EECS 4314 - Advanced Software Engineering

Apache Hadoop Concrete Architecture

Kimberly Bonilla (218091306)
Pawanjot Randhawa (219732718)
Kennie Oraka (219163104)
Aishwarya Narayanan (219350248)

York University, Lassonde School of Engineering
November 2025

Abstract

This report explores the concrete architecture of the MapReduce subsystem in Apache Hadoop, analyzed using *SciTools Understand*. The goal was to uncover the actual code structure, internal dependencies, architectural styles, and design patterns implemented in Hadoop 3.4.2. We then went further to highlight how MapReduce works alongside YARN (for resource management) and HDFS (for data storage).

The results show that MapReduce operates through a combination of Layered, Client-Server, and Master-Slave architectural styles to enable parallel processing, fault tolerance, and resource coordination across nodes. Key subsystems such as JobClient, ApplicationMaster, Mapper, Reducer, and ShuffleHandler collaborate to manage job submission, scheduling, and data movement between YARN and HDFS.

Beyond the structural analysis, the report discusses concurrency mechanisms that allow tasks to execute in parallel, presents a use case illustrating the full job lifecycle (from submission to output), and finally, reflects on lessons learned throughout the architecture extraction process. Together, these findings help connect Hadoop's high-level design ideas with how they actually work in code, thus showing how its architecture actually takes shape in practice.

Table of Contents

1) Introduction

1.1) Overview:

Hadoop is a Java-based open source framework that manages the storage and processing of big data by utilizing distributed storage and parallel processing [3]. Its architecture is built around three main subsystems: HDFS for storage, YARN for resource management, and MapReduce for computation. This report focuses on the MapReduce subsystem, which forms the core of Hadoop's data processing layer. MapReduce enables parallel processing by dividing data into smaller chunks that are processed independently across multiple nodes.

1.2) Extraction/Derivation Process:

The extraction process was carried out using *SciTools Understand* on the Hadoop 3.4.2 source code to identify the concrete architecture of the MapReduce subsystem. The first step was to import the full Hadoop source into Understand and analyze the directory structure. From this, a custom architecture model named ApacheHadoop_Concrete was created (See figure 1), containing only five relevant top-level modules: hadoop-common-project, hadoop-hdfs-project, hadoop-mapreduce-project, hadoop-yarn-project, and hadoop-tools.

Each module was reviewed to see whether it directly supports MapReduce or interacts with it. Parts of the system that aren't involved in processing (like authentication, native clients, or NFS-related packages) were left out so the focus stays on the components that handle data processing, resource management, and storage operations.

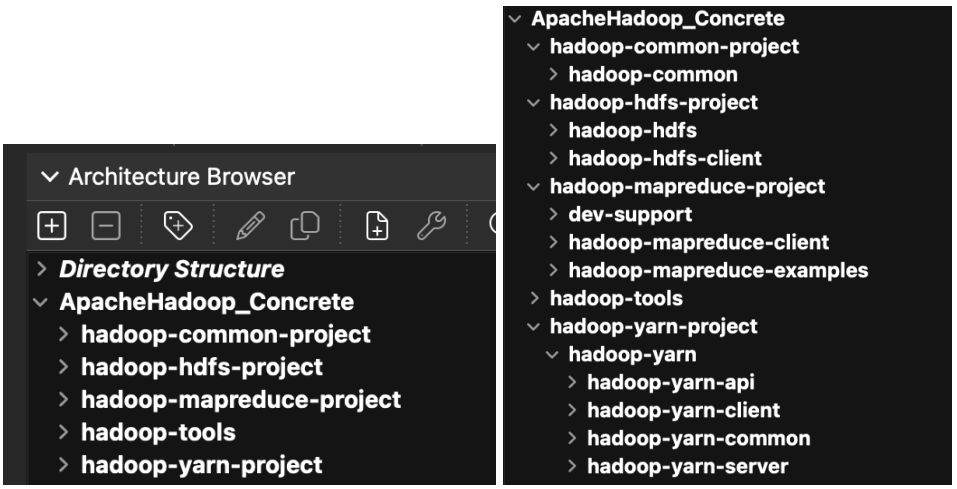

Figure 1: Apache Hadoop Concrete Architecture directory grouping

2) Concrete Architecture - System Overview

2.1) Apache Hadoop Concrete Architecture and its top-level subsystem interactions:

