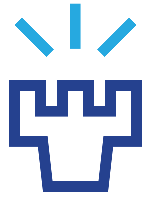

FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Vili Tiinanen

SUPPORTING INCREMENTAL BUILDS IN A DYNAMIC DISTRIBUTED BUILD ENVIRONMENT

Master's Thesis
Degree Programme in Computer Science and Engineering
September 2022

Tiinanen V. (2022) Supporting Incremental Builds in a Dynamic Distributed Build Environment. University of Oulu, Degree Programme in Computer Science and Engineering, 47 p.

ABSTRACT

Incremental building in continuous integration is a rare sight and could be used more than it is in the present day. This master's thesis aims to demonstrate how incremental building could be enabled in a continuous integration server environment, which is also dynamic and distributed by its traits.

This study develops a solution called "workspace cache". Its design is based on existing literature and organizational experience. The technological components of the workspace cache are selected based on the existing production system of the client of this study. The produced solution is by its nature a proof-of-concept implementation, and a production implementation requires further work.

Evaluating the solution in the production environment is not feasible, and therefore this study constructs a simulation environment for the workspace cache, where the efficiency of the solution can be measured. The solution is tested using two build systems, make and Gradle, and two popular open source software projects that are made with different programming languages. The solution is benchmarked against clean software builds. In addition to build times, source code cloning times and the correctness of the build outputs are measured.

The resulting solution enables incremental builds in the target context. The build systems and software projects that the solution was tested with do not guarantee the generalizability of the solution. With make, the correctness of the build outputs produced by the workspace cache does not fully fulfill the set correctness criteria. Future research work should focus on getting the builds from workspace cache to be fully correct, and on making the workspace cache more efficient than a remote cache.

Keywords: Build optimizations, Continuous Integration, Design Science Research, Distributed systems, Release engineering, Software builds, Software Defined Systems

Tiinanen V. (2022) Inkrementaalisten ohjelmistokoontien tukeminen dynaamisessa hajautetussa koontiympäristössä. Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 47 s.

TIIVISTELMÄ

Inkrementaaliset ohjelmistokoonnit ovat vähän käytetty optimointitekniikka jatkuvassa integraatiossa, ja sitä voitaisiin hyödyntää nykyistä enemmän. Tässä diplomityössä demonstroidaan, miten inkrementaaliset ohjelmistokoonnit voidaan tuoda jatkuvan integraation palvelinympäristöön, joka on luonteeltaan dynaaminen ja hajautettu.

Tässä tutkimuksessa kehitetään ratkaisu nimeltään "workspace cache". Ratkaisun suunnittelu perustuu aikaisempaan kirjallisuuteen ja asiakasorganisaation kokemukseen. Workspace cachen teknologiset komponenttivalinnat perustuvat asiakkaan olemassa olevaan tuotantoympäristöön. Työssä tuotettu ratkaisu on luonteeltaan konseptitoteutus, ja ratkaisun tuotantoympäristöön vieminen vaatii lisätyötä.

Ratkaisun arvioiminen tuotantoympäristössä ei ole mahdollista, joten tässä tutkimuksessa rakennetaan ratkaisulle simulaatioympäristö, jossa ratkaisun tehokkuutta voidaan mitata. Ratkaisua testataan kahdella eri koontijärjestelmällä, Gradlella ja makella, sekä kahdella suositulla avoimen lähdekoodin ohjelmistoprojektilla, jotka käyttävät toisistaan eriäviä ohjelmointikieliä. Ratkaisun suorituskykyä mitataan tyhjästä muodostettua ohjelmistokoontia vastaan sekä nopeuden että oikeellisuuden osalta. Koontienluonnin lisäksi työssä mitataan lähdekoodin kloonaukseen ohjelmistovarastoista kuluvaa aikaa.

Toteutettu ratkaisu mahdollistaa inkrementaaliset ohjelmistokoonnit kohdekontekstissa. Tutkimuksessa käytetyt koontijärjestelmät ja ohjelmistoprojektit eivät takaa ratkaisun yleistettävyyttä. Koontijärjestelmä makella ja workspace cachellä tuotetut ohjelmistokoonnit eivät täysin täyttäneet niille asetettuja oikeellisuusehtoja. Tulevaisuuden tutkimustyön tulisi pyrkiä saamaan workspace cachen koontiulostulot täysin oikeellisiksi sekä parantamaan sen suorituskykyä siten, että se olisi tehokkaampi kuin etävälimuistiratkaisut.

Avainsanat: Hajautetut järjestelmät, Jatkuva integraatio, Julkaisunsuunnittelu, Kehittämistutkimus, Ohjelmistomääritellyt järjestelmät, Ohjelmistokoonti, Ohjelmistokoonnin optimointi

TABLE OF CONTENTS

ABSTRACT
TIIVISTELMÄ
TABLE OF CONTENTS
FOREWORD
LIST OF ABBREVIATIONS AND SYMBOLS
1. INTRODUCTION.......... 7
   1.1. Definitions.......... 7
   1.2. Research Problem.......... 9
   1.3. Research Methodology.......... 10
2. BACKGROUND.......... 13
   2.1. Build Systems.......... 13
      2.1.1. Build Dependencies.......... 13
      2.1.2. Build Dependency Challenges.......... 14
      2.1.3. Caching and Sharing Builds.......... 15
      2.1.4. Correctness and Efficiency.......... 16
   2.2. Continuous Integration.......... 17
      2.2.1. CI Server Infrastructure.......... 18
      2.2.2. Build Systems in CI.......... 19
   2.3. Virtualization and Orchestration.......... 20
3. WORKSPACE CACHE.......... 21
   3.1. System Requirements.......... 21
      3.1.1. Gathering the Requirements.......... 21
   3.2. Simulation Components.......... 22
      3.2.1. Virtual Machines.......... 22
      3.2.2. Kubernetes Cluster.......... 23
      3.2.3. Kubernetes Client.......... 23
      3.2.4. Ceph and Rook.......... 24
   3.3. Solution Development.......... 25
      3.3.1. Persisting Workspace.......... 25
      3.3.2. Simulating CI Environment.......... 27
4. EVALUATION.......... 30
   4.1. Research Setting.......... 30
   4.2. Data Analysis.......... 32
5. DISCUSSION.......... 39
   5.1. Requirements.......... 40
   5.2. Challenges.......... 40
6. CONCLUSION.......... 42
7. REFERENCES.......... 43

FOREWORD

I would lie if I would say that this thesis project would not have been a great challenge for me. In spite of hardships, this thesis was still finished in time. The whole journey of the masters studies have given me a lot and taken almost everything. It has laid a new path in front of me to carry on with, and I sincerely think that learning will never end.

I am ever grateful to my family and parents, they have given me their unconditional support in life, without them, I would not be here today. I appreciate my employer for offering the opportunity for this thesis. Special thanks go to my colleagues who helped me during the thesis, you people made this possible.

Oulu, September 29th, 2022

Vili Tiinanen

LIST OF ABBREVIATIONS AND SYMBOLS

API	Application Programming Interface
CI	Continuous Integration
CPU	Central processing unit
CSI	Container Storage Interface
DAG	Directed Acyclic Graph
DSR	Design Science Research
DSRM	Design Science Research Methodology
ID	Identification
HTTP	Hypertext Transfer Protocol
JAR	Java archive file format
OS	Operating System
PV	Persistent Volume
PVC	Persistent Volume Claim
RAM	Random Access Memory
SDS	Software defined storage
VCS	Version Control System
VM	Virtual Machine
ZIP	Archive file format

1. INTRODUCTION

Evolving software systems are shown to increase in complexity, change regularly and are under constant growth [1 p.7]. This growth of source code is the cause of endless edit-compile-test loops, which again are the source of feedback for the developers. When software systems grow large, their build processes become time-consuming. Long build processes lengthen the developer feedback cycle and detain the development activity. This edit-compile-test rounds should be reduced to the minimum time possible in order to keep the development activity productive. The most time-consuming parts of this loop are compiling and testing, therefore the time savings need to come from those actions.

Build tools handle the compilation and testing of the project source code, and with bigger projects, build systems orchestrate the different processes to build the software. Most of the build systems have build optimizations built-in for reducing the build time, but their usage in a continuous integration tools are not self-evident. This thesis is trying to bring the build system optimization called incremental building to the continuous integration server environment, where architecture of the continuous integration server is dynamic and distributed, in order to shorten the feedback loops of the developers of the customer organization.

This thesis produces an approach called workspace cache for enabling incremental builds in the continuous integration process, and thus reduce the edit-compile-test feedback time to the developer and to amplify the utilization of idling Continuous Integration (CI) server workers.

The approach applies the features of distributed computing and cluster infrastructure to achieve its objective. The requirements for such solution are derived from existing literature and from the organizational experience. These requirements are later combined to form the final requirements for the solution.

The solution is demonstrated in a simulation environment which tries to mimic a real world computer cluster through virtualization, and likewise, attempts to represent the production environment of the client of this study for possible later migration of the solution to the production environment. The effectiveness of the solution is measured by comparing the results from using the solution against software builds that are built from clean sources.

1.1. Definitions

This section explains the key definitions of this work. These definitions are essential to understand the purpose and goal of this thesis.

In this thesis, a build system is any piece of software that provides means for constructing and parsing the dependency graphs, which represents the dependencies among the files in a software project. Build systems play a pivotal role in automating the development processes around modern software systems [2, 3, 4, 5]. The purpose of the build system is to conduct tasks like compiling, testing, packaging and deploying software systems [2, 5]. With modern software development organizations, these tasks could be extended to anything, from setting up a test environment, to sending an HTTP request to the coffee machine of the development team. In this thesis, we are interested

in only in the build tasks that require significant amount of computing like compiling and testing, since they are the most time-consuming part of the software build.

Computation output can be incrementally updated without recomputing from scratch, this however requires that the program tracks its dynamic data dependencies [6]. The time saved from not having to compute from scratch can be significant [6]. Hence, incremental builds reduce build times to a degree where they can be considered crucial for a modern build systems [5]. Incremental builds do not only benefit builds where there is a small change in the source code, it also speeds the building of different software configurations [7]. With different software configurations, incremental builds can be faster than clean builds, even if they might produce incorrect results [8]. Hence, incremental builds should be a good optimization technique for a general CI service provider to support for.

Memoization is an old programming optimization technique. It is used to speed up programs by storing the result of a computationally expensive function and returning the cached result when the same input is confronted again [9]. It must be noted that memoization is a way to change the computational complexity of the function; time cost is traded with space cost by storing the result somewhere and reusing it when an input repeats. The requirement for caching the result is that the function needs to be deterministic and work like a pure function [6, 9], for given input the function must always produce the same output. Any non-determinism in the function itself would cause a faulty output.

Build caching is the way build systems memoize task outputs, and it is known to save time and resources when building software projects [10, 11]. Considering that we first have to perform a clean build, every build task is executed and the output of each task is stored by its inputs. When a second build invocation is issued, after a change in the source code for example, the build system checks for existing outputs for the given inputs, if it finds suitable output, it retrieves the already computed output from the cache to avoid re-execution of the given task. Thus making the build overall faster in the subsequent runs.

There is a slight difference in the terms when we consider incremental building and build caching. It is often hard to determine which of those terms in build system refers to as the true "incremental building". Build caching is a technique to achieve incrementality within builds. Therefore, it is one of the ways build systems can achieve incremental building.

Continuous integration is a software development practice where developers integrate as frequently as they can, preferably with daily basis [12]. There are differences on how continuous integration practice is implemented across the industry, and there is no singular consensus between the practitioners of a homogenous continuous integration practice [12]. Particular tooling is not required within the continuous integration practice, as the practice itself is more important [13], albeit the tooling might improve the CI practice implementation [12].

The CI server is one of the most important tools of a development team [14]. The main purpose of the CI server is to pull the source code from the version control repository and check for any changes, if any are found, a list of commands to trigger the build is executed [14]. This continuous building and testing in the CI server maintains the quality and consistency of the software system by rigorously checking it, over and over again, automatically. The CI server can offload the work of checking the source

code to a worker node. CI worker is essentially another computing resource connected to the CI server, which executes a job defined by the CI server.

The issues that the distributed CI environment have, are the same that can be considered as issues of any distributed system. For example, there might be problems with geographical scalability [15 p.20], administrative scalability [15 p.21] or replication [15 p.26]. In a distributed CI environment, there are numerous geographically dispersed computers working in different roles, that are interconnected through a network but are still perceived as a single system. Therefore, a distributed CI environment fulfills the definition of a distributed system: "A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system" [15 p.2].

Achieving incremental builds in a dynamic distributed environment is challenging, as the earlier CI server software builds need to share their build data with subsequent builds. This creates a data dependency between the builds of the CI servers and, in contrast to stateless clean builds, it causes the software builds to be stateful. Thus, the issue could be translated to a data distribution problem, where effectively sharing the earlier build data could be the key to attain incremental builds in a dynamic distributed environment.

This thesis tries to fill the gap between the distributed CI environment and the incremental builds. By that, the aim is to reduce the feedback time to developers and to amplify the utilization of the CI workers.

1.2. Research Problem

The objective of this thesis is to design, develop and assess a solution that would enable incremental builds in a dynamic distributed build environment. The research questions should provide insight on how the proposed solution should be designed and developed. To achieve this goal, the following research questions were defined:

RQ1: How are incremental builds enabled?

The first research question is aimed towards understanding what underlying mechanisms make incremental builds possible. By answering it, the reader should be able to grasp the essence of incremental builds.

RQ2: What are the differences of running incremental builds in the CI server compared to the machine of the developer?

The second research question examines the gap between CI and incremental builds, especially what makes it so hard to perform incremental builds in CI compared to the machine of the developer.

RQ3: Can the incremental builds be enabled in dynamic distributed build environment?

The third research question tries to explore the possibility of having incremental builds in a dynamic distributed build environment, which is essentially similar environment that the customer of this study is having. By answering the last question, there should be consensus if the incremental builds are possible in the given setting.

