Workflow Trace Extraction Framework for Distributed Computing Analysis

CSE2000 Final Report

Unveiling Workflow Traces and Resource Utilisation for Enhanced Analytics

Group 12A
Atour Mousavi Gourabi
Lohithsai Yadala Chanchu
Henry Page
Pil Kyu Cho
Tianchen Qu

Supervisor: dr.ir. J. A. Pouwelse

Teaching Assistant: Timur Oberhuber

Client: Laurens Versluis on behalf of ASML

Delft, 2023

Preface

We are a group of five computer science students at TU Delft who all have extensive experience in software development and data processing. For the CSE2000 Software Project course, we have developed a framework that can gather execution metrics from big data applications. This project showcases a concrete plugin developed for Apache Spark along with a core module that is extensible to other big data applications.

The focus of this technical report is to provide a comprehensive understanding of the research, underlying technology, development, and evaluation of the developed product. To fully comprehend the entire report, it is essential for individuals to possess a foundational understanding and practical knowledge of how Apache Spark works. We hope that this report inspires readers to dive into the world of big data tooling and massive computing systems.

Readers that are particularly interested in product design and analysis can go to Chapter 5 and Chapter 6 respectively. Those that are more interested in the background research and reasoning that went into this project can go to Chapter 2 and Chapter 3.

This project would not have been possible without Laurens Versluis, the representative for ASML who offered the project to our group. We cannot thank him enough for his constant guidance and input regarding industry-standard technology throughout the project. In addition, we would like to thank our teaching assistant Timur Oberhuber for guiding us through the development and project process. Furthermore, our thanks go to our supervisor Johan Pouwelse and technical writing teacher Matthijs Looij for their supervision of the research and writing process. Lastly, special thanks to the developers of Apache Spark that maintain the framework and its status as one of the most important free and open-source software packages.

Delft, May 7, 2025

Atour Mousavi Gourabi, Lohithsai Yadala Chanchu, Henry Page, Pil Kyu Cho, and Tianchen Qu

Summary

As the importance of recording distributed computing execution metrics increases, the need for a unified format to store this information arises. The Workflow Trace Archive (hereinafter, WTA) addresses this need by creating a unified, open-source trace format that can be widely used and understood by many developers and researchers. The WTA is currently the most comprehensive archive of workflow data in the world. The presence of a tool capable of producing traces in the WTA format is extremely critical, as it enables the effortless generation of these traces by executing a plugin alongside the program operating on a distributed system. The objective of this report is to present a tool that generates WTA traces from Spark execution information, which will make it easier to share knowledge about the execution of distributed computing systems.

The plugin extracts the relevant information from a variety of sources. A combination of metrics received from Apache Spark and various other data sources including Perf and the Linux filesystem is used to gather information which is useful to distributed computing researchers.

The plugin's architecture was designed to be extensible and modular such that additions could easily be made by future open-source developers who want to extend the plugin. The plugin is split into a core component and an adapter component. The adapter component is split into different modules, each of which contains the logic to handle their respective big data application. The core component houses the code that is independent of the application being used, such as the Parquet writer that is used to produce the WTA output file. This design ensures that the plugin is extensible such that other big data applications (e.g. Apache Flink for stream processing) could be supported in the future. Furthermore, the plugin is kept extensible in such a way that new data sources can easily be added or swapped out if extra or different information is needed. It was also important that a maintainable design was chosen so that new developers could easily become accustomed to the existing codebase and start contributing.

Since the tool will be run on systems with a heavy load, it is imperative that the plugin is as efficient and scalable as possible. To validate the effectiveness and performance of the plugin, the benchmark run-time of Spark jobs with and without the plugin were compared. The final benchmark resulted in an overhead of below two per cent, thereby demonstrating that the plugin does not incur a significant performance overhead. The use of our plugin during job execution is more advantageous compared to alternative solutions like a log parser that operates after the job is completed. This preference stems from the fact that our plugin, when added dynamically to the job, is capable of extracting a higher volume of information.

Contents

Chapter 1

Introduction

As distributed computing scales globally, so does the importance of monitoring such systems. Workflow traces contain information about the execution of tasks performed in a distributed system. These traces are commonly used for monitoring and can identify inefficiencies and system failures. The Workflow Trace Archive (hereinafter, WTA) format is a unified trace format that captures specific information related to the execution of large-scale data processing tasks. This format allows researchers to create models and simulations for workflow execution. By analysing traces, developers and researchers can enhance the efficiency of big data systems, gaining a deeper understanding of the underlying processes behind distributed computing jobs. The founders of the WTA envision it to be an open repository for researchers to use this data for further analysis [1]. They argue that the availability of trace information from distributed computing systems in the WTA facilitates experimentation, boosts productivity, and encourages innovation. Numerous scientists work on a vast number of cloud systems for various industrial use cases. The exchange of knowledge is limited due to the lack of standardisation and tooling.

The objective of this report is to present a tool that generates WTA traces from Spark executions, which will make it easier to share knowledge about the execution of distributed computing systems. The design choices that have been made will be examined thoroughly. Having meticulously evaluated the advantages and disadvantages, the rationale for placing importance on specific aspects of the decisions made for this particular application, such as system design, performance, and usability, is explained. Since our tool will be run on systems with a heavy load, it is imperative that the plugin is kept as efficient and scalable as possible. Furthermore, it is crucial that this tool is kept extensible for other distributed computing frameworks since the need for a tool to generate these traces for other distributed computing frameworks might arise later on. Therefore, multiple approaches will be investigated to find the approach that results in the most performant and extensible product.

The report is set out as follows. In Chapter 2 the problem will be analysed in more detail; further context will be provided and the feasibility of potential solutions to the problem will be evaluated. After this, the risks and ethical implications that may arise from this project will be explored in Chapter 3. In Chapter 4, a detailed explanation is provided regarding the development methodology, accompanying a discussion of the project process and an evaluation of its alignment with our planned methodology. Chapter 5 goes over the software architecture and design. Potential extensions to the project will be considered after which the final product and its practical analysis are discussed in Chapter 6. Finally, in Chapter 7, the report will reach its conclusion with a discussion on the future potential of the product.

Chapter 2

Problem Analysis

Prior to seeking potential solutions, it is crucial to obtain a comprehensive understanding of the problem at hand. This chapter delves into the problem and investigates diverse approaches for its resolution. Section 2.1 gives a brief introduction of the technical background related to the problem. Section 2.2 discusses how different stakeholders seek to benefit from our product. Section 2.3 briefly discusses additional things that were investigated within the context of the project. This is followed by Section 2.4 where the feasibility of the solution is considered, concluding with Section 2.5, a risk analysis of the project.

2.1 Problem Statement

The problem addressed in this section is the difficulty of comparing and analysing Spark execution information across different platforms due to varying tooling and formats. To overcome this challenge, the objective is to develop an application that seamlessly converts Spark execution information into the Workflow Trace Archive (WTA) format. The WTA format provides a platform-independent way to capture and share workflow execution data, enabling researchers and industry practitioners to analyse and optimise distributed workflows effectively. The application will be pursued in one of two ways. It will either feature real-time processing using a plugin or it will be a log parser that parses logs from the Spark history server. The converted data will be stored in Parquet files and can be enhanced with resource utilisation data from Prometheus. By addressing this problem, the project aims to provide a valuable tool for academia and industry, facilitating the publication and analysis of workflow traces and fostering advancements in distributed computing.

2.1.1 Workflow Trace Archive

The increasing complexity of these workflows introduces additional challenges to understanding how a workload behaves in a distributed setting. A distributed workflow trace is essentially just a report of tasks and operations that are performed by nodes in a distributed system. Commonly used for monitoring and instrumentation, it is particularly useful for identifying inefficiencies and failures in a system. The Workflow Trace Archive is a collection of workflow traces that are in the WTA format. Researchers can use these traces to "create models [...] and simulations to replay the execution of a workflow in a controlled environment" [1, p. 2170]. The WTA format, as illustrated in Figure 2.1, is a unified trace format that supports the analysis of more advanced metrics. This allows researchers to use data analysis tools specifically made for this format to enable optimisation in a distributed computing environment. By analysing the trace data, developers and researchers can identify where errors and inefficiencies occurred and debug the system. The significance of this unified trace format stems from the availability of crucial trace information from distributed computing systems, which can be easily accessed by researchers to ease experimentation, thereby augmenting productivity and fostering innovation.

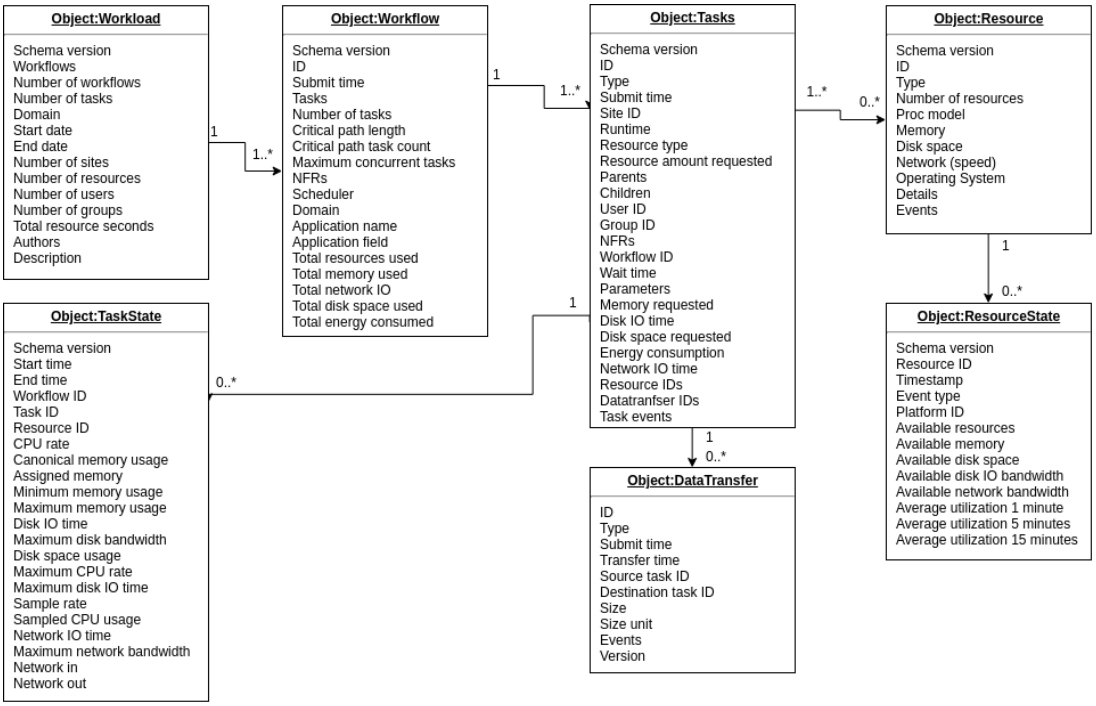


Figure 2.1: Schema of the traces as objects [2]

2.1.2 Apache Spark

Apache Spark [1] is a large-scale data processing engine with APIs for many different languages such as Scala and Python. The project is maintained by the ASF.[2] It makes extensive use of data parallelism when running jobs and is able to do so across clusters of machines, providing implicit fault tolerance. Furthermore, its ability to perform in-memory computing significantly increases the processing speed of its jobs. This feature alongside its support for SQL queries, machine learning, and graph algorithms makes Spark one of the most popular tools in the domain since 2014.

2.1.3 Problem Investigation

The main problem ASML, as represented by Laurens Versluis (hereinafter referred to as the 'client') is facing is that even though Spark execution information is widely used in the industry, it is not easy to compare and analyse this information across different platforms as the tooling may be different. The client has identified the WTA format as the solution to this problem, as it provides a platform-independent way to capture and share workflow execution data. However, the client does not currently have a tool that can convert Spark execution information into the WTA format seamlessly. The lack of a tool to convert distributed computing traces to the WTA format hinders the publication of such traces by academia and industry.

The primary objective of this project is to create an application that converts Spark execution information into the WTA format. The client has stated that this conversion can be achieved in two ways: real-time processing using a plugin or parsing the logs generated by the Spark history server. Furthermore, they have specified that the Spark execution information should be parsed into Parquet files. The client has emphasised the need to develop the product in a modular manner, allowing for future support of additional data sources such as Apache Flink [3]. Furthermore, the client has requested the inclusion of resource

[1] https://spark.apache.org/
[2] Apache Software Foundation
[3] https://flink.apache.org/

utilisation data from Prometheus and other instrumentation data obtainable from the Spark API to enhance the Spark execution information [4].

2.2 Stakeholders

In this section, we will introduce the key stakeholders involved in the project and outline their respective roles and responsibilities. These stakeholders play a vital role in the successful development and implementation of the solution. The stakeholders include the developers, the client, AtLarge Research, and the distributed computing community.

