version:
        runs-on: ubuntu-latest

        steps:
        - uses: green-coding-solutions/eco-ci-energy-estimation@v2
          with:
            task: start-measurement

        - uses: actions/checkout@v4

        - uses: green-coding-solutions/eco-ci-energy-estimation@v2
          with:
            task: get-measurement
            label: 'repository checkout'

        - uses: actions/setup-node@v4
            with:
            node-version: '20'

        - run: npm install -g bats

        - run: bats -v

        - uses: actions/upload-artifact@v3
          if: always()
          with:
              name: lap-energy-data
              path: /tmp/eco-ci/lap-data.json
```

Figure 2.6: Usage of EcoCI for the $custom - workflow$ sample

download this artifact, which is, in this case, the $lap - data.json$ file containing the four different energy consumption-related estimates for the measured step [15].

EcoCI provides the energy measurements for validating the efficacy of the proposed solution. As contributions have been made to improve EcoCI, it is important to note that instead of using an official release of EcoCI, most workflows used in the practical part of this thesis will use a specific commit made to EcoCI. As will be explained later, these commits are part of our contributions made to match the hardware requirements of the runners used for running the experiments in this work.

3

Related Work

This chapter examines the existing literature related to the topic in question. It extracts the relevant information as well as limitations of past solutions in order to form the solution to the main problem. It also reviews the alternative tools to those used in the proposed solution.

3.1. Build Optimisation

The build system of an application handles the translation of source code into deliverables, in most cases, both the documentation and the executables. Depending on the complexity of the code base and the requirements for the deliverables, the build system may involve thousands of commands that result in undesirably long execution times. The build process is one that developers must often interact with several times in a day, to check whether the changes they have made to the product have not broken the build. The use of build is often direct and local in these cases, when the developers check the builds on their machines after making a change [23]. While optimising build jobs directly within CI pipelines has only recently started receiving attention in scientific literature, optimising builds locally, or outside the pipeline, has been a subject of intensive research efforts for longer. Some approaches have even developed into features that are commonly used in real-world practices. One of them is an approach similar to partial compilation concept, known as incremental builds. The other is caching, which can be applied in many different fields, however in this section it will be reviewed in the context of the build process.

3.1.1. Incremental Builds

For the purpose of faster and cheaper building of the application code, package managers have recently started developing features that can save builds. One these features is incremental building, or incremental compilation, which aims to leverage changes made in the code base since the last build process to determine which parts of the next build process are necessary and which are safe to skip.

For Java, the most common package managers that offer a version of such feature are Gradle and Maven. Gradle explains its concept of incremental build as "a build that avoids running tasks whose inputs have not changed since the previous build". To use this feature, developers define tasks and their respective inputs and outputs, where the task pertains the build flow itself, with input being the code base and the outputs contain the build results, such as compilation and test results. Given these tasks, or builds, if no changes have been made to the inputs of the task, the task itself is not executed in the next build. It is only executed if there have been changes made to the input, or in this case, the code base. Especially for multi-task projects, where developers may define separate build tasks for each module, this approach has potential to skip a significant amount of compilation and test execution compared to the more traditional approach of building the entire code base with every build command [18]. The alternative popular Java package manager, Maven, also offers incremental compilation feature as a part of its compiler plugin used in the builds. When enabled, its incremental compilation approach, all sources are recompiled if any files have been added, deleted or modified. The modules are recompiled separately in case the JAR files they depend on have changed. If the incremental compilation is disabled,

Maven only compiles the changed classes. However, Maven documentation itself advises against this option, as the classes that may depend on the changed files are not identified or recompiled, leading to a high probability of skipping compilation of files that were broken by proxy [29].

The incremental building, or incremental compilation, implemented by the mention package managers show potential in saving the workload in each build job by use of caching and change-guided selection of the subset of files that need recompilation. However, the main limitation of using incremental build in Gradle is its heavy dependence on manual configuration by the developer. As per Gradle's documentation on their *Incremental Build* feature, the Gradle tasks must be defined with their respective inputs and outputs, where the selective building hinges on whether the inputs have changed since the last build. In case that the developers define as input the whole source code related to the project, many non-breaking changes may pose as triggers to running the build, even if the changes detected do not require the whole build to be executed. To our best knowledge, there is no sufficient research into how much build savings can Gradle's incremental building yield in an average project, whether built locally or in the CI process. In case of Maven's incremental compilation, there is a lack of granularity when it comes to determining of recompilation. If sources are added, deleted or changed, full recompilation is triggered, regardless of the real extent of the effect of the changes made. These limitations, mainly the requirement of developer's designing of tasks together with coarse granularity with which builds are saved, call for additional investigation into lowering the amount of manual configuration necessary as well as finding the real extent of the effects of changes made since the last build.

3.1.2. Caching

Caching refers to saving of results of computations for the purpose of reusing them rather than generating them again. Applicable especially in cases when results do not need to be updated from job to job, caching can save computation efforts in many different applications. The work of Gallaba et al. has explored application of caching mechanism in build execution of CI pipelines. The authors motivate the use of caching as an alternative to or in combination with the existing approaches such as Gradle's dependency-based optimisations. The presented approach, *KOTINOS*, uses caching of initialisation and installation of dependencies by encapsulating them in a Docker container. If the dependencies and related scripts have not changed between the CI runs, the Docker environment that was constructed at the first build is reused again instead of being built anew. If the dependencies or the scripts did change, the Docker container is rebuilt such that it can be used in the subsequent pipeline runs. This approach has shown that in 91% of studied cases, cached build results can be used, validating the concept of using caches in CI build jobs [10].

While caching is not directly used within the proposed solution of this thesis, it is an idea central to the future integration of the proposed solution in real-world practice. Its use in Gallaba et al. proves that there is promise in reusing build results. While the mentioned work does not cache compiled files but rather dependencies and installed environments, this research raises an opportunity to cache other parts of the build process, such as the compiled version of files.

3.2. Optimisations of Build Jobs in CI

An important interaction with building is related to CI, where the CI server regularly builds the latest code with the goal of identifying possible breaks early on [23]. According to Gallaba et al., 35% of the CI runtime is spent on building, or compilation and testing, making it the most time- and resource-demanding job in the CI process. With that, the authors call for more research effort to be put into making compilation and testing more efficient, as these improvements will yield largest reductions to CI workload, and as a result, reductions to both energy demands and duration of the CI process [11]. While it is a new, emerging field scientific literature, several works have already proposed different ways of approaching optimisation of the build jobs within the CI process.

The work of Abdalkareem et al. published in 2019 started investigating possible ways of reducing the overall workload by examining the commits for which the developers manually skipped CI pipeline execution. Under the assumption that developers only order CI pipeline skipping in cases where they deemed it completely safe, the authors manually examined a commit database of TravisCI and extracted the patterns found in CI-skipped commits. The patterns found were split up into two main categories based on whether the authors could identify the reasons for skipping CI solely from the

repository, or not. For the first category, they found that most of the commits developers decide to skip CI pipeline execution for are those that change non-source code files, such as documentation files. For the other category, where the authors could not identify the precise reason for skipping CI, most of the commits involved changes made to source code files. For the first category, with reasons for skipping CI easily collected from the repository, the authors derived a rule-based technique that was meant to automatically detect and label commits that were safe to skip. The latter category of reasons was omitted from further optimisation efforts due to the difficulties of extracting specific rules from these commits. The technique showed promise with its ability to reduce the number of commits that trigger the CI pipeline by 18.16% on average on unseen data. It also consulted developers to evaluate the real need for a tool that automatically flags commits to skip the CI pipeline for. They found that 75% of developers deem it to be nice, important or even very important to have such tool available, further motivating the efforts done in the CI optimisation field [2].

Abdalkareem et al. continued to build upon their previous findings in 2020, when the authors published a paper presenting a machine learning approach to predict the commits for which CI process can be skipped. To train the machine learning model, specifically a tree classifier model, they use 23 features. Particularly, they allocate 5 features to reflect the rules defined in their previous work from 2019, 17 features for other commit-level features from the TravisCI database of CI-skipped commits, and finally one feature to consider the commit message, as it is believed that this description developers provide for the changes can increase the accuracy of prediction. When tested on unseen data, the classifier developed in this work yielded an average F1-score of 0.79, which shows that the combined level of precision and recall is 2.4-times higher than that of the random-guesser baseline. Compared to the state-of-the-art technique - the rule-based techniques - the tree classifier showed 56% improvement [1].

Jin et al approached the optimisation from a different perspective, mainly by examining the results of CI pipelines and using the patterns found to predict failures in builds. The paper has found that on average, 84% of all builds in the CI pipelines are passing builds, which supports the motivation for building a CI-skipping framework by offering potential for substantial cost savings. Arguing that the existing predictors heavily rely on the past results of build execution, the authors make a distinction between first and subsequent failures, aiming to predict the former. Their solution, *SmartBuildSkip*, predicts the first failures using build features and project features. To collect build features, the authors extract information from the code base the changes it is subject to, such as the number of source files changed since the last build. The project features are concerned with meta-data such as the number of developers working on the project. Following the in-depth research into these features and their correlation with build failures, it has been shown that build features, particularly the number of source code files and lines changed since last build, have proven to be the most significantly correlated features in first failures, meaning that the higher the number of source code-related changes, the higher the probability of build failure. The technique showed an improvement compared to the baseline machine classifier of failures with up to 9% improvement in recall and 7% improvement in precision of safe build skipping. In terms of the potential to save builds using their predictor, the technique was found to be capable of saving up to 61% builds, although with only catching 73% of failures immediately [19].

Building on their past solution, *SmartBuildSkip*, together with reviewing and making use of the CI-skip rules developed by Abdalkareem et al. [2][1], Jin et al propose an even more sophisticated solution, *PreciseBuildSkip*. The first part of their research focused on evaluation the CI-skip rules and found that neither of the rules guarantee that the pipeline execution in builds can be safely skipped, and many of them provide few or no opportunities for cost savings. Therefore, the authors supplemented the existing rules with additional ones, referred to as CI-Run rules. These consider change-specific information as well as meta-data, such as changes in configuration files and addition of new platforms respectively. For their proposed solution called *PreciseBuildSkip*, which predicts the builds that can be skipped, the authors train it as a cross-project predictor using several different projects in its training process. In its most conservative configuration, which is also the safest, it provides 5.5% cost savings while correctly predicting all build failures. While other configurations can bring up to 48% savings in costs, they also lower the number of failures caught to roughly 45% [20].

Despite the successful applications and promising results of the papers mentioned above, the previous works share certain parameters to their solutions, leading to targeting only a very specific subset of possibilities with regards to optimising the CI process. One of the common attributes is their focus

on commit-level skipping of pipeline execution, where rule-based techniques and machine learning approaches decide whether the CI pipeline is completely executed, or alternatively, completely skipped for a given commit. Another shared limitation of the previous works is the focus on non-source code changes - most rules are based on changes in documentation files or meta files such as $.gitignore$. Due to the complicated nature of application code, these works often seize to analyse and therefore create rules for skipping any commits that involve a source code change. Recalling the findings of Jin et al. which show strong correlation between source code changes and build breaks, the situation demands more investigation into how changes in source code affect the CI pipeline build execution.

3.2.1. Test Selection in CI

Although the primary focus of this thesis is not on the test phase of CI, its integral relationship with build jobs in terms of environment and processes makes it an area worth investigating further to augment the research done in this thesis.

A paper from 2019 by Li et al. targets the test phase of the CI pipeline, and aims to optimise it by performing guided test selection. The underlying mechanism of their test-selecting framework is static dependency analysis aided by a set of dynamic rules. This combination of techniques is applied to a new commit pushed to the repository and as a result, the test phase of the CI pipeline is amended to only execute the tests for the code that has been marked as affected. In terms of improving efficiency of computation, it can reduce the test suite size executed in the pipeline by CI 92% when compared to running the full test suite, and by 48% when compared to ClassSRTS (the baseline) [21].