The main aspect of the literature review was to assemble existing literature that would cover the research topic. This included topics: Build systems, Continuous Integration, distributed systems and software defined storage (SDS). The websites of the organizations behind the tools that were used in this study were examined as non-scientific literature.

The literature review defined the backbone to the requirements for the proposed artifact. These literature-based requirements were combined with the application domain requirements to form a final list of requirements for the solution. The combined requirement list steered the design, development and assessment of the final design artifact in the design process.

1.3. Research Methodology

A design science research methodology (DSRM) is utilized in carrying out this study. As the nature of this research is objective-centered, DSRM supports it by ensuring validity and rigor of the study. In this study, a formal sequential process was followed, and it was initiated from the problem identification and motivation activity. Figure 1 is adapted from [16] and it represents a DSRM process model that is adapted to the context of this research, and it also displays the objective centric approach of this study. The DSRM process activities are presented more specifically in the next sections.

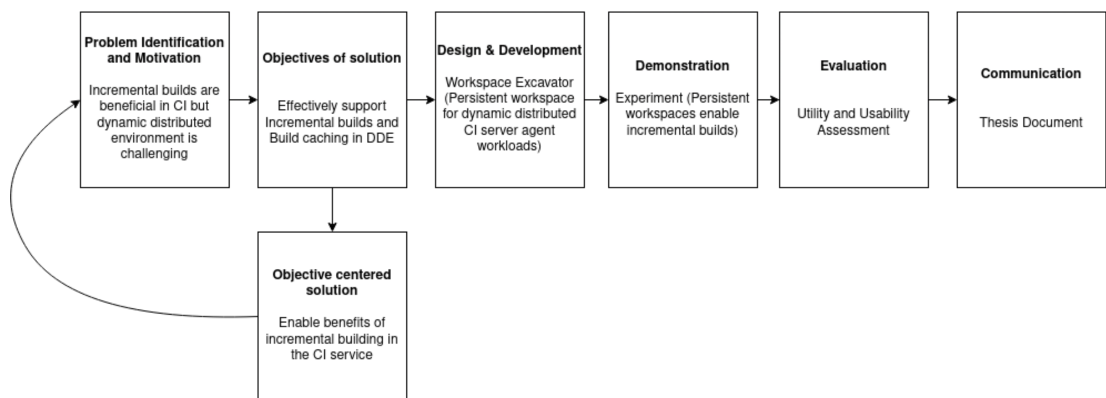

Figure 1. The DSRM process model adapted into this research context

1. Problem identification and motivation

Incremental builds and build caching are enabled by the ability of the build system to use up-to-date checking for its build rules during build time [17]. Being able to utilize incremental building and build caching can reduce build times tremendously [11]. Faster builds translate to faster feedback for the developers [4, 5]. Running

CI service infrastructure in a dynamic distributed environment, such as Kubernetes, introduces challenges as the data from previous builds is not present. Contrast to how developers can reuse the build data on their own computers with build systems.

Various solutions [10, 18] exist for making build output data available for later builds. However, none of those solutions take into account the human factor, where humans are error-prone and do not produce optimal build rules that capture build dependencies accurately [5]. The fact that multiple papers [5, 19] introduce tools, which help humans to detect build rule discrepancies, is a strong indication towards the need to address the human factor in the design.

The previous solutions are build system specific. The way one particular build system shares and stores the build outputs is different to other build systems, and thus the outputs can not be shared between them. There are no generally available solutions to address this issue, but others have started to look for different approaches [20, 21]. The solution might indeed be elsewhere, on a different level than build systems. This study tries to explore and produce a way of sharing and storing the build output inside a CI service environment in a build system agnostic way.

2. Objectives of the solution

This study proposes a file-system level approach for supporting incremental builds and build caching in a dynamic distributed CI environment. The solution should enable a build system agnostic way to support reusage of build output. The solution should outperform clean builds to prove its efficacy. Incremental builds are not useful if the outputs of the builds are incorrect. Therefore, the solution should produce sound builds. From the CI service provider standpoint, the solution needs to be usable by its end-user, meaning that some kind of interface needs to be made available and that interface should have good user experience.

3. Design and development

During the development and design of the artifact, software engineering best practices were used. Recent technological solutions such as software defined storage are considered amidst of the development process. Furthermore, prior studies relevant to this research were taken into account during the artifacts' development phase. The documentation of the process of design and development of the artifact is present in this thesis throughout, but Chapter 3 embodies development and design details the most.

4. Demonstration

A simulation in the target context was conducted to demonstrate the utility of the solution. The real world usefulness of the solution is proved by showing that it functions in the target context. The generality of the solution is demonstrated in the simulation by trying it with two different build systems. The design and implementation details of experimentation are more specifically documented in Chapter 4.

5. Evaluation

To evaluate the correctness of the solution, all the solution enabled build outputs are compared with clean build outputs for soundness. The efficiency of the solution is evaluated by comparing the solution enabled builds with clean builds. The usability of the solution was not evaluated, but rather discussed due to the limited scope of the thesis. The evaluation context, assessment method, analysis and findings are documented in Chapter 4.

6. Communication

The DSRM process model defines the structure of this paper. The communication in this research process happens through this thesis document. In this thesis document the problem and its importance is shown and the produced artifacts' utility, novelty, effectiveness and the rigor of its design presented.

2. BACKGROUND

This chapter introduces the prior literature. The literature review consists of exploring subjects such as continuous integration, build systems, and CI server infrastructure. In addition to these subjects, the following topics are explained as they form core concepts of this study: orchestration systems, distributed storage and existing implementations of incremental building and build caching in CI environment.

Past literature explains the need to shorten the feedback loops in continuous integration setting [11, 22] with incremental building, but does not provide any concrete means to apply this in practice, only guidelines and principles. They also show that a significant amount of time is wasted on the idling CI workers, therefore incremental building could reduce computing resources for software building and in addition to it could save developer attention and time. Small cycle times are not only for the cloud service development, shorter cycles are also possible in safety-critical systems like in automotive software [23].

2.1. Build Systems

Build automation tools gather all the commands and knowledge required to build a software project, and ease the maintenance of build descriptions [22]. Compared to plain shell scripts, build systems are able to infer from dependency graphs which build tasks need to be performed to achieve a build [22]. Having a full view on the build task dependencies give build systems the ability to skip unaffected build tasks [18, 22]. Skipping unaffected build tasks reduce the amount of work to be done and shortens the build time.

2.1.1. Build Dependencies

Build systems track their data dependencies and model it with directed acyclic graphs (DAG) [17, 18, 22] where the arrows between nodes represent dependencies [18]. Nodes that have an outgoing arrow are called targets and the ones without, source files [18]. If a given node A has an arrow to a node B, that means node A depends on the node B, and node B is needed before node A can be generated [18]. The dependency graph can be constructed from a build file [18], which is used with every build system to capture the dependencies between source files and build target. The Build file works as the recipe for the build targets, it lists the ingredients (source files) and methods (build commands) to produce the outcome (build targets).

Figure 2 is adapted from [18] and it represents a simple C project dependency graph where intermediate files of the compilation unit [18] are omitted, since they are temporary. Each compilation unit gets compiled into an object file (*handler.o*, *database.o* and *main.o*). All the object files are linked into the final executable *program*.

Let us consider a build system called *make* [24] as the first example, since most build systems use a similar approach to the building as make [3, 4]. As make performs a clean (scratch) build, it traverses the project DAG in the topologically sorted order, and

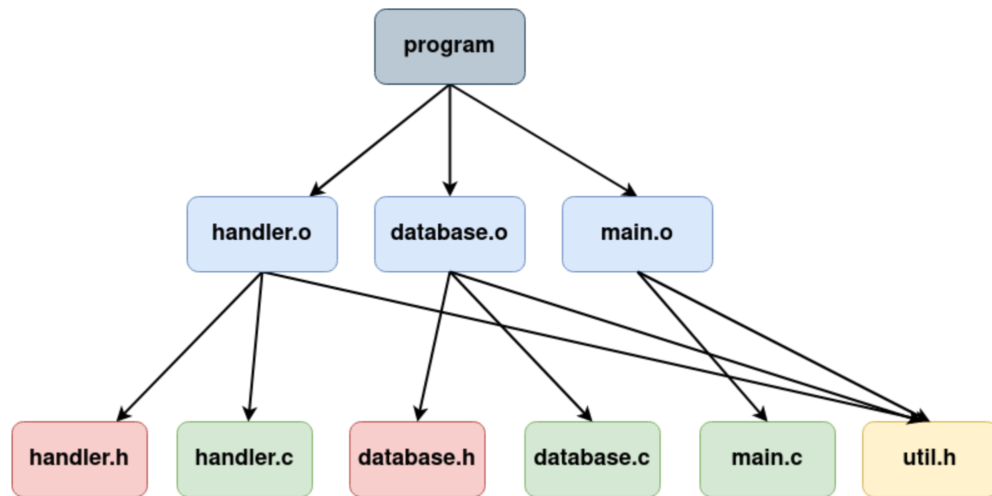

Figure 2. Structure of a dependency graph from a small C project.

for each build target it executes the build commands. In the later runs, the intermediate targets are available for reusage. Make compares timestamps between source files and the corresponding object files to check, whether there is a need for building the target again. Thus, if the timestamp tells that the source file is older than its object file, then that target needs no compilation. Furthermore, if the object file is older than the source file, then make knows that the source file has changed and runs build commands to recompile the object file. Any newly compiled object file is thus newer than the build targets depending on it. Which means that all the dependent build targets should be recompiled too. This action cascades throughout the dependency tree until the top is reached. Moreover, If multiple build targets depend on the same source file and that source file is modified, all the build targets are recompiled. Meaning that a minor change in a single header file might re-trigger the compilation of all build targets. [18]

In the example C project, All the source files are compiled at a clean build. This generates *handler.o*, *database.o*, *main.o* and the final binary executable file *program*. If *handler.c* is modified, its corresponding object file target *handler.o* would be generated, old *database.o* and *main.o* could be reused, then all the object files would be relinked into *program*. On the other hand, modifying header file *util.h* would cause all the object files to be regenerated and *program* to be relinked. [18]

2.1.2. Build Dependency Challenges

Dependencies are in the key role for build systems. Failing to describe the dependencies of a project can cause incorrect software builds [18]. Two types of faults can occur with software builds due to incorrect build definitions: missing inputs or missing outputs [5]. Redundant dependencies lead to unnecessary build actions [18]. Both, manually maintained and automatically derived dependencies of a *Makefile* are prone to redundancy, but the human maintained ones are more so than the automatic ones [18].

Build systems that rely on timestamps are prone to unnecessary recompilation [18]. Even if the source file of a given target object file stays the same internally, but

its timestamp happens to be newer than its corresponding object files, it means the recompilation will happen [18]. This is unnecessary in a sense that the source file did not actually get modified [18]. On the other hand, a build tool like *SCons* [25] and a compiler cache like *ccahce* [26] calculate the checksum of a file and use that to check whether the file has changed, this enables SCons to avoid the unnecessary compilation issue and prevents ccache from redundant compilations [18]. In turn, if the project has thousands of source files, the calculation impact could be considerable and slow the build down.

Stale artifacts are caused by the failure to re-generate build targets whose source files were changed. Failing to capture changed source files and then regenerating the final executable and build targets can lead to runtime errors or to incorrect build outputs. Even following the best practices for tracking dependencies can not always ensure correct builds. [5]

The example build system, make, does not detect if the build command of the build rule is changed. This is problematic, as this might lead to stale artifacts. Considering, no source files are modified, but the build command of one build target gets changed. The invocation of the build would not consider anything to be regenerated, although the command to build the target was different. Distinct to make, a build system called *Ninja* [27] can detect such a change in the build command. [4]

A build system tracks the changes to required or provided files, but the build can become inconsistent by not being able to track build *meta-dependencies*. If the code of the builder is changed, the build itself does not account for that, which in turn, can lead to inconsistent build results. Almost all the build systems ignore this type of dependency. [3]

2.1.3. Caching and Sharing Builds

Incremental builds enable reusing products of the last build, but the usefulness of incremental building deteriorates as the changes to the source files get bigger. Sharing the products of a build to geographically distant machines or to developers is difficult. By caching builds, these issues can be addressed. [22]

Caching build might help individual developers to improve their local build performance, by being able to reuse CI build cache remotely [22]. Build times can be reduced locally and in CI by implementing good data structures, algorithms and correct shared build cache [22]. This blurs the boundary between local and CI builds and has the potential to achieve a situation where the same build step would never see its second run [22]. Lastly, caching build results is not easy to implement correctly and may be impractical. For example, accessing files over a network might be slower than rebuilding locally. [11]

Tool like the previously mentioned ccache perform a specialized caching for a restricted set of compilers [22]. Generic and correct build caching is difficult, because builds depend on their environments [22]. Such "generic" cache is implemented with varying levels of isolation in *Gradle* [22, 28].

These remote cache backends are often HTTP services [10, 29], where the data is accessed with Create, Read, Update, and Delete (CRUD) operations. It is worth noting, that no general build cache is available, which would allow caching builds

from multiple different build tools. The optimization of these remote cache HTTP services are not in the scope of this thesis.

An environment cache is about caching tool chains, libraries and runtimes that are needed to perform the build in a build job processing node. Before the commands that CI configuration specifies can be run, the processing node of the build job needs to be initialized and prepared. These build processing environments rarely change dramatically over the lifetime of a software project, and installing dependencies at the start of each build wastes time. This can be addressed by caching the build processing environment as a snapshot (like a container image) and in case the specification for the dependencies change, a new snapshot is created. This allows the builds to pull a ready-made processing environment, and by that reduce the build time. [20]

These caching ways can be visualized by their file-level granularity. Figure 3 displays how these ways can be visualized. Starting from the most granular level, compiler cache works with a single source file and object output. Build cache handles build tasks and can cache multiple files associated to the build task. The environment cache includes the entire file system, including programming language runtimes and tool chain binaries. Having heterogeneous caching methods introduces complexity, since each implementation requires an individual remote backend and each remote backend is created with different languages, developers and incentives in mind. Without the usage of a specific remote cache backend, these caching methods cannot be used to share build outputs.