2.2.1 Developers

The developers are responsible for delivering the final solution and ensuring compliance with all the designated functional and non-functional requirements as specified by the client. Moreover, the developers are responsible for ensuring that the quality of the final solution adheres to a certain standard through verification and testing. As developers, our involvement in the project offers us an opportunity to acquire valuable experience in working with Spark and develop our software engineering practices within the context of a distributed computing project.

2.2.2 Client

The client is responsible for setting the initial requirements and expectations for the solution and providing feedback throughout the software development process, which is a core principle of the Agile software development methodology. This will be achieved by meeting with the client at least once a week. Successful completion of this project means that the client will receive a functional product that they can use to convert Spark execution information into traces that adhere to the correct format.

2.2.3 AtLarge Research

This research group bears the responsibility of hosting the WTA and serves as a stakeholder due to the potential benefits they can derive from our tool. Similar to the client, they stand to benefit from the application's ability to facilitate the conversion of Spark execution information into the WTA format. This is expected to lead to an increase in the number of researchers using and uploading traces to the WTA.

2.2.4 Distributed Computing Community

The greater community can benefit from this tool, as it will be made available as free and open-source software (FOSS). This enables anyone involved in distributed computing to utilise it and derive value from analysing traces originating from Spark workflows. Furthermore, FOSS can serve as a source of inspiration for future projects, making our product a potential source of inspiration as well.

2.3 Context Investigation

In this section, we delve into the context investigation related to our project. This consists of exploring existing solutions, and the insights gained from users and expert consultations.

[4] https://prometheus.io/

2.3.1 Existing Products

Currently, there are no available products capable of converting Spark execution history to the WTA format. Nevertheless, there exist tools that can facilitate the conversion of Spark execution information to alternative formats by leveraging the Spark API. Specifically, the investigation focused on the SparkMeasure plugin, which uses Spark event listeners for capturing metrics at the task and stage levels [3]. Using SparkMeasure, one can write these metrics to disk as well as synchronise them with other applications such as Apache Kafka [5]. If the objective is to process Spark execution information in real time, SparkMeasure could potentially be employed as a learning tool as it makes use of the Spark Listener interface. This would be beneficial in the development process as one can always refer to this open-source solution. The problem with this solution however is that it had no external integration with external performance monitoring tools, which may limit its usefulness in larger Spark deployments. Furthermore, the repository states that SparkMeasure has several limitations, such as not exposing the time spent on I/O operations, which the WTA format requires.

If the goal is to directly parse Spark event logs, it would be necessary to consult other less recognised examples. The SparkMeasure repository also has a working proof-of-concept of parsing event logs using Spark SQL, which provides us with an example of parsing event logs after the respective job has finished. Consulting this resource would be beneficial if the decision is made to parse the event log instead of creating a plugin. The problem with using this directly is that it was a very simple proof-of-concept, and it was not created in a modular manner. The adoption of this approach would necessitate a significant amount of manual refactoring.

In the requirements of Section 4.2, the client also specified that they wanted to monitor the energy consumption of a given task, as this is one of the fields that is specified by the WTA format. The Spark API does not provide support for this metric and thus we investigated JoularJX, which collects CPU usage information and tries to calculate the power consumption for every method [4]. It is important to note that this library is experimental and has some additional performance overhead. Another method we investigated is the PAPI library [5], [6], which enables the gathering of performance metrics but requires writing additional C code and performance overhead from using Java Native Interface [7] to integrating the C code into Java.

Considering the significant difference between the WTA format and other formats that represent execution data, it would be challenging to incorporate any existing solutions. Furthermore, we concluded that any existing Spark plugins were not comprehensive enough for our needs due to the lack of metrics needed for the WTA format. This justifies the need for a bottom-up approach.

2.3.2 Users and Expert Consultation

It is important to listen to and actively consult field experts. The client has advised us to reach out to the maintainers of the Distributed ASCI Supercomputer (DAS) to test our application in a truly distributed setting [8]. In addition, the client has given us some contacts who are experts in the related field of massive computing systems whom we can consult with. As the client will be one of the direct users of this application, the user requirements are directly modelled in the set of requirements that the client has proposed. To gain further domain-specific knowledge on Spark and determine acceptable levels of performance impact for plugin development in both academic and industrial contexts, we consulted with our supervisor and client to seek out experts they may know of.

[5] A distributed event streaming platform for handling real-time data.

2.4 Feasibility Study

In this section, we conduct a feasibility study to assess the practicality of our project. We evaluate the time constraints, and available resources, and compare the plugin and parser approaches. These considerations guide our planning and decision-making process, ensuring the successful development of our solution.

2.4.1 Time Constraint

Considering that the final product is expected to run on servers running critical applications in the real world, a lot of weight must be placed on optimisation and verification. The optimisation will require extensive planning for the best-suited design and implementation details. During the planning phase, we will also set up the project structure and automate any verification processes, such as the use of GitLab CI/CD [9] and Maven [6], to save as much time as possible during development. We expect the planning phase to finish no later than the end of week two.

During the project, we will input a minimum of 40 hours a week, although the actual number will most likely be higher than this.[7] If we finish our weekly planned tasks ahead of schedule, we will immediately follow up with the tasks for next week. This will give us more time to deal with verification later on which will allow us to identify problems that arise that could delay the delivery of the product. We expect the development phase to be finished no later than the end of week nine.

Nevertheless, we must consider the possibility of not being able to implement all of the functionality within the given time constraints for this project. In such a case, we may want to narrow down the scope of the project to focus on the most important functionalities, ensuring that the end product meets the client's expectations while remaining feasible for development. To adequately prioritise all these problems, we used the MoSCoW[10] prioritisation method to divide these problems into must-haves, should-haves, could-haves, and won't-haves. More on this can be found in Section 4.1.

2.4.2 Available Resources

We have compiled the following libraries and tools to carry out the project. These have been carefully thought of and consulted with the client in order to create an efficient working environment and deliver a final product that meets the client's needs.

As the application is to support Spark 3.2, we can leverage the post-Spark 3.0 plugin API for both the driver and executor sides of the system. For the development process, Maven will be used to manage the project, while we make use of GitLab CI/CD to automate the code integration and validation. This will be combined with a number of linters, such as Spotless[8] and Checkstyle,[9] to adhere to style guidelines throughout the project. These linters complement each other as Spotless focuses more on code formatting and ensures a consistent code style while Checkstyle helps identify other code quality concerns like code smells and maintainability issues through the use of extensive rule sets to ensure good practices are being followed.

In terms of testing, we will conduct functional tests and use JaCoCo to continuously generate test reports that monitor comprehensive testing metrics during development. To ensure our test suite suffices, we look at the testing metrics compiled by JaCoCo. In addition, PITest will be used for mutation testing in order to enforce us to write an adequate amount of boundary tests, especially for the core of the application. The coverage thresholds used for both types of tests are specified in Section 4.3.

As mentioned in Section 2.5, the application will be tested on DAS-5 to verify that the application works

[6] https://maven.apache.org/

[7] The expected amount of work for a 15ECTS course over a ten-week period such as in CSE2000 is 40 hours per week.

[8] Code formatter for various programming languages

[9] Quality checker of coding standards for Java programming language.

in a truly distributed setting. This will allow us to overcome any limitations imposed when evaluating the product in a local environment. Furthermore, to reinforce the significance of adhering to industry standards, we will incorporate the widely recognized TPC-DS benchmark into our performance evaluation process. The purpose of this comprehensive benchmark is to thoroughly test the system's performance in various scenarios, making it a versatile benchmark. This guarantees that our product can efficiently handle higher workloads whilst minimizing unnecessary overhead.

2.4.3 Plugin and Parser Approaches Compared

Our biggest decision to make for the project during the planning stages was the choice between making a plugin to function as the adapter layer over the core module or a log parser. To assess this, the benefits and drawbacks of both approaches needed to be considered. With a plugin, we get direct access to the Spark instance, its driver, executors, and machines through Spark's plugin API. This means that we will be able to collect additional metrics directly in real-time. We will thus not rely on separate logs or data sources for metrics such as energy consumption, which a parser would have to do.

The major drawback of this approach, however, is the potential overhead that it could introduce to the Spark job that is running, both memory- and time-wise. Another important consideration when creating a plugin is that it should not interfere with the Spark job, as the plugin is a non-critical part of the run. This means that when it fails, it should not cause the Spark job to fail as a whole but instead, it should handle these cases itself and let the Spark job continue and finish gracefully later on.

The main benefit of using a parser is that the client would be able to parse Spark history logs. This means that in contrast to the plugin approach, traces can be created from old runs prior to the creation of the tool. The main drawback is the increase in difficulty when enriching traces with data that does not originate from Spark itself such as energy consumption metrics. This requires the application to be fed other server logs, after which we would have to perform sequence alignment on the separate data sources. Another consideration is that users might not have access to these more detailed logs anymore, which could lead to traces that are almost empty.

For the run-time overhead problem, we ran timed benchmarks using TPC-DS queries[10] and data. We ran these benchmarks on two local Spark configurations: one with a regular Spark application with no plugin and the other with the Spark application monitored by the SparkMeasure plugin. We ran the benchmarks using the Java Microbenchmark Harness framework[11], with both fifty warm-up and iteration runs. During benchmarking, logging to the terminal was set as minimal to minimise any background subprocesses. The Spark job used for benchmarking was configured to run one executor instance using four cores. The benchmarks were run on a machine using 16GB of RAM and an AMD Ryzen 5 3600 processor, boasting six cores and a clock speed of 3.60GHz. The results of the benchmarking were an estimated run-time of 105915 ms for the plain Spark run and no more than 107472 ms for the run with the SparkMeasure plugin enabled. This translates to an estimated difference (and thus overhead for SparkMeasure) of less than two per cent between these runs, which is well below the overhead threshold the client deemed acceptable from a previous meeting. This, along with the client's desire to incorporate energy consumption data, has led us to select the plugin approach over the log parsing approach

2.5 Risk Analysis

In this section, we conduct a risk analysis to identify potential challenges and mitigation techniques. We assess the risks associated with hardware, technical skills, licensing, an overhead of dependencies, and a lack of support and options. By understanding these risks, we can proactively address them and ensure the project runs smoothly.

[10] Standardized SQL queries with a wide variation in complexity and range of data scanned [11].
[11] Tool for creating benchmarks of Java programs

2.5.1 Hardware Analysis

This project goal is to build a plugin for Spark, which is expected to run on large-scale distributed systems. Therefore, it is necessary to test and evaluate our plugin on a distributed system. For this, we intend to utilise the DAS-5 distributed supercomputer [8]. In case this presents a bigger challenge than anticipated, we will still carry out the testing and evaluation, but in a local environment. While this would not be as informative as when using DAS, it would not cause great risk for the project.

2.5.2 Technical Skill Analysis

It is safe to say that we are already well-acquainted with most of the skills and knowledge base needed for this project. This includes Java, Spark, JaCoCo, PITest, and Spotless. A quick recap of some skills might be needed, such as for Spark, but this should be quick and bear no threat to the project. On the other hand, there are some skills which we are not familiar with. These include tools such as Docker and Maven, operating the DAS-5, and benchmarking. However, these shortcomings will not block the development of the product as they mostly require attention during the second half of the project life cycle.

2.5.3 Licensing

As our project is intended to be hosted as FOSS after the completion of the internship, we are exposed to some legal considerations regarding the licensing of the software. To be considered FOSS, the code is to be licensed under an open-source license, preferably one that is quite permissive. For this, we will need to ensure the compatibility between our licenses and the licenses of our dependencies. One such example which has sparked some controversy in the past would be the GPL licenses [12], [13], which mandate license compatibility in all projects that make use of GPL dependencies.[12] One of the most popular open-source licenses which will be able to avoid the most common licensing issues as with GPL, would be the Apache-2.0 license [17], [18]. Another benefit of it is that it allows the absence of copyright headers in all files, covers patent problems, and is already in widespread use. It does have one drawback, namely, that we cannot link to code licensed under GPLv2. The only way to be able to do this is when we use a copyleft license such as GPLv2 ourselves, which is something we want to avoid at all costs.

2.5.4 Dependency Overhead

As the project progresses, we will have to keep an eye on the size of the dependencies we use and critically question whether we need a big portion of the dependency or only a small part of the functionality for our application. If we need only a small part of the functionality offered by the dependency, implementing the necessary behaviour in-house must be considered. If this is not taken into account, we will end up with a massive file which contains the dependencies that will have to be distributed to each of the nodes in the cluster, increasing the overhead for starting up and running the plugin which is problematic since it delays the jobs which are executed on each node within the cluster. Therefore, it is best to keep the size of the dependencies as low as possible by selecting lightweight dependencies and, if that is not feasible, implementing the desired behaviour ourselves.

2.5.5 Lack of Support and Options

Since our project is located in a niche, we do not have an abundance of tools to help us build our project and the tools that we do have are often lacking. A preliminary check of the Spark plugin documentation tells us it is severely lacking as many API methods are undocumented. This means that we will have to get comfortable with figuring out what methods do from reading the source code, which is more time-consuming than reading the documentation. Furthermore, another tool we will need is a Parquet

[12] Viz. viral licenses and the Linux "cancer" [14]–[16]

writer, of which there are not many for Java. Taking a quick look at the code shows us that using it will introduce a lot of unnecessary dependencies and thus bloat our project. This means that we might have to fork some dependencies and modify library code to make it more suitable for our application.