The results of this approach prove that it can yield significant savings in the test phase of CI. Considering the direct dependency of testing on the compilation of files, which is the main subject of this research, this work has inspired the approach we later use in determining the partial compilation sequence. The assumption is that this connection and the similarities between the building, specifically the compilation, and the testing could mean that building could be optimised in a manner much like testing - specifically, by using static dependency analysis.

3.3. Measuring Energy Consumption

Energy consumption patterns of CI pipelines and measuring of real consumption inside the runners has recently been addressed in a thesis work by Limbrunner (2023), mentioned in Chapter 1. In this thesis, the author developed a framework, called the *Planterary Framework* that measures the energy consumption of the hardware that executed the CI pipeline. The tool offers a fine-granularity in measuring, where each step of the pipeline can be measured separately. The framework, which can be integrated into the CI pipeline and report on its energy consumption patterns, retrieves existing logs from DevOps platforms and runner's API technology to calculate the relevant energy metrics. Besides providing an in-depth analysis of energy consumption of different jobs and job steps in the pipeline explained in Chapter 1, the author also suggests some category-related, generic ideas for potential improvements in energy efficiency. For the build job category, the main suggestions include caching of build results and skipping of unaffected build steps [22].

The solution of Limbrunner is one of the first elaborate investigations into CI-related energy consumption. While the analysis of consumption patterns sheds light on this uncharted field, the integration of this framework presented some difficulties in preliminary research done for this thesis. The implementation of the *Planetary Framework* is accessible freely on GitHub, however, the integration of the framework into the CI pipeline used in this thesis proved difficult, partly due to outdated language versions in the code base. The second reason for difficulty of use could be attributed to the proprietary nature of some components. The thesis of Limbrunner was written in collaboration with a private company, and as a result, some necessary components of the *Planetary Framework* are not freely available. Therefore, instead, this thesis adopted EcoCI, a GitHub Actions-based plugin for CI pipeline energy consumption estimation, explained in better detail previously in Chapter 2.

4

Approach

This chapter introduces the conceptual overview of the proposed solution. The fundamental principles are explained from a high-level perspective to present to the reader how the solution addresses the challenges outlines in the problem statement.

4.1. Framework Overview

The main objective of the framework is the optimisation of the CI pipeline in terms of energy consumption. The framework targets a specific stage of the CI pipeline - the build job. One of the main tasks carried out within such build job is the compilation of source code files. With the aim of reducing the build job workload, which in turn reduces energy consumption, the framework employs partial compilation, a concept explained earlier on in Chapter 2. The framework operates on component-level, where by the term *component* we mean the methods and fields of classes.

To present a clear idea of how the framework integrates into the regular interaction between the developer and the CI process, the pipeline and the application moment for the framework is illustrated in Figure 4.1.

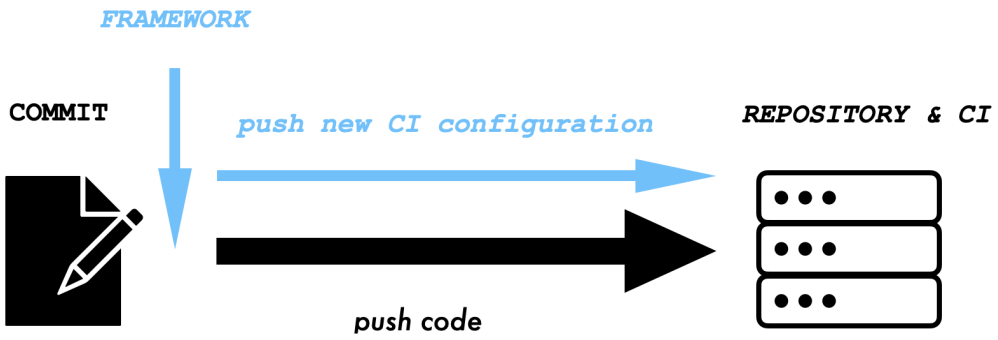


Figure 4.1: Framework integration in practice

First, the developer makes changes to the original, or pre-commit version of the code base. Then, at the moment they commit these changes to the remote repository, the framework intercepts the pipeline. Normally, the next step would be pushing the post-commit code to the remote repository and running CI as normal. However, when the framework is used, it processes the commit and outputs the updated configuration for the CI pipeline, which contains the instructions for the framework-derived partial compilation. Then, when the developer pushes the commit, the updated code is pushed as well as the output of the framework - the CI configuration. The CI pipeline is then amended accordingly, and the cycle continues as it normally would.

The central idea of the framework is to use partial compilation to reduce the workload without sacrificing the reliability of the result. In short, the partial compilation sequence should be capable of catching errors that full compilation does. However, for this to be true, files to be compiled cannot just be chosen randomly. To select relevant files for compilation and filter out files that do not need to be compiled, the framework uses two main inputs: the changes made in the current commit and the dependencies between different parts of the code.

4.1.1. Commit Analysis

The idea of using the code changes that have been made in the commit revolves around the assumption that the previous commit, has passed the CI pipeline, and therefore, the framework considers the previous version of the code as checked. Given that the last version is marked as checked by CI, and that the latest commit introduced a set of changes made on top of this checked version, it can be derived that the next CI pipeline must only check the validity of the updated code, or the part of the code base that has been affected by the changes made in the commit, while skipping the checking of files that have not been affected since the last version. In the scenario specific for the proposed solution, in the upcoming CI pipeline process, pertaining to the latest commit, only the files affected by the latest commit changes would be compiled, and the files that remained unaffected since the previous version would not be compiled.

However, the set of changes in the commit does not present enough information for the framework to build the partial compilation sequence. This is due to the fact that the code base is inherently interconnected, as some parts of the code base use other parts of the code base. Assuming that each class is fully contained in its respective file, a given class Dog, which contains $method1()$, is called by $method2()$ that resides in class Cat. In case the latest commit makes changes to $method1()$ in class Dog, and the framework would only compile the files where the changes have been made, then the file of the Dog class would be compiled. However, the changed made to $method1()$ may also reflect in $method2()$residing in the Cat class file. Since there were no changes to this file, the Cat class will not be recompiled, and the possible breaks inflicted in $method2()$ of class Cat will go uncaught. This simple scenario would already violate the validity of partial compilation guided solely by the file changes pertaining to the latest commit.

The problem could be simply reformulated as follows. The commit information provides a list of *explicitly changed* files, however, due to the dependencies inherent to the application code, these *explicit* effects alone do not provide sufficient information for constructing a partial compilation sequence that would be as reliable as full compilation. This is because of *implicit* effects, in which explicit changes can affect unchanged files via dependencies that connect the explicit change with said file.

4.1.2. Dependency Analysis

The framework identifies the dependencies using dependency analysis techniques. It does so by capturing three types of dependencies. The first one is the dependency created by inheritance, which works on the class level. The second one is the dependency derived from caller graphs and abstract syntax trees (ASTs), which works on the component-level, where under the term *component* we refer to class methods and fields. Finally, the third type of dependency that is extracted is the type dependency, where a field or a method parameter or return type is a custom class in the code base. The results of such analysis can be visualised as three graphs, each depicting the network of one of these types of dependencies.

For the first graph, the node set is composed of nodes that correspond to all classes in the code base. All existing edges represent an inheritance relation between the two classes that the edge connects. Every edge is directed, originating in the super-class and pointing to its respective sub-class. Note that in a regular UML graph, the dependency is usually drawn in the opposite direction, i.e. the edge normally originates at the sub-class and points towards the super-class. However, for the custom dependency graph used by the framework, the goal is to model the graph such that when it starts at a node which is a super-class, it can easily track the outgoing edges to get to nodes that are dependent on the considered node.

In the second graph, nodes correspond to all methods and fields that exist in the code base. For convenience, we refer to methods and fields as components. In this graph, edges are created between

the nodes in case one of the components refer to or call another components. These edges are also directed, originating at the component being referred or called, and pointing to the method that calls it, signifying that the destination node of the edge is dependent on the implementation of the origin node.

In the third graph, we also use directed edges, which signify the dependency of a field or a method on a specific type. The origin node is always a class, as it represents a type, and the destination node is always a method or a field node. Such a dependency exists when a method has parameters of this type or returns an object of the type. Similarly, a field is always dependent on its type. If this type is a custom class defined in the code base, there is an existent dependency that translates into an edge in this graph, pointing from the type class to the field.

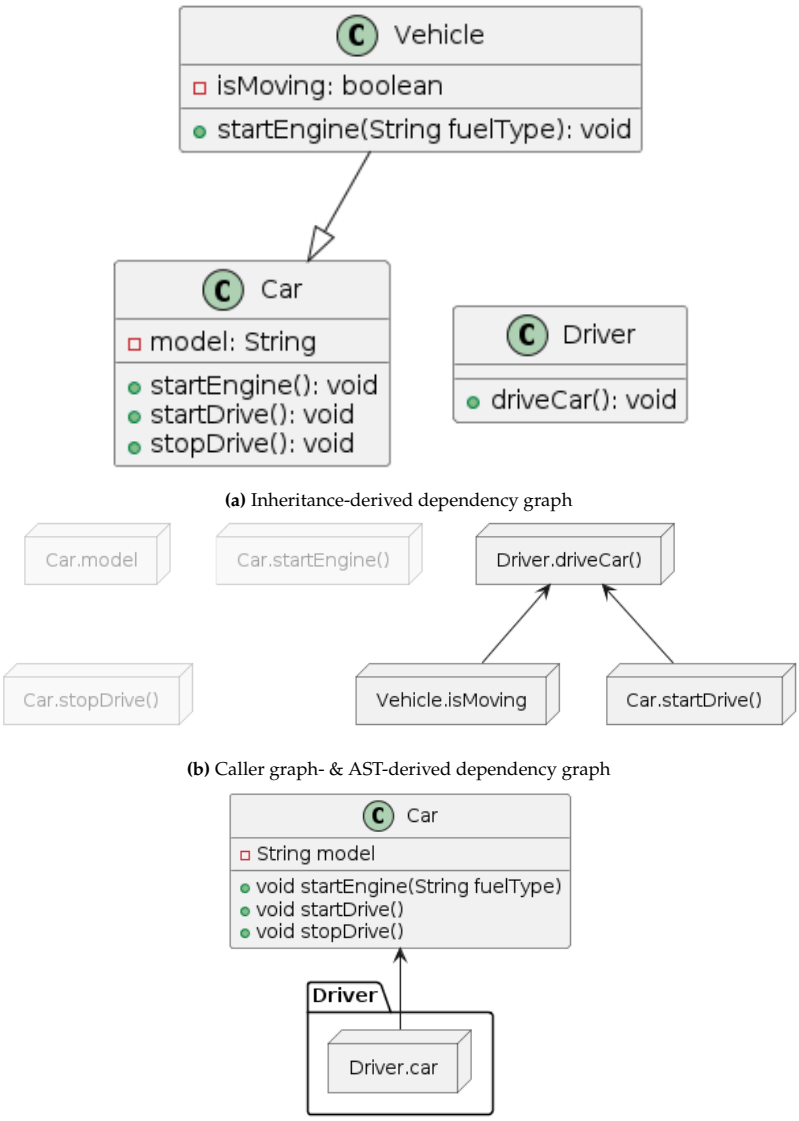
An example of such graphs is shown in Figure 4.2. The inheritance graph in Figure 4.2a shows all three classes in the respective sample code base - $Vehicle$, Car and $Driver$. In this case, class Car inherits from class $Vehicle$, and the relation is depicted in the inheritance dependency graph with an edge directed from the super-class $Vehicle$ to its respective sub-class Car. Since there are no sub-classes or super-classes to the $Driver$ class, there are no outgoing or incoming edges from or to its node. In Figure 4.2b, all methods and fields of the code base are represented by the nodes in the graph. For simplicity, we only look at the connections related to the $Driver$ car and its only method, $driveCar()$, and disregard the grayed-out nodes of unrelated methods and fields in the code base. There are two incoming edges to its node labelled $Driver.driveCar()$. The leftmost edge originating at the field $isMoving$ of class $Vehicle$ signifies a dependency of $Driver.driveCar()$ in on this field, which was found by analysing the AST analysis, revealing that $Driver.driveCar()$references $Vehicle.isMoving$. The rightmost edge originating at the node of method $startDrive()$ defined in class Car signifies that this method is a caller of$Driver.driveCar()$. Finally, in the type dependency graph in Figure 4.2c, the graph contains both class and component nodes. The connection between class Car and the field car declared in class $Driver$ signifies that the type of the field is Car. As the edges point from the class to the dependent component, we have a directed edge originating at Car class, pointing to the $Driver.car$ field.

Note that these three graphs could be connected into one larger graph, in which the node set is composed from classes and components. Dependencies are considered to be of equal importance, therefore, it would not be necessary to differentiate the different types of edges. We separate the graphs here for logical simplicity.

4.1.3. Partial Compilation Mechanism

The commit information and the dependency graphs are used in combination by the framework in the process of selecting relevant files to compile. The conceptual overview of the process can be seen in Figure 4.3.

The pipeline consists of three main stages. After the developer commits their changes, the framework reviews the commit and extracts the information about the changes made to the original, or pre-commit version of the code, as described in 4.1.1. In this example, the only change in the commit was made to the method $Car.startDrive()$. Next, the framework constructs the dependency graph from the post-commit version of the code as described in 4.1.2. In this stage, it also reviews the information extracted from the commit to find the explicit changes. In this case, the changes have been made in $Car.startDrive()$ method. Consequently, the framework marks the corresponding nodes in the dependency graphs. In case of the component dependency graph, it marks the node of the changed method, while in the inheritance graph it marks the class in which this method is defined. In the type dependency graph, we mark the Car class as changed. When all explicit changes are marked in the dependency graph (marked in the graph by red colour), for each marked method, the graphs are walked by following all the outgoing edges and tracking all nodes visited until all paths are fully explored. For distinction between the explicit and implicit changes, they are marked by red and blue colour in the graph respectively. In this case, an explicit change was made in the$Car.startDrive()$method which is connected has one node dependent on it - the $Driver.driveCar()$ method. When all paths originating at the $Car.startDrive()$ node in the component graph are explored, all nodes visited during the traversal process are marked. In this case, it is only one path, which leads to marking of $Driver.driveCar()$ node. In the inheritance graph, the framework follows the same approach, however in this case, class Car does not have any sub-classes and therefore there are no outgoing edges from its node. As a result, there are no implicit



(a) Inheritance-derived dependency graph

(b) Caller graph- & AST-derived dependency graph

(c) Type-dependency Graph

Figure 4.2: Dependency graph construction

changes made in this graph. Similarly, in the type dependency graph, we mark the $Driver.car$ node as an implicit change, since it depends on an explicitly changed class Car. In the final stage, where the files to be compiled are selected, the framework collects the methods corresponding to all the marked nodes in the dependency graph, both the explicit and the implicit changes found (i.e. both the red and the blue nodes). Finally, the marked methods are mapped back to their classes, and these classes are then mapped to the files they reside in, also referred to as parent files. These files are then all marked to be compiled in the respective build process within the CI pipeline, automatically omitting all files that have not been affected by changes, and therefore do not need to be checked in the CI.

4.2. Git-related Use Cases

The framework, given its approach, is suited to a specific scenario in terms of the Git commit history. The scenario is visualised in Figure 4.4.

The figure depicts the line of commit history leading up to the latest commit. The initial commit of the history is fully compiled. Every addition to the initial code version introduced with commit #0 is compiled in the subsequent commits, as it would be classified as an explicit change. Therefore, it can be

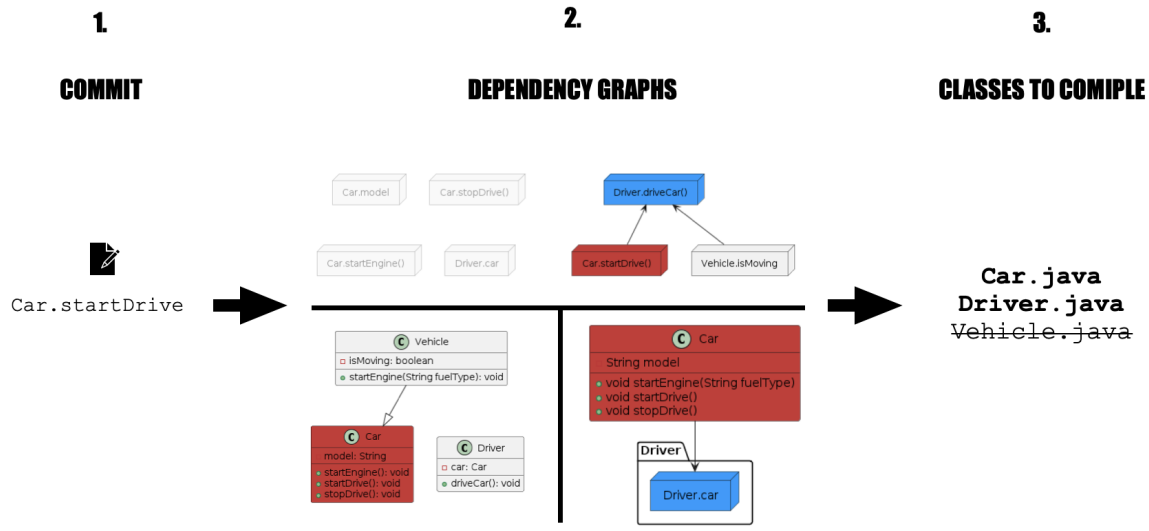


Figure 4.3: Framework pipeline

derived that every commit after the initial commit would be valid in terms of the CI pipeline, regardless of whether full compilation or partial compilation was used between the commits. Recalling the assumption that the CI pipeline is always passing for the previous commit, the framework is specifically suited to be applied if the commit before the latest one did not exhibit any issues that would cause the pipeline to fail. In this case, it is safe to assume that the version of the latest commit is quality-checked by CI, and this in turn allows the framework to instruct the latest commit CI configuration to only compile the explicit changes and their effects.

4.3. Change-related Use Cases

When a developer interacts with the source code, there is a number of different changes that can be done. Classes, methods and fields can be added or deleted, or alternatively, changes can be made to existing code components. The solution proposed in this thesis reacts to all the different types of changes to the method by checking where additions or deletions have been made. Subsequently, it uses these changes to find all class files potentially affected by the changes made. This section unpacks each particular use case and explains how the mechanism of the framework operates to target each situation with relation to the commit information and the dependency graph. The assumption for each use case below is that the changes happen in isolation, that means, if the given use case is the singular change in the latest commit. 4.3.3 explains how the framework addresses commits with different combinations of changes.