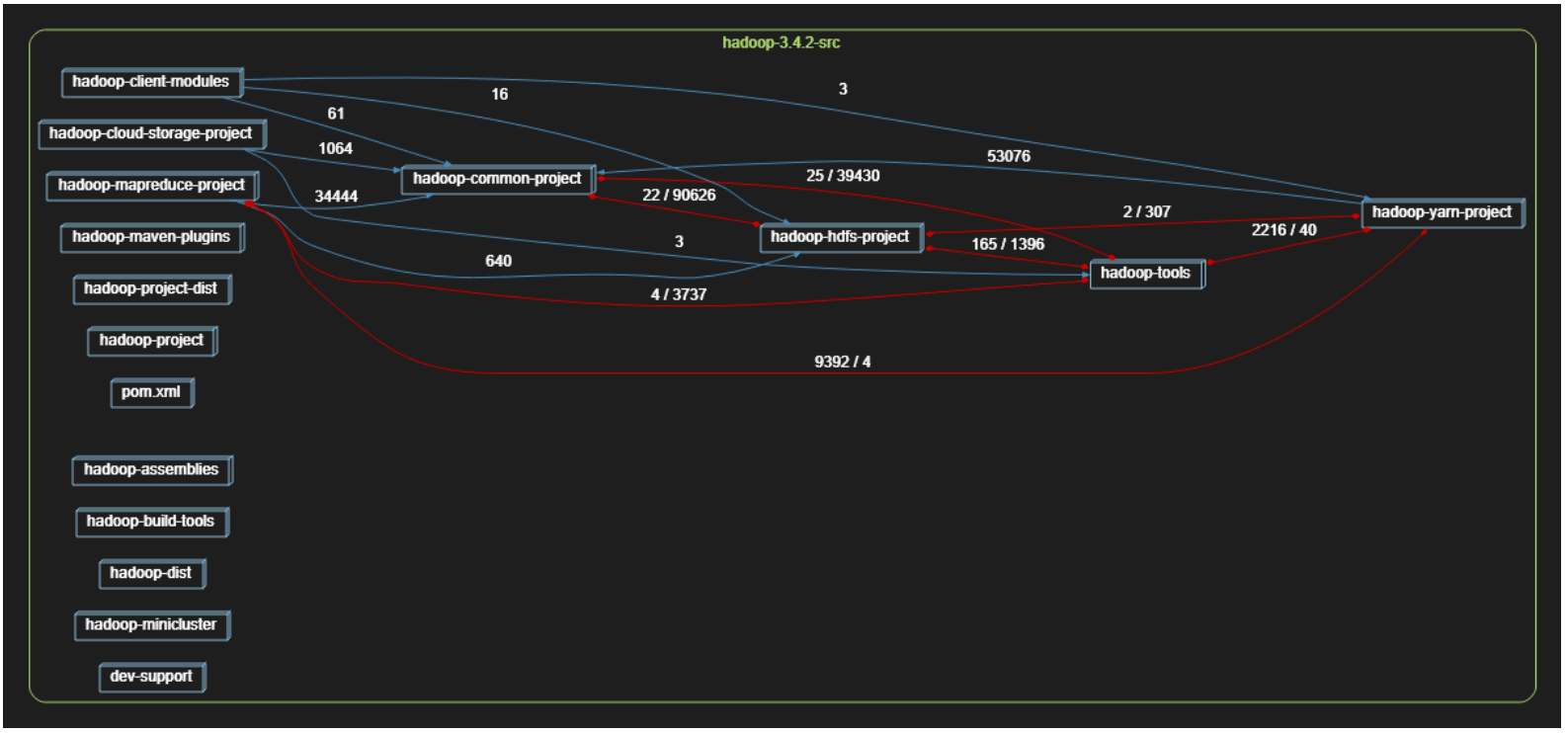

Figure 2: Top-level Subsystem Interactions in Apache Hadoop

Figure 2 shows how the main subsystems inside Apache Hadoop 3.4.2 depend on each other in the actual source code. Each box is a top-level module, and the arrows trace who uses what. The blue arrows show normal dependencies (when one part of the system uses code or functions from another). The red arrows show the reverse (when other modules depend on it or when there's a two-way link between them).

The hadoop-common-project provides the shared utilities that nearly everything else relies on. hadoop-hdfs-project builds directly on hadoop-common(manages how data is stored and accessed). Then, hadoop-mapreduce-project performs the actual data processing and uses hadoop-hdfs, because it needs file-system access for input and output. Similarly, hadoop-yarn-project (the cluster resource manager) depends on hadoop-common and interacts with both HDFS and MapReduce. Some MapReduce components call into YARN, and certain YARN modules still reference MapReduce code.