Chapter 3

Values and Ethical Implications

Something we should never overlook would be the ethical implications of the project: how will it affect people, benefit them as well as pose potential risks to them? We will analyse these risks with regard to two areas: personal data in Section 3.1 and data management and transparency in Section 3.2. Then we will discuss the situation when a vulnerability is discovered in Section 3.3, and end on one particular security case: side-channel analysis as discussed in Section 3.4.

3.1 Personal Data

The project's ethical considerations mainly revolve around the information contained within the Spark history execution logs, which may pose certain data privacy-related concerns. Despite initial concerns about personal data privacy, a thorough analysis of our project concluded that it is not directly related to personal data safety. The risk of violating personal data privacy is minimal due to the fact that our plugin will only collect the working metrics such as energy consumption and memory usage. In other words, we will only collect how the Spark job is running instead of what it is doing. This will not include any information about the actual content of the process. Furthermore, after consultation with ethics experts dr. Ishmaev and ir. Chotkan, it was determined that the risk associated with this information is minimal. As traces are not personally identifiable data, it cannot be used to target specific individuals.

Regardless, there are still some ways to indirectly deduce some personal data from our plugin. Discussion on such issues and risks will be presented in the following sections.

3.2 Data Management and Transparency

Though the plugin has little risk associated with it regarding personal data privacy, it could cause problems for corporate users. The cluster traces we extract might contain sensitive information the company would not want to disclose.[1] Since the gathered metrics will provide an image of how servers are organised and used, this could give corporate rivals and potential bad actors important information about the system that could be exploited.

As mentioned before, the WTA project includes a trace anonymisation script in its tooling repository [19] that currently uses regular expressions to match things such as IP addresses and email addresses. It replaces these with a hashed representation of these fields. A hash is a technique to transform the original data into a seemingly random data chunk that is very hard to crack, and thus it is able to pseudo-anonymise data. However, from an ethical perspective, this may not be the most effective method of anonymising the data because it could still enable bad actors to try and validate the existence of certain information in this hashed format. Following consultation with ethics experts, we identified the need to balance the

[1] i.e. the type of machines they use to run their jobs.

requirements for detailed statistics in research with the potential risks associated with targeting individuals and corporations and choose an appropriate resolution of data stored in the generated traces. In other words, we must balance research interests with those of corporations. Over-anonymisation of the data makes it less usable to researchers, while too little anonymisation may potentially harm users that opt to publish their traces. Failure to consider both interests could result in decreased participation in the trace community.

3.3 Potential Vulnerability Discovery

In case we discover potential vulnerabilities in the codebase of another organisation, we will follow the guidelines of responsible disclosure [20]. This means we will first let the developers[2] of that software package know about the vulnerability and give them ample time to fix it. In the case they ignore the vulnerability and fail to roll out a patch within a reasonable timeframe, we will publicise the vulnerability ourselves in order to make users of the software aware of the flaws and force the hand of the maintainers.

3.4 Inference of Private Data Using Side-Channel Analysis

Our requirements mandate the measurement of power-consumption data during a Spark job, which poses an ethical concern. High-resolution data for this metric potentially enables the inference of private data using side-channel analysis [22]. When Spark is executing jobs it might internally generate cryptographic keys. High-resolution energy consumption data could be used to deduce the key that is generated [23]. If the key is later used for encrypting the internal communications between different Spark processes, privileged information might be obtained by the attacker. Hence, our application's design must consider this ethical concern to effectively address and prevent any potential risks (by potentially reducing the resolution of the energy consumption data), particularly in security-critical domains.

[2]Please do note that in case vulnerabilities in Apache projects like Spark or Parquet are discovered, we would contact the dedicated ASF Security Team [21] instead of the project's maintainers.

Chapter 4

Development Methodology

The preceding sections comprehensively addressed the research and analysis conducted as part of the project's validation study. The focus will now shift towards the practical aspects of the project. Section 4.1 explains our approach to drafting the initial set of requirements. Sections 4.2 and 4.3 detail the functional and non-functional requirements we have elicited to achieve a successful product. Readers who are interested in learning about our team's communication methods and the effective utilisation of the SCRUM methodology are encouraged to refer to Appendix D.

4.1 Requirements Engineering

When planning a project, it is important to recognise and define the needs and demands of your stakeholders. In order to do this, we have drafted up a set of requirements which the product should fulfil. These requirements are split into two: functional and non-functional requirements. The former concerns the functionality of the final product, while the latter is more concerned with performance, quality assurance, and internal standards. In the following section, we will elaborate more on the requirements we have drafted in collaboration with our client.

4.2 Functional Requirements

Using the MoSCoW prioritisation model, we have divided the functional requirements of the project into four classes: must-haves, should-haves, could-haves, and won't-haves [10]. The must-haves constitute our minimum viable product and are thus the core features the application will absolutely need in its minimal form; the should-haves constitute important features which are not necessary for the minimum viable product; the could-haves are some additional features the application could benefit from but are not essential; the won't-haves are features that we believe are nice to have but not feasible for this iteration. The won't-have category is mainly used to prevent scope creep from occurring during the project.

4.2.1 Must-haves

1. The application shall at least support Spark 3.2.
2. The application shall require the user to specify the authors and domain[1] of a trace.
3. The application shall ask the user for a description of the trace and notify them when this is left blank.
4. The application shall collect the resource and hardware-related metrics[2] in the trace.

[1] The domains in the WTA are biomedical, engineering, industrial, and scientific.
[2] E.g. the maximum CPU usage, total resources used, or amount of resources requested

5. The application shall collect the memory-related metrics[3] in the trace.
6. The application shall collect the disk-related metrics[4] in the trace.
7. The application's output shall adhere to the WTA format.
8. The application shall collect the metadata of the Workload object.
9. The application shall output the generated trace to the disk.

4.2.2 Should-haves

1. The application shall output the generated trace in Parquet format using the Snappy compression algorithm.
2. The generated trace shall include a Parquet file naming format for versioning of the application.
3. The application shall generate INFO-level logs.
4. The user shall have the option to use stage-level metrics rather than Spark's task-level metrics.
5. The application shall collect energy metrics[5] in the trace where possible.

4.2.3 Could-haves

1. The application shall use Prometheus metrics to enrich the traces.
2. Users shall have the option to automatically validate the generated traces.
3. Users shall have the option to automatically anonymise their traces.
4. The application shall support S3 as a storage location for the generated traces.
5. Users shall be given the option to automatically upload their traces to Zenodo after anonymisation and validation.
6. The application shall collect data transfer-related metrics[6] in the trace.
7. The application shall support the integration of GPU metrics from NVIDIA Rapids.

4.2.4 Won't-haves

1. The application shall extend support to Spark 2.4.
2. The application shall include an adapter for Flink.
3. The application shall include an adapter for Hadoop MapReduce.

4.3 Non-Functional Requirements

1. The project shall be written in Java 11 LTS.
2. The project shall use Maven as the build tool.
3. The project shall be created in a modular fashion such that new adapters can augment the project.
4. The core project shall be able to handle and aggregate multiple data sources.[7]

[3] E.g. the total memory used requested
[4] E.g. the total disk space used or the disk I/O time
[5] E.g. the total amount of energy consumed
[6] E.g. the submit or transfer time
[7] E.g. some combination of Prometheus and Spark

5. The codebase shall have both a branch and code test coverage of at least eighty per cent, under the condition that every test case has at least one meaningful assertion.
6. The project shall use JaCoCo to generate test coverage reports.
7. The project shall incorporate mutation testing,[8] with test coverage of at least sixty per cent.
8. The project will be licensed under a permissive open-source license such as Apache License, Version 2.0.
9. The pipeline will integrate Spotless and Checkstyle to adhere to style guidelines.
10. The application shall never interfere with a Spark job through its exceptions.
11. The final product shall not exceed a ten per cent slowdown when running the TPC-DS benchmarks over a plain Spark run, while minimising the slowdown as much as possible.

4.4 Code Development Tools

We will use several technologies such as Apache Spark, Java, and Lombok during the development of the plugin. The plugin is extensible and is designed such that new data processing frameworks can be easily added in the future. We decided to add support for Apache Spark first (as opposed to other frameworks) since it is the most widely used large-scale data processing framework. It also has an existing plugin API which will be very useful when designing the plugin. Not only does this allow us to abstract away the logic of sending the information from the executor to the plugin, but it also gives us greater control of when metrics are processed through the use of callbacks in the API. We can motivate the choice of Java as the client also wanted the product to be written in Java or Python. Java is the preferred language for Apache Spark due to its static typing, our familiarity with it, and its vast ecosystem and extensive libraries that seamlessly integrate with Spark applications. Lombok addresses the verboseness that comes with Java. Lombok will help increase readability and development speed since it uses annotations that auto generates boiler plate code.

We will also be using several tools to make contributing code in a group environment easier. We will be using Git for version control, and GitLab for hosting code remotely. We will also be using GitLab features such as boards and labels to manage issues. The issue board will be used to keep track of the status of each issue and will have multiple tags for each issue including their MoSCoW priority, the sprint that they belong and the type of issue (e.g. refactoring, bug fixing, development). We will also use milestones to manage our sprints group issues. We will also be actively monitoring burn-down charts to ensure we are on track. The GitLab requirements functionality will also be utilised to ensure that our issues covered the requirements as proposed by the client. Pipelines and merge trains will be extensively used to ensure that merge requests which are merged simultaneously do not break functionality. For the branch schedule, we will keep the main and development branches protected from code that is not reviewed and merge all the feature branches to the development branch before going into the main branch. This ensures all contents on the main branch will be fully functional, while the development branch acts as a sort of buffer to make sure that everything works together. Once an issue is assigned, the assignee shall open one new branch for it with a merge request. The project will be hosted in two separate repositories: the development repository, which contains all the project code, and the internal repository, which stores project-related documents such as the code of conduct, meeting agendas, and minutes. In case we need to fork other repositories, we will create separate repositories dedicated to each fork.

[8] I.e. PITest

4.5 Testing Infrastructure

To ensure we deliver a robust, working product, a strong testing infrastructure needs to be in place that thoroughly examines all aspects of the product's functionality and its life cycle. This infrastructure must be already in place from the early stages of the project. Otherwise, we run the risk of either building up a huge technical debt which will either take a considerable amount of time to fix or bugs that might not get caught until after the product has been delivered. Our testing infrastructure thoroughly examines any new feature added to the product throughout the development life cycle both in the feature itself and its interaction with related components. This ensures that any potential bugs or flaws are caught in the early stages, allowing the performance standards to hold after integrating features and resolving bugs in the final product.

4.5.1 Testing Methods

We currently employ the following testing methods in our project:

- Unit testing – tests individual units of classes and their methods
- Mutation testing – tests whether the test cases can detect potential errors by intentionally introducing bugs in parts of the source code
- Integration testing – tests the interactions between the combined software modules as a group
- System testing – tests the system as a whole from start to end

As per non-functional requirements 5 and 7 defined in Section 4.3, we achieve a branch and code test coverage of at least eighty per cent. For a more detailed view of the test coverages of the different types of tests see Appendix B. However, it is easy to 'cheat' by creating test cases that cover the source code but do not make meaningful assertions. Therefore, we also ensure that at least one meaningful assertion is in place for each test case. While there exist tools that can detect the presence of specific types of test assertions, there are no tools that can detect the absence of assertions in general. Therefore, we would have to rely on the developers and code reviewers to manually inspect the validity of the test cases.

4.5.2 Pipeline

Manually running the entire testing process mentioned above can be a long and tedious task. In addition, it is not safe to assume that every developer will run all the tests for every merge request to the main branch of the repository. Mistakes can happen which significantly increases the chances of faulty source code being added to the main branch. To minimise this risk we make use of GitLab CI/CD's pipeline tool as mentioned in Section 2.4.2. By carefully defining the packaging configurations and creating custom Docker images, we were able to run all the testing methods defined above in the pipeline (see Appendix F). Moreover, any testing reports and outputs generated from running tests are temporarily saved such that developers can inspect the results and check whether any errors have occurred.

Chapter 5

Design and Architecture

This chapter will go over the architectural design and functionality of the final product. Section 5.1 explores the design choices we have made along with some justifications as to why we made them. Section 5.2 subsequently explains the implementation details of the proposed design. Section 5.3 provides more insights as to how each component within the system works. Finally, Section 5.4 covers the future extensibility of the final product.

5.1 Design Choices

When creating a software system it requires design choices that are backed by logical reasoning and/or hard numbers, while simultaneously meeting the client's needs. Our final product is aimed towards researchers and engineers that are involved in distributed systems with large-scale software, Apache Spark in particular. Therefore, we have had to make design choices that have as few dependencies as possible for simplicity and as little overhead as possible to not compromise performance too much.