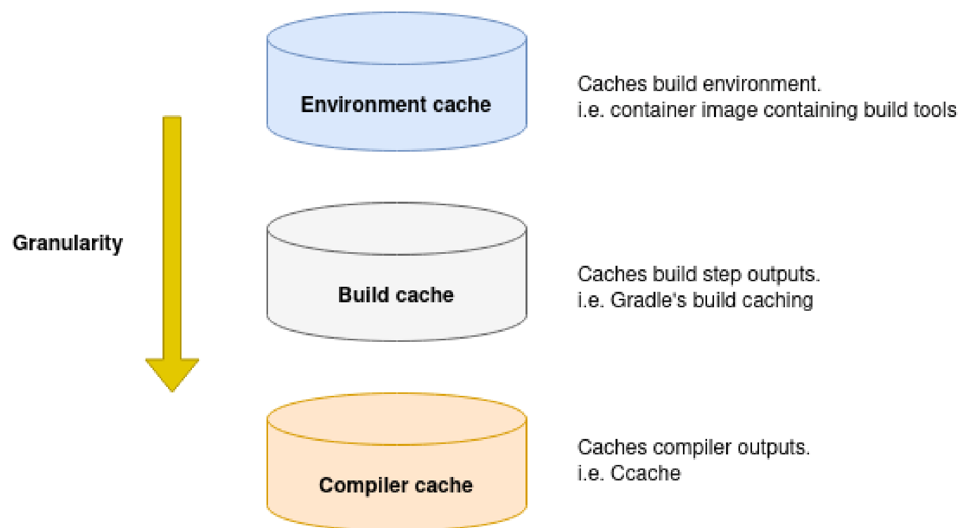

Figure 3. Levels of caching with software builds. The file level granularity lowers downwards

2.1.4. Correctness and Efficiency

Build speed is oftentimes traded off with build *correctness* [20], reusing files from previous builds could introduce improper files to the software being built and result into different output as compared to clean build [7]. To prove that an optimization

like incremental building provides correctness, the build output of a clean build should match the build output contents of an incremental build [7]. The correctness of build systems is a hard problem [7], but not in the scope of this thesis since we are trying to enable incremental building through existing build systems with existing functionality, we are not trying to create a new build system or try to add a new feature to existing build systems to ensure correctness, we expect it from the existing build system.

Maudoux et. al. [22] argue that build tools that claim to be correct should maintain some state on the disk or cloud. The storage location could be locally on the source tree or outside of it.

While correctness helps to avoid future issues, efficiency reduces the time taken with the edit-compile-test loop [22]. Thus, incremental builds should outperform clean builds, as build outputs from previous builds are present and available for reuse [7]. Unused but declared build dependencies lower the overall efficiency of the build [22]. One should therefore expect the incremental build to be more efficient than the clean build.

2.2. Continuous Integration

In the last two decades, continuous software engineering, continuous activities and DevOps have become the de facto practices in the software industry [30]. The goal is to do software development in a more rapid pace and still maintain the quality and consistency. This is only achieved if all faults are discovered early on and reported back to developers as soon as possible [30].

Automation is the principle which enables short lead times, fast delivery and rapid feedback [31]. This automation requires tooling [23] *i.e.*, version control systems (VCS), build systems, test frameworks and CI servers. Automation is already used at the machines of the software developers of today. Compiling source code into binary files or testing a change to the source code is automated by build systems. The faster the build task runs, the faster the feedback is received. When the developer wants to integrate his code back to the mainline, he commits his code to the version control system. From there, continuous integration server notices that there is a change request and verifies through regression tests that the change is integrable to the mainline code base. The quicker the developer gets his feedback of the change request eligibility, the quicker it can be adjusted. If the change request is merged into mainline, even more tests and compiling is done by the CI server until it delivers out the final artifact.

In a classical release engineering pipeline [32] the feedback from the different pipeline stages is relevant to release engineers, release planners, developers and operators. The fast moving pace of modern software development puts immense pressure on this pipeline to bring more frequent releases and shorter development cycles in order to ship software faster. Making the edit-compile-test loop as short as possible does not only cut the development time of the project, but also amplifies software evolution [11].

The build systems are part of the release engineering pipeline and are used by the developers to test their code locally. It is also used with continuous integration to run tests and to create software artifacts. The continuous integration environment however

differentiates from the local environment of the developer, making it difficult to use the optimizations of the build systems in a CI setting.

2.2.1. CI Server Infrastructure

Continuous integration server is one of the most important tool of a software development team. Its purpose is to check out the code from the repository for any changes and, in case of any, run the defined build commands to trigger the build. The task of the continuous integration server is to check whether the newly introduced changes to the source code still pass the quality stages it was configured to run. Multiple self-hosted continuous integration tools exist, like Jenkins, TeamCity and Bamboo. There are also hosted products such as CloudBees and TravisCI. [14]

CI servers can work as a single monolithic system, but there is a limit to that. With monolithic systems, the scaling is limited to the vertical axis. Thus, at a certain point, the building and checking of the software is blocked by the underlying servers' hardware ability to run those workloads simultaneously. The server simply chokes on the work that it gets, causing software builds to queue up. This monolithic build capacity can be changed by introducing horizontal scaling to the CI server. This is usually done by using a master-slave architecture, where the CI server acts as the master for slave worker nodes that conduct most of the work, like building the software [33].

Idling static non-virtualized distributed worker nodes of the CI server is a waste of computing resources, as they can be sitting and waiting for work. If the computing resources are not deployed on-premise or in the cloud, where virtualization is possible, there will be a significant financial impact from the operational costs [34]. To address this, virtualized dynamic distributed worker nodes that spawn when work is available, would reduce these operational costs by releasing computing, hardware and energy resources when there is no work to be done.

Jenkins is an open-source CI tool, made with Java. It automates building, testing and deploying of software products. It has many plug-in options available, and its user base is large. It supports all currently relevant version control systems. Jenkins can be extended over multiple machines for more computing resources. This can be done by connecting additional worker nodes to the master instance, thus increasing the overall computing capacity. [23, 33, 35]

We look at the continuous integration servers from the point of view of Jenkins [35], as it is so widely used for building and testing software [33]. Additionally, Jenkins integrates nicely with deployment technologies (such as Kubernetes [36]) [33], which is important as we are interested in dynamic distributed worker schema, where the worker node spawns when there is CI work available.

To support different platforms [33] or to provide infinite on-demand cloud workers [37], the continuous integration server has to be able to distribute its workload. Jenkins can work in a master-slave architecture, where the master node can execute multiple jobs on multiple different slaves [33]. This master-slave architecture is visualized in Figure 4. Bringing physical machines up when a build job is available is not feasible, and virtualization is a way to address that [34]. The dotted lines in the figure mean that virtual machines and containerized slaves can be dynamically scaled out.

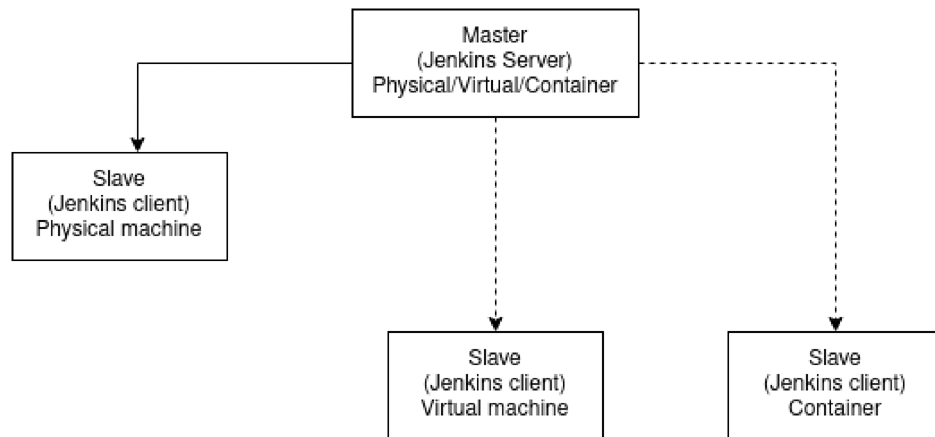

Figure 4. Jenkins Master-slave Architecture. It is possible to use different types of slave agents.

Running software builds in a distributed environment is established in the past literature [38, 39]. But that literature either focuses on a single build system (*CloudBuild*) [38] or they present an incremental build algorithm [39]. The problem is, that the CloudBuild specific things are not agnostic to other build systems and that a general pseudocode algorithm does not help the current situation with the build systems. Different build systems implement a vast range of diverse algorithms. Migrating the already used build system to a new one with any software project is a major endeavor and not a painless task [40], thus it is not likely that an already established software project would be willing to change their build system in to the one "true" and "correct" build system.

2.2.2. Build Systems in CI

Incremental builds are rarely enabled in continuous integration, since correctness is usually preferred over compilation speed. The release engineering practice is all about tooling and particular importance is in the build systems, as they enable build optimizations for faster builds. To enable incremental builds in the CI environment, the build system should: (1) produce *correct* build outputs identical to clean builds, (2) provide control over the build environment, (3) provide build linearity by building chronologically organized software configurations (small incremental changes to the source code rather than big, unrelated ones). [11]

The build optimizations are often unused, as the builds in continuous integration are built from clean sources. Release engineers want better quality guarantees for their build products rather than fast compilations. There are also challenges with enabling incremental builds on continuous integration servers, as a distributed cluster of workers does not guarantee that the consecutive builds are assigned to the same worker each time, thus making incremental builds somewhat useless. [11]

2.3. Virtualization and Orchestration

Virtualization abstracts the server hardware for software applications [41]. Virtualization gives the flexibility to adjust computing resources for a given application dynamically [41]. Packaging and running applications in virtualized environments make the manageability of those applications better [41]. Often times different applications and the servers are only used a brief moment, leaving them running otherwise idle. Even when dedicated machines run idle, their power costs, cooling costs, space requirements and other requirements continue. The idling of machines is costly for organizations [42]. Virtualization ensures that the server is utilized to its full extent and that multiple applications can run on a single physical server [42].

In addition to virtual machines, containers have been another emerging virtualization technology. Containers have gained popularity due to high performance, lightweight and higher scalability over virtual machines. The major difference between virtual machines and containers is their architecture. Virtualization in containers is done at the kernel level, and therefore containers share the host operating system (OS). With virtual machines, each virtual machine (VM) has its own guest OS. Using containers results in high performance, smaller memory footprint and reduced infrastructure cost. [43]

To manage virtual machines or containers at scale, there has to be some kind of cluster management. Cluster management is about provisioning and managing resources in a cluster. The solutions that are used for virtual machines for cluster management are called *virtual machine management platforms*. In turn, the solutions for containers are called *container orchestration systems*. [41]

Orchestrating appertain to automatically provision and manage underlying resources such as storage or VM [44]. Container orchestration is used for scaling and moving containers between different hosts [45]. It is also used to allocate resources for containers and to load balance containerized services [45]. Container orchestration tools help to manage hundreds of containers at once [45]. Container orchestration simplify container operations, brings more resilience and improves security [45].

Hundreds or thousands of CI workers put a lot of strain on the cluster storage. *Software defined storage* (SDS) tries to address issues with traditional storage systems, especially in the large data center scale [46], as the modern workloads with the emerging paradigms like machine learning and internet of things requires ever-increasing amount of data analyzing and storage across multiple nodes [47].

Kubernetes is an open-source container orchestration system that was originally developed by Google and later handed over to Cloud Native Computing Foundation for hosting [48]. Kubernetes specifies a set of abstracted building blocks that provide mechanisms to orchestrate and scale container workloads. Kubernetes handles the controlling of the containers in the background, whereas the developers are mainly concerned on what kind of workloads they want to have running in it. Kubernetes is usually deployed across multiple machines for resiliency. Single node installations of Kubernetes are used for development usage rather than as a production environment, as they miss the high availability traits and can not recover from sudden crashes. [36]

3. WORKSPACE CACHE

For this study, a simulator was developed to simulate the behavior of a Continuous Integration workload using SDS to enable incremental builds in a container orchestration system. Likewise, a solution called *workspace cache* was created to enable the incremental builds in a dynamic distributed environment. The development process is part of the DSRM, as it was presented in the Chapter 1.

3.1. System Requirements

System requirements are gathered and specified to provide a guideline how the designed artifact should be made and which evaluation criteria it should fulfill. The developed requirements were defined in two domain areas. The first domain consists of gathering background knowledge from the existing literature and from the other relevant research work. The second domain covers the real world need to get incremental builds enabled in a dynamic distributed environment.

3.1.1. Gathering the Requirements

In this study, the requirements are combined from the literature and from the organizational needs of the client of the research. The client of this study serves a company-wide standardized CI service for other software development teams. The customers of the client conduct programming in various domains, but mostly focuses on embedded software. There are hundreds of developers using the service, and the service runs thousands of source code verifying checks per day. The organizational requirements were collected from the stakeholders of the CI service through meetings and workshop sessions. The participated stakeholders titles were ranging from release engineers to software engineers, architects, product owners, business owners, and these people were working across multiple software development teams inside the company of the client of the study. The summarized requirements are visualized in Table 1. The collected requirements were:

- The developed system should provide correct build results. Compared to the clean builds, the incremental builds should be able to produce the same build outputs. If incremental builds are not able to provide correctness, then the benefits of incremental builds are worthless. A release engineer has to have a strong trust on the builds in the CI system. That trust should not be undermined by producing incorrect build outputs.
- The developed system should be efficient. When a clean build and an incremental build execution times are compared, the incremental build should provide prominent benefits. Thus, the building incrementally should reduce the build times. If this is not achieved, the extra overhead build with the solution is created for nothing.
- The developed system should support multiple build systems. As a CI service provider, the clients might use a diverse set of build systems. As long as the

build system supports incremental builds, the solution developed should provide means to use it.
- The developed system should use a Kubernetes based container orchestrator. Although multiple container orchestration platforms exist, the system should use something that uses Kubernetes at its core, since the client of this study requires it.
- The developed system should be compatible with Jenkins. The solution that the developed system uses should be able to be used in Jenkins, natively or through plugins.