The numerical labels (for example, “34444” or “22 / 90626”) indicate the count of calls or dependencies between the components (so higher-number links mark heavier coupling).

2.2) Concrete vs Conceptual Architecture:

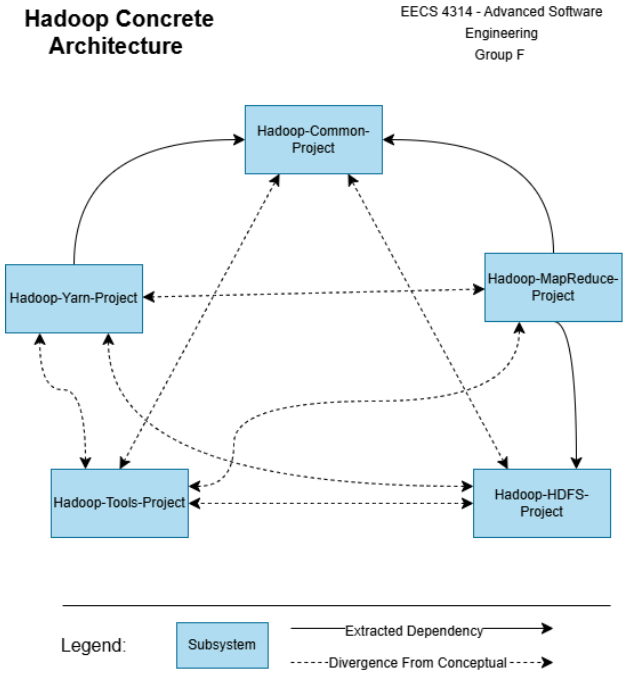

Figure 3: Hadoop Concrete Architecture

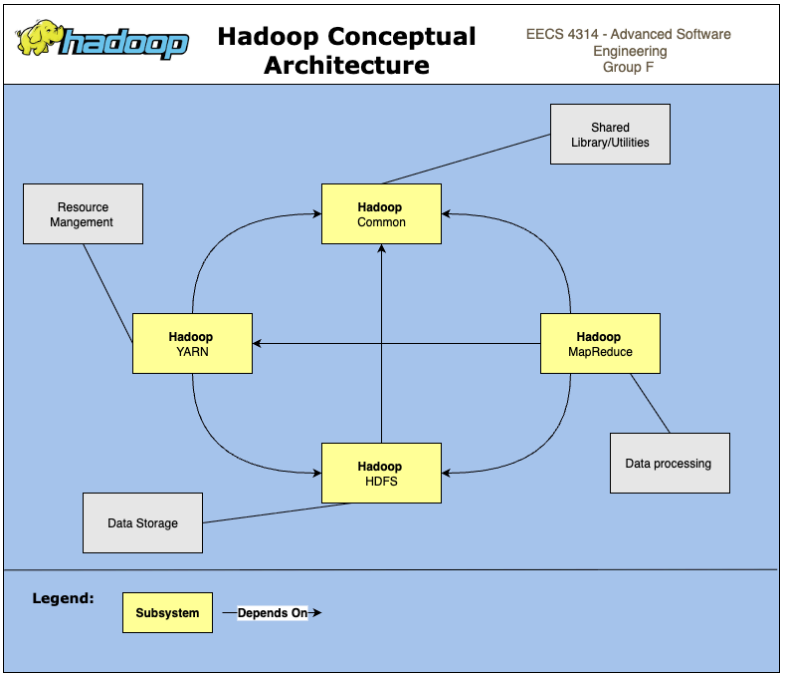

Figure 4: Hadoop Conceptual Architecture (Assignment 1)

Figure 4 presents the conceptual architecture developed by the group in Assignment 1 (Hadoop Conceptual Architecture), while Figure 3 shows the concrete architecture derived from the source code analysis using *Understand*. To improve the clarity, the architecture was redrawn based on the top-level subsystem interactions in Apache Hadoop from section 2.1, with a focus on the five most relevant top-level modules. Dashed lines were added to highlight the differences between the conceptual and concrete model, demonstrating the divergences that are further investigated in later sections.

One major difference between both architectures is the addition of a new subsystem Hadoop-Tools-Project, which contains a collection of utility modules/functionalities that other

subsystems use. It is a mutual dependency with all the other systems because it may also rely on functionalities provided by other components. A mutual dependency was also discovered between HDFS and both Common and YARN, which was not originally seen in the conceptual design. The reason is that HDFS interacts with data access applications managed by YARN [1]. Another discrepancy was found between YARN and Mapreduce. Initially, it was assumed that YARN did not depend on Map Reduce, however the analysis revealed that YARN needs mapreduce to use its resources it manages and monitor the progress to properly coordinate the execution. Lastly , all dependencies that involve Hadoop-Tolls-Project are dashed because this subsystem was not present in the conceptual architecture.

