EECS 4314 - Advanced Software Engineering

# Apache Hadoop Discrepancy Analysis

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

York University, Lassonde School of Engineering
November 2025

## Abstract

This report performs a discrepancy analysis on Apache Hadoop and its Mapreduce subsystem by comparing conceptual and concrete architecture models from previous assignments. Unexpected discrepancies are identified and investigated in order to understand the rationale behind important design decisions.
Using the reflexion model, the report also reveals the major changes the Mapreduce system had going from versions 1 to 2, as well as the motivation behind these changes, we then propose revisions to the conceptual/concrete architecture to align it with real system behavior, and discuss the limitations of our findings.
Additionally, the report examines concurrency handling, and explores a real world use case through a sequence diagram to better understand the system practical applications. The report concludes with a set of lessons learned about the role of reflexion modeling in understanding the Apache Hadoop system, including insights drawn from our own discrepancy analysis and how these findings inform real-world software architecture practices.

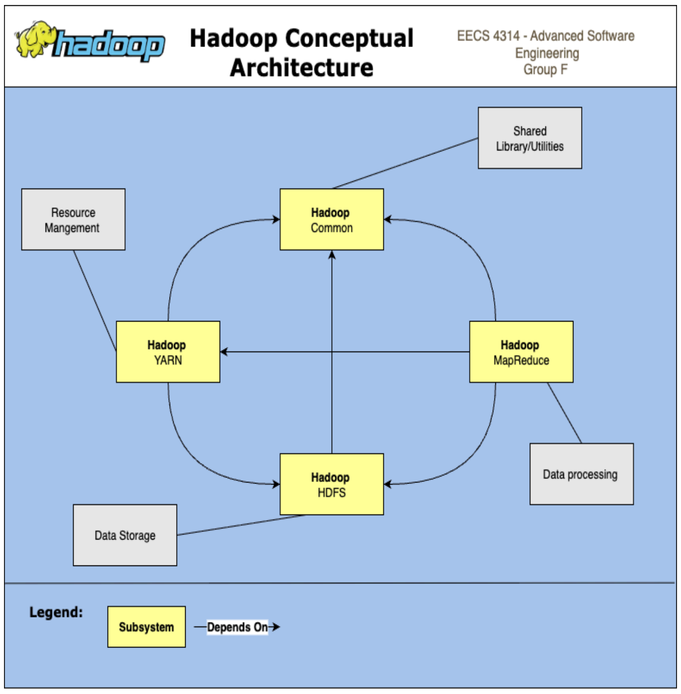
## Table of Contents

## 1) Introduction

This report analyzes the differences between the conceptual architecture of Apache Hadoop (developed in Assignment 1) and the concrete architecture extracted directly from *SciTools Understand* with the Hadoop 3.4.2 source code in Assignment 2. By comparing what we expected to see with what actually exists in Hadoop 3.4.2, we use the reflection model to identify key discrepancies, explain why they occur, and update our conceptual/concrete model accordingly. We focus on top-level subsystem interactions and a deeper MapReduce analysis, outlining how Hadoop evolved from its original MRv1 design to the modern YARN-based MRv2 architecture.