Table 1. Requirements for the design artifact

Requirement ID	Requirement description
R1	The solution should provide correct build results
R2	The solution should be efficient
R3	The solution should support multiple build systems
R4	The solution should be able to run in Kubernetes infrastructure
R5	The solution should be compatible with Jenkins

3.2. Simulation Components

This section describes and illustrates the tools and systems that were utilized in the artifact design process. The used components resemble the production environment of the client to a degree that the results from the simulation environment are comparable to it. Deeper review of the production environment of the client is avoided at the request of the client. The simulation approach was selected because the testing of the solution was easier and faster than the building and using of the solution in the production environment. The proof-of-concept approach of the solution and simulation was justified for the client, as it would later serve as the basis of further decision-making for this type of feature in the production environment. First, the underlying technology is presented. Later, the proposed solution is laid out on the foundations of the presented technology.

3.2.1. Virtual Machines

Virtual machines were used to provide a reproducible environment for the container orchestrator platform. As shown in Figure 5, a single desktop with AMD Ryzen 2700X

CPU and 32 GB of RAM, was used in this study, and it was divided into 3 smaller "machines" with virtualization. Usually, container orchestrators clusters work in a manager-worker schema, where managers handle the orchestration and scheduling of the containers, workers do the actual workload [45]. For this study, one of the virtual machines with 2 CPU cores and 6 GB of RAM was reserved for the cluster manager node. The remainder two virtual machines with 3 CPU cores and 6 GB of RAM per each were reserved for the worker nodes. Each virtual machine used Ubuntu 20.04 as their guest operating system.

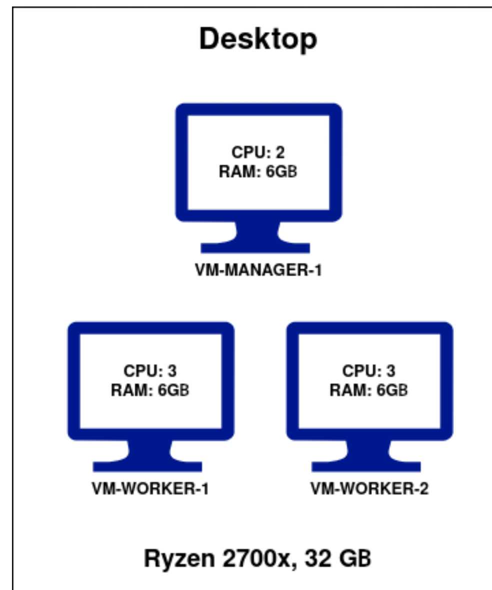

Figure 5. Virtual machines for the simulation

3.2.2. Kubernetes Cluster

Running a large Kubernetes cluster in the production environment is a challenging and laborious task. A local, small scale Kubernetes cluster allows simulating the true CI workload and offers all the same functionality as a larger one would. This study uses Microk8s [49] from Canonical to form the Kubernetes cluster.

3.2.3. Kubernetes Client

There are official Kubernetes client libraries for various programming languages and many more as a community maintained projects [50]. This study uses the official Python client library to interact with the Kubernetes Cluster. Figure 6 shows an example of client library usage where a pod is created, then all the pods are listed to see if the pod exists, after that the given pod is deleted, finally all pods are listed to see if the given the deletion of the pod was successful.

```python
from kubernetes import client, config

config.load_incluster_config()
v1=client.CoreV1Api()
pod=client.V1Pod()
spec=client.V1PodSpec()
pod.metadata=client.V1ObjectMeta(name="busybox")
container=client.V1Container()
container.image="busybox"
container.args=["sleep", "1800"]
container.name="busybox"
spec.containers = [container]
pod.spec = spec
ret = v1.list_namespaced_pod(namespace="default")
for i in ret.items:
    print("%s  %s  %s" % (i.status.pod_ip, i.metadata.namespace, i.metadata.name))
v1.create_namespaced_pod(namespace="default",body=pod)
ret = v1.list_namespaced_pod(namespace="default")
for i in ret.items:
    print("%s  %s  %s" % (i.status.pod_ip, i.metadata.namespace, i.metadata.name))
v1.delete_namespaced_pod(name="busybox", namespace="default", body=client.V1DeleteOptions())
```

Figure 6. Creating and deleting a pod with the Kubernetes client made with Python. The client was used to interact with the simulation environment.

3.2.4. Ceph and Rook

Kubernetes provides good portability on stateless applications, but two obstacles remain on stateful applications; (1) storage infrastructure abstraction and (2) portability of the actual data. Storage infrastructure abstraction hides the underlying storage infrastructure diversity, and this could be done in Kubernetes with the use of Storage Classes (SC) [51] and the Container Storage Interface (CSI) [52]. With Storage Classes, developers are able to request persistent storage just by the name of the storage "profile". CSI adds block and file storage to containerized applications and abstracts the management interface of different storage providers by providing a common cross-platform API. With CSI, the storage lifecycle operations are abstracted in such a way, that there is a common interface for volume creation, deletion, mounting and snapshotting. The SC and CSI abstractions allow developers to build and deploy containerized applications on heterogeneous storage infrastructure without the need to know about the fundamental storage architecture or the storage vendor. Data portability comes in to play, when one would want to have an ability to move the entire application stack and its data to multi-cluster or multi-region environments. An example where data portability would come to play is where the application and its data needs to move to a different region cluster to ease the load impact on your primary cluster, this requires moving the data to the second cluster first and then firing up the application in the second cluster. Moving the data becomes the issue, and it is then important to invest on the performance and efficiency of the data transportation.

To have the ability to use CSI properties with Kubernetes, one storage backend had to be chosen, that would allow dynamic provisioning of Persistent Volumes (PV). Ceph is a distributed storage system that provides file, block and object storage in a large production clusters. It also has the needed CSI abilities for this study. Rook is an orchestration tool that automates the deployment of Ceph inside the Kubernetes cluster with specialized Kubernetes Operators.

3.3. Solution Development

This section presents the development process of the simulation system with the proposed solution and also presents the interplay between different components. In this section, the proposed solution is formulated and reasoned based on the previous literature and from the organizational experience.

3.3.1. Persisting Workspace

Developers build source code on their machines all the time. After each consecutive build, the source tree and file system have all the previous build outputs passing along. Therefore, the build system can then reuse the existing outputs in the consecutive builds. This kind of reusage of build outputs is not usually used in CI server setting, as it is commonly done with remote caches, but they do not provide generality in a sense that none of the remote build cache backends are compatible with some other build system. Enabling a similar kind of reusage, as on the desk of the developer, requires a persistent storage for each CI worker in order it to work. Dynamic distributed build environments are ephemeral by nature, as they get destroyed after each build they are assigned to do. This makes it hard to coordinate the storage persistence on dynamic distributed build environments.

Persisting workspaces are already possible with static CI workers. However, static CI workers are not dynamic by their nature. Static slaves can not be scaled out easily and even if they could be, one would not be able to share the build state from an existing worker to a freshly deployed worker replica. Also, static workers just sit idle and wait for work. This waiting is a waste of resources, if we consider that the underlying infrastructure has maintenance and operation costs. Only when the CI server needs to offload its build work to the workers, the workers become useful.

Remote caching build outputs is a working solution, but the lack of general remote build cache is problematic from the perspective of a CI service provider. Clients might use a heterogeneous selection of build systems, and it might be that the given build system that the customer uses, does not have remote caching backend implemented. If one would want to make such general remote cache, it would be very hard to support the modelling of the builds of each and every build system. Likewise, from the infrastructure perspective, all different remote caches would need a different backend to be deployed. Maintaining, operating and optimizing the usage of those remote cache backends would turn out to be laborious. The cache might get corrupted, it would require multiple backends per customer, and it would only multiply as the CI service customer count grows. The remote cache backends are usually HTTP based file servers. HTTP servers are a well established technology and studied widely regarding optimizations and implementation. The problem with remote caches is that implementations are too diverse.

Eventually, the build outputs are pulled in to the source tree one way or another, file by file, or all files at once. Remote caches get these files to the file system one by one, by requesting them from the backend. This study considers the all files at once, as the alternative solution. If the persistent workspace and moving of all build output files at once is combined, it might be possible to pass the build outputs

to different builds for reusage. Persisting the workspace in a dynamic distributed build environment, where the CI worker might pop up at any given underlying cluster node, is problematic. All the build files need to be moved between containers through network, since the containers might be geographically sparse and might not be located at the same underlying hardware node. This shifts the problem in to an infrastructure and a software problem, where the networking plays a big part, as it works as the file transfer layer.

Kubernetes has file storage abstractions like the CSI and Storage Classes, whom help to abstract the storage backend from the container. These two abstractions allow the solution to mount the persistent storage volume as the CI worker workspace. Later, when some container wants to have the same workspace mounted, it just refers to the workspace volume mount with the same persistent volume name. This allows the build output "state" to be passed along with the storage volume.

The build system should behave as it does on the machine of the developer. First building the source code, then file system preserves the outputs. Later build invocations are able to use the old build outputs from the file system, therefore reducing the build time. At the least, a correct build system should preserve some kind of build state on the disk [22].

The proposed solution is shown in the Figure 7. The idea is to carry on the state from the build pod to the next by keeping it in the PV. Figure 7 shows a sequence diagram of a developer requesting a change in the source code by issuing a pull request to the VCS. First, cluster admin makes sure there is a SC available. Then the developer issues the pull request. CI server starts by requesting a PV through a PVC, and later it gets it from the External Provisioner. Now that the PV is ready, the pod with the PV mounted in it could be created. When the pod is up and ready, the actual checking of the source code could begin. The CI server clones the code under the change, when done, it builds and tests the code. The result is notified towards the VCS and back to the developer. In Figure 7, it is assumed that some changes were made to the original change request and the source code has to be checked again. Now the persistent volume is already there and there is no need to request it. However, a new pod with the persistent volume mounted in it is created and the cloning and checking is performed like before. But this time, all the previous build outputs should be available for the build system and the duration of the build should be reduced, thus giving the feedback faster to the developer. Once the pull request is done, the CI server tears down the persistent volume to prevent wasting storage capacity.

Persisting the workspace is orthogonal to the remote caching. These techniques should be able to work simultaneously and bring the benefits of both. Persistent workspace should provide up-to-date build outputs and downloading any files should be unnecessary. If something has to be built, the remote cache would then serve the needed files and remove the necessity of compiling the output.

This solution should provide the missing linearity to builds. With this solution, a given CI worker in a cluster should have access to the build products of the previous builds, not requiring a clean build. The communication and caching of the builds happen on the infrastructure level, where the previous build state is reference by a persistent volume name.

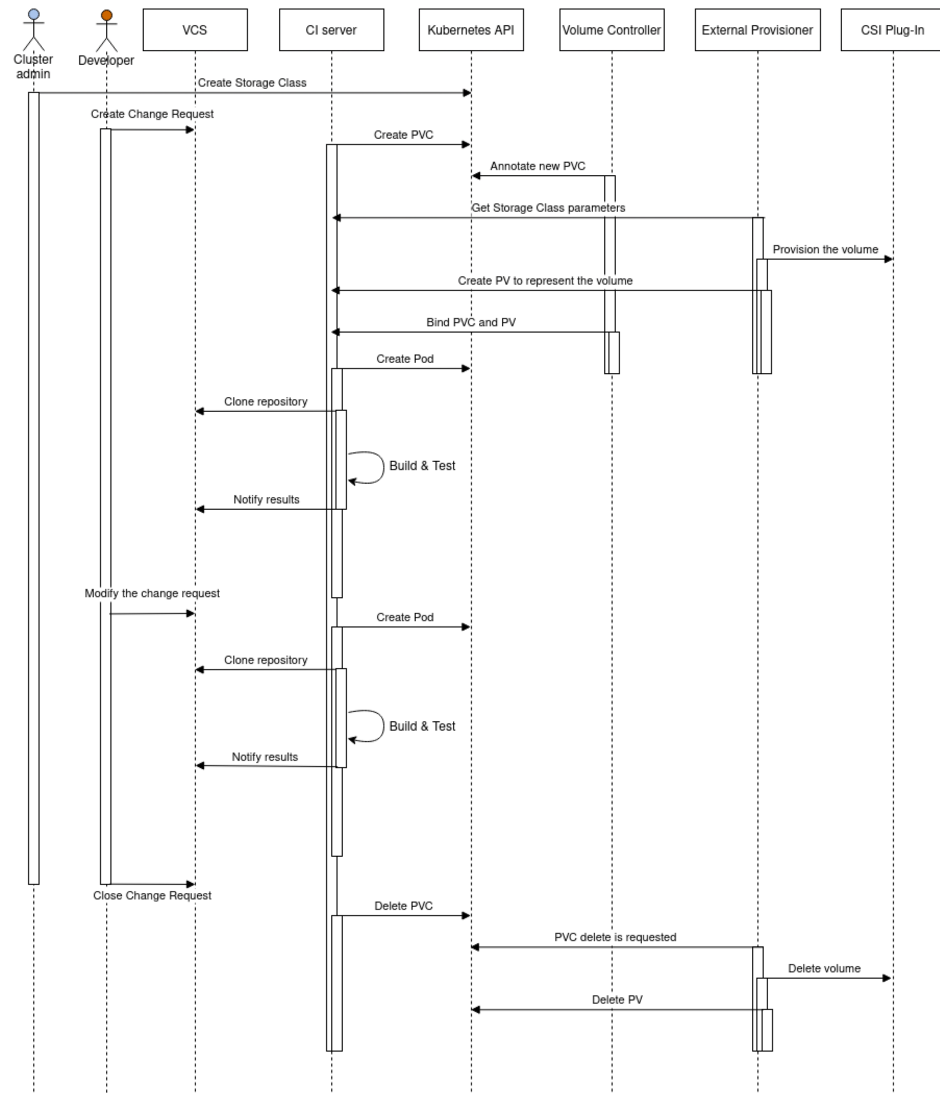

Figure 7. Persistent Volume holding the build state throughout the Change Request. Illustrating the possible workflow of developers and cluster admins in the production environment.