3) MapReduce Internal Subsystems & Interaction

3.1) Description of Subsystems:

Figure 5: Top-Level MapReduce Subsystems

Figure 5 illustrates the internal subsystems of MapReduce, where the first subsystem is Hadoop-MapReduce-Client, a core component of MapReduce. It contains the client API, job submission logic, and communication with YARN, responsible for tuning, implementing and configuring MapReduce jobs [2]. This subsystem is useful in ensuring jobs are properly submitted to the cluster for execution. The next subsystem is mainly used as a demo, providing example programs such as WordCount, Sort, and Random writer. These belong to Hadoop-MapReduce-Examples, where these ready examples help users understand how to structure and use MapReduce. Hadoop-MapReduce-Lib is the next subsystem, which is a utility/shared library that allows distribution of additional JARs and native libraries to the map and reduce tasks using a designated API. Typical contents of Lib include Java libraries and input/output formats definitions and more that support MapReduce[3]. Hadoop-MapReduce-Dev-support is another subsystem found in the source code under hadoop-mapreduce-examples/dev-support. It contains one file used for developers called findbugs-exclude.xml. This is used to configure FindBugs and help developers in testing their

MapReduce code. The final subsystem, Hadoop-MapReduce-Conf, is part of the configuration subsystem that contains XML configuration files like mapred-site.xml and yarn-site.xml that allow users to define job submission parameters. When a job is submitted, the client has to upload both the job JAR and the configuration files to the system directory on the file-system for execution[2]. This allows MapReduce to execute the jobs according to the specific parameters of each submitted job.

3.2) Interaction of Subsystems in MapReduce:

For the interaction of the MapReduce subsystem, we used Understand to further group the main sub-modules based on what they do and where they appear in the codebase. We ended up with six key subsystems: JobClient, ApplicationMaster, Map Phase, Shuffle & Sort, Reduce Phase, and Input/Output Format.

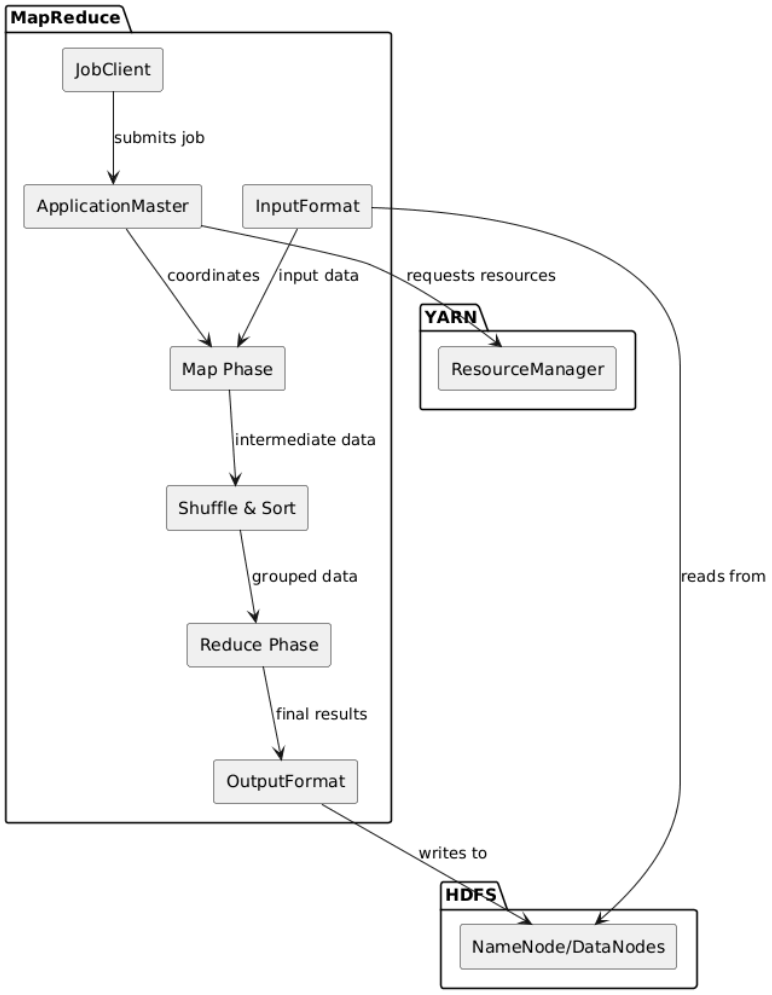

Figure 6: Data Flow for MapReduce

As shown in Figure 6, everything starts when the user submits a job through the JobClient. This component packages the code, input paths, and configuration, then sends a request to YARN's ResourceManager to start an ApplicationMaster (a small controller that manages just that job).