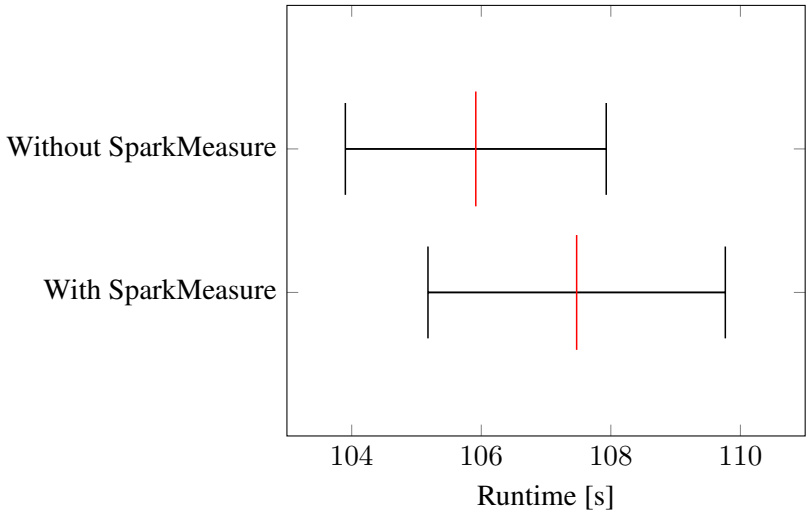
5.1.1 Plugin Option

As mentioned previously in Section 2.1.3, we were given the choice between creating a plugin or a log parser. The client advised us to measure what the potential overhead of a plugin could be and recommended a maximum overhead of 10%. We ran the TPC-DS benchmark using an existing plugin called SparkMeasure to see what overhead we could expect from our own product if we were to make a plugin. The overhead of running a Spark job along with the SparkMeasure plugin ended up being 2-3% which was well below the limit set by the client.

This works very well with sequential data. I just have two datapoints I want to compare with confidence intervals plot two data points with confidence intervals LaTeX ANSWER | PHIND MODEL For comparing two data points with confidence intervals, you can use the pgfplots package in LaTeX to plot the points and their confidence intervals. Here's an example of how to do this:

In addition to the benchmark results, we carefully weighed the pros and cons of both options when it came to development and usability. In the end, we have decided that the plugin approach is more suited to our client's needs and demands. This decision is further supported by data not provided by Spark, but accessible only via a plugin[1].

[1] I.e. energy consumption metrics



5.1.2 Core Component

For the development of the plugin, we have developed a modular design for the plugin with a bare-bone module of core logic such that adapter layers can be built on top of it. Basically, a Spark adapter module will be created based on the core module. This Spark adapter will run alongside a Spark job and collect any metrics, converting the data to the WTA format and generating Parquet[2] files. Furthermore, this design choice allows other adapters to be built using the core module. This includes but is not limited to Flink and Hadoop's MapReduce [24], [25]. Another advantage of this design choice is that future developers can easily expand the project's coverage since this project is also intended to become open-source for the community.

5.1.3 Integration of External Resources

The product requires the integration of a few external resources for its basic functionality. For instance, some hardware-level metrics such as energy consumption can not be collected from Spark alone. Since these metrics are still an important part of the WTA format, we have decided to use the `perf` performance analysis tool. `perf` is used to collect low-level performance metrics of computer systems running on the UNIX-based operating system [3] with kernel versions 2.6 and later. We believe this is sufficient for these advanced hardware-level metrics as we do expect the vast majority of distributed systems to run on Unix-based operating systems with kernel versions greater than 2.6 since versions older than that are likely no longer maintained.

We originally planned to use Prometheus to gather additional metrics for the traces. After a thorough investigation, however, we discovered that Prometheus itself doesn't directly gather metrics but instead uses client libraries to gather metrics. Moreover, it is primarily used as a monitoring and alerting tool for time series data rather than a resource gathering tool [26], which is out of the scope of this project. Most importantly, any possible metrics Prometheus could have provided were already available from other libraries we made use of. Thus we came to the conclusion of not using Prometheus as it doesn't provide much value in the context of our project.

We were initially planning on using Apache Kafka to continuously handle the incoming stream [4] of metrics. However, there were several issues with this approach. Using Kafka meant having to set up the cluster programmatically, which is a huge implementation issue. Moreover, the conventional application of Kafka did not align well with our requirements, given its nature as a distributed streaming platform,

[2] Parquet is an open-source column-oriented data storage format in the Apache Hadoop ecosystem
[3] https://perf.wiki.kernel.org/index.php/Main_Page
[4] Collection of data objects that will later be used to aggregate

which we deemed excessive for the scope of this project. Furthermore, we investigated other lightweight streaming libraries, such as Java Stream Ops [5], however, this would introduce an unneeded dependency which would increase the size of the final file. This becomes problematic due to the reasons identified in 2.5.4. Moreover, most lightweight streaming libraries have not been maintained for approximately 3/4 years. Thus we made our own metric streaming infrastructure which implemented only the functionality we needed. This gave us more control over the implementation.

To store our data and generate output, we were initially planning to use Apache Arrow [6], an in-memory data format that can be used to store columnar data that can be written to Parquet. When the Spark application is done, we aggregate the data in the stream and generate Parquet files from them. However, further research revealed that Arrow requires manual memory allocation to write to Parquet files. In addition, Arrow itself used the Apache Avro [7] module to generate Parquet files. After careful consideration, we have decided to use Avro directly instead of Arrow, which does not require manual memory management.

5.2 Architectural Overview

Combining all the design choices from the previous section, we have created the following reference diagram (Figure 5.1) that explains the interactions between the core components of our product.

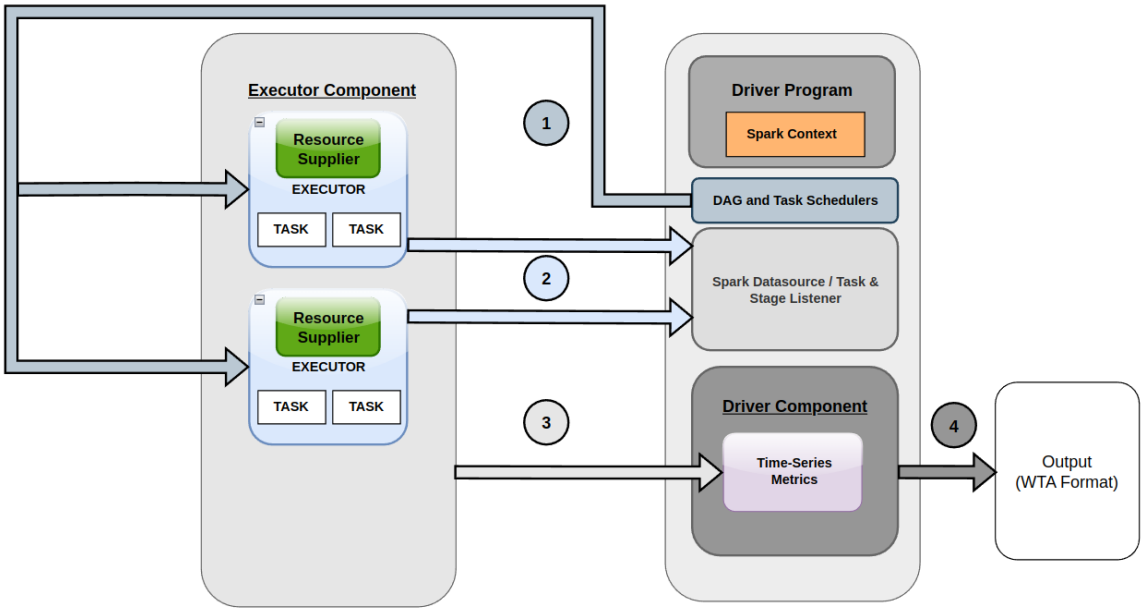


Figure 5.1: Reference Diagram for our Spark Plugin Architecture

1. The task scheduler receives sets of tasks from the DAG (Directed Acyclic Graph) scheduler at every stage and then dispatches these tasks to individual executors.
2. The Heartbeat [8] is sent by the executor to the driver once every 10 seconds to send metrics. These are intercepted asynchronously by the SparkListenerInterface.
3. RPC (Remote Procedure Calls) [9] are sent using the in-built Spark API for executors to communicate additional resource utilisation information collected from the resource suppliers.
4. Once all jobs have ended, the workload is finished and all job-related formats will be serialised into

[5] https://github.com/nanosai/stream-ops-java
[6] https://arrow.apache.org/
[7] https://avro.apache.org/
[8] Heartbeat refers to a periodic message sent by the executor nodes to the driver node to signal its liveliness and availability
[9] communication and interaction between different components of a Spark application running on distributed nodes

the Parquet format.

Utilising the aforementioned architecture, the transmission of data within the plugin is smooth and adaptable. This section, along with Section 5.3, will provide further insight into the implementation specifics.

To gather task, stage, and job-level data, the SparkListenerInterface is used. This interface provides callback functions tied to Spark life cycle events, enabling users to retrieve information like identifiers and submission or completion times.

The collected information is transformed into Java objects which are stored to be serialised into Parquet. To address the limitation of excessive memory consumption, it is impractical to store all this information solely in the driver's memory. As a solution, serialisation and de-serialisation techniques are applied (see Section 5.3.3). Upon completion of the application, all of the data is ultimately outputted to Parquet format (as seen in Figure 5.2).

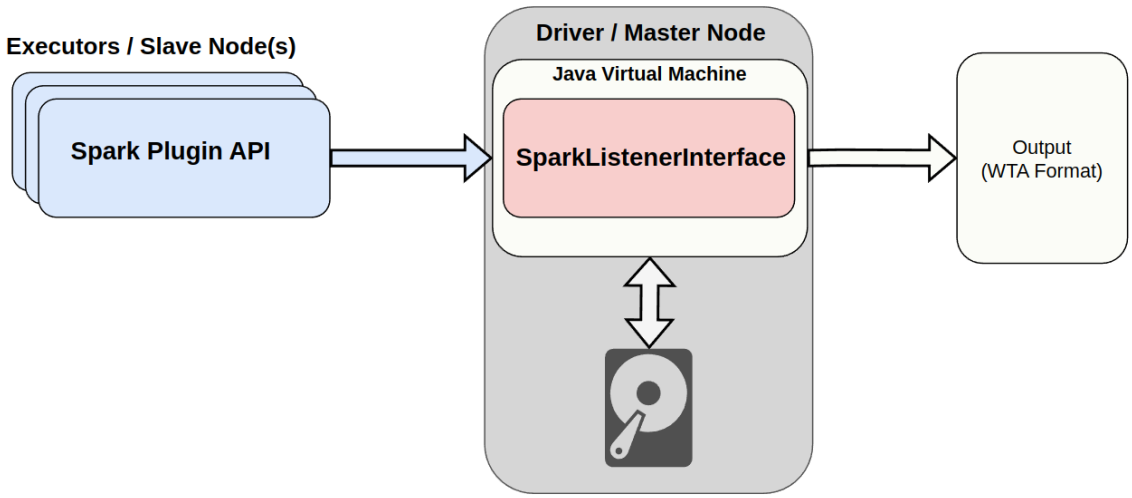


Figure 5.2: Data Flow Visualization

In order to capture additional resource utilisation metrics at the executor level, The Spark Plugin API was leveraged. This API allows developers to inject custom code during executor initialization [27]. This capability is crucial as it enables the plugin to make UNIX calls through the Java Virtual Machine without introducing additional run-time dependencies.

Running within the executor node's context provides numerous opportunities for implementing additional instrumentation functionality. However, these extensions will not be explored within the scope of this report.

Utilisation metrics are acquired periodically from metric suppliers (see Section 5.3.2). The interval for this data collection is known as the 'resource ping interval' which can be adjusted by end-users to achieve the desired level of granularity of the metrics themselves.

To mitigate the performance impact of suppliers taking longer to return metrics, asynchronous programming techniques utilising Java's CompletableFuture API [28] are employed. This allows for the parallelisation of supplier queries, preventing them from blocking each other. As a result, metrics can be obtained within a reasonable timeframe, ensuring efficient retrieval of data.

Once the metrics are collected, they are stored in a buffer on the executor node. This buffer is periodically synchronised with the driver, which subsequently inserts the metrics into a stream (as seen in Figure 5.2). This interval is referred to as the 'executor synchronization interval'.

By providing this level of control, the plugin allows end-users to optimise the synchronisation interval

based on the geographical distribution of executors. For instance, if the executors are deployed in remote locations, a shorter synchronisation interval may introduce additional overhead in terms of data packet transmission over the network. In many cases, however, a synchronisation interval of -1 is sufficient. This means that the metrics are not buffered, and are instead directly synchronised as soon as they are obtained.

Finally, data at the driver node is consumed from the stream and written to Parquet using the WTA writer (see Section 5.3.4).