4.3.1. Classes

Classes can be either added, changed or deleted from the code base. While some of these changes can be solved by only compiling the explicitly changed files, changes that alter the dependency graph may trigger compilation of some implicitly affected related files in the respective pipeline.

An important thing to note is that in this section, we explain the class-level use cases which mainly deal with class-level dependencies. Therefore, within this section, when we refer to the dependency graph, we mean to refer to the class dependency graph only, as the component dependency graph is not relevant for this category of use cases.

Addition

With regards to the mechanism of the framework, addition of a new class in a new file always leads to the same result in terms of partial compilation decision. Assuming that the addition of a new class is the singular change in the commit, the only file that is marked for compilation is the file where the class was added. The reason behind this behaviour is that when a class is added to the code base, the only dependencies that can be created are the dependencies of the new class on other classes. There is

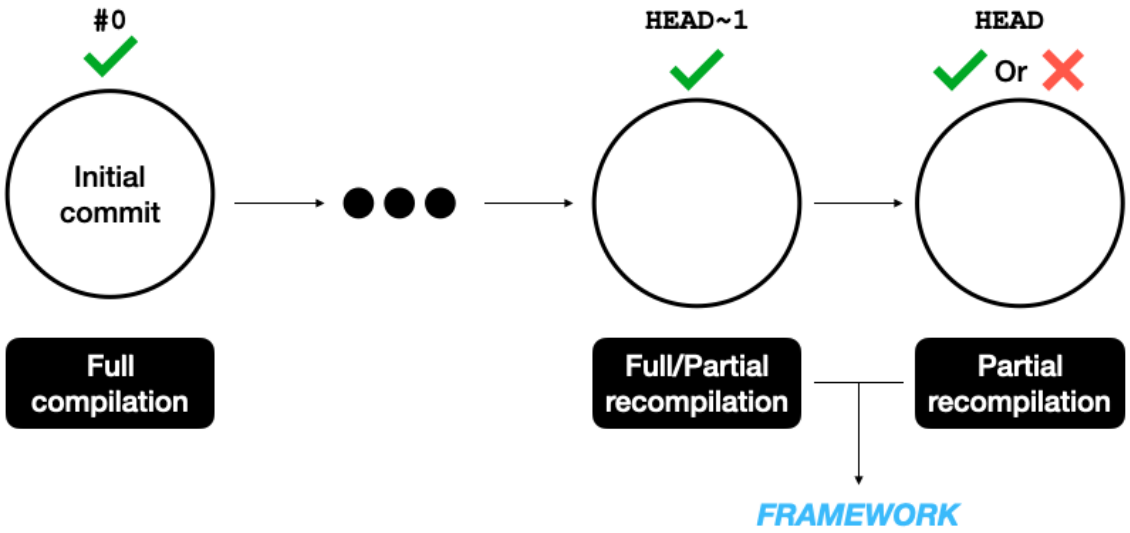


Figure 4.4: Framework application in Git history

no way to create dependencies of existing components on the new class without editing the existing components themselves. Therefore, if the addition of a new class file is the only change made within a commit, it will be the only file that is compiled in the next build job. Since it is an explicit change, it will always be compiled. Consequently, any potential break introduced with the addition of this file is guaranteed to be reported. This implies that the result of partial compilation will never miss a break, and therefore always reliably provides a result identical to that of a build job with full compilation.

Change

The developer can also make changes to existing classes. The framework distinguishes two types of such changes. The first type of change is the one made to the "outer" definition of the class, such as its name or its definition in terms of extending a class. The second type of change is that made to the implementation of the class, in which the changes are made to its components - methods and fields. This section discusses the first type of change, while the second type of change will be discussed in 4.3.2.

When the outer definition of the class is changed, the decision of the framework in which files should be compiled again hinges on how the class is related to other classes in the class graph. An example such inheritance dependency graph can be seen in Figure 4.2a.

- **Changing a class that has no dependencies or only incoming dependencies**: If a class definition is changed and the class itself has no other classes dependent on it, i.e. no incoming edges in the dependency graph, only the changes class itself is marked for compilation. For demonstration, given a dependency graph in Figure 4.2a, if the name of class Driver is changed, only that class file would be recompiled, as there are no other files dependent on it.
- **Changing a class that has outgoing dependencies**: In case the class that is changed has sub-classes, the class itself is marked for compilation as well as all sub-classes. This is because the implementation of sub-classes depends on the changed super-class, therefore, their validity may be compromised and must be checked in the build job. For example, given the dependency graph in Figure 4.2a, if the name of class Vehicle is changed, its file would be marked for compilation as it contains an explicit change, but with it, the file of class Car would also be compiled as it is inheriting from class Vehicle.

In summary, if there are no classes that inherit from the changed class, only the parent file of the class that has changed is marked for compilation, as it contains an explicit change. In case there are class that inherit from the changed class, they are also marked for compilation. Note again that this only applies to changes made to the outer class definition, for example class access level, class name and its inheritance relations. Changes to the internal implementation of classes is more nuanced and therefore the framework analyses these changes on a lower granularity level, particularly, method- and field-level,

discussed in 4.3.2.

Deletion
Similarly to the case of class changes, when class deletion is encountered in the commit, the framework constructs the partial compilation sequence based on which of the following situations applies to the circumstances. In summary, deletion of a class will only cause its former sub-classes to compile in the build job, while in the other cases, no files will be compiled. Note again that the other dependency leveraged by the framework hinges on the use of methods and fields. In case that the methods and fields inside the deleted method are referenced by other components, it is addressed with a specific approach explained in 4.3.2.

4.3.2. Methods & Fields

Additions, changes and deletions of methods are addressed in the same way as they are for fields. Therefore, in this section, we discuss the use cases that apply to both of them. For simplicity, we refer to methods and fields as components.

It is also important to note that when a change is made in a component, it is reflected in both the inheritance dependency graph and the component dependency graph, unlike in the case of changes to classes, where only the inheritance dependency graph is used to determine partial compilation.

Addition
Given that additions of new classes that may include components have already been discussed in 4.3.1, this section explains the case of adding a component to an existing class.

In case of adding a component, the same mechanism applies as it does with adding a class. In simple terms, if a component is added to the class, the parent file of that class is explicitly changed, and therefore automatically marked to be compiled in the subsequent CI build job. While this component can change the dependency graph when being added, for example, by calling an existing method from an existing class, this process can only make the added method dependent on other components, not vice versa. To create such dependency, the other component would have to be changed, implying that its parent file would also be explicitly changed, hence marking it to be compiled as well. As a result, the parent file of the component would always be the only file marked for compilation.

Change
In case of an existing method, the developer can decide to make changes to the method. The framework treats the whole method as one unit, such that if a change is made anywhere in the method, the method is marked as explicitly changed in the dependency graph. This also implies that in practice, whether a change is made in the signature, i.e. the input parameters or the return type of the method, or a change is made inside a method's body, the outcome for the framework is the same - the method has simply been changed, and therefore it is marked for compilation.

Rather than the nature of the change made, what essentially determines the partial compilation sequence is the changed component's connections in the dependency graphs. To determine which files must be compiled given a change is made to a certain component, the same procedure of graph walking is applied as it is in case of class changes. However, instead of only traversing the inheritance graph, the framework traverses the component dependency graph as well. An example of how this works is depicted in Figure 4.3. First, the component that has changed, in this case the $Car.startDrive()$ method is marked in the component dependency graph. Subsequently, the class Car, which has been affected by the change made in its method, is marked in the class dependency graph. Both graphs are traversed from the explicit change to gather implicit changes. Finally, the affected files are collected and marked to be compiled.

Deletion
Deletion of a component entails the removal of all lines that define the component. Identically to when a change is made to a component, the selection of files to be compiled in case of a component deletion is determined by the dependencies of the deleted component. Therefore, the framework approaches deletion in the same way as the changes explained in 4.3.2.

4.3.3. Combination of Changes

Developers can also make multiple changes in a commit, and these changes can be of multiple different types. For example, a developer can change a method in class A as well as delete class B in the same commit.

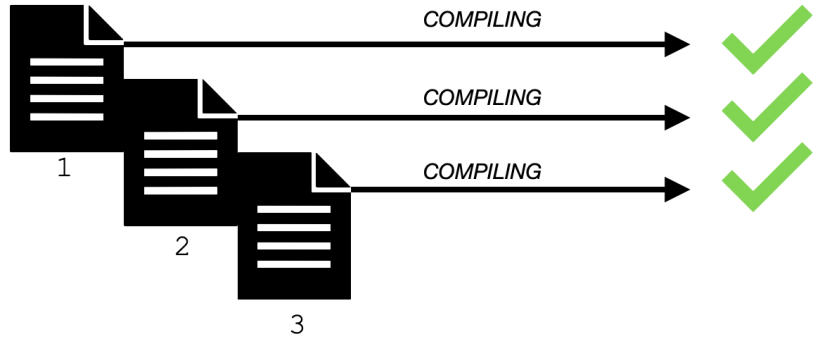
The framework also has the capability to process such multi-change commits. It does so by targeting each change, one at a time. Given the pipeline shown in Figure 4.3, the whole pipeline would be run for each change in the commit. When all changes are processed, all files that were marked for compilation in each pipeline are gathered into a set. Finally, the build job is instructed to compile this accumulated set of files, targeting the potential breaks of each change within the one CI pipeline run.

4.3.4. Experimental Setup

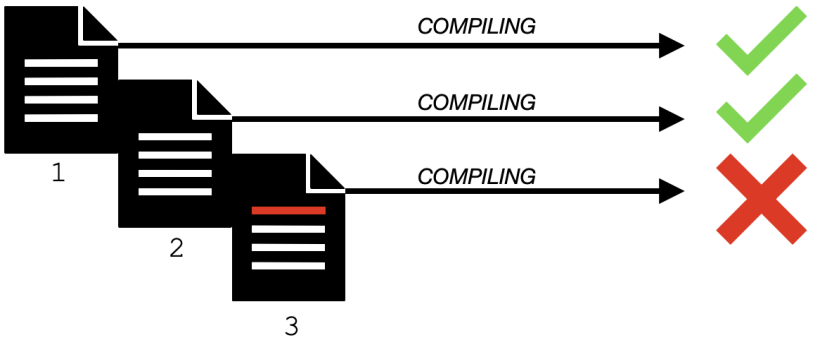
The first part of the composite research question addressed by this thesis concerns itself with the reliability of the approach of partial compilation construction explained in this section. Given the Git-related and change-based use cases, the validity of the approach in the defined scope can be evaluated by targeting these use cases. This subsection explains how the experiments for the validity were devised and how the results of these experiments aid in answering the first sub-question.

Build Outcome

Given the description of the mechanism used internally within the framework, there are two main distinctions to be made with regards to testing the validity. The first one is the outcome of the build job, which can be either a passing, or a failing result. For demonstration, an example of the two outcomes of compilation is depicted on Figure 4.5.



(a) Successful compilation



(b) Failed compilation

Figure 4.5: Two outcomes of the build job

In this sample code base, there are only three files for simplicity, marked 1, 2 and 3. Figure 4.5a shows a build job with a passing outcome, where all three files are compiled successfully. Figure 4.5b shows a build job with a failing outcome, where a mistake in file 3 leads to a failing compilation, which translates into the build job failing. To explain the relevant cases for checking the validity of the framework, it is important to compare the situations from both the baseline and the optimised build job execution, or full compilation and partial compilation respectively. In the passing case, both the full compilation and

the partial compilation will always deliver the correct result. Regardless of the partial compilation file set, i.e. whether one, two or no files are compiled, all files will be compiled successfully. However, this is not guaranteed in the failing case. Since it compiles all files, full compilation is guaranteed to find the compilation-breaking fault present in file 3 in the failing case in Figure 4.5b. However, in case of partial compilation, the set of compiled files may not contain the faulty file *3*. In this case, if, for example, only files *1* and *2* would be compiled, the fault would be missed, leading to an invalid passing result. With that, to evaluate whether the used partial compilation technique delivers a valid, reliable result, it is only relevant to test whether the partial compilation can catch the same faults as the full compilation approach.

Category of Breaking Change
Another categorisation relevant to validity evaluation of our partial compilation approach is the categorisation of changes. There are two main types of changes differentiated by the framework. The first one are the explicit changes, which are all the changes done by the developer and reported in the commit information. The second category includes all changes that have not been made directly by the developer, but are rather the effects of the explicit changes made through their connecting dependencies. As explained earlier in this chapter, to ensure all explicit changes are checked in the next build job, all files that were explicitly edited by the developer are automatically marked for compilation in the next CI build. Since we narrowed down the experiments to check validity of build failures only, we need to look at two types of breaks - those made explicitly and those that arise in implicit changes. If a break appears in an explicitly changed file, this break would always be checked in both full and partial compilation approach, as all explicit changes are automatically compiled. Therefore, in this case, partial compilation would always give a valid result - identical to the one produced with full compilation. This leaves the breaks made by implicit changes. Full compilation would always find catch such a break in the code base, as it compiles all source files. However, in case of partial compilation, what must be tested is whether the breaks in implicitly changed files are caught. This is an essential validity check, as it will show whether the partial compilation technique applied in the proposed solution manages to capture all relevant implicit changes.

Category of Explicit Change
Finally, it is important that the framework can detect faults in all the listed use cases. Given the scope presented in terms of what changes are processed by the framework, the set of experiments must cover each use case listed. To demonstrate the ability of the framework to detect different types of dependencies, the experiments test for implicit breaks inflicted through vertical and horizontal dependencies separately.

Recalling the use cases, the scope covers additions, deletions and changes made to classes, methods and fields. Changes done to methods and fields are addressed in the same way, however, for completion, they are evaluated as separate categories. What is omitted from the tests is addition of classes and components, as addition cannot introduce breaks in implicitly changes files. With that, the experiments must cover deletion, implicit breaks by vertical dependency, and implicit breaks by horizontal dependency for classes, methods and fields.

To summarise all mentioned requirements, all validity experiments to be conducted in order to answer RQ1a are presented below:

- **Classes**
  - Implicit break by deletion
    - Vertical dependency
    - Type dependency
  - Implicit break by change
    - Vertical dependency
    - Type dependency
- **Methods**
  - Implicit break by deletion

        - Vertical dependency
        - Horizontal dependency
    - Implicit break by change
        - Vertical dependency
        - Horizontal dependency
- **Fields**
    - Implicit break by deletion
        - Vertical dependency
        - Horizontal dependency
    - Implicit break by change
        - Vertical dependency
        - Horizontal dependency

4.4. Sample Project for Evaluation

To evaluate the validity of the framework by conducting the experiments listed above, we created a project that contains the necessary data structures and dependencies between them.

For simplicity, the project features three classes in their respective file, namely $Vehicle$, Car and $Driver$ class. The sample project code is shown in Figure 4.6. Its UML diagram and the corresponding framework dependency graphs are shown in Figure 4.7.

The sample project was specifically designed to be light-weight and fit the needs of the experimental setup. To fulfil the requirements for experiments, there is exactly one link made between classes and components for each use case.

From Figure 4.7b, we can see that there is a vertical dependency created by having class Car inherit from class $Vehicle$. For simplicity, in the graph, the only class that lists its methods is the $Vehicle$ class, as the information we extract from this graph is the inheritance relationship as well as the super-class implementation, namely the field $isMoving$ and the method $startEngine(fuelType)$, such that we know which methods defined on the super-class $Vehicle$ can affect its sub-class Car. The inheritance on its own enables us to test whether the framework can catch a break caused by the super-class being changed or deleted. The use of the inherited field $isMoving$ and the method $startEngine(fuelType)$ in class Car enables us to evaluate how the framework addresses breaking deletion and changing of components with vertical dependency.

Finally, Figure 4.7c is more relevant for the experiments related to horizontal dependencies, i.e. when a field or a method are used by another component. Since the method $Driver.driveCar(car)$ uses the $Car.model$ field and also calls the $Car.startDrive()$ method, this enables us to test whether breaking deletion and change of these components are caught by the framework.

Note that for the experiments, some dependencies may be dropped purposefully to isolate different use case experiments.

Listing 4.1: Car class