Once the ApplicationMaster is running, it uses InputFormat to figure out how to split the data into chunks that can be processed in parallel. It then asks YARN to allocate containers for the Map tasks, which begin reading their data from HDFS, processing them, and producing intermediate key-value pairs. The Shuffle & Sort phase moves these key-value pairs to the right Reducers, and then the Reduce Phase then aggregates this data and writes the final output back to HDFS through the OutputFormat component. When all tasks finish, the ApplicationMaster updates the JobClient which reports completion to the user.

The dependency graph in Figure 7 also confirms the above information, and how these subsystems fit together in the real code. This setup keeps the system scalable and fault-tolerant, meaning if a task fails, YARN can restart it without affecting the whole job.

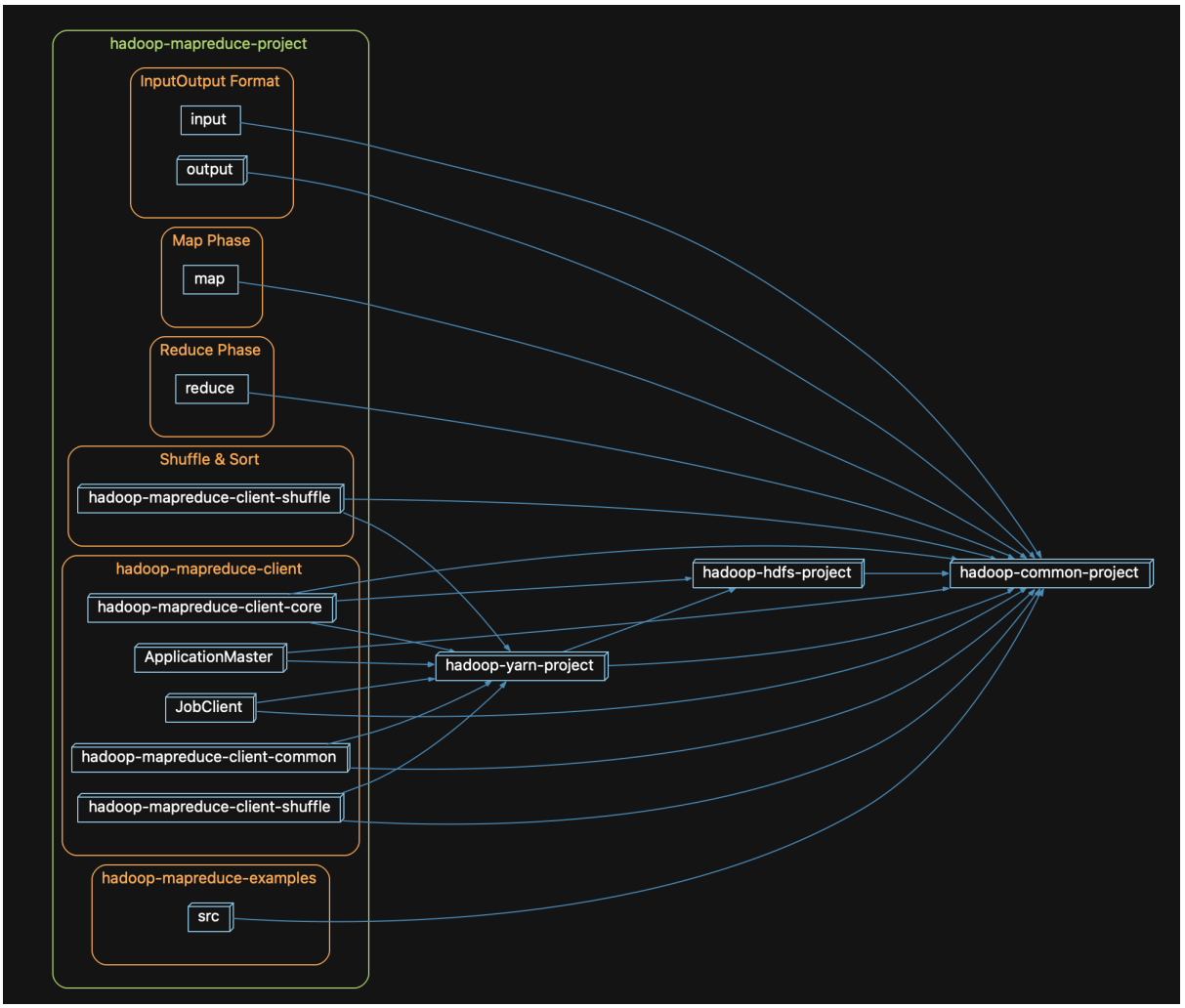

Figure 7: Dependency Diagram for MapReduce Using Understand

4) Architecture Styles

4.1) Layered:

We noticed that Mapreduce followed a layered architecture, in which different components are contained in hierarchical layers that build on the services provided by the layers below

them. At the top, we have a processing layer that contains the core working parts, including the input/output format, mapping phase, shuffle and sort, as well as the reduce phase. This depends on the layers below it, namely the client layer which is responsible for coordinating tasks. This layer contains the client core, the application master, as well as the job client. Below this we have the infrastructure layer, which both of the previous layers depend on in order to function. The infrastructure layer contains YARN, HDF, and Hadoop common, providing scheduling, resource management, and storage solutions.