3.3.2. Simulating CI Environment

The simulation cuts few corners compared to Figure 7 as setting up the CI server is unnecessary. This study simulates CI server with a developed Python script to be able to benchmark the solution efficiently. Further, the creating of the pull request could be simulated by running consecutive commits of a project, which would represent the minor changes that the developer does for the code.

For the simulation, two open-source projects were chosen from the GitHub. The first one is FFmpeg, an open-source multimedia conversion tool. It is popular in GitHub, and it is well known project overall (over 30k stars in GitHub). The main reason to choose FFmpeg was to have a project with make as the build system. Make was already discussed in the Chapter 2 and according to the previous literature it can not be considered as a correct build system. The second project for the simulation was

spring-boot, a boilerplate repository for Java based web development framework called Spring. It was also very popular, having over 61k stars in the GitHub. The reason to choose spring-boot, was to have a Java project with Gradle as the build system. The Gradle would give a good comparison point for the make. From both of the projects, 10 chronological commits (not necessarily consecutive) were chosen to have a progressive code changes to benchmark the solution against.

In the Kubernetes cluster of the simulation, illustrated in Figure 8, a Ceph storage platform was deployed in to the cluster with the help of Rook. Each cluster node had an OS disk and a 30 GB virtual disk dedicated to the Ceph pool. Out of those disks, a Ceph block pool was made, where the SC could then request block volumes from. By requesting a PVC, a pod could get a persistent volume to mount into and at the same time a place where the state of the software build could now persevere. Also, it would be easy to start from a clean slate by just deleting the existing PVC and creating a new one.

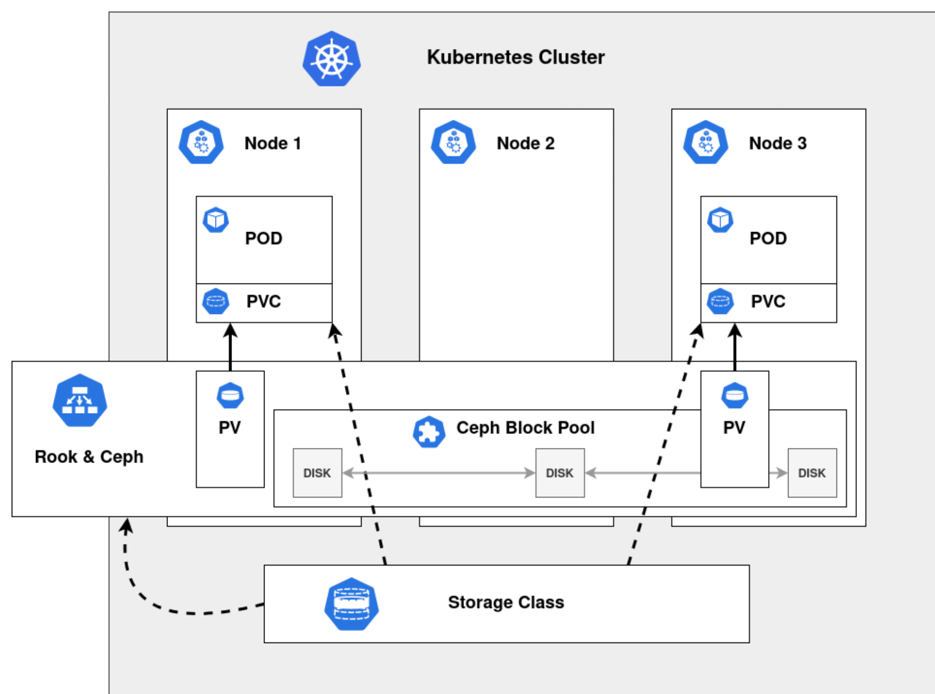

Figure 8. Hyper-converged Ceph storage in Kubernetes Cluster, like it was used in the simulation.

Caching the workspace can be thought as more granular than environment cache. Environment cache stores whole file systems, whereas workspace cache stores only the source tree of the given project. Build cache stores only the build task outputs and thus, the workspace cache can be seen more coarse than build cache. Therefore, to original Figure 3, a new level of a software build cache can be added. Figure 9 has the workspace cache added between the environment cache and the build cache.

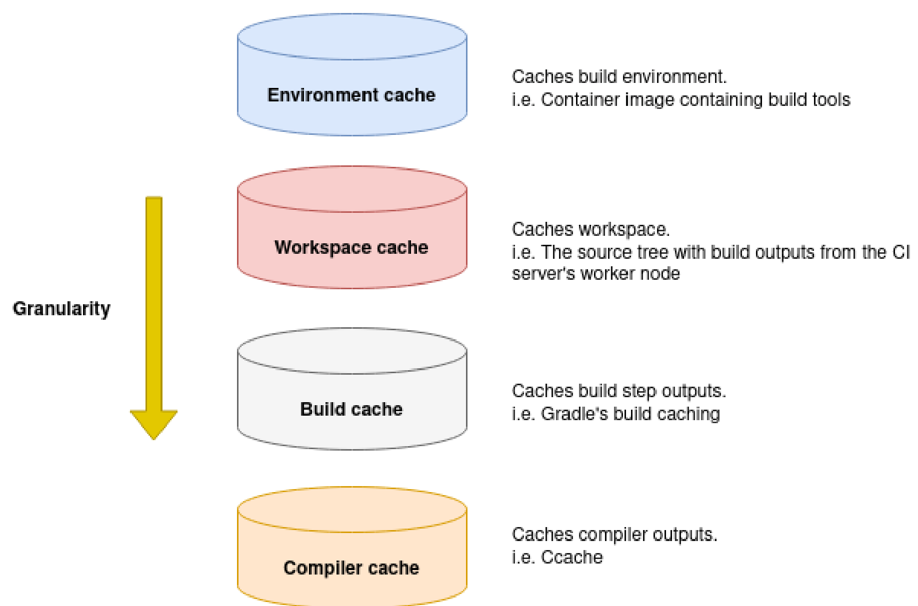

Figure 9. Addition of workspace cache to the levels of caching with software builds

4. EVALUATION

The meaning of the evaluation activity is to find out how effective the designed artifact is [53]. The evaluation activity can also be used to distinguish shortcomings and enhancements for the developed solution [53]. These both aspects are useful for understanding the research problem and therefore help to solve it [53]. Thus, to evaluate the real world angles of the designed artifact and to find out the impact of it on the research problem, an evaluation experiment was conducted. The main target of the evaluation was to assess the design artifact using the requirements set in Table 1.

4.1. Research Setting

The overall objective of the experiment was to assess the effectivity of the persisted workspace approach for incremental builds. More over, it was interesting to see if the developer feedback time could be reduced by persisting the workspace in the CI worker. However, this study is based on previous literature and organizational needs. The composed requirements in Table 1 create the backbone of the evaluation criteria of this study, and therefore assessing them is the main objective of this chapter.

To ensure that the required properties of the system are fulfilled, some quantitative measurements were taken to assess the fulfillment. Some requirements could be met without measurements, but rather with reasoning and by justification.

To ensure that the build outputs are correct, the outputs from incremental builds should be equivalent to the ones from the clean builds. In order to measure this, we had to capture the meta information about the build outputs of each project, something that would identify the file contents. Each software project has different build outputs, but usually these projects aim to build at least one binary or package as their main output. Just like in the case of Figure 2 where the *program* was the main build output.

The project *FFmpeg* main build outputs were easy to find from their GitHub-page README. The files that were used to compare as build outputs in this study were; *ffmpeg*, *ffplay* and *ffprobe*. All of them were binary executables. An SHA-256 checksum was used on each of the main build outputs of the *FFmpeg*, to capture the main output file contents properly for the comparison.

The main build outputs of the project spring-boot were difficult to find. As it was a Java based project, the JAR files produced after the build could be considered as the main build output. But the *spring-boot* produced a lot of JAR files from the modules it had. Therefore, it was hard to determine which was the main outputs from all of them. In the end, it was seen best to use all the JAR files as the reference for checking the build output correctness. With the project spring-boot the checksum comparison was not used. This was due to the nature of JAR files, they are archive files like ZIP files. Each build invocation might embed a timestamp or change the file ordering inside the archive file, causing the checksum to be different each time. Thus, it was seen best to resort to file sizes on the disk as the comparison standpoint with the JAR files.

Later, the file checksums and the file sizes that were gathered from the simulation runs were simple to compare against. The comparison of checksums enabled to calculate correctness rate for each project's build simulations. As seen in the Equation

(1), the correctness rate of the project's build outputs of the project was calculated rather simply.

An example: with project's build simulation, correctness rate r is

$$r = \frac{c}{s}, \tag{1}$$

where c is the amount of the correct build outputs, and s is the amount of all the successful builds.

To capture the efficiency of the builds, the time taken to build the whole project was measured. As Figure 7 shows, the source code checking phase involved cloning the source code from the VCS and after that, building and testing the source code. Now, as there were two distinct phases of the simulation runs, both of them were recorded individually. One could argue that the cloning of the source code is not part of the build process, but it is a part of the CI server source code checking process. The different phases durations were captured with a program called *time*, which runs specific commands with the given arguments and when the command finishes, *time* gives the timing statistics about the program run. These timing statistics were captured in the measurements for each simulation run for later analysis.

In order to see which simulation runs were successful, the return code of the simulation runs were captured. As the simulation phases were defined in a singular bash script and executed in it sequentially as functions, it meant that if any of the commands inside the script were to fail, the return code would be something other than zero. This was later used to determine if the simulation build was successful or not.

For both of the simulation projects, 10 chronologically ordered commits were selected for the simulation runs from their commit history. Each potential commit was first tested with the local development environment so that the build would actually succeed and produce a proper build output. This selection process of commits caused the selected commits to not be strictly consecutive, but they still were chronologically ordered. Thus, the selected commit chain could still be thought of as the developer was adding small changes to the source code. Table 2 and Table 3 show the final commits that were selected for the simulation runs, the top most hash represents the oldest and the bottom most the newest. The simulation run starts from the oldest commit and progresses towards the newest.

Table 2. Used FFmpeg commit list

ID	Commit hash
1	e71d5156c8fec67a7198a0
2	d2d8b9b972ba2df6b2a2eb
3	ab05e9a7f277b3eb47b23d
4	98ec4261fd75b47a18cedf
5	84241e63cf2f3cc8f7d8a1
6	52a14b8505923116ed6acc
7	2c77d9150d101582b75d8a
8	075c2308e3eb0421c4f708
9	710dce131fdb6c1ebec1f2
10	b90341d1d5585b7181873a

Table 3. Used spring-boot commit list

ID	Commit hash
1	f9db6ad2378a11254dec63
2	c45d3e50ef9f462d16fecd
3	2d9dfe9bc5a9f29e695ec5
4	e9673fd62296f4052a24fb
5	36a60d8e2e1b487ef1700c
6	f433d250e1c2a20451ef20
7	d354c03e6398f49fb12531
8	b2e63db57fc91c74aaaf65
9	85ad4fd4b43ec9de962109
10	1ba1eff63a6791476e0fed

The data collection span across 6 days, since the simulation cluster was unstable. Mostly, the cluster would hold together for a day and then go to a bad state where no simulation runs were possible to run without manual intervention. Once the cluster got back up in a good state, new simulation runs could be run again. This constant patching of the cluster caused the data to be recorded in chunks, but the dataset should then also capture the unreliability of the cluster when the software builds are run in it.

4.2. Data Analysis

Figure 10 displays how the data of the FFmpeg simulation runs looked like. Out of the 270 clean and incremental built commit pairs of the project FFmpeg, 210 were found to be successful to a degree where the clean build and the incremental build were being able to be compared against each other. Out of those 210 runs, 6 runs were incorrect, resulting in a correctness rate of 97.14%.

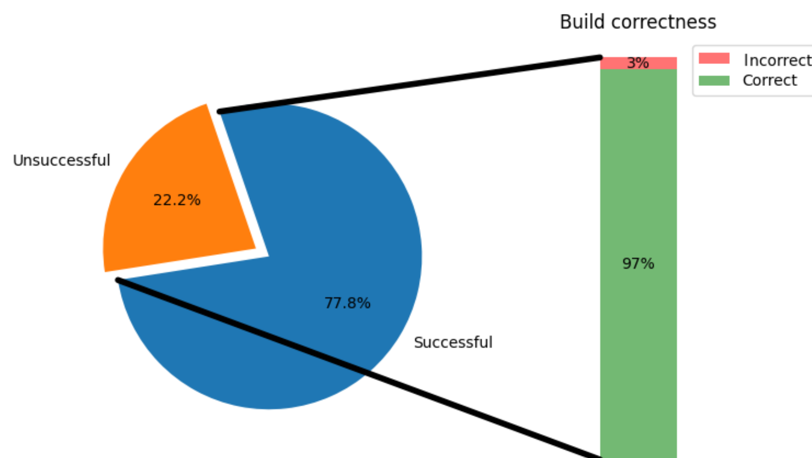

Figure 10. Data from the project FFmpeg. The ratio of successful and unsuccessful simulation runs. 97.14% of the successful builds were correct.