```
package dummy;

class Car extends Vehicle {
    String model;

    {
        System.out.println("Static initializer: Value of x in superclass: " + isMoving);
    }

    @Override
    public void startEngine(String fuelType) {
        super.startEngine(fuelType);
    }

    public void startDrive() {
        System.out.println("Starting the drive...");
    }

    public void stopDrive() {
        System.out.println("Stop the drive...");
    }
}
```

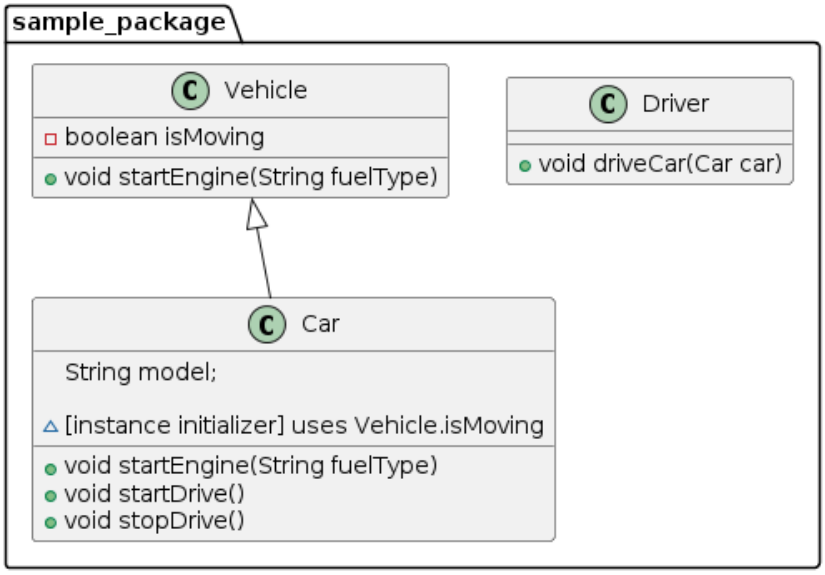
Listing 4.2: Driver class

```
package dummy;

class Driver {
    Car car;

    public void driveCar(Car car) {
        System.out.println("Driver is driving the car...");
        car.startDrive();
        if (car.model == "Q7") {
            System.out.println("This car is cool.");
        }
    }
}
```

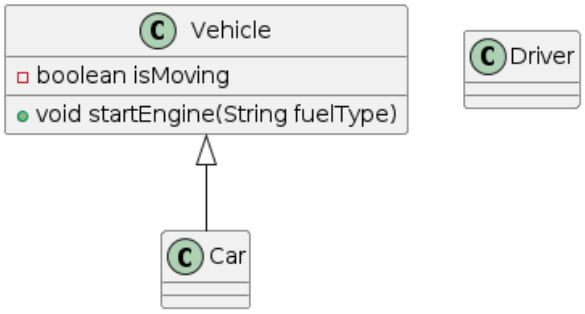
Listing 4.3: Vehicle class

```
package dummy;

class Vehicle {
    boolean isMoving;

    public void startEngine(String fuelType) {
        System.out.println("Vehicle is starting engine with " + fuelType);
    }
}
```

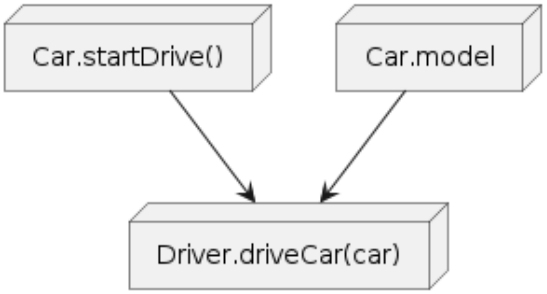
Figure 4.6: Sample project code base



(a) UML of the sample project



(b) Inheritance-derived graph for sample project



(c) Caller graph- & AST-derived dependency graph for sample project

Figure 4.7: UML and dependency graphs for sample project

5

Implementation

This chapter explains the technical aspects of the solution in detail. This chapter focuses on explaining how the fundamental concepts of the proposed solution outlined in the previous chapter are implemented in practice.

Before explaining the implementation of the framework, it is important to recall our assumption of full compilation being done before we use the framework. Given that the code base has been fully compiled and that since this full compilation there have only been passing pipelines, we can guarantee the validity of the code base in its latest version.

For a better understanding of how the framework operates in practice, its integration into the developers interaction with Git and the CI process is shown in Figure 5.1.

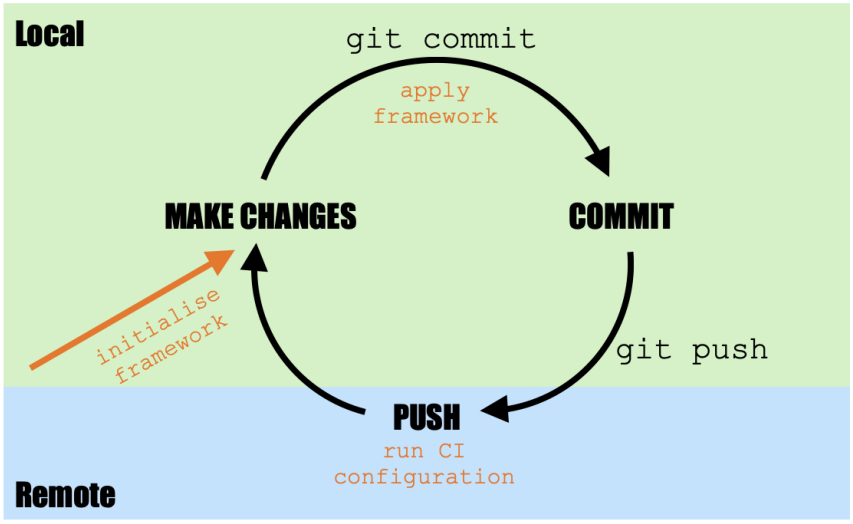


Figure 5.1: Framework integration into the developer-CI process interaction scenario

Before the framework is used to enable partial compilation in the CI pipeline, it must be initialised within the code base where it is to be applied. This is a one-time entry point for the framework which must happen before any changes to the latest version are made. This initialisation moment enables the developer to then use the framework in the cycle of making changes, committing them, and pushing them to the GitHub platform which employs GitHub Actions to execute the CI checks.

After the initialisation, changes are made to the code base by the developer. When they are satisfied with their changes, they perform the $gitcommit$ command to save the new state of the code base. It is at this moment that the framework applies itself, determines the partial compilation configuration for

the CI process and updates the commit data to reflect this. This entire process is taking place on the local machine used by the developer. Later, by performing the $gitpush$ command, the developer-made changes and the framework-made configuration for the CI process are uploaded to the remote repository on GitHub, where GitHub actions executes the CI pipeline process. The receiving of result of the CI feedback closes an instance of such cycle. For the next commit, since the framework has been initiated and it has saved the state of the latest commit, the developer does not have to initialise again, nor take any extra action to make use of the framework in the future cycles.

5.1. Framework Implementation

The pipeline of the framework in terms of its inputs, outputs and the internal components is shown in Figure 5.2.

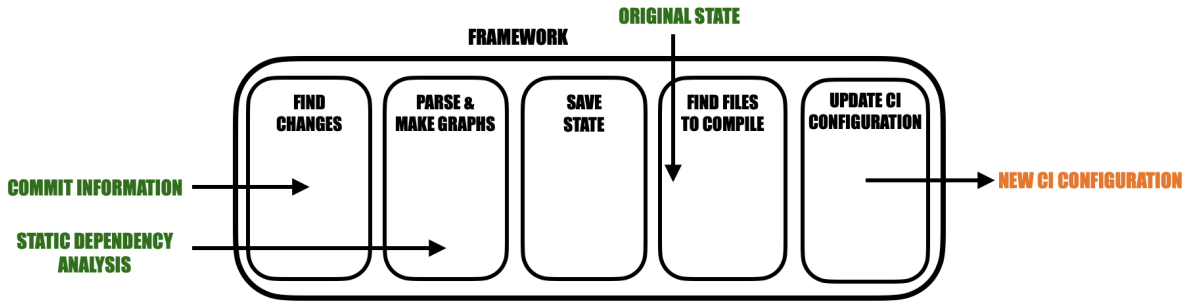


Figure 5.2: Framework pipeline: inputs (*green*), outputs (*orange*) and internal processes

As can be seen from the figure, the framework must first collect the commit information and the results of the static dependency analysis (SDA) to be able to function and product the partial compilation sequence in form of a list of files to compile. This section will explain each component of the implementation in greater detail.

All components of the framework are written as Python scripts. This architecture was chosen to allow for modular changes, as is explained in the subsections below.

5.1.1. Inputs

The two inputs the framework must get prior to being applied are the commit information and the static dependency analysis. By commit information, we mean the data we can extract from the commit, particularly the names of changed files and the types of changes made to them. This includes information about whether a given file has been added or deleted, or in case of changed files, which lines were added or deleted. For this implementation, we use Git and its interface to extract this information. For the static dependency analysis, in this thesis, we make use of Doxygen, a dependency analysis tool suitable for Java-based projects [6].

The components dedicated to processing each of the inputs are separated. This is to allow developers to change the input sources, for example change the static dependency analysis tool to some other tool than Doxygen. To do so and still be able to use framework, they would have to change only the component that processes that input source, for the example above, they would need to adjust the parsing process to fit the format of the incoming data.

5.1.2. Finding Changes

To extract the changes made within the commit, we use the Python $Repo$ library which allows the script to communicate with the local Git system. Encapsulated in the $diff\_extractor.py$ file, the main purpose of the script is to find all changed files and all changed lines within them. As the $Repo$ library can access the commit information directly from the code, no explicit input is necessary. To satisfy the use cases within our scope, we do not differentiate between line additions and deletions, and simply find the range of lines that have been changed. The result of this component is a Python dictionary object, where the keys are the names of the files changed, and the associated value for each key is the set of line numbers that have been changed within the file.

5.1.3. Parsing & Construction of Graphs

In the parsing phase, the results of the SDA are processed and translated into data structures which model the classes, methods and fields and the connections between them. It is this phase in which the SDA is mapped into the dependency graphs that are later used together with the changes made to determine which files must be compiled. The parser, encapsulated in the $sda_parser.py$ file, takes as input the XML files produced by Doxygen, where each class has its respective documentation file. In these files, Doxygen encodes the information about the classes themselves as well as their super-classes (or sub-classes) to inform on vertical dependencies. It also contains documentation of horizontal dependencies, for example, for each of its methods, the class file documentation also lists its callers across itself and other classes. As well as these dependencies, these files store information on types that are referred to by components, for example, if a field has a custom type, there is an explicit reference to that type in this XML documentation.

To construct the dependency graphs, the framework makes use of two custom objects, the $Class$ and the $Component$ object for the inheritance-derived and the caller graph- and AST-derived graphs respectively. Since the framework treats class methods and fields similarly, the $Component$ class is used to represent both these components. The UMLs for these two classes are depicted in Figure 5.3.

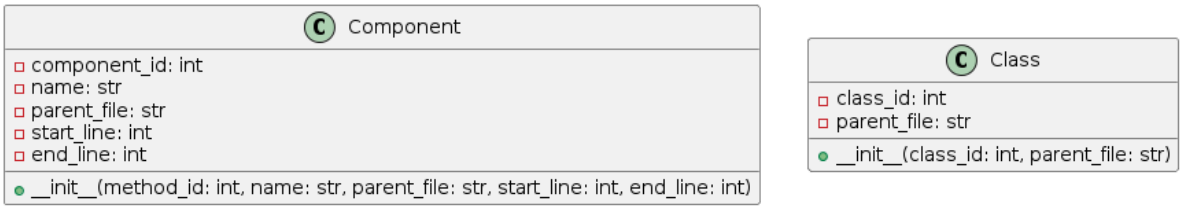


Figure 5.3: Custom objects for the technical representation of the dependency graphs

Note that the two attributes shared by $Component$ and $Class$ object are the $component\_id$ or $class\_id$ respectively, and the attribute called $parent\_file$. The IDs of both objects are necessary for identification within the graphs, and the path of the respective parent file is needed to easily add the relevant file to the partial compilation sequence whenever the class or the component are marked for compilation. In terms of changes to a class, the framework does not differentiate between the changes based on where in the class have they occurred. Therefore, it is sufficient to only store the corresponding file, as any change in the class will lead to marking the parent file for compilation. However, to maintain a finer-granularity when it comes to components, the $Component$ class contains two additional pieces of data - the $start\_line$ and the end_line of its declaration, which respectively hold the number of the first and the last line within which that component is declared. Having this information, the framework can later map the lines that may have changed to the exact lines where the components occur in the code base, and use this information to identify the implicit changes.

The parser sequentially reads all the provided XML files that contain class documentation and subjects these XML files to further processing. From each file, the parser first extracts the class definition and creates the corresponding $Class$ object and with that also extracts its inheritance-related dependencies, i.e. its super-class or its sub-classes. Following this, the parser continues to analyse the code in greater depth by parsing the data on existing methods and fields within that class. When reading this data, it also extracts the horizontal dependencies for each component, i.e. the callers of methods and the components referencing fields, as well as the type dependencies of fields and methods. Finally, when all files are fully processed, the parser outputs five different data structures, particularly dictionaries, or maps. Two of the dictionaries are the $class\_dict$ and the $component\_dict$, which map the IDs of classes and components to their corresponding $Class$ and $Component$ object respectively. The other three dictionaries track the dependencies - the $class\_dependency\_dict$ and $component\_dependency\_dict$ store the dependencies between classes and components respectively. The $type\_dependency\_dict$ holds the dependencies between classes, or types, and components. The keys and the values of the class and component dependency graphs are the class and component IDs respectively. Given a key ID of a class k, and the set of values that map to this key, which are the IDs of a class $v_1, v_2, ..., v_n$, all these classes are dependent on class k. Same applies in case of the $component\_dependency\_dict$ and $type\_dependency\_dict$. This enables a simple look up of a class' or a component's information

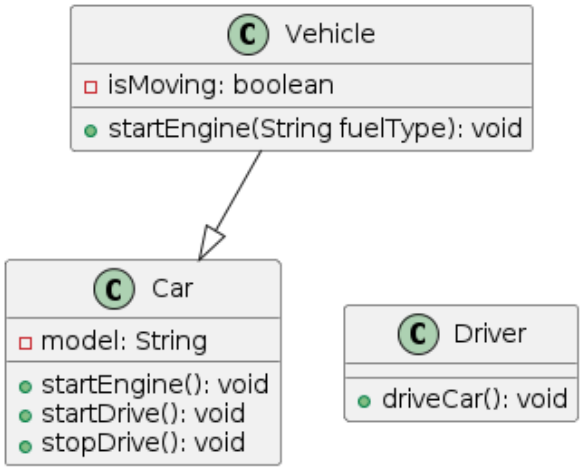
as well as its associated dependencies. Essentially, these five dictionaries together model the directed dependency graphs explained in **??**. An example of how the *class_dict* and the *class_dependency_dict* together model the inheritance dependency graph is depicted in Figure 5.4.