Figure 8 below showcases this layering by illustrating the ordering of the subsystems and the dependencies between them.

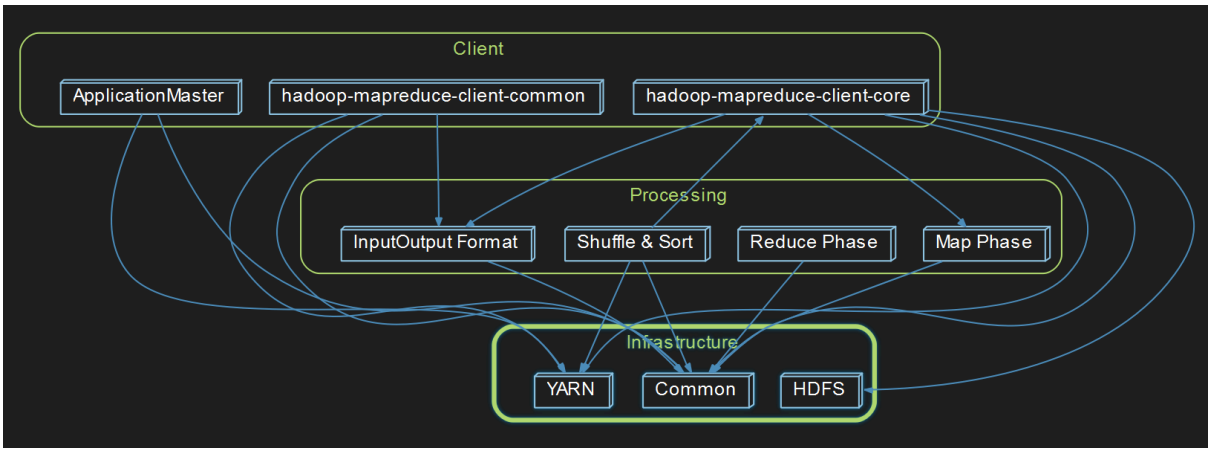

Figure 8: Layered Dependency Diagram for MapReduce Using Understand

4.2) Client-Server:

Figure 9: Client-Server Architecture and screenshot of MRAppMaster.java file

After examining the source code, a Client-Server pattern was also identified within Hadoop, where Hadoop-MapReduce-Client depends on Hadoop-Yarn-Client. This relationship is highlighted in part of the MRClientService.java file (Figure 9), located within the hadoop-mapreduce-client-app subdirectory. The MRClientService file is responsible for managing the communication between MapReduce and YARN Resource Manager. This class imports several YARN APIs and service packages such as yarn.exceptions.YarnExceptions, yarn.webapp.WebApp, yarn.ipc,and YarnRPC. These imports indicate that MRClientService directly interacts with various services of YARN, which shows a concrete dependency in the source code. Architecturally, the dependency demonstrates the Client-Server pattern, where MapReduce acts as the client requesting resources, while YARN acts as the server that handles job scheduling and resource allocation.

4.3) Master-Slave:

Figure 10: Screenshot of MRAppMaster.java file

In the MapReduce architecture, the ApplicationMaster acts as the master. It coordinates the work that happens across the cluster. Each MapTask and ReduceTask then acts as a slave, carrying out the actual computation assigned to it. Figure 10 highlights part of the MRAppMaster.java file in the hadoop-mapreduce-client-app sub level, it shows the key imports that make the master-slave coordination possible.