5.3 Major Components

Our final product consists of several major components. These components are decoupled from each other with each one serving a unique purpose. In general, these major components all describe key parts of the workflow of our final product. They play roles in the retrieval, storage, aggregation, and the output of data. By delegating the responsibility to smaller components, it enables scalability and extensibility, ensuring that other big data applications can be easily supported.

5.3.1 Spark Listeners

We implemented our own Spark listeners to capture the data necessary for the WTA trace format. The SparkListenerAPI supplies some of the required fields. These listeners are triggered after the completion of tasks, stages, jobs, and applications, and they offer metrics that are related to these recently finished processes. These metrics cover memory usage, disk space utilisation, resource consumption, run-time, relationships with other processes, and more. Within these listeners, certain callback methods are responsible for aggregating these metrics to generate statistical values for the WTA traces. For example, they calculate the total memory consumption, the standard deviation of disk usage among tasks, and the median run-time. After the application has ended, a callback function is invoked within a listener and the collected data is written to Parquet.

5.3.2 Information Suppliers

The SparkListenerInterface does not provide all the required metrics for the WTA trace format. This is due to the fact that the trace format requires metrics about the resources used in the job (as seen in Figure 2.1), which is not accessible from Apache Spark itself. In order to obtain resource utilisation metrics that align with the data from the actual machine, we need to gather information from alternative sources. These sources, known as 'information suppliers', can gather data from diverse entities like the Java Virtual Machine or the UNIX Shell. To achieve this, all suppliers must adhere to a standardised interface that ensures data provided to the plugin adheres to a certain format. This interface also guarantees that suppliers deliver information asynchronously, preventing the query from blocking. The metrics include but are not limited to things such as energy consumption and the average load utilisation of a particular node within a cluster. Currently, the suppliers we use get information from dependencies such as perf [10], iostat [11] and dstat [12]. Next to this, we also gather data from the pseudo-filesystem file system proc [13] in order to enrich our traces.

5.3.3 Streaming Engine

The streaming engine is an engine we developed from scratch to handle large amounts of processed objects provided by the SparkListenerInterface and the information suppliers. As per Section 4.3, we

[10] https://perf.wiki.kernel.org/index.php/Main_Page
[11] https://linux.die.net/man/1/iostat
[12] https://linux.die.net/man/1/dstat
[13] https://man7.org/linux/man-pages/man5/proc.5.html

stated that the plugin should not interfere with the Spark job. Without implementing any optimisation, if we were to store these objects in the memory of the driver, it would rapidly exhaust its available memory, and eventually result in an out-of-memory error. The streaming engine has the capability to process data from different data sources, assigning a separate data stream to each source. Each data stream efficiently manages the serialisation of objects to disk once a specified number of objects (by default 1800) have been added. When data aggregation is required, the engine intelligently deserialises objects in batches for further processing. To handle heavier workloads, the streaming engine is designed to scale effectively by utilizing Java thread pools. This non-blocking approach ensures that expensive I/O calls do not impede the performance of the rest of the system.

5.3.4 WTA Writer

The Parquet writer writes the WTA trace to the user-specified location. For most objects in the WTA format, it will dynamically generate the schema of their corresponding Parquet files based on the availability of each field by utilising the Java reflection API, after which it writes the corresponding Parquet file. For this purpose, we used the Apache Parquet and Avro libraries together with our own implementations of their interfaces in order to allow our users to avoid a big dependency, namely a local Hadoop installation. These implementations have afterwards been submitted to the Parquet project. The workload object on the other hand is written directly to a JSON file as specified by the format.

5.4 Future Extensibility

While our product is aimed at Spark, the core module can be used to build new adapters for other platforms. Spark is just one of many popular distributed data processing engines used in the industry. Other possible engines include but are not limited to Flink and Hadoop's MapReduce. While their internal process differs from each other, the high-level process required to extract execution information is similar. Some common behaviour for this extraction includes common collectors for gathering metrics, metric streaming infrastructure, and the generation of Parquet files. We believe that by capturing this common behaviour in our core module, we will allow other developers to build their own adapters with ease. One thing to keep in mind is that the performance overhead could differ between adapters. This is not only due to the nature of the different processing engines but also the potentially different designs on the adapter, such as when building a log parser instead of a plugin. In the following sections we will mention some ways the core functionality of the plugin could be extended.

Chapter 6

Evaluation

Before going into concluding remarks, it is crucial to conduct a thorough assessment of not only the final product but also the project life cycle. Section 6.1 elaborates on the requirements fulfilled by the completed product. Section 6.2 delves into how effectively the dependency overhead issue outlined in Section 2.5.4 is handled. Section 6.3, dives into the evaluation of the final product, where it explores the usability of the plugin. Section 6.4 elaborates on the verification of the system. The chapter concludes with Section 6.5 with a comprehensive performance analysis using benchmarks.

6.1 Fulfillment of Client Requirements

All of the must-have requirements as specified by the client were met after 3 sprints. The subsequent sprints were augmenting the must-have requirements with additional resource-utilisation metrics that were collected from the Java Virtual Machine or additional UNIX dependencies. This was crucial as it allowed us to fill in other parts of the trace format such as the resource and resource state objects.

In terms of collecting resource utilisation metrics, we have encountered a significant limitation concerning the outputted Parquet file. Our current approach only allows us to capture resource utilisation metrics at the executor level, rather than at the task level. Consequently, certain fields, such as energy consumption within the task object, may be duplicated across multiple tasks executed on the same executor. It is worth noting that addressing this limitation falls outside the scope of our project, and is primarily due to the limitations of the Apache Spark API itself.

The plugin operates independently without causing any disruptions to the ongoing Spark job. In the event of an error on the driver side, the Spark job will continue unaffected, although the plugin will cease to collect metrics. Our comprehensive test suite, consisting of unit, integration, and system tests was used to verify this behaviour. In the case of an executor failure, the plugin will simply halt recording resource utilisation metrics from that particular executor. Furthermore, the failure of the executor will be logged on the command line. To validate this functionality, we intentionally terminated an executor process on DAS-5, and the plugin resumed its operation seamlessly without any errors.

During the development process, we also realised that some requirements were simply infeasible during our time constraints. For example, could-have requirement 6 mentioned in 4.2.3 was found to be difficult to implement due to the limitations of the Spark API, which did not provide a lot of data related to the WTA data-transfer metrics. In consultation with the client, we prioritised the stability of the application over the implementation of these could-have features.

6.2 Dependency Minimisation

Bloated dependencies were a huge problem as they refer to third-party libraries that are not necessary to run the application but packaged into the application file regardless [29]. As a result, the size of our first application file was more than 200 megabytes. This was not acceptable as the client requested an application file size of no more than 20 megabytes. This meant that drastic resizing was necessary. This required careful, systematic dependency packaging to exclude any dependencies not necessary to run the application. As mentioned in the previous paragraph, we even implemented part of our own Parquet writer to write the WTA trace while avoiding the Hadoop dependency, which itself was almost 50 megabytes. In the end, we were able to shrink the application file down to around 500 kilobytes, which is an approximately 99.75% reduction in file size.

6.3 Usability

Enhancing the usability of the final product is imperative as it directly affects the overall user experience. Based on our analysis, we have identified that the most challenging aspect of using our product revolves around the configuration and installation steps. These steps need to be supported by comprehensive documentation. To address this concern, we have developed detailed step-by-step instructions for the installation and a clear explanation of the necessary dependencies required to run the plugin.

To help users install supplementary dependencies, we have created four `Dockerfiles`. These `Dockerfiles` install all the dependencies the Spark adapter needs which can be configured through a Docker image. We provide these images for four relevant Linux distributions so that they can be used by users deploying Spark in containerised environments. The distributions for which we currently support this are Linux Alpine, CentOS, Debian, and Ubuntu. We included these four because Alpine is often used for containerised applications in general, CentOS is often used to run such clusters because of its stability, also by ASML, and Debian and Ubuntu remain the most popular Linux distributions. Sadly, we could not include a full `perf` installation in this `Dockerfile`. This is because `perf` heavily relies on the kernel and must be compatible with it. As Docker containers do not virtualise this part of the system, it will still need to be enabled by the user, as stated in the documentation.

The end-user is provided with options to customise our application wherever possible, as specified in the documentation under the configuration section. These options allow users to modify parameters such as the frequency of resource utilisation metric collection and the output directory for the generated traces.

Once the user correctly configures the plugin, it will collect metrics alongside an active Spark job. To enhance usability, we have incorporated custom log messages in addition to the existing Spark debug messages. This feature provides the end-user with a comprehensive overview of the Spark job's progress and the plugin's actions at any given moment. Additionally, the plugin logs any abnormal behaviour by appending a custom error message on top of the existing Spark log.

To ensure the validity of the generated trace, we have included documentation that instructs users to run the verification scripts hosted on the client's WTA Tools repository. These scripts can be utilised for verification purposes.

In addition, end-users have the option to expand the capabilities of the product. This situation may arise when there is a requirement to collect metrics from another big data application, such as Apache Flink, or when additional resource utilisation metrics need to be gathered from a different source. The modular design of our application allows for extension with new functionality, provided that users follow the instructions outlined in the documentation. For a detailed overview of the code documentation check Appendix C.

6.4 System Verification

During the second half of the project, we started to employ manual testing of the product in the production environment and stress testing. Although the test suites mentioned in Section 4.5.1 do provide a certain level of robustness to the product's development life-cycle, it does not guarantee how well the product itself will hold when run under the production environment nor when it is pushed to its processing limits. Manual testing ensured that the product ran smoothly in the production environment and stress testing helped to define the limitations of the product and explored ways to extend it.

We regularly ran our plugin on the DAS-5 as mentioned in Section 2.4.2. In order to ensure that our plugin could handle long-lasting workloads effectively, we conducted 12-hour test runs. Initially, we encountered errors related to the memory allocation for the driver node. Through stress testing, we were able to pinpoint bottlenecks within the plugin that needed to be addressed. This involved freeing up memory by clearing unnecessary information from maps and utilising the streaming engine instead of solely relying on memory. After implementing these changes, the plugin ran smoothly and produced properly formatted traces.

Unfortunately, we couldn't perform extensive stress testing on DAS-5 for an extended period of time due to limitations in slot allocation. The following section provides information on the benchmark testing carried out on DAS-5.

6.5 Performance Analysis using Benchmarks

As per non-functional requirement 11 specified in Section 4.3, we regularly ran benchmarks after our MVP was completed. Similar to the initial benchmarks performed in Section 2.4.3, these rounds of benchmarking were done using TPC-DS queries and data on DAS-5. It is important to note that for the plugin, the time it took for generating the trace files was not factored into the benchmarks as that is not a data processing task.

When benchmarking in a distributed environment, it is necessary to run multiple iterations due to the nature of variability in distributed computing [30]. TPC-DS does not specify a minimum number of iterations for running the queries, thus it is up to the engineers to decide these numbers based on the circumstances. To obtain a reliable and consistent measurement, we decided on 20 iterations. We chose this number since we discovered several bugs in the code that severely affected the performance and reliability of the software. Therefore, these bugs had to be fixed before the final benchmarking. In the end, there was only enough time to benchmark for ten hours, which is equivalent to 20 iterations.

The final benchmarking used 20GB of TPC-DS data and all 99 TPC-DS queries per iteration. The data could not be any bigger due to the disk space limitations per users of the DAS-5.

The following resource settings of the DAS-5 and Spark were used to benchmark under a realistic production environment:

- 10 nodes in the cluster (1 master, 9 workers)
- 9 Spark executor instances
- 30GB memory per Spark executor instance
- 4 cores per executor instance

The following violin plot shows the distribution of iteration run-time between Spark jobs without the plugin and Spark jobs with the plugin:

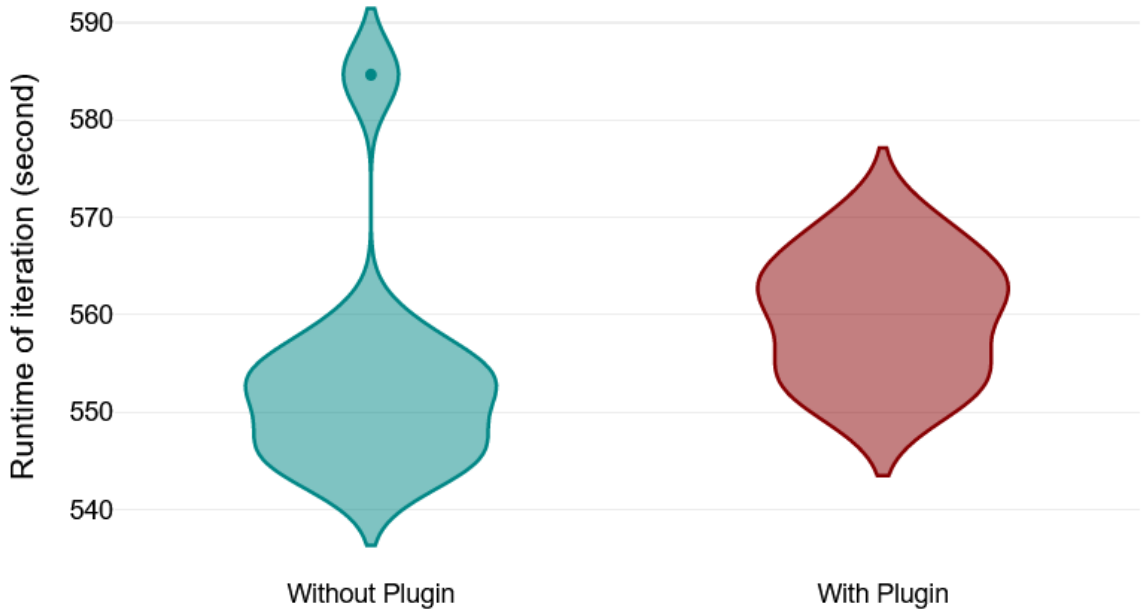


Figure 6.1: Violin plot of the distribution of each iteration run-time

As presented in Figure 6.1 there is a slight overhead in the general distribution of Spark jobs with the plugin. This is to be expected as running a Spark job with the plugin will require extra processing from the system. In addition, there are outliers deviating from the distribution of Spark jobs without the plugin. These outliers can be seen as true anomalies as they lie outside more than two standard deviations away from the average, which is often used to detect anomalies [31]. Therefore, the performance overhead and statistical analysis were done without these anomalies as seen in Table 6.2.