```
class_vehicle_id : Class<Vehicle>
class_car_id     : Class<Car>
class_driver_id  : Class<Driver>
```

(a) *class_dict*: ID-to-Object class dictionary

```
class_vehicle_id : [class_car_id]
class_car_id     : [ ]
class_driver_id  : [ ]
```

(b) *class_dependency_dict*: ID-to-IDs class dependency dictionary



(c) Inheritance graph

Figure 5.4: Framework-built class dictionaries translating into the corresponding inheritance graph

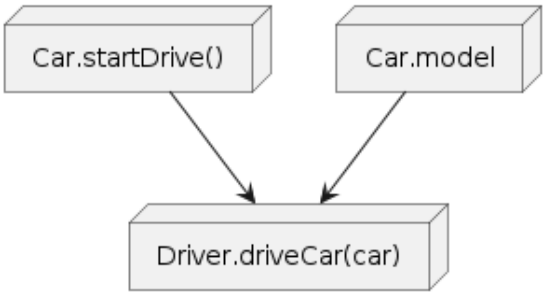
In this example, we have three classes in the code base: *Vehicle*, *Car* and *Driver*. In the *class_dict* in Figure 5.4a, the IDs of the classes represent the keys (on the left-hand side) that map to the corresponding *Class* object. Then, we have the Figure 5.4b dictionary, in which each class ID maps to a set of class IDs. In this case, the class IDs that correspond to *Car* and *Vehicle* class are empty, indicating that there are no classes that inherit from them. The *Vehicle* class ID maps to the ID of class *Car*, indicating that *Car* inherits from *Vehicle*. Connecting the information contained in the dictionaries, this data can be mapped into the inheritance graph in Figure 5.4c and vice versa.

5.1.4. Saving the State

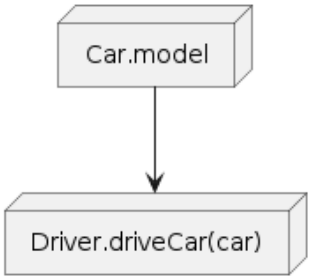
While this step of the pipeline is not directly necessary to the commit being currently processed by the framework, saving of the state is necessary to perform to ensure correct processing of the next commit. In this case, the state refers to the classes and components in the code base as well as their dependencies. Therefore, in implementation, this refers to creating a persistent storage for the *class_dependency_dict*, *component_dependency_dict* and *type_dependency_dict*, such that this data can be reused by the next execution of the framework. This function is encapsulated in the *create_pickle.py* script, in which these four dictionaries are "pickled" using the Python module *pickle*. This process essentially serialises Python objects, and since dictionaries are considered objects as well, pickling the dictionaries converts them into a byte stream which is saved in a file with a *.pickle* extension. Later, when the original state of dependencies are reused, this file is "unpickled" and the byte stream is deserialised into the three dependency dictionaries dictionaries such that they can be used in the code again.

Conceptually, saving of the state is an important part of the process, because the performance of the framework hinges on comparing the original state, i.e. before the commit changes have been made,

and the latest state of the code structure and dependencies. This is particularly aimed at the deletion use cases. If deletion occurs in the current, whether it is of a class or a component, taking into account only the latest state is not sufficient. This is due to the fact that if the state is fetched after such deletion, the new state will not be aware of the deleted class or component, neither will it reflect the previously existing dependencies. This situation is visualised in Figure 5.5



(a) Dependency graph before deleting the $Car.startDrive()$ method



(b) Dependency graph after deleting the $Car.startDrive()$ method

Figure 5.5: Changes in state and the resulting loss of information

In this example, assume that the only change in the current commit would be the deletion of $Car.startDrive()$. Given that there are no other changes made in the commit, it means that the effects of this deletion have not been corrected in the method that calls the deleted method - $Driver.driveCar(car)$. As this method depends on the $Car.startDrive()$, it should be marked as an implicit change, and should be compiled to check for potential breaks. In the situation where we would only consider the state of the code base as it is in Figure 5.5b, the framework is not aware that there is a method $Car.startDrive()$. Furthermore, it also loses the notion of the dependency between the deleted method and $Driver.driveCar(car)$ method, and as a result, falsely does not mark it for compilation, leaving a potential break in the $Driver.driveCar(car)$ method unchecked. For this reason, deletion, and in some cases changes, can only be performed correctly when the framework knows the original state of the code base and compare it to the latest state, ensuring that components and dependencies lost between the two code base versions are taken into consideration when constructing the partial compilation sequence.

5.1.5. File Selection for Partial Compilation

The final stage of the framework's production of the partial compilation file set harnesses all the intermediate outputs of the stages before hand and processes them to find the correct files. Besides the class and component dependency dictionaries and the commit changes, for this phase it is also necessary to collect the state of the original version of the code, before the commit changes happen. For reference, this previous state either comes from the previous run of the framework, or in case of a cold start, it is produced in the initialisation phase. The algorithm for this final stage can be seen in Algorithm 1.

The core of the algorithm can be divided into four logical modules. We start by comparing the dependency dictionaries created from the latest code version, i.e. the state of the code base at commit moment, and the previously saved state. Here we compare the dependencies - if a dependency exists in the old state but does not exist in the new state, we ensure that the components that were the dependent

Algorithm 1 Identification of files for partial compilation in pseudocode

```
1: Input: changedFiles, changedLines, classDict, componentDict, oldClassDependencyDict, oldComponentDependencyDict, oldTypeDependencyDict, classDependencyDict, componentDependencyDict, typeDependencyDict
2: changedComponents ← []
3: changedClasses ← []
4: filesToCompile ← []
5: // Step 1a: Compare class dependencies from past to latest
6: for all classDependency in oldClassDependencyDict do
7:    if classDependency not in classDependencyDict then
8:       for all dependentClass in oldClassDependencyDict (recursively) do
9:          add dependentClass.ID to changedClasses
10:      end for
11:   end if
12: end for
13: // Step 1b: Compare component dependencies from past to latest
14: for all componentDependency in oldComponentDependencyDict do
15:   if componentDependency not in componentDependencyDict then
16:      for all dependentComponent in oldComponentDependency (recursively) do
17:         add dependentComponent.ID to changedComponents
18:      end for
19:   end if
20: end for
21: // Step 1c: Compare type dependencies from past to latest
22: for all typeDependency in oldTypeDependencyDict do
23:   if typeDependency not in typeDependencyDict then
24:      for all dependentComponent in oldTypeDependencyDict (recursively) do
25:         add dependentComponent.ID to changedComponents
26:      end for
27:   end if
28: end for
29: // Step 2: Given the changed files, find all classes and components
30: for file ∈ changedFiles do
31:   fileComponents ← findComponentsInFile(file, componentDict)
32:   for line ∈ changedLines[file] do
33:      for component ∈ fileComponents do
34:         if line overlaps with component then
35:            add component.ID to changedComponents
36:         end if
37:      end for
38:   end for
39:   fileClass ← findClassOfFile(file, classDict)
40:   add fileClass.ID to changedClasses
41: end for
42: // Step 3: Analyze implicit changes in components and classes (recursive)
43: for changedComponentID ∈ changedComponents do
44:   find all dependentComponents of changedComponentID in componentDependencyDict (recursively)
45:   for dependentComponentID ∈ dependentComponents do
46:      add dependentComponentID to changedComponents
47:   end for
48: end for
49: for changedClassID ∈ changedClasses do
50:   find all dependentClasses of changedClassID in classDependencyDict (recursively)
51:   for dependentClassID ∈ dependentClasses do
52:      add dependentClassID to changedClasses
53:   end for
54:   find all dependentComponents of changedClassID in typeDependencyDict (recursively)
55:   for dependentComponentID ∈ dependentComponents do
56:      add dependentComponentID to changedComponents
57:   end for
58: end for
59: // Step 4: Find parent files for each list of changes and add them to filesToCompile
60: for changedList ∈ [changedComponents, changedClasses] do
```

ones in the lost dependency are marked for compilation. Then, framework finds which components and classes have been changed by iterating through the changed files and matching them to $Class$ objects, and then iterating through changed lines to $Component$ objects. All classes and components that were changed are then marked in the dependency graphs. For each marked class, all class IDs that map to it in $class\_dependency\_dict$ are marked as changed. Same happens for the component and type dependencies with their respective $component\_dependency\_dict$ and $type\_dependency\_dict$ structures. Finally, when all explicitly and implicitly changed classes and components are gathered, we find their respective parent files which together form the list of files to compile.

5.1.6. Editing the Configuration

For the partial compilation to be executed when building, the configuration of the CI process has to be edited before it is executed.

In the framework, we use the Maven $pom.xml$ to configure the compilation process. After the partial compilation is determined and the framework produces the list of files to be compiled, it passes the this list to the module called $configuration_editor.py$. This script edits the $pom.xml$ of associated with the code base by finding or creating the necessary XML tags to configure the Maven Compiler Plugin and overrides the full compilation by listing the files selected to be compiled.

This is the final stage of the framework pipeline. After this step is executed, the configuration in the $pom.xml$ together with the code changes made in the commit is uploaded to GitHub where the CI pipeline is triggered and executed by GitHub Actions with the amended configuration.

6

Experimentation

This chapter explains the set up that has been produced in order to run CI execution experiments. It describes what an experiment entails, as well as the experimental environment in terms of hardware and configuration.

6.1. Experimental Pipeline

In our study, an experiment entails a set number of executions of the CI pipeline on a specific commit, or in other words, a specific version of the code base. Our experiments are initiated by an automated pipeline which pushes the specific commits to GitHub, which triggers the GitHub Actions-hosted CI execution.

We run each experiment in two different versions. The first version is the *baseline*, which refers to the execution of an experiment without applying the framework. In the other version which we refer to as *optimised*, we apply the framework which updates the CI configuration before pushing it to GitHub and running the pipeline. This is an essential part of our experimentation, as we are ultimately comparing the results of the optimised version, in which we use our framework, to the baseline version of the CI execution, in which we do not apply the framework.

Note that the every aspect of the experiment is identical for both runs, except the $pom.xml$ file, as in the optimised version, the framework edits the $pom.xml$ configuration before finalising the commit. This means that the YAML file that defines the workflow for the CI pipeline remains identical, as well as all other files in the code base.

6.1.1. Experiment Manager & Data Collector

To automate the running of experiments, we created an experimental tool that takes as input the unique IDs of commits (SHA) and runs experiments on these commits a custom number of times in the two versions - baseline and optimised. When a certain commit run is finished, the experimental tool collects specified data for our results. This must happen before the experiment is run the next time, as logs and artifacts for the commit-related CI pipeline are only stored for the latest run. After the data is collected and downloaded by the tool locally, the running of the experiments continues in this manner until all experiments are done. In the end, the results of the experiment, both the baseline and the optimised versions, are gathered in a specified directory, ready for analysis and post-processing.

6.2. Experimental Set-up

In this section, we navigate through the technical aspects of the environment in which we conduct our experiments.

6.2.1. Hardware

This section describes the details of the hardware used to perform the experiments. It is particularly important, as energy consumption estimation produced by EcoCI is heavily dependent on the hardware

specifications.

Runner
To maintain a consistent environment for our CI pipeline execution, we created a runner from an Ubuntu-based Docker image which was operating on a remote server. The Dockerfile that defines this Docker image creates the complete environment necessary for running the workflows used in our experiments. We opted for the ephemeral runner, which shuts down when a workflow job is finished, and automatically restarts itself to take on the next workflow job. Such runner is well-suited for the energy experiments we conduct in this thesis, as every runner execution goes through an identical cycle of start up, execution, and shut down.

The machine that runs the Docker image, informally known as GreenServer, For reproducibility, the mentioned Dockerfile is made available on GitHub [25].

Experiment Management & Data Collection
To mitigate the limitation of the GitHub Actions platform due to which artifacts from a given run are lost if another run is initiated, we combined the automated experiment manager with the data collector. This enabled us to collect data from a given workflow run immediately after that run was executed, and continue with re-running the experiments only when we have collected the data from the previous run.

The experiment manager and data collector scripts were run on MacBook Pro 15" 2019. Note that this machine was used to instruct the runner to run the experiments via GitHub API and collect the run-related artifacts via the same API. Since the runner is an independent machine, the validity and energy consumption data are independent from this device.

6.2.2. Energy Metrics
In some of the conducted CI execution experiments, our main focus is the measurement of energy consumption and other metrics that can capture potential improvements in efficiency. To record this data, we use the EcoCI GitHub Actions plugin, which measures these energy metrics during a pre-defined phase of the CI pipeline execution. We use this section to explain the four different metrics:

- **CPU Utilisation (%)**: The CPU utilisation, measured in percentages, is the ratio between the time that CPU is actively executing instructions and the total time. For example, if the total duration of the measured execution was 100 seconds, and the CPU utilisation would be 8%, it would mean that 8 seconds out of the whole execution time. Higher CPU utilisation may signify that the processing unit needs to use up more of its capacity to execute the instruction sequence as it must process the workload for a longer amount of time. On the other hand, a lower CPU utilisation percentage suggests that the currently executed instructions are not excessively straining on the available resources, and therefore may be more efficient.
- **Total Energy Consumption (Joules)**: The total energy consumption, measured in Joules (J), is the value that represents how much energy has been used to execute the measured experiment. The higher the total consumption, the more energy must be used to execute the experiment, while lower total consumption would signify that less energy had to be spent.
- **Average Power Output (Watts)**: The average power output value shows us how much power is produced by the system over a given period of time, in our case, during the experiment. If we compare this value between two different experiments, we can conclude which experiment is more energy-efficient relative to the other experiment.
- **Duration (seconds)**: The duration is simply the number of seconds it takes to execute the experiment - in our case, how long does it take to execute the build job within our CI pipeline. While shortening the duration of the build job is not the primary focus of this study, it is closely correlated with the total energy output, and can shed light on the overall workload executed in each experiment.

7

Results

This chapter introduces the results of the experiments that were executed to answer the presented research questions.

7.1. Validity in an In-Vitro Project

First, we show the results of the validity-focused experiments run on the sample project presented in **??**. Before showing the results of the experiments, we present a short description of experimental setting specific for these experiments.

7.1.1. Experimental Set-Up

The experiments are heavily dependent on the configuration of the workflow file containing the build job specification. The pom.xml and its configuration is closely linked to the workflow, as the pom.xml file holds the configuration for the Maven commands. However, in this case, for simplicity, we show only the workflow build command and explain what Maven life cycle phases it executes in practice based on the pom.xml configuration. The build job portion of the workflow file can be seen in Section 7.1.1.

With this command, we first execute the $mvnclean$ command to ensure a clean environment for the experiment, as this removes any cached content that may interfere with the experiments. This is followed by a $mvncompile$ command, written with the configuration $compiler : compile - fpom.xml$. This ensure that in the compilation phase, the build process uses the compiler plugin configuration defined in the $pom.xml$ file.

7.1.2. Validity Results

The RQ1a is concerned with the reliability or the validity of the CI process result when the proposed solution is applied. To answer this subquestion, we executed the experiments that were devised to cover the scope we defined for the proposed solution. These experiments are listed in **??**. In this section, we present the results that prove the validity within the defined scope, collected in Table 7.1. We also present the energy results from the 10 experiments conducted.

The table shows the meta data about the CI process of each experiment, comparing the baseline and the optimised results. The "Captured Break" column is the most important one in answering the research question, as it shows whether the partial compilation managed to catch the break in the code in the same way as the full compilation, which is the baseline. The last two columns show the number of classes that were compiled in the process during full compilation and the partial compilation respectively.