The clone times of the FFmpeg part of the dataset are visualized in Figure 11. The clean workspace had to clone the source code repository each time, causing the simulation to spent time downloading files from the VCS for 50 seconds minimum. On the persistent workspace counterpart, the first commit had to be cloned into a clean volume, but all the subsequent ones can benefit from the persistency of files. Thus, the cloning of the source code in the persistent workspace after the initial commit have to only fetch the missing files rather than all the files. The times from the clean workspace clones are more dispersed than the persistent equivalents.

The line graph in Figure 12 shows the median clone times from different commits with FFmpeg. The persistent workspace seems to save a minimum of 50 seconds from the source code clone time, with some commits even more than that, up to 100 seconds. With the first persistent workspace commit clone, the clean one outperforms the persistent one. This could be due to the overhead caused by the distributed storage.

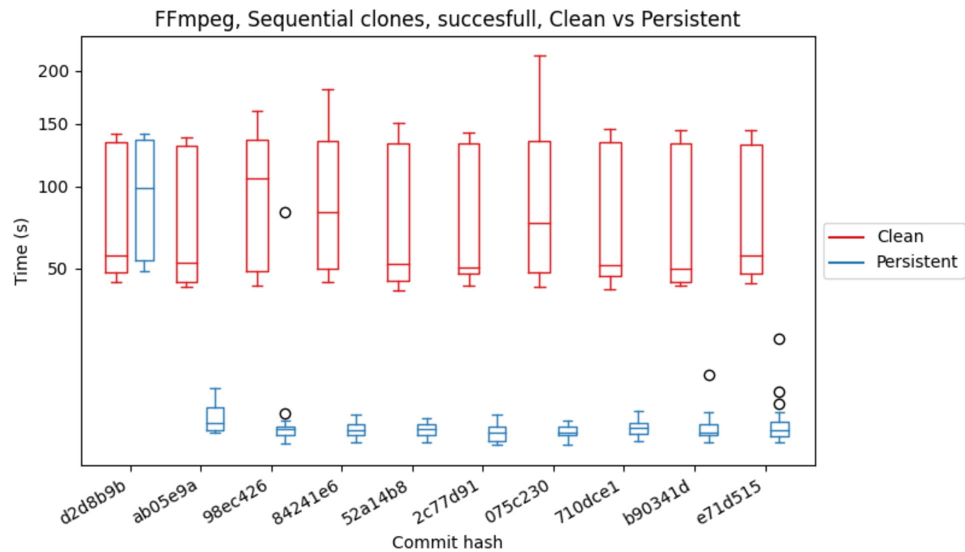

Figure 11. Data from the project FFmpeg. Box plot that shows the structure of the data. Deviation with the clean clones is large and with the persistent clones there is not much of deviation. There are not many outliers in the data.

The overhead of abstracting the storage and using the storage over the network slows the storage down. With the clean workspace, the storage volume is served from the host machine where the pod runs, reducing the overhead.

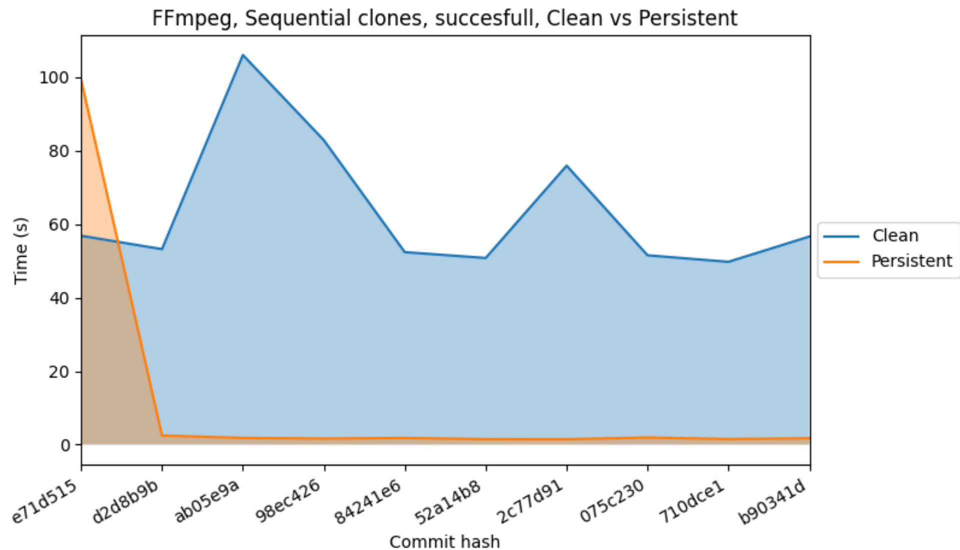

Figure 12. Data from the project FFmpeg. Line graph of the most common clone times from the data. Persistency consistently reduces the clone time compared to a clean clone by a large margin.

The build times of the FFmpeg part of the dataset are visualized in Figure 13. The clean workspace had to build all the build targets each time, causing the simulation to spent time on building at least 300 seconds minimum. On the persistent workspace

counterpart, the first commit had to build all the build targets from scratch, but all the subsequent ones could benefit from the persistency of files. Thus, the building in the persistent workspace after the initial commit has to only build the missing build targets rather than all the build targets. The times from the persistent workspace builds are more dispersed than the clean equivalents.

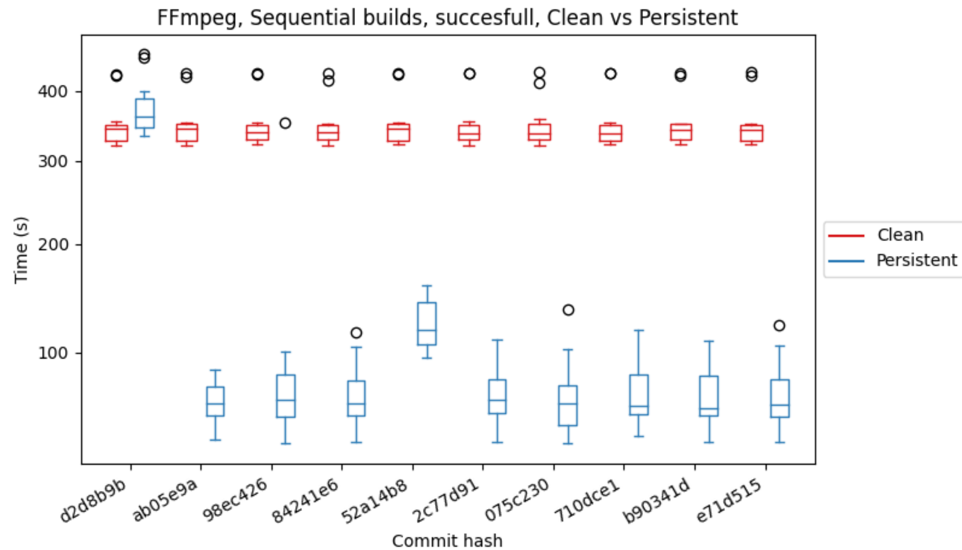

Figure 13. Data from the project FFmpeg. Box plot that shows the structure of the data. Deviation with clean builds is small. The deviation with persistent builds is moderate. Some amount of upper outliers can be detected.

The line graph in Figure 14 shows the median build times from different commits with FFmpeg. The persistent workspace seems to reduce the build time to a minimum of 200 seconds, with some commits even more than that, up to 275 seconds. With the first persistent workspace commit build, the clean one outperforms the persistent one. The difference is smaller than in Figure 12. It could be that the source code cloning has a bigger load on the file system than the building action. With the source code building, the CPU does most of the work and the storage is not in the highlight as long as the file system can feed the files into a build system with appropriate speed.

Figure 15 displays how the data of the spring-boot simulation runs looked like. Out of the 220 clean and incremental built commit pairs of the project spring-boot, 155 were found to be successful to a degree where the clean build and the incremental build were being able to be compared against each other. Out of those 155 runs, none were incorrect, resulting in a correctness rate of 100%.

The clone times of the spring-boot part of the dataset are visualized in Figure 16. The clean workspace had to clone the source code repository each time, causing the simulation to spent time downloading files from the VCS, for 18 seconds minimum. On the persistent workspace counterpart, the first commit had to be cloned into a clean volume, but all the subsequent ones can benefit from the persistency of files. Therefore, the cloning of the source code in the persistent workspace after the initial commit have to only fetch the missing files rather than all the files. The times from the clean workspace clones are more dispersed than the persistent equivalents.

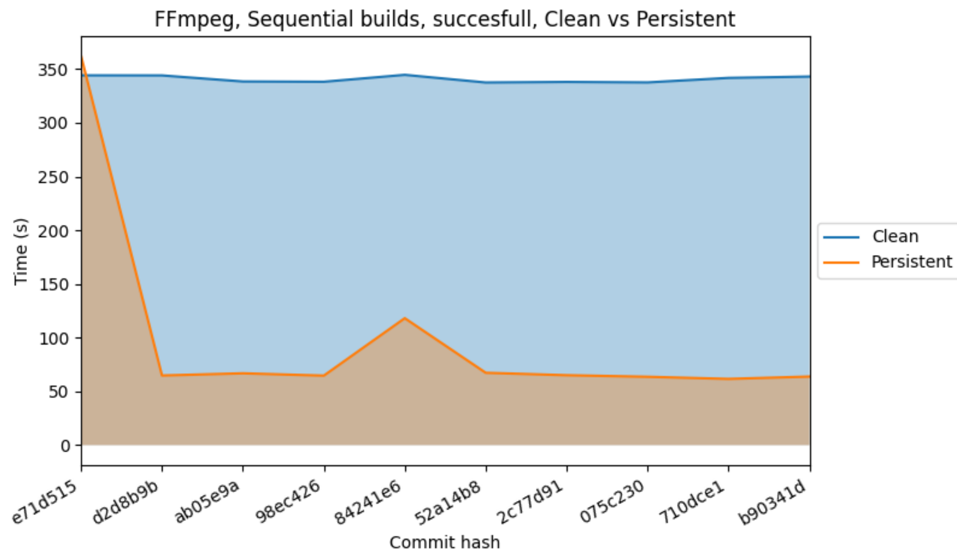

Figure 14. Data from the project FFmpeg. Line graph of the most common build times from the data. Persistency reduces the build times manyfold.

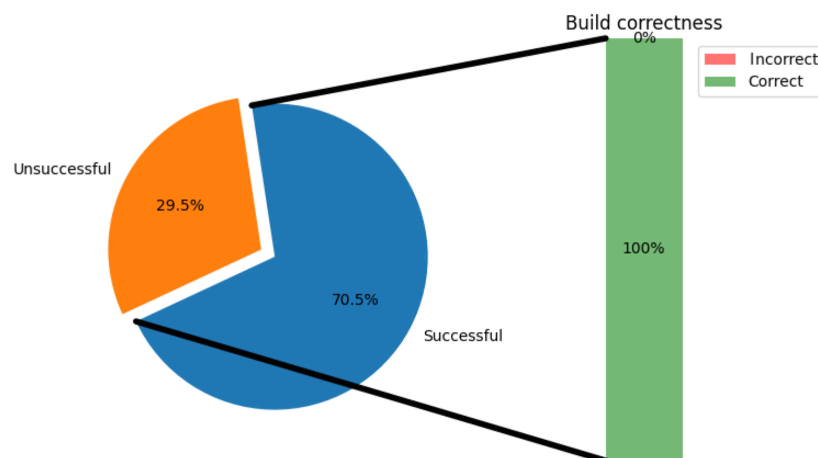

Figure 15. Data from the project spring-boot. The ratio of successful and unsuccessful simulation runs. 100% of the successful builds were correct.

The line graph in Figure 17 shows the median clone times from different commits with spring-boot. The persistent workspace seems to save a minimum of 15 seconds from the source code clone time, with some commits even more than that, up to 25 seconds. With the first persistent workspace commit clone, the clean one outperforms the persistent one. This could be due to the overhead caused by the distributed storage.

The build times of the spring-boot part of the dataset are visualized in Figure 18. The clean workspace had to build all the build targets each time, and the persistent did not. On the persistent workspace counterpart, the first commit had to build all the build targets from scratch, but all the subsequent ones could benefit from the persistency of

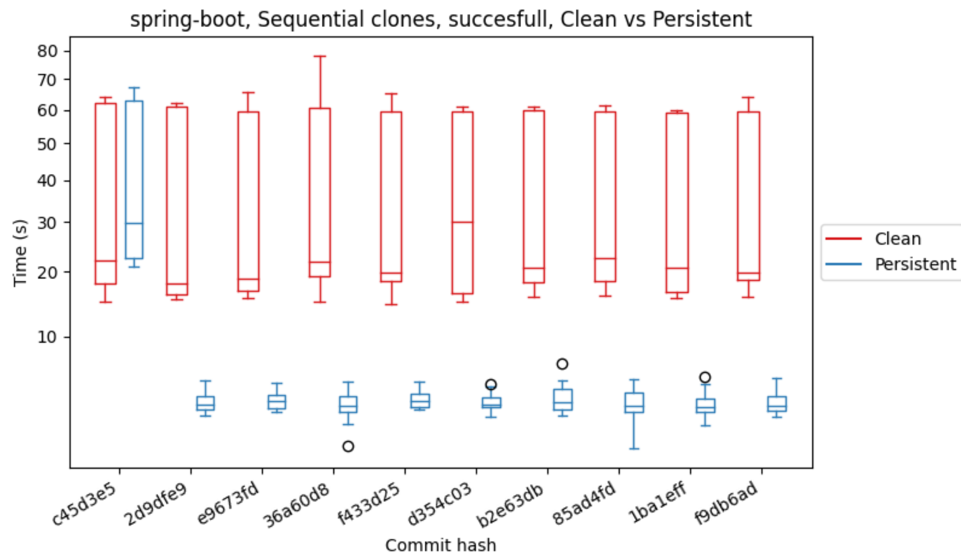

Figure 16. Data from the project spring-boot. Box plot that shows the structure of the data. Deviation with the clean clones is large and with the persistent clones there is not much of deviation. There are not many outliers in the data.