The ApplicationMaster is responsible for managing this entire process. It communicates with YARN's ResourceManager through the RMContainerAllocator (4), asking for containers to

run the Map and Reduce tasks. Once resources are assigned, the ContainerLauncher (3) starts each task on its respective node. The ApplicationMaster manages the state of every task through Task and TaskAttempt objects (1). Communication and synchronization between the master and slaves are handled using TaskEvent and TaskAttemptEvent (2). On the slave side, the logic for actually executing the tasks is implemented in the MapTask and ReduceTask classes (found in hadoop-mapreduce-client-core). These classes are responsible for reading input splits, processing key-value pairs, and writing output to HDFS.

5) Concurrency

Concurrency is an important aspect of the MapReduce subsystem, promoting efficient processing of large datasets. When a client submits a job, it is first split into smaller map and reduce tasks that can be executed in parallel on different cluster nodes. During the mapping phase, several map tasks run simultaneously, with each task processing a portion of the input data. Reduce tasks wait to execute until all map tasks are complete, ensuring data consistency while still maximizing concurrency [2].

The configuration subsystem determines how many tasks can run concurrently, optimizing resource usage based on the available cluster capacity. The libraries subsystem ensures tasks run efficiently in parallel by handling scheduling, communication between nodes, and execution using threads or processes. MapReduce automatically reruns failed tasks on available nodes, ensuring uninterrupted processing [2].

6) Use Case

A use case of the MapReduce subsystem is a word count tool, which demonstrates how MapReduce can process large amounts of text data by dividing the workload into smaller tasks that run in parallel across the cluster. As shown in Figure 11, the process begins with the input stage, where a text dataset is provided for analysis. During the splitting stage, the text is divided into smaller segments and each is assigned to a separate map task. In the mapping stage, each map task scans its segment and outputs key-value pairs for every word it processes. Next, the shuffling stage groups key-value pairs by word, collecting all occurrences of each key across the cluster. In the reducing stage, reduce tasks sum the counts for each word to calculate the total frequency. Finally, the output stage produces the list of words with their corresponding frequency counts. This use case highlights how MapReduce distributes and coordinates tasks to efficiently handle large datasets [2].