```
1         run: mvn clean compiler:compile -f pom.xml
```

Table 7.1: Comparison of Baseline and Optimised Results

Category	Experiment	Break Captured	Baseline # Classes	Optimised # Classes
Class	Deletion	✓	2	1
	Change	✓	3	2
	Type deletion	✓	2	1
	Type change	✓	3	2
Field	Deletion (horizontal)	✓	3	2
	Deletion (vertical)	✓	3	2
	Change (horizontal)	✓	3	2
	Change (vertical)	✓	3	2
Method	Deletion (horizontal)	✓	3	2
	Deletion (vertical)	✓	3	2
	Change (horizontal)	✓	3	2
	Change (vertical)	✓	3	2

7.2. Energy Consumption in Controlled Environment

While the first sub-question of our research questions focused mainly on the validity of the CI process outcome, we also conducted a preliminary study into the energy consumption of the pipeline during these experiments. The boxplot results for all four estimates - the CPU utilisation (in percentages), energy consumption (in Joules), power output (in Watts) and duration (in seconds) can be seen in Figure 7.1.

7.3. Energy Efficiency in Real Repositories

In order to answer the second sub-question, our experiments investigated whether the proposed solution can reduce the energy consumption in CI pipelines in real-world projects and their respective commits. This section describes the experimental set up for answering RQ1b as well as the results of the associated experiments.

7.3.1. Experimental Set-up

We conduct two experiments to investigate the impact of the framework on the CI pipeline energy efficiency. The first experiment focuses on compilation only, which leads to a simpler integration into an existing workflow and demonstrates the effect of the framework in the compile phase in isolation. In the second experiment, we apply the framework to the original command from the project's CI workflow, such that we can observe the effect the workflow may have in practice.

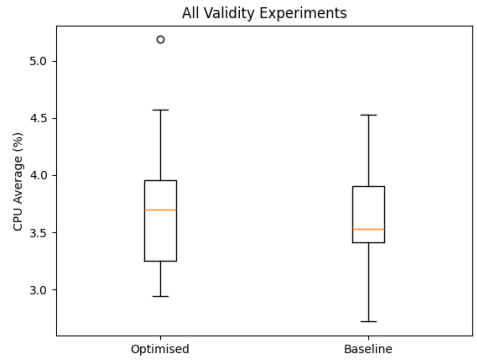
7.3.2. Real-world Project

The main aspect is meant to illuminate to what extent can the proposed solution reduce energy consumption in the CI process.

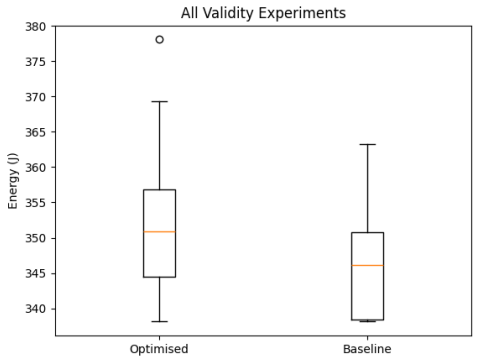
To evaluate the performance of the proposed solution in a real-world scenario, we chose an open-source project from the Apache Community with several requirements:

1. **The majority of the code is written in Java**: Since the workflow only considers Java source files, it is important to select a project with enough Java files to avoid unexpected situations and possible associated complications, such as erroneous partial compilation sequences due to missing dependencies and disregard for non-Java files.
2. **The project contains a GitHub workflow with Maven commands**: This is important as the framework exclusively uses the Maven package manager and its commands to configure the CI pipeline execution.
3. **The project's workflow contains at least one build-related command that compiles the source code**: A command that runs the compilation process is necessary such that we can observe the effect of the framework, as it targets compilation in particular.

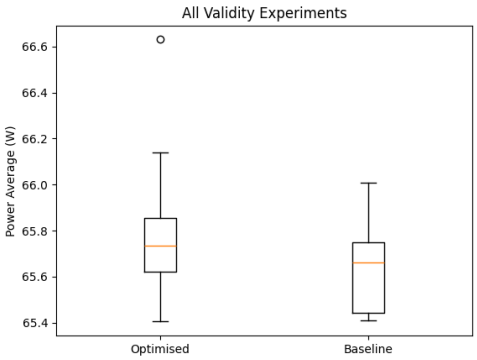
To satisfy these requirements, we chose an Apache Community project $commons - bcel$. With that, we



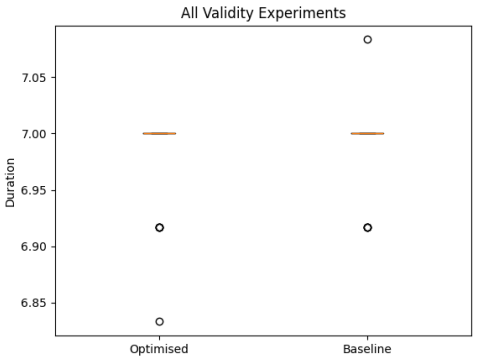
(a) CPU Utilisation (%)



(b) Energy Output (Joules)



(c) Average Power Output (Watts)



(d) Duration (seconds)

Figure 7.1: Four subfigures arranged in two rows.