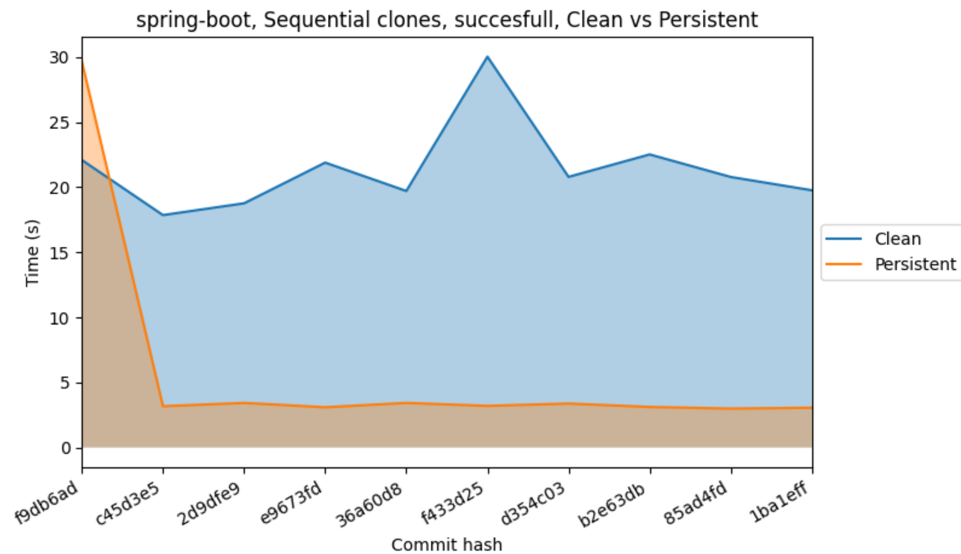

Figure 17. Data from the project spring-boot. Line graph of the most common clone times from the data. Persistency consistently reduces the clone time compared to a clean clone by a large margin.

files. Thus, the building in the persistent workspace after the initial commit has to only build the missing build targets rather than all the build targets. Compared to the FFmpeg builds, with spring-boot, the clean and persistent builds are more close to each other. There are commit builds where the clean ones outperform the persistent counterparts, but also commit builds where persistent outperform the clean ones.

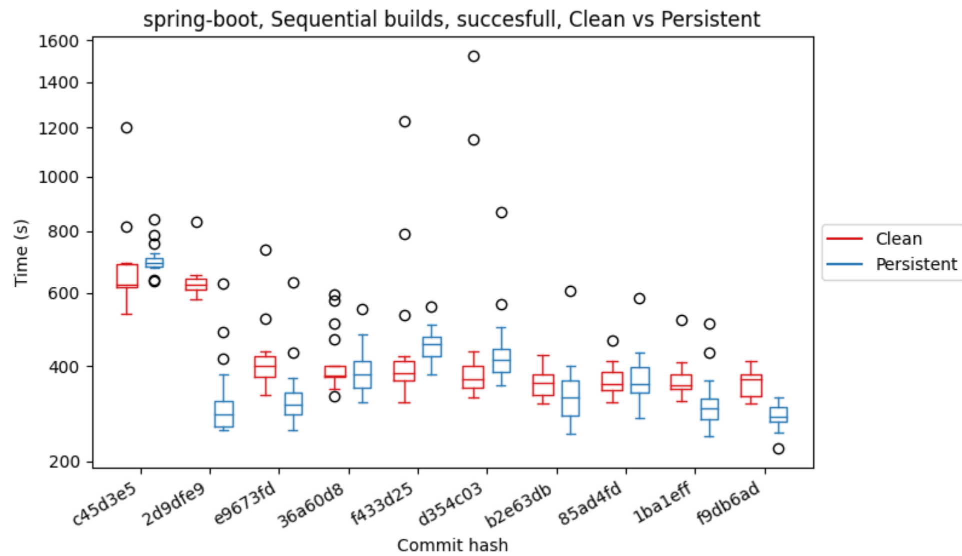

Figure 18. Data from the project spring-boot. Box plot that shows the structure of the data. Deviation between the build data is moderate. Some commits have big upper outliers.

The line graph in Figure 19 shows the median build times from different commits with spring-boot. There is a commit where the persistent build saves over 300 seconds from the build time compared to the clean build, and commits where the clean build is approximately 100 seconds faster than the persistent build. One can not tell if there is an actual performance gain with persistent builds according to this graph.

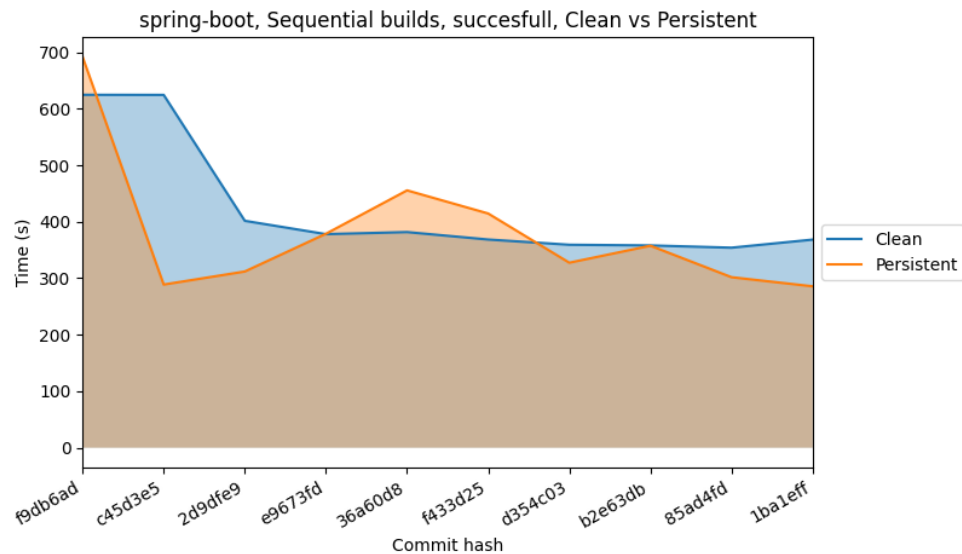

Figure 19. Data from the project spring-boot. Line graph of the most common build times from the data. Persistency does not bring that much of an advantage over clean builds.

One thing that could explain the uniformity in Figure 19 build times is remote caching. From the source code of the spring-boot repository, it could be checked that the remote caching feature in the configuration files of Gradle was turned on. Furthermore, the collected debug output verified that where the clean builds were able to use the "FROM-CACHE" feature of Gradle and the persistent equivalent was using the "UP-TO-DATE" most of the time, but also used the "FROM-CACHE". This means that both, clean and persistent builds used the remote caching feature of Gradle, but the persistent could also reuse the locally cached tasks and check those tasks' status with up-to-date check.

The slowness within some commits with the persistent builds could be by the reason of the overhead that the distributed storage creates. Persistent builds being faster at some commits could be by cause of remote cache serving files slower or that there were more files to pull from the remote cache with those commits.

5. DISCUSSION

The DSRM method was used to answer the research questions described in the Chapter 1. The first research question, **how are incremental builds enabled?** was answered by the process of reviewing the existing academic literature. The incremental builds are made possible by the build systems. Builds systems that implement up-to-date checking for their build tasks are able to produce the benefits of the incremental builds. However, it must be noted that no build system is the same as the other. Therefore, different implementations do exists, and they might not be trivial to support from the perspective of CI service provider.

The answer to the second research question, **what are the differences of running incremental builds in the CI server compared to the machine of the developer?** was also provided by the existing literature and further established from the experience of the organization during the design and development cycle. The literature review identified that in the CI server setting, release engineers tend to aim for correctness over the reduced feedback time. Furthermore, the CI servers usually lacks the linearity required for the incremental builds. These findings were used in the requirements of the solution in order to produce a working solution in a CI server setting. The proposed solution should not affect the correctness of the builds, since build systems are the ones responsible for the build correctness. Therefore, it is arguable if the way the correctness is measured here is the right kind. From the existing literature, it was prominent that the remote caching could be a good solution for CI servers. However, the need to deploy remote cache backend servers for each build system there is, was not considered as desired, but it did inspire the proposed solution of workspace cache.

The answer to the third research question, **can the incremental builds be enabled in dynamic distributed build environment?** was answered by design, developing and evaluating the proposed solution. The design and the development cycles provided the basis for the workspace cache. The organizational requirements provided the dynamic and distributed nature of the infrastructure environment. The incremental build requirements like efficiency and correctness were conducted from the existing literature. The solution, workspace cache, did produce satisfactory results with the given two software projects selected for this study. Thus, Incremental builds can be enabled in the context of dynamic distributed build environment. Some arguable observations from the final solution were found, like the correctness measurement used in this study and the effect of the overhead for the distributed storage, but they are left for the future researchers to further study on.

This work proves to the client of this study that the incremental builds can shorten their software development feedback cycles in their CI server environment. The solution of this study works as a proof-of-concept implementation and works as the background knowledge for their further decision-making towards incremental build topic. Although, the simulation environment does not represent the actual production environment, the idea is transferrable in to it. The production implementation requires more work and testing to be viable.

5.1. Requirements

To conclude, how the proposed solution artifact fulfills the requirements defined in Table 1. In this section, each requirement is checked over individually to assess its fulfillment.

R1: The solution should provide correct build results
Not fully achieved. The proposed solution could not produce 100% correct build outputs. It is still debatable if this is caused by the proposed solution itself. Build systems are the source of correctness, and old imprecise build systems do not fulfill the needs of the software developers of today. Testing the building of the FFmpeg project with another (correct) build system, would probably produce 100% correct outputs.

R2: The solution should be efficient
Achieved. The solution seemed to generally reduce build times, but the spring-boot project utilized remote-caching to an extent where the clean builds could be faster in some cases.

R3: The solution should support multiple build systems
Achieved. The solution supports build systems make and Gradle. The proposed solution also demonstrated its orthogonality to remote caching with Gradle. The solution is not tested with other build systems, but if any given build system supports checking up-to-date build files from the source tree, then the build system can be considered supported. It is also important that the possible build system follows the definition of the correct build system [22], as it has affected the foundations of the proposed solution.

R4: The solution should be able to run in Kubernetes infrastructure
Achieved. The solution runs in Kubernetes and is built upon the principles of data storage of Kubernetes.

R5: The solution should be compatible with Jenkins
Achieved. The solution uses PVC name as the reference to the given workspace. Jenkins has Kubernetes-plugin where one can define a build pod with the PVC name to persist the workspace.

5.2. Challenges

The overhead of running the storage over the network has an effect on the results. The distributed storage in the Kubernetes is not equal with the ephemeral disk that the clean builds use. This is expected since the networking bandwidth in the simulation environment was limited to the emulated network drivers of the VirtualBox. A micro benchmark was conducted with a tool called fio [54] on both of the storage options. According to the measurements, the volume from the Ceph storage had an average speed of 6603 KiB/s and the ephemeral volume had an average speed of 20.6 MiB/s.

The simulation would be more balanced if the storage options would be closer to each other with their speeds.

The gathered data from spring-boot project leaves an open question about the feasibility of the workspace cache against remote caching. Although, the workspace cache is more of a generic solution for all correct build systems, the build times when the remote cache is turned on is minimal or even inferior.

This study assumes that the simulation cluster nodes are co-located in a single location, without geological distance. In a more real world setting, the cluster nodes might be located in different geological regions and be mapped in to different zones. As of current, this simulation does not consider this situation and therefore can not evaluate it.

Incremental builds in a CI environment is not studied much, and future researchers could try to fill this gap with further studies. This study uses the CI infrastructure to forge a solution for this issue, but more could be done on the remote caching side to provide a build system agnostic remote cache, removing the need to maintain multiple remote caching backends.

The experiment was conducted with 10 chronologically ordered commits for each of the projects. It can not be said with confidence that the results will generalize to a broader setting. To have the results more generalized, different sorts of projects should be taken to the experiment to cover the variety of different build systems and programming languages. Furthermore, more than 10 commit chains should be tested and possibly have multiple commit chain per project which to test. It is hard to implement such extensive experiment, since it would take a lot of resources from the researchers, but it would ultimately produce a generalizable result.

According to the results of this study, the first commits that are build with workspace cache take the same time or more than the clean build. This means, that if a change request is issued, the initial commit does not benefit from the incremental building. All the consecutive commits to the change request on the other hand will, but it would be more beneficial to the developer to get the advantage of incremental builds already from the first commit on. There could be a way to warm the workspace cache beforehand somehow. which would have some level of build outputs available for the first commit. More research is needed to find out how the workspace cache could be warmed up with build outputs to get the incremental build benefits from the start of the change requests.

The Kubernetes cluster that was created for the experimentation broke down multiple times during simulations runs. Usually, the cluster lasted for a day and then one or both of the worker nodes would disconnect from the manager node. This was partly recognized to be a problem with the virtualization platform which provided the emulation of the networking, but was not fully extinguish and the cluster would still lose connectivity to each other periodically. Moreover, the block pool of the Ceph storage that was deployed inside the Kubernetes did get corrupted once during the study and this caused the Kubernetes to be unable to use Ceph storage. The corrupted Ceph block pool issue was resolved with a deployment of a fresh pool with fresh disks. The steps of the deployment of this experiment were documented, and therefore it was easy to conduct the re-deployment. For a more reproducible version, Vagrant [55] files could be developed in order to set up the whole simulation environment with a single command.

6. CONCLUSION

The main objective of this thesis was to find out if incremental builds can be enabled in dynamic distributed build environment in order to shorten the time taken to deliver the feedback from the checking of the source code to the developers. The secondary objective was to elucidate the backgrounds of incremental building inside a continuous integration setting, as it is not researched that much and could be clarified to the organizational stakeholders.