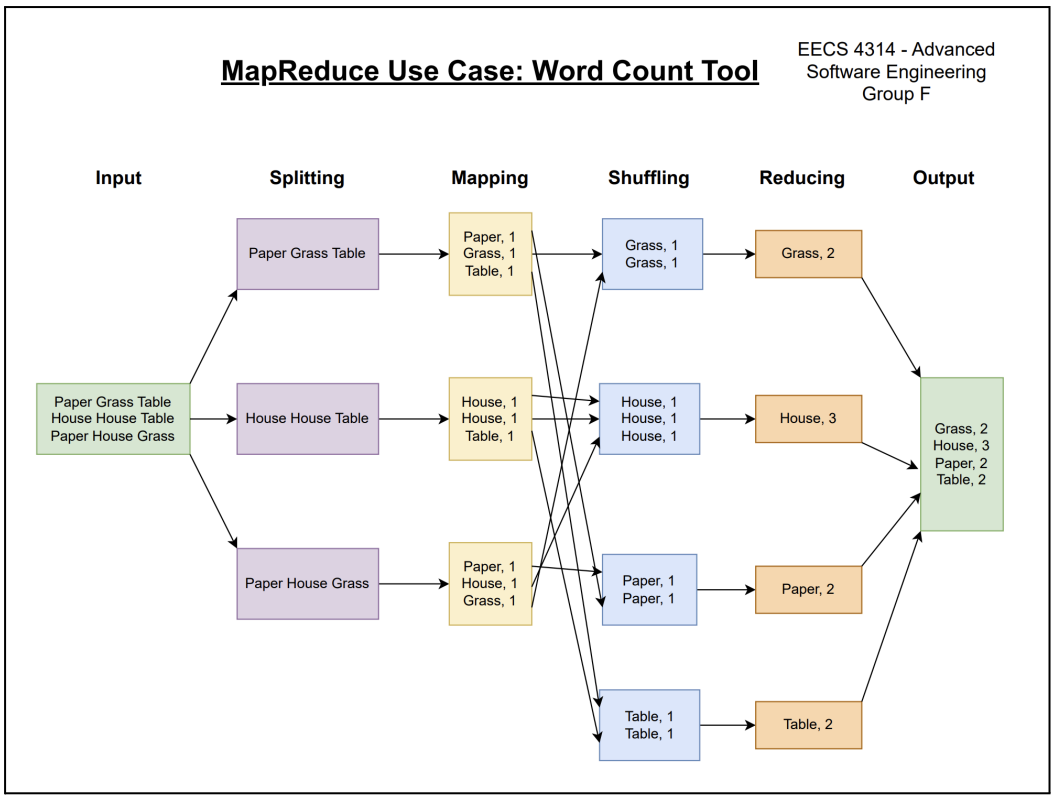

Figure 11: MapReduce data flow for a word count tool

7) Lessons Learned

One of the key things we learnt in this assignment was how different conceptual and concrete architectural views can be. From doing the last report we thought that we had a good understanding of how the various systems would interact with each other, but by using Scitools Understand to create our concrete architecture, many different relations and subdependencies were revealed.

This project also highlighted the importance of tools like understanding, as reading through the source code to figure out how everything connects is difficult when you are unfamiliar with the project. SciTools Understand simplified this process of identifying various connections in the system and because it can be used on any codebase, gaining experience with this tool is a valuable skill that extends beyond the scope of this project.

We also learnt about some of the limitations of this tool, namely the focus on static code dependencies. Dynamic interactions, such as runtime behavior or dependencies created through reflection or configuration were not captured, which limited the completeness of the analysis. While this can be problematic when trying to understand the systems complete behaviour, Scitools Understand still provided a strong foundation for analyzing the Hadoop's overall structure.

Lastly, we learned how larger scale systems implement various architectural styles that we have covered in class. It is one thing to know and understand the styles on a theoretical level, but being able to see these styles working together to achieve a reliable and scalable system gave us a newfound appreciation for what we have learnt.

8) Conclusions

This report looked at the concrete architecture of Apache Hadoop at a high level, and looked at the concrete architecture of MapReduce in more detail. Using SciTools Understand, we were able to extract and visualize the actual code-level structure of both systems, allowing us to derive their concrete architectures directly from the source code. At the top level, we looked at some of the differences between the conceptual and concrete views, while for Mapreduce we explored the interaction in its subsystems and its implementation of various architectural styles. We also examined how concurrency is handled in the system, and looked at a use case to understand real world applications.

Overall, working on this project left us with a deeper understanding of how large scale systems like Hadoop, and their subsystems like Mapreduce are structured and designed. The assignment helped bridge the gap between theoretical concepts and their practical implementations in real software systems.

9) References

[1] A. Lambda, "Understanding basics of HDFS and YARN," *Cloudera Community*, August 30, 2018. [Online]. Available: https://community.cloudera.com/t5/Community-Articles/Understanding-basics-of-HDFS-and-YARN/ta-p/248860#:~:text=When%20that%20quantity%20and%20quality,that%20eluded%20previous%20data%20platforms.

[2] Apache Hadoop, "MapReduce Tutorial," *Apache Software Foundation*, August 20, 2025. [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

[3] Hadoop, "MapReduce Tutorial," *Apache Software Foundation*, May 18, 2022. [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html

[4] Apache Software Foundation, "Apache Hadoop," *Apache Hadoop Project*, [Online]. Available: https://hadoop.apache.org/

[5] Databricks, “Hadoop Ecosystem,” *Databricks Glossary*. [Online]. Available: https://www.databricks.com/glossary/hadoop-ecosyste

10) Appendix

i) Data Dictionary:

Term	Definition
HDFS (Hadoop Distributed File System)	Responsible for distributed data storage across multiple nodes.
NameNode	Master node in HDFS that manages metadata, file structure, and regulates the system
DataNode	Worker node that stores data blocks and performs read/write operations (data processing).
Block	A fixed-size unit of data (typically 128 MB) into which files are divided for storage.
YARN (Yet Another Resource Negotiator)	Resource management subsystem responsible for scheduling and allocating jobs.
MapReduce	Data processing subsystem that divides workloads into map and reduce phases for parallel computation.
JobTracker / TaskTracker	Components of early Hadoop versions that managed MapReduce job coordination and execution.
Hadoop Common	Shared library that provides configuration, logging, and I/O utilities for all Hadoop components.

ii) Naming Conventions:

The naming conventions used in our described architecture follow the normal terminology and abbreviations within the Hadoop framework.

Subsystem Names: Each major component; HDFS (Hadoop Distributed File System), YARN (Yet Another Resource Negotiator), and MR (MapReduce).