```
1             run: mvn clean compiler:compile -f pom.xml
```

```
1             run: mvn --show-version --batch-mode --no-transfer-progress -DtrimStackTrace=
                  false -Ddoclint=none -Dcommons.japicmp.
                  breakBuildOnBinaryIncompatibleModifications=false
```

also chose a random passing commit from the repository's commit history.

7.3.3. Experimental Set-Up

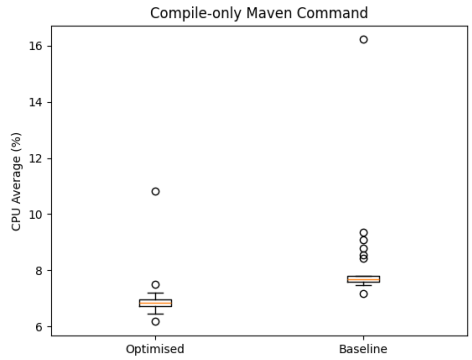
For each of the two experiments, the CI process is run with a different workflow command. The workflows for the compile-only experiment and the original command experiment can be seen in Section 7.3.3 and Section 7.3.3.

For the compoile-only workflow, we use a command identical to the one used for the in-vitro project experiments, shown in Section 7.1.1. It simply compiles the source files, specifying that the compiler plugin should use the configuration from the *pom.xml* file. The original command workflow is more nuanced. All the flags that follow the *mvn* command are meant to make for simple and clean logging of the build process within the CI environment, with the exception being the last two flags. These two flags override the *pom.xml* settings such that the Java 8 DocLint feature is disabled and therefore cannot break the build. The last flag is specific to the *japicm* Maven plugin, ensuring that if the plugin detects binary incompatibilities, it will not break the build, and therefore will not cause the CI pipeline process to return a failing result. The most interesting feature of this command in terms of energy consumption readings is the different Maven life cycle phases triggered within this process. While no specific Maven command has been included in the instruction, given the *pom.xml* configuration and the workflow command for building, the CI process will execute all life cycle phases up until the *verify* phase. This

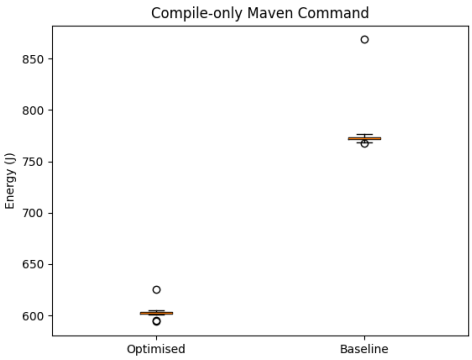
implies more workload in the latter experiment compared to the first, simpler experiment, where we only run the *validate* and the *compile* life cycle phases. The Maven life cycle is explained in greater detail in 2.

7.4. Energy Consumption in Practice

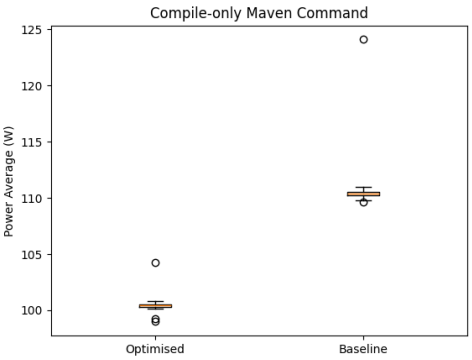
The results of the energy consumption of the compile-only experiment and the experiment with the original Maven command from the workflow file are shown in Figure 7.3 and Figure 7.4 respectively.



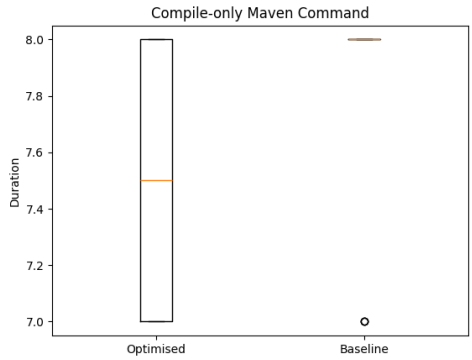
(a) CPU utilisation (%)



(b) Total energy consumption (Joules)

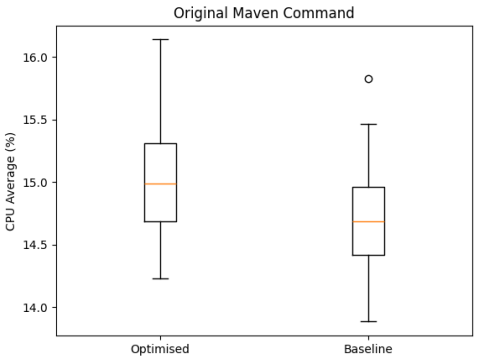


(c) Average Power Output (Watts)

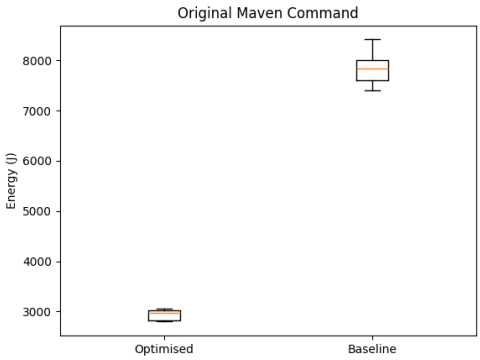


(d) Duration (seconds)

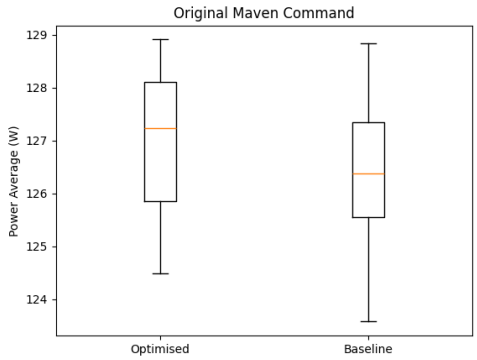
Figure 7.3: Comparing the optimised and baseline energy measurements for the compile-only workflow in common-bcel experiment



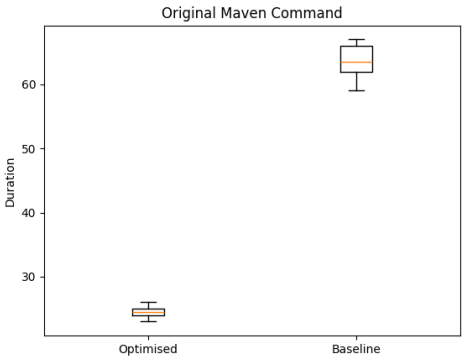
(a) CPU utilisation (%)



(b) Total energy consumption (Joules)



(c) Average power output (Watts)



(d) Duration (seconds)

Figure 7.4: Comparing the optimised and baseline energy measurements for the original workflow in common-bcel experiment

8

Discussion

This chapter analyses the results presented in the previous chapter in detail.

The experiments conducted in this study yielded results which were summarised in the previous chapter. To better understand the behaviour of the proposed solution, we use this chapter of the report to analyse the results.

8.1. Validity & Energy Efficiency in Controlled Environment

We first analysed the results of the RQ1a-related experiments in the sample project, or in other words, the controlled environment. We analyse the validity results as well as the energy results collected during the experiments.

8.1.1. Validity

In terms of validity, the Table 7.1 shows that all experiments devised to cover the use cases for the framework have produced a result identical to the full compilation. That is, the partial compilation solution has managed to catch the respective break of every particular experiment and as a result, it has also produces the same feedback as the baseline. The table also shows other meta data, such as the number of classes that are compiled by the proposed solution and the number of classes compiled by the baseline. Comparing these two columns shows that partial compilation is being employed successfully, as the partial compilation compiles less files than the full compilation without compromising the CI execution result.

8.1.2. Energy Consumption in an In-Vitro Project

In terms of energy consumption data from the sample project, the results show counter-intuitive numbers. Despite the fact that we compile less files in the optimised version of the CI execution, the total energy consumption, CPU utilisation and the average power output estimates are higher when the solution is applied. While the result may be unexpected, there are several reasons why the energy output of the optimised version of the framework may be higher than the baseline.

The first aspect which may be causing the problem may be bias in the measurements. When reviewing the graphs shown in Figure 7.1, it can be seen that the Y-axis scale is very granular, and that the differences between the mean values as well as the outliers compared between the optimised and the baseline version are minimal. For example, the CPU utilisation in percentages displayed in Figure 7.1a shows less than a 0.5% difference between the mean values for the optimised and baseline experiment, with the means being 3.75% and 3.6% respectively. If we compare the total energy consumption of the optimised and the baseline version of CI execution in Figure 7.1b, we can observe that the mean values of energy consumption are 351 and 347 Joules respectively. In case of the average power shown in Figure 7.1c, we can see even smaller, decimal-scale differences, with the optimised and baseline power output at 65.75 and 65.68 Watts respectively. However, if we compare the duration of the CI execution in Figure 7.1d, we can see that the mean time it takes to execute the build job is equal for the optimised

and the baseline version. In fact, the outlier values for the two experiments show lower outliers for the optimised version, where the duration could get as low as 6.84 seconds, while the lowest baseline outlier appears at 6.93 seconds. The baseline experiment also exhibits outliers above the mean time, where an execution took 7.08 seconds. While these differences may be insignificant in practice in case of all four measurements, the duration values show more favourable results for the optimised version while other values demonstrate more efficient result in the baseline version. This inconsistency in behaviour, especially between strongly related variables such as the total energy consumption and time execution, further strengthens the theory that there may be bias present in the measurements.

While bias may be a more likely explanation for the small difference between the optimised and baseline energy-related values, there is a possibility that optimised version may present additional overhead within the CI pipeline due to the additional configuration. When Maven reads the *pom.xml* file with the configuration produced by the framework, it must perform more parsing to read the files to include in compilation, and possibly also more processing later, in which it has to match the source files to the ones marked to be compiled during the compilation stage. While such overhead may be insignificant in a larger repository, in a small-scale project such as the one designed to test the framework's reliability in the defined scope, it may be adding more workload for the CI execution machine without a significant compensation in compilation-related energy savings.

Perhaps the most important take-away from these results is that the workload savings brought by the framework may not be significant in small-scale projects. In each validity experiment, we only skip one source file compilation process. These results show that in such cases, the framework is not yielding the desired results as it works on file basis. If a project contains a small amount of files to begin with, partial compilation does not yield significant benefits in terms of energy efficiency, if compared to full compilation.

8.2. Energy Consumption in a Real-World Project

To evaluate the benefit that may be brought about by the framework in a real-world scenario, this study has conducted experiments to simulate practical operation of the framework as reliably as possible. To do so, and to observe the effects of the framework in different extents of operation, we conducted two experiments - the compile-only workflow experiment and the original build command experiment. The results can be seen in Figure 7.3 and Figure 7.4.

8.2.1. Energy Consumption in Compilation

Since the framework targets compilation in the CI pipeline build job, we conducted the compile-only experiment to observe the energy efficiency improvements for this particular phase. As can be seen in all four energy metrics graphs in Figure 7.3, the proposed solution outperforms the baseline in each metric. An interesting contrast can be seen in the total energy consumption in Figure 7.3b. The partial compilation in the optimised version saves energy compared to the baseline, where the optimised and baseline mean energy consumption is 603.02 and 775.59 Joules respectively, bringing 22.2% improvement in energy efficiency. Similarly, we see that the recorded CPU utilisation is also lower in the partial compilation experiment, with the mean values being 6.96% and 8.11% respectively. The lower CPU utilisation value shows us that when the optimising solution is used, the CPU has less workload to process in general, leading to having the CPU execute instructions for less time. A closely related metric to the processing unit utilisation is the average power output of the experiment. The average power output of the optimised experiment with partial compilation was measured at 100.50 Watts, while the baseline experiment power output value was 110.80 Watts. A lower average power output in the optimised experiment indicates that the system consumed less energy, on average, over the measured period. This suggests that partial the optimised framework has effectively reduced energy consumption compared to the baseline. The last metric we measured was the duration in seconds. The optimised version execution being by 0.4 second faster on average compared to the baseline. While we would expect a greater duration improvement considering the closely correlated total consumption, the reason for a small improvement in this metric may be due to the fact that we cannot collect sufficiently precise time-related data due to its coarse granularity. The reporting used for the results can only measure the runs in full seconds, but no decimal points. This may mean that these results may not be capturing the smaller differences in duration brought about by the proposed optimisation, which may explain its

unexpectedly insignificant correlation with the total energy consumption. While the difference may not seem significant, the shorter duration in the optimised version still shows that less workload is being processed in the CI execution, which is a desirable direct effect of the partial compilation mechanism being utilised. The fact that the CI pipeline-executing hardware executes the build job faster while the other energy metrics are kept low, suggests that the partial compilation version is more efficient than the baseline version with full compilation.

8.2.2. Energy Consumption in a Full Build Job

To explore the possibilities of integrating the framework into existing workflows as well as its potential effect on the developer-defined build job as a whole, we also conducted an experiment in which the framework is applied to the original command in an existing workflow. This application has shown great potential for both energy consumption and execution duration improvements. The results are shown in Figure 7.4.

In the optimised version of this experiment, we observed a very significant improvement in the total energy consumption of the CI build job. Compared to the baseline, in which the mean value of total energy consumed was 7825.58 Joules, the optimised version saved around 62.47% of energy, with its mean consumption of 2938.52 Joules. Such a difference measured over 30 experiments shows that the optimised version using partial compilation approach is consistently performing more efficiently than the full compilation in the baseline version. For the duration of the build job, which is closely correlated with the total consumption metric, we observe a very similar trend. While the full compilation experiment took 63.4 seconds to complete on average, the optimised version exhibits greater efficiency, reducing this duration to a mean of 24.53 seconds, yielding a 61.3% improvement in time requirements of the CI build process. A counter-intuitive result was observed in the remaining two efficiency-related metrics, particularly the CPU utilisation and the closely correlated average power output. The CPU utilisation has been shown to be higher for the optimised version with a mean value of 15.1%, compared to the baseline version with a mean of 14.74%. Similarly, the power output overtime was measured at 126.88 and 126.38 Watts for the optimised and the baseline version respectively. This presents a result that appears to be opposing to the observations we gathered in the compilation-only experiment, in which we have seen the optimised version of our experiment exhibit a lower value for each of these metrics. Given these results and the results for the compile-only experiment, it appears that the behaviour of the CPU utilisation and the power output may be build command-dependent. The $mvn\ verify$ command has more instructions to execute than the $mvn\ compile$ command, which is likely the reason why the processing unit utilisation and the power output are higher overall for the latter experiment. Given the results of the two experiments with different commands, it is possible that the optimised version, which has an effect on the execution of Maven life cycle goals, may be adding extra workload and raising these metrics as a result. However, it is also important to note that the differences between the optimised and the baseline version's metrics in the compile-only experiments are more significant than those in the original command experiment. For example, while the average power output in the baseline is more than 10 Watts, or 10% higher in the compile-only run, the difference between the means in the original command experiment is only 0.5 Watts, or roughly 0.4%. Comparing these values, we see that the increase in CPU utilisation and power output is less significant in the original command experiment compared to the increase in these metrics in the compile-only experiment. Finally, while these metrics illuminate the efficiency of the hardware managing the given workload, our main focus throughout this study is the total energy consumption, which was improved by the optimised version in every conducted experiment.

8.2.3. General Conclusion on Energy Consumption

In both experiments, the compile-only and the original command experiment, we observed a higher total energy consumption by the build job, proving that the proposed solution utilising partial compilation has the potential to execute the CI process more efficiently.

The experiment where we integrated the proposed solution into the original workflow showed the greatest improvements. These results provide an enlightening insight into the capabilities our proposed solution has in terms of bringing efficiency on a full build job scale, including phases such as testing on top of compilation. The improvements are likely brought about by the fact that skipping compilation also includes exclusion of test files, and consequently the conducted tests. However, while these results

are an exciting perspective of the framework's potential in the future, it is important to build up to such integration through systematic research starting with the compilation-only experiment we conducted.

On the other hand, the compile-only experiment serves as a direct answer to the RQ1b research question, proving that in the compilation step of the build job, the proposed solution can reduce the workload conducted, leading to lower energy consumption as well as the duration of the compilation build step. We found that the solution can save 22% of the total energy consumption, which proves that the proposed solution is a viable concept in increasing the energy efficiency within the CI build process. It is possible that this number could be even higher for other projects, however, even if we consider the recorded savings and extrapolate this scenario to every commit, this solution can make a considerable leap towards making the build process more sustainable. Moreover, as we have seen in the original command experiment, the improvements made by the solution in the compilation stage may propagate and multiply the energy savings across other parts of the build job in the future.

8.3. Threats to Validity

Before drawing conclusions from the presented results, it is important to understand the various factors that may compromise the integrity and generalisability of the study. In the following section we examine the internal and external threats to validity and discuss how they can be mitigated.

8.3.1. Internal Threats to Validity

In this section we investigate the potential problems that may compromise the reliability of our results. We discuss different aspects that may be affecting the results of our experiment in ways that we cannot accurately account for. We examine these confounding variables and also present the employed strategies to mitigate their undesirable effects.

Energy Measurements

A major aspect of validity regarding the energy consumption results is the accuracy of its measurement.

Conducting only one run of an energy consumption experiment does not yield enough certainty in its validity. The measurement we conduct may be an outlier to the true underlying distribution of the energy consumption, which would give a false base for generalisation. As the energy consumption depends on a myriad of factors, some of which are difficult to quantify, we conduct several measurements to mitigate the bias. Specifically, we conduct each experiment with 30 repetitions and average the results to minimise the effect of potential outliers.

Energy consumption of software execution is strongly correlated to the hardware on which it is run, as well as the state of the hardware prior to running the experiments. If an experiment is run immediately after the machine boots, it is likely that the internal temperature of the hardware will be cooler, leading to less energy being consumed compared to the situation in which the hardware has been warmed up before. Therefore, before running the CI execution experiments, we include a set up procedure during which we push the base commit for our experiments and trigger its CI execution. This procedure sets the necessary state for the experiments, it also warms up the runner for the experiments upcoming experiments. Additionally, we also randomise the execution on experiment level. Specifically, for each experiment, we run each the optimised and the baseline version 30 times such that the order of execution of the 60 individual CI pipeline executions in total is randomised.

Another hardware-related aspect that could hinder the validity of the result is the length of pause between the experiments. From an experimental point of view, it is difficult to find the exact moment of CI execution termination, as the communication with GitHub Actions API is periodic and initiated from outside of the platform. This means that if we check for termination of the current experiment n every X seconds, and then set a pause time of Y seconds, the range of the pause between the termination of experiment n and the initiation of experiment $n + 1$ ranges from Y to nearly $X + Y$. This uncertainty of the real pause duration between experiments could lead to changes in the state of the hardware which we cannot account for [4]. With the purpose of mitigating this problem, we set the runner to be ephemeral, forcing it to shut down after each job and restart for the next one. This way, the runner shuts down exactly when a experiment is finished, and turns back on when another experiment is started, minimising the range of inter-experiment pause duration.

Finally, it is important to note that the energy consumption measurements provided by EcoCI are estimated by hardware specifications. If the CPU model of the used runner machine is not recognised by EcoCI, the tool generalises the attributes of the CPU such as Thermal Design Power (TDP) or the total number of threads. Since these attributes have great influence over the final estimate, such generalisation may yield results that are less precise. To mitigate this, we add the specification of the runner's CPU model into the EcoCI open-source project and use it to obtain more accurate results of the real energy consumption during CI pipeline execution.

8.3.2. External Threats to Validity

In this section, we examine the potential problems that with generalising the proposed solution.

Project Setup

While the underlying mechanism of the proposed solution leverages characteristics that could be extracted from other programming languages, applying the framework to a project with a different language does not guarantee the performance observed in this study. A valid use of the underlying partial compilation mechanism hinges on an accurate dependency representation obtained using static dependency analysis. With other languages, it is not guaranteed that the dependency analysis results will capture the same types of structures and their connecting links. In case it would fail to do so, the partial compilation may not be determined correctly, leading to invalid results. We targeted Java because of its popularity and widespread use over the last decades, with the purpose of ensuring that a large number of existing and new projects could apply the framework to increase the CI process efficiency.

Similarly, we used Doxygen as our static dependency analysis tool for the in-vitro and real-world Java projects. However, different SDA tools may have different capabilities and scopes when it comes to capturing the types of dependencies necessary for our solution. Moreover, different SDA tools may report results in different formats which require different. We encourage future users who may want to use a different SDA tool to first understand the tool's approach and working, as well as its results. To allow for this, we ensure that the module for parsing the SDA results is separated from the other logic in the framework's code base, allowing the developer to change the internal parsing, only needing to ensure that the protocol with the next layer is satisfied.

It is important to understand the scope of using our proposed solution, in terms of use cases in both the Git perspective and the code changes made in each commit. In terms of using Git, we can guarantee that making a new commit with a given set of changes is processed as documented and shown in the experimental setup. However, cases such as amending a commit, or creating a commit after a rebase or a merge operation, have not been tested within this study, and therefore can not be guaranteed to work as expected. For the change-related use cases, we have scoped the types of changes that the framework covers and processes, however, this list may not be exhaustive when it comes to all possible code changes.

8.4. Limitations

Here we list the known limitations of the proposed solution and its integration in practice. We list the limitations and how they may impact the use of the framework. Later, in the 9, we address how these limitations may be mitigated in the future.

8.4.1. Static Dependency Analysis

As we explained in the 2, static dependency analysis is a technique of obtaining the dependencies between different parts of the application code. However, while we do not need to execute the code to find these dependencies, there is a subset of potentially important dependencies that may be missed by static analysis. Past work has observed that when conducting static analysis, the resulting dependency graph do not contain some other dependencies that can be find only by running the code or using a form of dynamic rules. Therefore, the validity of the proposed solution hinges on the type of dependencies within the code, as we make partial compilation decisions based on only the dependencies which can be extracted statically.

8.4.2. Solution Integration

The proposed solution and the configuration it produces is strongly connected with the configuration file for the project. In our case, the two most relevant configuration files are the workflow file and the (parent) *pom.xml*, where the latter file can prove problematic in terms of integrating the solution. The *pom.xml* file contains the configuration for all project-related processes, including the behaviour during local and CI builds. Due to existing configuration and potential interfering plugins, there may be problems later on with Maven attempting to parse the configuration. Following is the list of problems we have encountered when setting up the experiments:

- When the *pom.xml* is edited, the license is automatically removed, often causing a failure in the build due to license enforcing rules
- Exclusion or inclusion of files from or in the compilation may already be present, potentially clashing with the compiler configuration devised by the framework
- If caching is used, it often has to be configured for every plugin used in the build configuration.

9

Conclusion & Future Work

In this thesis, we have shown that the proposed solution has the potential to bring notable energy savings into the CI process. It is not only the partial compilation that has shown its advantages. What is more, the techniques that together guide the file selection have managed to do so in a manner that, in the defined scope, captures an identical result to the baseline, which is the full compilation of all source files in the code base.

9.1. Future work

While this work is not the first one to provide viable solutions of removing redundant workload from the CI pipeline process, it enters this field of research in its emerging stages. To the best of our knowledge, it is the first work that focuses on improving the CI process with the emphasis on energy consumption and a strong commitment to guaranteed reliability of the results despite the lowering of the overall workload. With the rapidly growing need for reflection on the energy consumption, the proposed solution is meant to offer a starting point for a more versatile solution. This envisioned solution will, in its ideal state, make reliable partial compilation, and possibly partial or selective testing, the new, sustainable standard of the CI process.

We consider the proposed solution to be one of the first building blocks to a new standard, and with that, know that the future holds many possibilities for improvement of its current version. To present the envisioned development of the proposed solution, we split the ideas into two main categories. The first category lists the improvements that are necessary to make the proof of concept more robust as well as ideas for improvement that are viable in the immediate future. The latter category presents the vision of the bigger picture in which the proposed solution figures as the base mechanism, listing ideas involving more extensive research and development on the path to shifting the standard CI pipelines to a less energy-demanding yet persistently reliable level of operation.

9.1.1. Immediate Future

In this section, we list and elaborate on the enhancements we believe are necessary to enhance the framework's robustness and safety in the near future. At the end, we also mention other research to illuminate the real impact of the existing solution.

Caching

In our solution, we made use of the caching mechanism within the workflow file to cache all compiled classes instead of recomputing all of them. However, this proved to be a relatively complicated task, as the caching has to be propagated to all involved plugins and processes via $pom.xml$. The idea behind this approach is that caching would provide the least invasive way of conducting the partial compilation. To elaborate, instead of only delivering the subset of compiled classes to subsequent processes such as testing, we would deliver the full set of compiled classes in the previous CI run, with only the files defined in the partial compilation sequence updated, replacing their outdated cached versions. With