The research problem was approached with DSRM methodology. The design and development phases were a continuous process, which lead to producing a system with the desired functionality. The produced artifact was assessed using the identified requirements, and it ended up fulfilling most of them. Some requirements were not fully met as there were resource and time constrains.

This thesis starts from problem identification, where the potential of incremental building in CI is shown and demonstrated. It is also demonstrated what have been the impediments for not using the incremental building in the CI setting. From the gathered requirements, a solution called workspace cache was developed. A simulation environment was set up to provide an assessing environment for the solution, and the solution was benchmarked against clean builds in it. Later, the workspace cache was assessed using the requirements that were gathered earlier.

Workspace cache as a solution does prove to speed up the software building process when remote caching optimization are not in use. The speed-up is so significant that it is feasible to start to try out the solution in a real world production setting. The workspace cache is not fully build system agnostic, to be fully compatible with all correct build systems the solution should be demonstrated to be able to work with them, meaning more research needs to be done. Furthermore, having remote caching feature enabled in build system seem to outweigh the benefits of the workspace cache, but this result is disputable as the distributed storage solution used in this study was rather slow compared to the ephemeral storage. The results of this study are promising, the workspace cache as a solution is shown to shorten the build times of a software build, it also provides a novel approach in bringing the incremental builds to continuous integration.

This study proves that the incremental building in CI setting is possible and potential. More research is to be done to prove that the solution works also outside the simulation environment. This thesis provides a concrete solution for a tricky problem, and the benefits of it can be measured in money or time, which ever is preferred. The issue with build correctness is caused on from the build system and not from the solution itself. Therefore, the correctness issue turns in to a problem of shifting software developing organizations away from using incorrect build systems in order to produce correct incremental builds.

7. REFERENCES

[1] Tripathy P. & Naik K. (2015) Software evolution and maintenance : a practitioner's approach. John Wiley & Sons, Hoboken, NJ, 418 p. DOI: http://dx.doi.org/10.1002/9781118964637.fmatter.

[2] Cao Q., Wen R. & McIntosh S. (2017) Forecasting the duration of incremental build jobs. In: Proceedings - 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, p. 524 – 528. DOI: http://dx.doi.org/10.1109/ICSME.2017.34.

[3] Erdweg S., Lichter M. & Weiel M. (2015) A sound and optimal incremental build system with dynamic dependencies. In: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA, p. 89 – 106. DOI: http://dx.doi.org/10.1145/2814270.2814316.

[4] Konat G., Erdweg S. & Visser E. (2018) Scalable incremental building with dynamic task dependencies. In: ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, p. 76 – 86. DOI: http://dx.doi.org/10.1145/3238147.3238196.

[5] Sotiropoulos T., Chaliasos S., Mitropoulos D. & Spinellis D. (2020) A model for detecting faults in build specifications. Proceedings of the ACM on Programming Languages 4. DOI: http://dx.doi.org/10.1145/3428212.

[6] Acar U.A. (2009) Self-Adjusting Computation: (An Overview). In: PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation, PEPM '09, p. 1–6. DOI: http://dx.doi.org/10.1145/1480945.1480946.

[7] Randrianaina G.A., Khelladi D.E., Zendra O. & Acher M. (2022) Towards Incremental Build of Software Configurations. In: ICSE-NIER 2022 - 44th International Conference on Software Engineering – New Ideas and Emerging Results, pp. 1–5. DOI: https://dx.doi.org/10.1145/3510455.3512792.

[8] Randrianaina G.A., Tërnava X., Khelladi D.E. & Acher M. (2022) On the Benefits and Limits of Incremental Build of Software Configurations: An Exploratory Study. In: ICSE 2022 - 44th International Conference on Software Engineering, pp. 1–12. DOI: http://dx.doi.org/10.1145/3510457.3513035.

[9] Pugh W. & Teitelbaum T. (1989) Incremental computation via function caching. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 315–328. DOI: http://dx.doi.org/10.1145/75277.75305.

[10] Lam A. (2018) Using Remote Cache Service for Bazel: Save Time by Sharing and Reusing Build and Test Output. Queue 16, p. 31–43. DOI: http://dx.doi.org/10.1145/3277539.3287302.

[11] Maudoux G. & Mens K. (2017) Bringing incremental builds to continuous integration. In: Proceedings of the 10th Seminar on Advanced Techniques Tools for Software Evolution. URL: http://ceur-ws.org/Vol-2070/paper-01.pdf, Accessed 29.9.2022.

[12] Ståhl D. & Bosch J. (2014) Modeling continuous integration practice differences in industry software development. Journal of Systems and Software 87, pp. 48–59. DOI: http://dx.doi.org/10.1016/j.jss.2013.08.032.

[13] Fowler, M., 2013. Continuous Integration. URL: https://martinfowler.com/articles/continuousIntegration.html. Accessed 21.7.2022.

[14] Meyer M. (2014) Continuous integration and its tools. IEEE Software 31, pp. 14–16. DOI: http://dx.doi.org/10.1109/MS.2014.58.

[15] Tanenbaum A.S. (2017) Distributed systems. Maarten Van Steen, Milton Keynes, 3rd ed. URL: http://www.distributed-systems.net, Accessed 28.9.2022.

[16] Peffers K., Tuunanen T., Rothenberger M. & Chatterjee S. (2007) A design science research methodology for information systems research. Journal of Management Information Systems 24, pp. 45–77. DOI: http://dx.doi.org/10.2753/MIS0742-1222240302.

[17] Cooper T. & Wise M. (1997) Achieving incremental compilation through fine-grained builds. Software - Practice and Experience 27, p. 497 – 517. DOI: http://dx.doi.org/10.1002/(SICI)1097-024X(199705)27:5<497::AID-SPE88>3.0.CO;2-W.

[18] Zhang Y., Jiang Y., Xu C., Ma X. & Yu P. (2015) ABC: Accelerated building of C/C++ projects. In: Proceedings - Asia-Pacific Software Engineering Conference, APSEC, p. 182 – 189. DOI: http://dx.doi.org/10.1109/APSEC.2015.27.

[19] Licker N. & Rice A. (2019) Detecting Incorrect Build Rules. In: Proceedings - International Conference on Software Engineering, p. 1234 – 1244. DOI: https://dx.doi.org/10.17863/CAM.35755.

[20] Gallaba K., Junqueira Y., Ewart J. & Mcintosh S. (2020) Accelerating Continuous Integration by Caching Environments and Inferring Dependencies. IEEE Transactions on Software Engineering 48, pp. 2040 – 2052. DOI: http://dx.doi.org/10.1109/TSE.2020.3048335.

[21] Sjøberg D., Welland R., Atkinson M., Philbrow P. & C. W. (1997) Exploiting persistence in build management. Software - Practice and Experience 27, p. 447 – 480. DOI: http://dx.doi.org/10.1002/(sici)1097-024x(199704)27:4<447::aid-spe93>3.0.co;2-s.

[22] Maudoux G. & Mens K. (2018) Correct, efficient, and tailored: The future of build systems. IEEE Software 35, p. 32 – 37. DOI: http://dx.doi.org/10.1109/MS.2018.111095025.

[23] Ebert C., Gallardo G., Hernantes J. & Serrano N. (2016) DevOps. IEEE Software 33, pp. 94–100. DOI: http://dx.doi.org/10.1109/MS.2016.68.

[24] Make - GNU Project - Free Software Foundation. URL: https://www.gnu.org/software/make/. Accessed 13.4.2022.

[25] SCons: A software construction tool - SCons. URL: https://scons.org/. Accessed 22.5.2022.

[26] Ccahce - Compiler cache. URL: https://ccache.dev/. Accessed 22.5.2022.

[27] Ninja, a small build system with focus on speed. URL: https://ninja-build.org/. Accessed 14.4.2022.

[28] What is Gradle? URL: https://docs.gradle.org/current/userguide/what_is_gradle.html. Accessed 15.4.2022.

[29] Build Cache Node User Manual | Gradle Enterprise Docs. URL: https://docs.gradle.com/build-cache-node/. Accessed 24.5.2022.

[30] Fitzgerald B. & Stol K.J. (2017) Continuous software engineering: A roadmap and agenda. Journal of Systems and Software 123, pp. 176–189. DOI: http://dx.doi.org/10.1016/j.jss.2015.06.063.

[31] Humble J. & Molesky J. (2011) Why enterprises must adopt devops to enable continuous delivery. Cutter IT Journal 24, pp. 6–12.

[32] Adams B. & McIntosh S. (2016) Modern release engineering in a nutshell: Why researchers should care. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, pp. 78–90. DOI: http://dx.doi.org/10.1109/SANER.2016.108.

[33] Cepuc A., Botez R., Craciun O., Ivanciu I.A. & Dobrota V. (2020) Implementation of a continuous integration and deployment pipeline for containerized applications in Amazon Web Services using Jenkins, Ansible and Kubernetes. In: Proceedings - RoEduNet IEEE International Conference. DOI: http://dx.doi.org/10.1109/RoEduNet51892.2020.9324857.

[34] Nie J. (2013) A Study on the Application Cost of Server Virtualisation. In: 2013 Ninth International Conference on Computational Intelligence and Security, pp. 807–811. DOI: http://dx.doi.org/10.1109/CIS.2013.176.

[35] Jenkins. URL: https://www.jenkins.io/. Accessed 5.6.2022.

[36] Kubernetes. URL: https://kubernetes.io/. Accessed 5.6.2022.

[37] Soni M. (2015) End to End Automation on Cloud with Build Pipeline: The Case for DevOps in Insurance Industry, Continuous Integration, Continuous Testing, and Continuous Delivery. In: 2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM), pp. 85–89. DOI: http://dx.doi.org/10.1109/CCEM.2015.29.

[38] Lebeuf C., Voyloshnikova E., Herzig K. & Storey M.A. (2018) Understanding, Debugging, and Optimizing Distributed Software Builds: A Design Study. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 496–507. DOI: http://dx.doi.org/10.1109/ICSME.2018.00060.

[39] Shweta M., John N. & Shenoy S. (2014) Improving enterprise build process using a workflow driven approach in a distributed environment: A generic technique to provide rapid integration feedback. In: Proceedings - 2014 4th International Conference on Advances in Computing and Communications, ICACC 2014, p. 214 – 217. DOI: http://dx.doi.org/10.1109/ICACC.2014.58.

[40] Maudoux G. & Mens K. (2019) Lessons and Pitfalls in Building Firefox with Tup. In: Proceedings of the Seminar Series on Advanced Techniques & Tools for Software Evolution. CEUR Workshop Proceedings vol. 2510. URL: http://ceur-ws.org/Vol-2510/sattose2019_paper_2.pdf, Accessed 29.9.2022.

[41] Sharma P., Chaufournier L., Shenoy P. & Tay Y.C. (2016) Containers and Virtual Machines at Scale: A Comparative Study. In: Proceedings of the 17th International Middleware Conference, Middleware '16. DOI: http://dx.doi.org/10.1145/2988336.2988337.

[42] Deka G. (2014) Cost-benefit analysis of datacenter consolidation using virtualization. IT Professional 16, pp. 54–62. DOI: http://dx.doi.org/10.1109/MITP.2014.89.

[43] Singh V. & Peddoju S.K. (2017) Container-based microservice architecture for cloud applications. In: 2017 International Conference on Computing, Communication and Automation (ICCCA), pp. 847–852. DOI: http://dx.doi.org/10.1109/CCAA.2017.8229914.

[44] Tosatto A., Ruiu P. & Attanasio A. (2015) Container-Based Orchestration in Cloud: State of the Art and Challenges. In: 2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems, pp. 70–75. DOI: http://dx.doi.org/10.1109/CISIS.2015.35.

[45] Malviya A. & Dwivedi R.K. (2022) A Comparative Analysis of Container Orchestration Tools in Cloud Computing. In: 2022 9th International Conference on Computing for Sustainable Global Development (INDIACom), IEEE, pp. 698–703. DOI: http://dx.doi.org/10.23919/INDIACom54597.2022.9763171.

[46] Jararweh Y., Al-Ayyoub M., Darabseh A., Benkhelifa E., Vouk M. & Rindos A. (2016) Software defined cloud: Survey, system and evaluation. Future Generation Computer Systems 58, pp. 56–74. DOI: http://dx.doi.org/10.1016/j.future.2015.10.015.

[47] Raghunath A., Zou Y. & Chagam A. (2020) On Evolving Software Defined Storage Architecture. In: 2020 IEEE International Conference on Cloud

Computing Technology and Science (CloudCom), pp. 57–64. DOI: http://dx.doi.org/10.1109/CloudCom49646.2020.00008.

[48] Kubernetes Documentation | Kubernetes. URL: https://kubernetes.io/docs/home/. Accessed 9.9.2022.

[49] Microk8s - Zero-ops Kubernetes for developers, edge and IoT. URL: https://microk8s.io/. Accessed 5.9.2022.

[50] Client Libraries | Kubernetes. URL: https://kubernetes.io/docs/reference/using-api/client-libraries/. Accessed 12.6.2022.

[51] Storage Classes | Kubernetes. URL: https://kubernetes.io/docs/concepts/storage/storage-classes/. Accessed 15.6.2022.

[52] Introduction - Kubernetes CSI Developer Documentation. URL: https://kubernetes-csi.github.io/docs/. Accessed 15.6.2022.

[53] Venable J., Pries-Heje J. & Baskerville R. (2012) A comprehensive framework for evaluation in design science research. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 7286 LNCS, pp. 423–438. DOI: http://dx.doi.org/10.1007/978-3-642-29863-9_31.

[54] fio - Flexible I/O tester rev. 3.30. URL: https://fio.readthedocs.io/en/latest/fio_doc.html. Accessed 27.9.2022.

[55] Introduction | Vagrant by HashiCorp. URL: https://www.vagrantup.com/intro. Accessed 10.7.2022.