For the measurements of the statistical summary we chose sum, variance, standard deviation, minimum, maximum, and average as shown in Table 6.2. They provide a comprehensive understanding of the distribution of the results and its central tendencies: the sum is used to calculate other statistics, the variance and standard deviation shows the variability of the data, the minimum and maximum show the range of values and potential outliers, and the average shows the central tendency of the result. The measurement unit in Table 6.2 is second, except for the overhead part which uses percentages.

Table 6.1: Statistical summary with anomalies

Groups	Count	Sum	Variance	SD	Minimum	Maximum	Average
Without Plugin	20	11077.389	132.260	11.500	543.063	584.693	553.869
With Plugin	20	11196.067	38.511	6.206	550.693	569.048	559.803
Overhead (percentage)					1.41	-2.67	1.07

Table 6.2: Statistical summary without anomalies

Groups	Count	Sum	Variance	SD	Minimum	Maximum	Average
Without Plugin	18	9908.121	24.099	4.909	543.063	558.318	550.451
With Plugin	20	11196.067	38.511	6.206	550.693	569.048	559.803
Overhead (percentage)					1.41	1.92	1.70

Our null hypothesis is that the two groups have no relationship in terms of the benchmark result, thus any

difference in the result is merely coincidence. Our alternative hypothesis is that there exists a relationship between the two groups. The P-value[1] from our statistical summary without anomalies is 0.00001, which is well below the 0.05 P-value threshold to reject the null hypothesis [32]. Even if we consider the statistical summary with the anomalies, the P-value is 0.049 which is still below the 0.05 P-value threshold. This means that the difference in the benchmark result is not a coincidence and using the plugin causes overhead. However, our overhead value is well below the ten per cent threshold as per the non-functional requirement 11 specified in Section 4.3. These can be seen in both Table 6.2 and Table 6.1. For a more detailed overview of the benchmarking info see Appendix A.

In conclusion, we can safely say that any overhead our plugin causes will have no significant performance impact on the running systems. In addition, running the plugin on a highly intensive job for 12 hours continuously didn't throw any errors. Therefore, we can say with a certain level of confidence that we have developed a reliable product that meets our client's needs.

[1]Statistical measurement used to validate a hypothesis against observed data.

Chapter 7

Conclusion

The objective of this report is to present a tool that generates WTA traces from Spark executions, which will make it easier to share knowledge about the execution of distributed computing systems. We have developed a WTA trace generation framework that retrieves data from the Spark application as a plugin and outputs the trace to the user-specified location. These traces contain data on a diverse set of metrics, including memory, disk, and energy-related data. Our product will also aggregate these low-level metrics retrieved from Spark and the operating system. These aggregated metrics can provide insights into the performance of Spark and allow researchers to further optimise it.

All of the must-haves and should-haves have been met. The majority of the could-haves we did not do related to the integration of the validation and anonymisation scripts after the traces were generated (see Appendix E). Since the plugin was designed with extensibility in mind, these can be added with ease. Increasing the general quality of the plugin code through refactoring code and increasing coverage of the different test suites was deemed to be of higher priority since validation and anonymisation could be done by manually running the respective scripts.

Our product has been through comprehensive benchmarking including TPC-DS benchmarking via DAS-5. It has also gone through stress testing to verify its robustness. From the result, our product does not pose a heavy burden to the Spark job as the run-time overhead stays well under 10%, following the client.

We have also looked into the possible risks and ethical implications of our product, concluding that it is safe from most attacks and only bears a small risk against side-channel analysis attacks, which can be prevented through proper configuration by the user.

While the product we provide is complete, it is intended to be published as an open-source tool for the distributed computing community and is designed with further future development in mind. We have focused a lot on the extensibility of our product during development. The product can be extended for other data sources or even different frameworks such as Apache Flink in the future. The project's source code will be published on GitHub together with the other WTA-related tooling. As it is FOSS, we welcome anyone to contribute to and improve the existing plugin with additional functionality and help the distributed computing community move towards a unified trace format.

Bibliography

[1] L. Versluis, R. Mathá, S. Talluri, *et al.*, "The workflow trace archive: Open-access data from public and private computing infrastructures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 9, pp. 2170–2184, 2020. DOI: 10.1109/TPDS.2020.2984821.

[2] @Large Research, *Trace format*, 2021. [Online]. Available: https://wta.atlarge-research.com/traceformat.html (visited on 06/25/2023).

[3] L. Canali, *Sparkmeasure*, 2017. [Online]. Available: https://github.com/LucaCanali/sparkMeasure (visited on 06/25/2023).

[4] A. Noureddine, "Powerjoular and joularjx: Multi-platform software power monitoring tools," in *18th International Conference on Intelligent Environments (IE2022)*, Biarritz, France, Jun. 2022, pp. 1–4, ISBN: 978-1-6654-6934-0. DOI: 10.1109/IE54923.2022.9826760.

[5] V. M. Weaver, D. Terpstra, H. McCraw, *et al.*, "PAPI 5: Measuring power, energy, and the cloud," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Austin, TX, 2013, pp. 124–125, ISBN: 978-1-4673-5779-1. DOI: 10.1109/ISPASS.2013.6557155.

[6] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 3, pp. 189–204, Aug. 2000, ISSN: 1094-3420. DOI: 10.1177/109434200001400303.

[7] *Java native interface specification contents*, Oracle, 2017. [Online]. Available: https://docs.oracle.com/en/java/javase/11/docs/specs/jni/ (visited on 05/03/2023).

[8] H. Bal, D. Epema, C. de Laat, *et al.*, "A medium-scale distributed system for computer science research: Infrastructure for the long term," *Computer*, vol. 49, no. 5, pp. 54–63, May 2016. DOI: 10.1109/mc.2016.127.

[9] GitLab, *GitLab CI/CD*. [Online]. Available: https://docs.gitlab.com/ee/ci/ (visited on 06/14/2023).

[10] S. Hatton, "Choosing the right prioritisation method," in *19th Australian Conference on Software Engineering (ASWEC 2008)*, 2008, pp. 517–526. DOI: 10.1109/ASWEC.2008.4483241.

[11] M. Poess, R. O. Nambiar, and D. Walrath, "Why you should run tpc-ds: A workload analysis.," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07, Vienna, Austria: VLDB Endowment, 2007, pp. 1138–1149, ISBN: 978-1-59593-649-3.

[12] *GNU general public license*, version 2, Free Software Foundation, Jun. 1991. [Online]. Available: https://www.gnu.org/licenses/gpl-2.0.html (visited on 05/02/2023).

[13] *GNU general public license*, version 3, Free Software Foundation, Jun. 29, 2007. [Online]. Available: https://www.gnu.org/licenses/gpl-3.0.html (visited on 05/02/2023).

[14] P.-E. Schmitz, *Why viral licensing is a ghost*, 2015. [Online]. Available: https://joinup.ec.europa.eu/node/147739 (visited on 05/03/2023).

[15] P. Albert, *GPL: Viral infection or just your imagination?* 2004. [Online]. Available: https://www.linuxinsider.com/story/gpl-viral-infection-or-just-your-imagination-33968.html (visited on 05/03/2023).

[16] Chicago Sun Times, *Microsoft ceo takes launch break with the sun-times*, 2001. [Online]. Available: https://web.archive.org/web/20010615205548/http://suntimes.com/output/tech/cst-fin-micro01.html (visited on 05/03/2023).

[17] *Apache license*, version 2.0, Apache Software Foundation, Jan. 2004. [Online]. Available: https://www.apache.org/licenses/LICENSE-2.0 (visited on 05/02/2023).

[18] Y. Loonat, *Which license should i use? MIT vs. Apache vs. GPL*, 2016. [Online]. Available: https://www.exygy.com/blog/which-license-should-i-use-mit-vs-apache-vs-gpl (visited on 06/14/2023).

[19] L. Versluis, *WTA tools*, 2019. [Online]. Available: https://github.com/atlarge-research/wta-tools (visited on 06/25/2023).

[20] M. W. Kranenbarg, T. J. Holt, and J. van der Ham, "Don't shoot the messenger! a criminological and computer science perspective on coordinated vulnerability disclosure," *Crime Science*, vol. 7, no. 1, 16, pp. 1–9, 2018. DOI: 10.1186/s40163-018-0090-8.

[21] Apache Software Foundation, *Asf security team*. [Online]. Available: https://www.apache.org/security/ (visited on 05/03/2023).

[22] L. Stankovic, V. Stankovic, J. Liao, and C. Wilson, "Measuring the energy intensity of domestic activities from smart meter data," *Applied Energy*, vol. 183, pp. 1565–1580, Dec. 2016, ISSN: 0306-2619. DOI: 10.1016/j.apenergy.2016.09.087.

[23] L. Lerman, G. Bontempi, and O. Markowitch, "Power analysis attack: An approach based on machine learning," *International Journal of Applied Cryptography*, vol. 3, no. 2, pp. 97–115, 2014. DOI: 10.1504/IJACT.2014.062722.

[24] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "Apache flink: Stream and batch processing in a single engine," *Bull. IEEE Comput. Soc. Tech. Committee Data Eng.*, vol. 36, no. 4, 2015.

[25] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004, pp. 137–150.

[26] P. Authors, *Prometheus - monitoring system & time series database*, 2014. [Online]. Available: https://prometheus.io/.

[27] L. Canali, *Sparkplugins*, 2019. [Online]. Available: https://github.com/cerndb/SparkPlugins (visited on 06/25/2023).

[28] *Java se 8 documentation - class completablefuture*, Oracle Corporation, 2014. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html (visited on 06/25/2023).

[29] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, "A comprehensive study of bloated dependencies in the maven ecosystem," *Empirical Software Engineering*, vol. 26, no. 3, p. 44, 2021. DOI: 10.1007/s10664-020-09914-8.

[30] A. V. Papadopoulos, L. Versluis, A. Bauer, *et al.*, "Methodological principles for reproducible performance evaluation in cloud computing," *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1528–1543, 2019. DOI: 10.1109/TSE.2019.2927908.

[31] R. Arava, *Anomaly detection — part 1*, 2020. [Online]. Available: https://medium.com/analytics-vidhya/anomaly-detection-part-1-acf1a993b573 (visited on 06/25/2023).

[32] G. D. Leo and F. Sardanelli, "Statistical significance: P value, 0.05 threshold, and applications to radiomics—reasons for a conservative approach," *European Radiology Experimental*, vol. 4, no. 18, p. 8, 2020. DOI: 10.1186/s41747-020-0145-y.

[33] R. Bevans, *One-way anova | when and how to use it (with examples)*, 2020. [Online]. Available: https://www.scribbr.com/statistics/one-way-anova/ (visited on 06/14/2023).

[34] N. Haugen, "An empirical study of using planning poker for user story estimation," *AGILE 2006 (AGILE'06)*, p. 34, 2006. DOI: 10.1109/agile.2006.16.

[35] *Migrating a project to scala 2.13's collections*, Scala. [Online]. Available: https://docs.scala-lang.org/overviews/core/collections-migration-213.html (visited on 06/25/2023).

[36] *Introduction to the build lifecycle*, Apache Software Foundation. [Online]. Available: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html (visited on 06/25/2023).

[37] N. Mündler, J. He, S. Jenko, and M. Vechev, *Self-contradictory hallucinations of large language models: Evaluation, detection and mitigation*, 2023. DOI: 10.48550/ARXIV.2305.15852. [Online]. Available: https://arxiv.org/abs/2305.15852 (visited on 06/21/2023).