```
public static void main(String[] args){
    Car[] cars = [new Car("Audi Q7"), new Car("Fiat Punto")]
    for (int i = 0; i < cars.length, i++) {
        cars[i].honk();
    }
}
```

(a) Traditional for-loop in Java

```
public static void main(String[] args){
    Car[] cars = [new Car("Audi Q7"), new Car("Fiat Punto")]
    for (Car car : cars) {
        car.honk();
    }
}
```

(b) Enhanced for-loop in Java

Figure 9.1: Sample code of a traditional for-loop and the enhanced for-loop syntax

that, we believe that implementing caching such that all relevant configuration would be updated as well, is the next step in enabling easier integration of the framework into a real-world project.

Static Dependency Analysis Research
During the research conducted in this study, our primary focus was understanding and working with SDA tools and dependencies they produced. There are generally two different aspects we recommend be researched in the future: the functionality and the efficiency of SDA tools.

When using Doxygen, we found that some dependencies are missed even though we would expect them to be recognised by an SDA tool. Particularly, we found that the tool cannot capture two expected dependencies. The first missing detection is that of a $super$ call, in which a sub-class calls a particular method of its respective super-class. The $super$ keyword is not detected as an important keyword. With that, Doxygen also does not match the call with a specific method. The second important dependency that is missed is that in case of using an enhanced for-loop when iterating through collections of items. Specific to Java, but also used in other languages, its syntax looks as shown in Figure 9.1.

The sample code snippet in Figure 9.1 shows a sample implementation of Java's traditional and enhanced for statement in Figure 9.1a and Figure 9.1b. In our preliminary research, we found that Doxygen will capture the dependency of method $main$ which calls the $Car.honk()$ method in case of the traditional for-loop, however, it will fail to capture the same dependency in the enhanced for-loop, despite the two expressions being synonymous. According to the documentation, the enhanced for statement is meant to make "loops more compact and easy to read". The authors of the Java documentation also explicitly suggest that developers use the enhanced version of the for loop. Given this preference, it is likely that the enhanced for statement is a popular choice when it comes to iterating through collections. As a result, many existing projects that may want to utilise our proposed framework, cannot use the framework reliably due to dependencies created in enhanced for loops being missed. With this, we suggest that more research effort is dedicated to better understanding of dependencies and how to ensure that SDA tools are up to date with the latest versions of Java and other programming languages.

The second important aspect that warrants further investigation is the efficiency of SDA tools. Depending on the size of the code base, SDA tools may consume a significant amount of energy and time to produce the data on existing code dependencies. From our preliminary research, the SDA tools do not use caching or change-specific analysis that would consider the previous state of the code base in terms of dependencies. Therefore, to better understand the energetic impact of conducting SDA and consequently create a more efficient solution, we call for more research into making these tools more sustainable, which is a particularly important task in minimising the extra energy required by our proposed solution.

Repository Size Threshold
In our energy-related results from using the framework within our minimal in-vitro project, we observed that the framework has not produced the desirable energy savings. However, we have seen on our

other experiments, particularly those conducted on a larger, real-world repository, that in this case, the framework has managed to save energy. These results point to the fact that there may be a threshold to the repository size in terms of number of lines of code as well as the number of files, such that partial compilation could omit a sufficient amount of files from compiling and thus save a significant amount of energy. With that, we believe it is important to find this threshold in future research to ensure we only apply the framework in cases where it has the potential to yield energy savings.

Energy Demands of the Solution
In this thesis work, we have proven that the proposed solution has the potential to make the CI build process more energy efficient. However, in this work, we do not take into account the energy consumption of the proposed solution.

One reason for this is that one of strongly correlated factors that determine the energy consumption of the framework itself is the SDA. Depending on the size of the code base, SDA may take a long time to complete, consuming a great amount of energy resources as it executes the analysis. This factor is quite variable, as the energy consumption varies between different SDA tools, as they use different processes. For the SDA tool alone, a separate research effort may be necessary to ensure that the SDA process becomes more energy-efficient. This could be done in terms of the framework's context only by, for example, running the SDA only on the files affected by the commit-changes.

Regarding the framework itself, in the near future, the pipeline of the framework could be measured on a local machine by different tools [5]. For a better understanding of how the process is correlated with the size of or the number of dependencies in the underlying code base. Finding out the energy-related behaviour of the framework would allow us to compare it to the energy savings in the CI pipelines, enabling calculation of the net benefit that the framework can produce.

9.1.2. Further Future

In this section, we introduce the larger scale on which we see this framework fully utilised to deliver maximum possible efficiency improvements.

Scope extension
Knowing the limitations of static dependency analysis, it is necessary to find the scope of build breaks that can be covered by this approach, and with that, identify other cases which this technique may miss. We know from previous work that static analysis methods may not be sufficient in capturing all relevant dependencies for a robust test selection. With that, this work also combines the static analysis approach with defined dynamic rules that extend the scope covered with only static methods [21]. To enhance our proposed solution and enable its usage across all cases, we propose additional research into build-breaking cases caused by missing dependencies. This could be done systematically, using the knowledge of dependency analysis and sampling more cases, or by an exploratory study, in which the framework is used across the pipelines alongside full compilation, to identify the cases in which it cannot reliably catch a build job break.Finding and covering all currently missed cases, by, for example, adding dynamic rules to catch the missing dependencies, will enable developers to safely use the framework without having to match it to the currently specific scope of usage.

Fail-safe Mechanism
While the ideal vision for the framework is that in which it is capable of catching any source-caused build break, identifying and covering all the cases is a difficult task. The situation is made worse by the fact that programming languages are always developing, possibly creating new potential for failures with every new functionality feature. For this reason, a viable future path for the proposed solution is its integration in a bigger framework that executes a fail-safe mechanism. For example, after a custom number of rounds of using partial compilation, we force a full compilation in CI execution to ensure that a break that may have been missed by the framework is caught eventually. The negative consequence of this approach is that developers must account for a possibly delayed accurate feedback. They may also have to get more involved in the configuration, and find the desirable trade-off level between accurate in-time response and the amount of energy-saving efforts. An addition to this approach could be combining the framework with existing successful approaches, such as *SmartBuildSkip* or *PreciseBuildSkip*, which could also serve as an indicator for break-prone commits and signal the instances in which full compilation may be preferable [19][20].

Bibliography

[1] Rabe Abdalkareem, Suhaib Mujahid, and Emad Shihab. "A machine learning approach to improve the detection of ci skip commits". In: *IEEE Transactions on Software Engineering* 47.12 (2020), pp. 2740–2754.

[2] Rabe Abdalkareem et al. "Which commits can be CI skipped?" In: *IEEE Transactions on Software Engineering* 47.3 (2019), pp. 448–463.

[3] David Brooks, Vivek Tiwari, and Margaret Martonosi. "Wattch: a framework for architectural-level power analysis and optimizations". In: *SIGARCH Comput. Archit. News* 28.2 (May 2000), pp. 83–94. ISSN: 0163-5964. DOI: 10.1145/342001.339657. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/342001.339657.

[4] Luís Cruz. *Green Software Engineering Done Right: a Scientific Guide to Set Up Energy Efficiency Experiments*. http://luiscruz.github.io/2021/10/10/scientific-guide.html. Blog post. 2021. DOI: 10.6084/m9.figshare.22067846.v1.

[5] Luís Cruz. *Tools to Measure Software Energy Consumption from your Computer*. http://luiscruz.github.io/2021/07/20/measuring-energy.html. Blog post. 2021. DOI: 10.6084/m9.figshare.19145549.v1.

[6] Dimitri van Heesch. *Doxygen*. YYYY. URL: https://www.doxygen.nl.

[7] Thomas Durieux et al. "An analysis of 35+ million jobs of Travis CI". In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2019, pp. 291–295.

[8] Omar Elazhary et al. "Uncovering the Benefits and Challenges of Continuous Integration Practices". In: *IEEE Transactions on Software Engineering* 48.7 (2022). 2570. DOI: 10.1109/TSE.2021.3064953. URL: http://dx.doi.org/10.1109/TSE.2021.3064953.

[9] Keheliya Gallaba. "Improving the Robustness and Efficiency of Continuous Integration and Deployment". In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, pp. 619–623. DOI: 10.1109/ICSME.2019.00099.

[10] Keheliya Gallaba et al. "Accelerating continuous integration by caching environments and inferring dependencies". In: *IEEE Transactions on Software Engineering* 48.6 (2020), pp. 2040–2052.

[11] Keheliya Gallaba et al. "Lessons from Eight Years of Operational Data from a Continuous Integration Service: An Exploratory Case Study of CircleCI". In: *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 2022, pp. 1330–1342. DOI: 10.1145/3510003.3510211. URL: http://dx.doi.org/10.1145/3510003.3510211.

[12] GitHub. *GitHub Actions REST API Documentation*. Accessed on: 20/04/2024. 2024. URL: https://docs.github.com/en/rest/actions?apiVersion=2022-11-28.

[13] GitHub. *GitHub Documentation*. Accessed on: 04/04/2024. 2024. URL: https://docs.github.com/.

[14] *GitHub Actions*. GitHub. URL: https://github.com/features/actions.

[15] Green Coding Solutions. *Eco-CI: Energy Estimation for Continuous Integration*. 2024. URL: https://github.com/green-coding-solutions/eco-ci-energy-estimation.

[16] Michael Hilton et al. "Usage, costs, and benefits of continuous integration in open-source projects". In: *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 2016, pp. 426–437.

[17] IEA. *Data Centres and Data Transmission Networks*. 2023. URL: https://www.iea.org/energy-system/buildings/data-centres-and-data-transmission-networks#tracking.

[18] *Incremental Build*. URL: https://docs.gradle.org/current/userguide/incremental_build.html.

[19] Xianhao Jin and Francisco Servant. "A cost-efficient approach to building in continuous integration". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 13–25.

[20] Xianhao Jin and Francisco Servant. "Which builds are really safe to skip? Maximizing failure observation for build selection in continuous integration". In: *Journal of Systems and Software* 188 (2022), p. 111292.

[21] Yingling Li et al. "Method-level test selection for continuous integration with static dependencies and dynamic execution rules". In: *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE. 2019, pp. 350–361.

[22] Nikolai Limbrunner. *Dynamic macro to micro scale calculation of energy consumption in CI/CD pipelines*. 2023.

[23] Shane McIntosh et al. "An empirical study of build maintenance effort". In: *Proceedings of the 33rd International Conference on Software Engineering*. 2011, p. 141. DOI: 10.1145/1985793.1985813. URL: http://dx.doi.org/10.1145/1985793.1985813.

[24] Sparsh Mittal. "A Survey of Techniques for Approximate Computing". In: *ACM Comput. Surv.* 48.4 (Mar. 2016). ISSN: 0360-0300. DOI: 10.1145/2893356. URL: https://doi.org/10.1145/2893356.

[25] nstruharova. *CIPipeline*. https://github.com/nstruharova/CIPipeline.git. 2024.

[26] John O'Duinn. *The Financial Cost of a Checkin - Part 2*. https://oduinn.com/2013/12/13/the-financial-cost-of-a-checkin-part-2/. Dec. 2013.

[27] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices". In: *IEEE Access* 5 (2017), pp. 3909–3943. DOI: 10.1109/ACCESS.2017.2685629.

[28] Daniel Ståhl, Torvald Mårtensson, and Jan Bosch. "The continuity of continuous integration: Correlations and consequences". In: *Journal of Systems and Software* 127 (2017), pp. 150–167. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2017.02.003.

[29] The Apache Software Foundation. *Apache Maven*. 2024. URL: https://maven.apache.org.