[38] M. A. Quidwai, C. Li, and P. Dube, *Beyond black box ai-generated plagiarism detection: From sentence to document level*, 2023. DOI: 10.48550/ARXIV.2306.08122. [Online]. Available: https://arxiv.org/abs/2306.08122 (visited on 06/21/2023).

Appendices

Appendix A

Benchmark Statistics

Table A.1: A table of sample queries from a sample iteration

Query number	Run-time (second)
1	15.670
2	7.307
3	1.613
4	31.637
5	8.829
6	2.633
7	2.675
8	2.576
9	22.193
10	6.409
11	12.682
12	2.283
13	2.380
14a	21.647
14b	17.985
15	1.960
16	8.196
17	4.463
18	5.319
19	2.137
20	2.129

For statistical analysis of the benchmark result, we used one-way analysis of variance, also known as one-way ANOVA. One-way ANOVA is a statistical method to determine the relationship between two or more unrelated groups. While there are different variations of ANOVA, we specifically chose one-way ANOVA as there is only one independent variable, which is the benchmark result [33]. The ANOVA table consists of the following components: sum of squares, degrees of freedom, mean squares, F-statistic, and P-value.

Table A.2: One-way ANOVA with anomalies

Source of Variation	Sum of Squares	Degrees of Freedom	Mean Squares	F-statistic	P-value
Between Groups	352.112	1	352.112	4.124	0.049
Within Groups	3244.645	38	85.385		
Total	3596.757	39			

Table A.3: One-way ANOVA without anomalies

Source of Variation	Sum of Squares	Degrees of Freedom	Mean Squares	F-statistic	P-value
Between Groups	828.602	1	828.602	26.135	0.00001
Within Groups	1141.384	36	31.705		
Total	1969.987	37			

Appendix B

Code Coverage Statistics

Table B.1: Test Types and Coverage

Type	Coverage
Branch (Spark Adapter)	83%
Line (Spark Adapter)	97%
Branch (Core)	81%
Line (Core)	85%

Appendix C

Code Documentation

We documented our code throughout the project, using both Javadoc and `README.md` files which are contained in the subproject's roots. The existence of Javadoc was enforced by our pipeline in order to allow anyone to generate a complete site of documentation using Javadoc tooling, as can be done using Maven's site goal in the final repository.

The READMEs we created provide information to users of our application on how to configure and get started with using it, a sample of which can be seen in Figure C.1. It explains the properties that need to be configured and it gives examples of how users can attach the plugin to Spark runs. It also provides an overview of our Maven configuration, including the profiles we set up and an overview of which goals those plugins are attached to in the Maven lifecycle. We include the commands we would expect users and later developers to use and need to package the application and generate the site. Next to this, we also provide information for developers who would like to contribute or extend our application in the README. We do this by providing them some specifications and guides on the code in the `core` module.

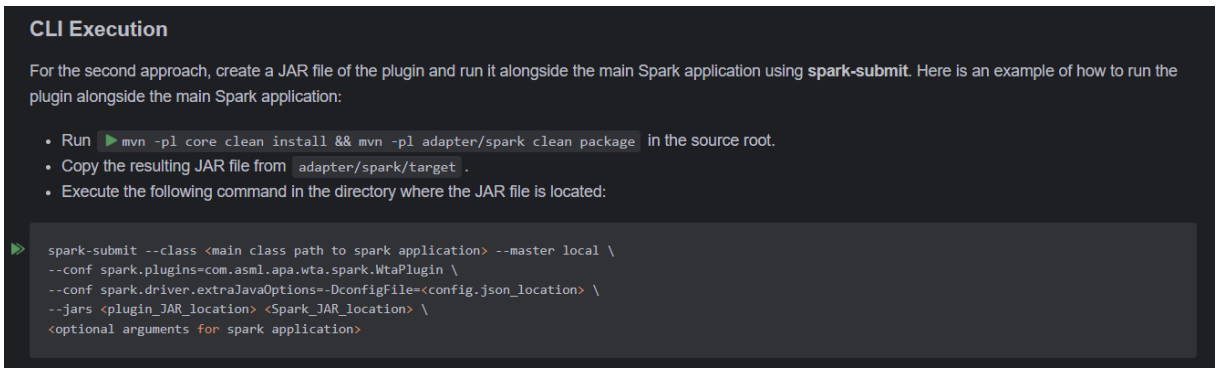


Figure C.1: CLI usage section in the README

Next to just the Javadoc and READMEs, our Maven configuration also includes a site. This site has information on the dependencies and plugins configured in Maven and numerous additional reports. Next to the Javadoc site and a browsable copy of our code, these reports include data on all kinds of areas such as code quality, test coverage, dependency updates, and the licenses used by these dependencies. This can provide users and contributors alike valuable insights into the repository and application. A sample of the site as it would be generated can be seen in Figure C.2.

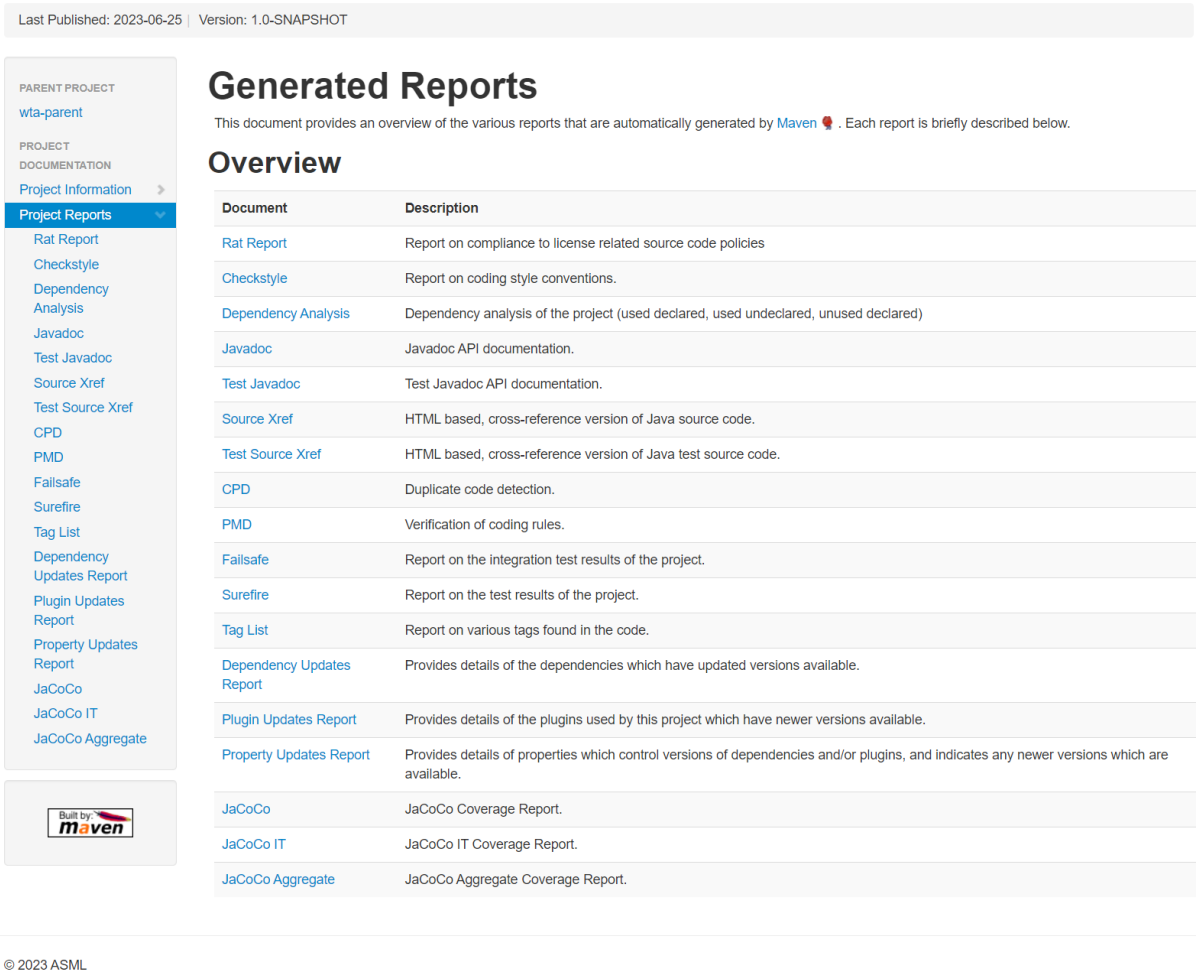


Figure C.2: The generated reports page of the Maven site

Appendix D

Development Process

D.1 Communication as a Team

Effective communication is vital for the success of any program development project. It ensures team members are aligned, informed, and can collaborate efficiently towards project goals. Three key communication practices in such projects are official weekly meetings, weekly retrospectives, and stand-up meetings. " Official weekly meetings bring the entire team together to discuss updates, challenges, and future plans. Led by a project manager or team lead, these meetings foster collaboration, distribute issues, and ensure everyone is on the same page. Team members share task updates, highlight roadblocks, and seek assistance. These meetings find out the issues that block development elsewhere and promote transparency, accountability, and knowledge sharing.

On the other hand, we will have weekly retrospectives that reflect on the past week and find possible improvements in the future. We would discuss the tips and tops for the team performance and propose solutions for the next week.

Stand-up meetings complement weekly meetings, providing frequent communication within the team. Occurring every other day, these short and focused gatherings allow for quick progress updates, accomplishments, and identification of roadblocks. Each team member shares their goals for the day and any assistance required. Stand-up meetings promote collaboration, align efforts, and maintain a clear project status.

Besides, we also keep online and offline communication channels to enable ongoing collaboration. Online channels include instant messaging platforms such as Discord, project management tools, and collaborative document-sharing platforms such as Overleaf and Google Docs. They provide flexibility for remote collaboration, real-time updates, and asynchronous communication. Offline communication, such as face-to-face interactions, strengthens relationships and team dynamics.

Overall, by implementing these practices and utilising different communication means, our team has been able to foster a productive and collaborative environment, ensuring the success of program development projects.

D.2 Weekly Goals and Reflections

Throughout the project, we used the Scrum methodology whilst planning our sprints on a weekly basis. To determine which issues would be addressed in the upcoming sprint, we conducted face-to-face meetings involving all group members. During these meetings, we took advantage of a technique known as Scrum Poker, which is a gamification of the process of estimating the effort and time required for each issue. Each group member provided input, enabling us to collectively assign expected time, weight, and people

to the issues. This approach not only fostered collaborative teamwork instead of individual task pursuit but also is known to provide more accurate time and difficulty estimations on issues [34].

Similarly, at the conclusion of each sprint, we engaged in a retrospective to reflect on the successes and challenges encountered. By conducting this retrospective just before planning the next sprint, we were able to consider the lessons learned from the previous sprint and adjust our plans accordingly. This iterative process allowed us to enhance our sprint planning on a weekly basis.

Through this iterative refinement, we observed that our time estimations for the issues became more accurate. Initially, we noticed a tendency to be overly ambitious in the early sprints, but by learning from each iteration, we were able to make more realistic estimations.

Furthermore, during the initial sprints, we identified an opportunity to improve communication by directly consulting team members who were more involved in specific issues. This adjustment helped us reduce unnecessary communication overhead.

Overall, by utilising the Scrum methodology, incorporating Scrum Poker, conducting retrospectives, and refining our communication practices, we were able to enhance our sprint planning and achieve greater accuracy in time estimations for the issues at hand.

During our sprint planning, we actively referenced the Gantt chart (see Figure D.1) that we initially developed at the start of the project. Since this Gantt chart primarily outlined the priority of various issues based on the MoSCoW model, it provided us with a high-level perspective on whether we needed to assign a greater or lesser number of issues in upcoming sprints to maintain our progress. Additionally, the Gantt chart included mentions of administrative tasks that needed to be accomplished alongside the product, enabling us to account for potential time-consuming activities that could impact our programming efforts.

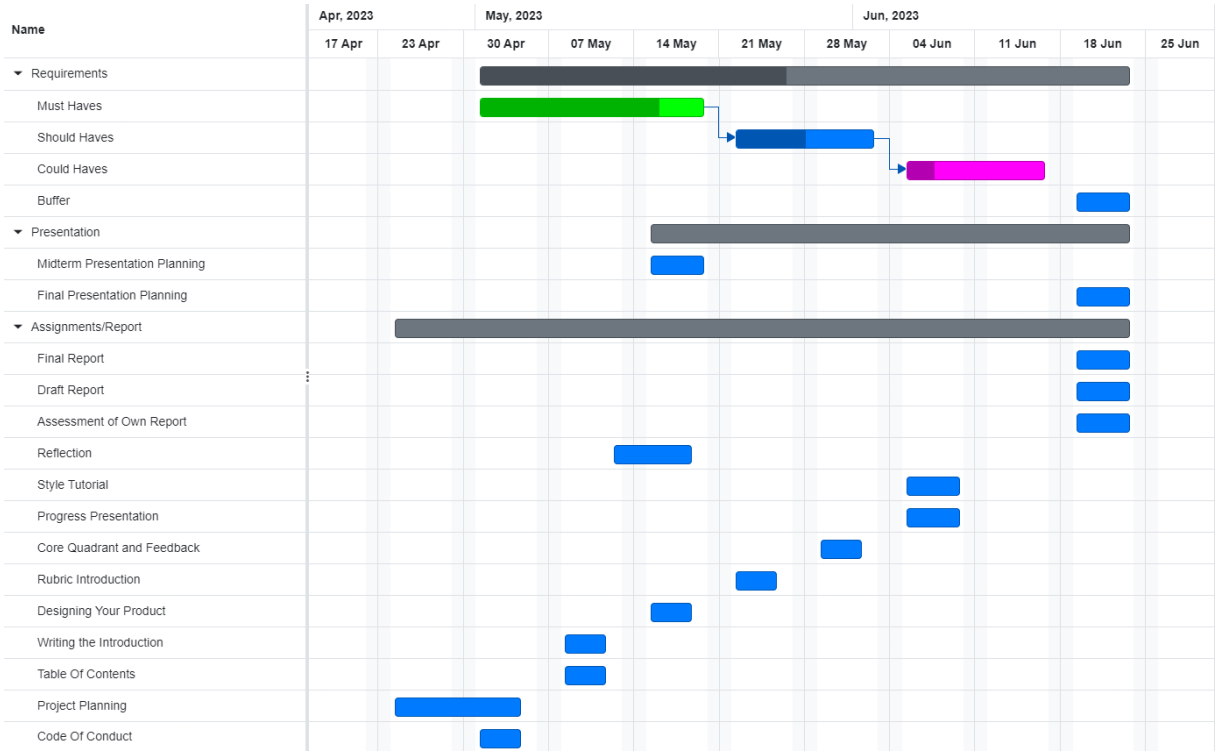


Figure D.1: Gantt chart detailing the planned process throughout the project

Appendix E

Requirements Completion Overview

Here we go over the original requirements and their completion status. As you can see we did not implement the could-haves. This was done in consultation with the client, who prioritised a stable system over these could-have features.

E.1 Must-haves

Requirement	Status
The application shall at least support Spark 3.2.	Completed
The application shall require the user to specify the authors and domain[1] of a trace.	Completed
The application shall ask the user for a description of the trace and notify them when this is left blank.	Completed
The application shall collect the resource and hardware-related metrics[2] in the trace.	Completed
The application shall collect the memory-related metrics[3] in the trace.	Completed
The application shall collect the disk-related metrics[4] in the trace.	Completed
The application's output shall adhere to the WTA format.	Completed
The application shall collect the metadata of the Workload object.	Completed
The application shall output the generated trace to the disk.	Completed

E.2 Should-haves

Requirement	Status
The application shall output the generated trace in the Parquet format using the Snappy compression algorithm.	Completed
The generated trace shall include a Parquet file naming format for versioning of the application.	Completed
The application shall generate INFO-level logs.	Completed
The user shall have the option to use stage-level metrics rather than Spark's task-level metrics.	Completed
The application shall collect energy metrics[5] in the trace where possible.	Completed

E.3 Could-haves

Requirement	Status
The application shall use Prometheus metrics to enrich the traces.	Not completed
Users shall have the option to automatically validate the generated traces.	Not completed
Users shall have the option to automatically anonymize their traces.	Not completed
The application shall support S3 as a storage location for the generated traces.	Not completed
Users shall be given the option to automatically upload their traces to Zenodo after anonymization and validation.	Not completed
The application shall collect data transfer-related metrics[6] in the trace.	Not completed
The application shall support the integration of GPU metrics from NVIDIA Rapids.	Not completed

E.4 Non-functional Requirements

Requirement	Status
The project shall be written in Java 11 LTS.	Completed
The project shall use Maven as the build tool.	Completed
The project shall be created in a modular fashion such that new adapters can augment the project.	Completed
The core project shall be able to handle and aggregate multiple data sources.	Completed
The codebase shall have both a branch and code test coverage of at least eighty per cent, under the condition that every test case has at least one meaningful assertion.	Completed
The project shall use JaCoCo to generate test coverage reports.	Completed
The project shall incorporate mutation testing, with test coverage of at least sixty per cent.	Completed
The project will be licensed under a permissive open-source license such as Apache License, Version 2.0.	Completed
The pipeline will integrate Spotless and Checkstyle to adhere to style guidelines.	Completed
The application shall never interfere with a Spark job through its exceptions.	Completed
The final product shall not exceed a ten per cent slowdown when running the TPC-DS benchmarks over a plain Spark run, while minimising the slowdown as much as possible.	Completed

Appendix F

Continuous Integration

To guarantee code quality during development we made use of GitLab CI/CD. To this end, we set up a pipeline to guarantee the correctness of our code continuously throughout the lifetime of the project. The pipeline we set up included numerous tasks designed to spot bugs, code style problems, and licensing issues early on. Because of this we were able to quickly catch a lot of bugs and regressions, which were then patched shortly.

Our pipeline starts off with jobs to build the necessary images, in order to ensure they are present in the repository when running the pipeline. To not waste resources we check for their presence in the package registry before spending the resources to build any images. We set up a base image with Ubuntu Focal that includes installations of Maven and Java 11. After this we set up two Spark images which are used for the end-to-end runs. The two images set up Scala and Spark on the base image we built. The Spark images differ in the Scala version. One is built for Spark with Scala 2.12, while the other is built for Spark with Scala 2.13, this is necessary as Scala 2.12 and Scala 2.13 are not compatible [35]. Then we set up three other images which layer installations of command line dependencies of the Spark adapter on top, these dependencies include `dstat`, `iostat`, and `perf`. These images are later used during integration tests and a portion of the end-to-end runs. The three images we generate are one which layers these dependencies on top of the base image, one which layers these dependencies on top of the Scala 2.12 Spark image, and one which layers these dependencies on top of the Scala 2.13 Spark image. The first is used for running the integration tests, while the latter two are used for running end-to-end jobs.

After setting up the images, we build and package our code. This is done in dedicated jobs which do not run anything but the building and packaging of our code. This means we do not run any tests here yet, this despite `package` and `install` being run after `test` in the Maven lifecycle [36]. Next to just packaging our standard JARs, this stage also includes packaging the end-to-end JAR. This is a JAR which includes the `EndToEnd` test class, which is needed to run the end-to-end jobs later on in the pipeline. We package these JARs for Spark versions relying on both Scala 2.12 and Scala 2.13.

After this stage of building and packaging our code, we go on to two testing stages. One concerns the tests in the `core` module, while the other is concerned with testing `adapter/spark`. The stage containing `core` related jobs runs its unit and integration tests. Next to these two jobs it also runs two more dummy jobs which merge and report the coverage statistics gathered from running the unit and integration tests from JaCoCo to GitLab CI/CD. We also run one more job to enforce a line and branch coverage of over 80%. The stage containing the `adapter/spark` related jobs runs slightly more jobs, having all the same jobs as the `core`, but also including the end-to-end runs. It includes four of these end-to-end jobs. Two running the end-to-end test using Spark on Scala 2.12 and Scala 2.13 respectively in an environment where none of the command line dependencies are met, and two running Spark on Scala 2.12 and Scala 2.13 respectively while all of the command line dependencies are installed, with `kernel.perf_event_paranoid` set to `0`, in order to allow the application access to the `perf` dependency.

Then finally, we have one last stage which includes the final code checks. This stage includes code style verification, dependency management, and static application security testing (SAST). To help with code style we integrated three tools: Spotless, Checkstyle, and PMD. We configured these where necessary, using a slightly modified version of the Palantire Baseline Checkstyle configuration[1] as our code style guideline. Next to these three linters, we ran two of GitLab's tools, its Gemnasium-based Dependency Scanning analyser and its Semgrep-based SAST. These made sure we did not get in trouble with our dependencies or introduce any security concerns into our application.

As you could have noted above, we split up our pipeline as much as possible. This allowed us to parallelise huge chunks of it. This meant we could run this extensive pipeline with thirty jobs in a short timeframe of around five minutes. Some jobs did, however, rely on others. An example of this is the Scala 2.12 end-to-end run with all command line dependencies, which depends on two other tasks: the packaging of the end-to-end JAR and the building of the Spark image that uses Scala 2.12 with the command line dependency `Dockerfile` layered on top. These dependencies and their chains become quite complex for pipelines as extensive as ours. These dependency chains can be captured as a DAG, as can be seen in Figure F.1, which can help with understanding the complex workflow of our pipeline.

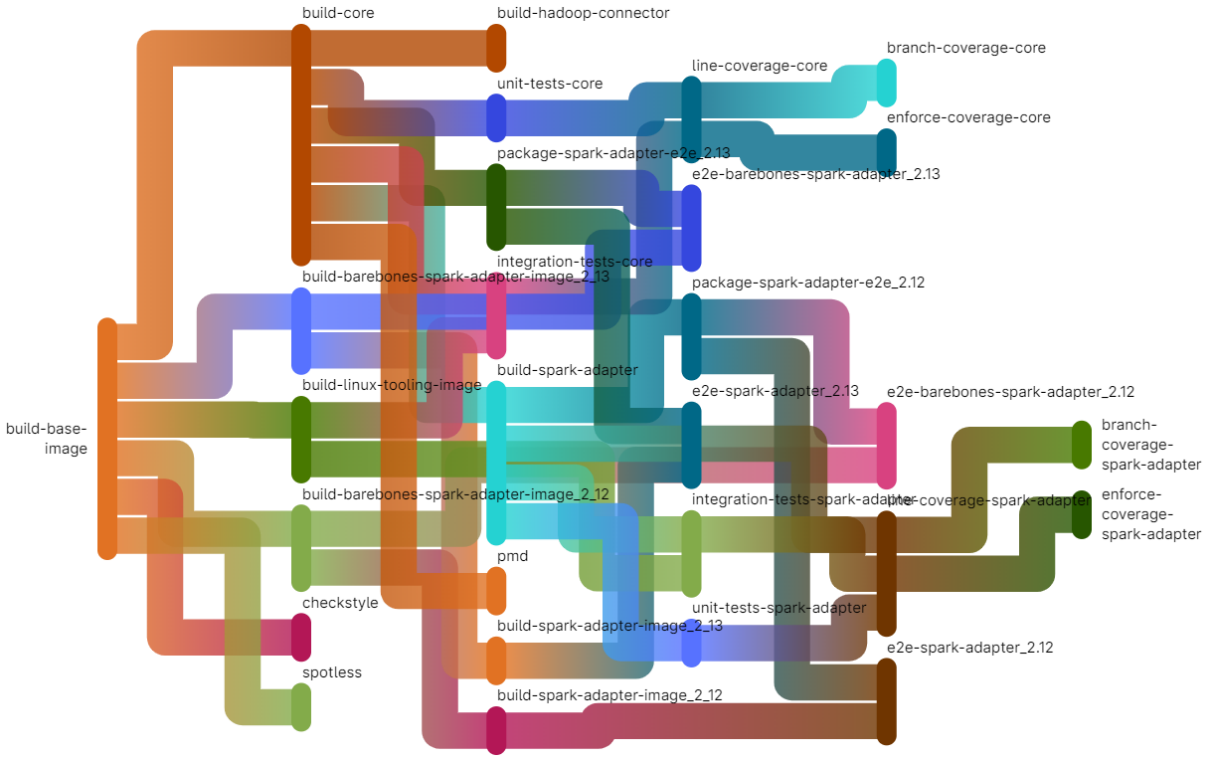


Figure F.1: Chart depicting our GitLab pipeline

[1] https://github.com/palantir/tracing-java/blob/develop/.baseline/checkstyle/checkstyle.xml

Appendix G

ChatGPT Usage

When writing a technical report, LLMs[1] are a powerful aid. They do, however, need to be handled with care. This comes as they oftentimes hallucinate, meaning they make up information on the spot [37]. Not accounting for this behaviour would seriously harm the credibility and integrity of the report. Next to this, they also have the tendency to copy the text verbatim from other sources without any citations, which is an ethical concern [38]. To mitigate any issues that can arise from this, we have made sure to only use the output of these models to reword various sections of our report. Because of the issues with respect to both the reliability and the ethics involved, we did not take any content-wise suggestions from said tools. Furthermore, all the suggestions that were given by these models were proof-read so that we could still ensure that the final piece of text was coherent.

The actions taken ensure that the usage of LLMs in our technical report strikes a balance between their beneficial capabilities and the potential risks they pose. Furthermore, the decision not to incorporate content-wise suggestions from LLMs reflects the importance of maintaining the reliability and accuracy of the information presented. Through thorough proofreading, we guarantee that the final text maintains its coherence and readability, enabling us to leverage the strengths of LLMs.

Delft, May 7, 2025

Atour Mousavi Gourabi, Lohithsai Yadala Chanchu, Henry Page, Pil Kyu Cho, and Tianchen Qu

[1] Large language models (e.g. ChatGPT, LLaMA)