## 2) Top Level Analysis

### 2.1) Conceptual vs Concrete



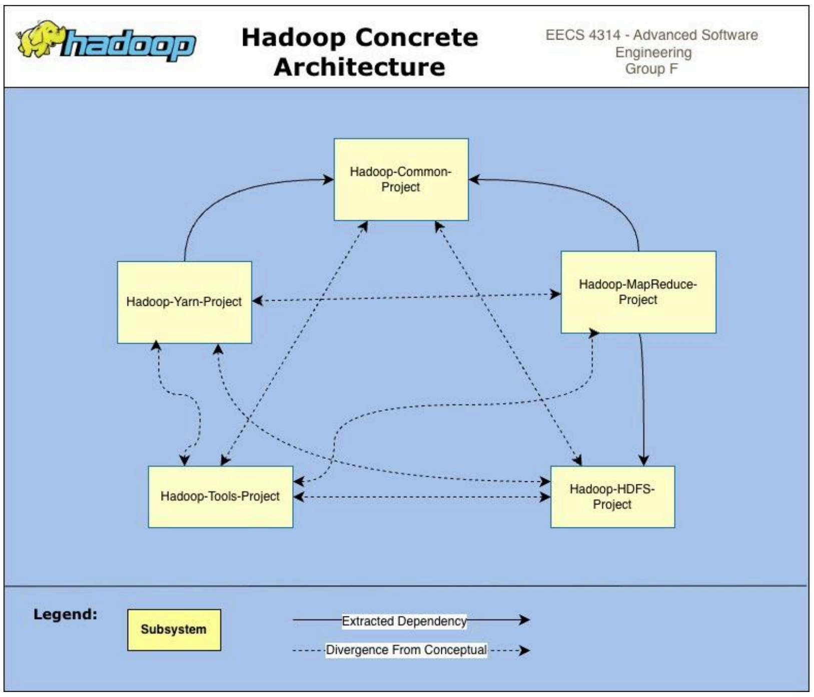


Figure 1: Conceptual Architecture (A1) Figure 2: Concrete Architecture (A2 revised)

This section outlines the differences observed at the top level of the Hadoop system architecture. Figure 1, displays the Conceptual Architecture our group developed in Assignment 1. Figure 2 displays the redrawn Concrete Architecture from Assignment 2 using *Understand*, with the five most relevant top-level modules. The conceptual architecture was derived by understanding Hadoop's documentation while the concrete architecture was obtained by understanding the source code (low-level implementation). In the subsequent sections, a detailed explanation explaining how we came to the redrawn concrete architecture, including the steps taken and the reasoning behind the changes done to the conceptual model. We then discuss the discrepancies found, rationale behind why these changes occur along with the fixes we made to the conceptual model.

## 2.2) Discrepancies

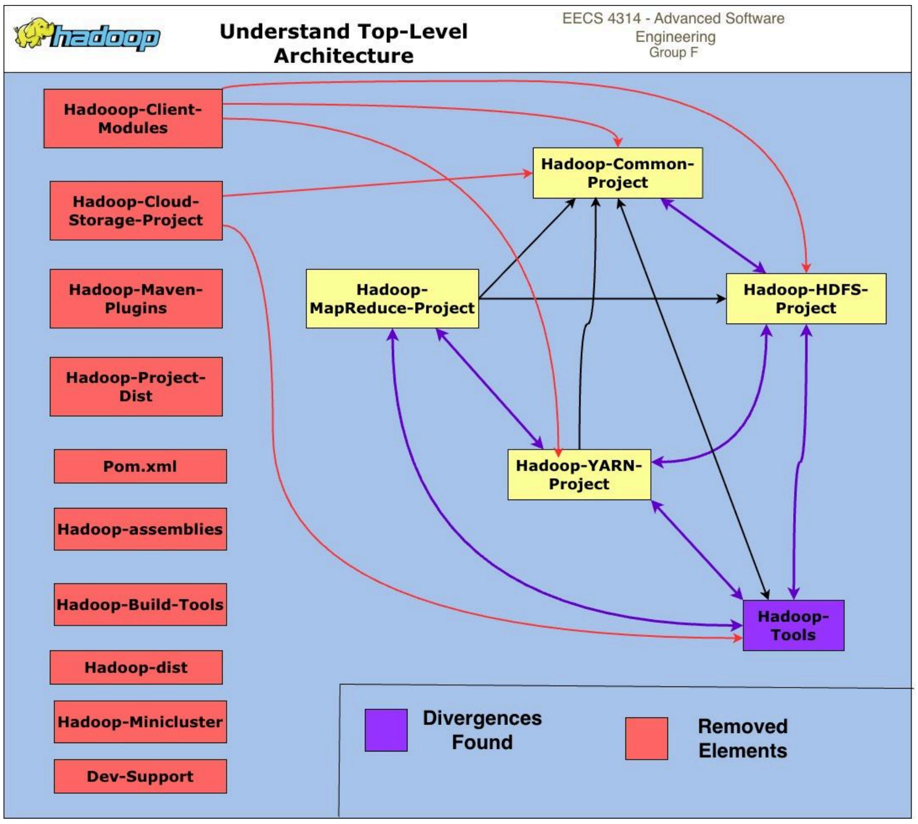


Figure 3: Top-Level Architecture from *Understand*

Figure 3 shows the architecture produced by *Understand.* Redrawing this diagram shows the discrepancies, removals and modifications identified to show what was kept /modified / fixed for our redrawn final Top-level architecture (Figure 2). When comparing our original conceptual architecture (Figure 1) with the model done by *Understand* (Figure 3) there are quite a few differences. Figure 3 was redrawn to highlight the discrepancies in purple arrows/boxes, and removed elements or dependencies in red boxes/arrows.
The most notable discrepancy is structural, the conceptual model is missing the Tools-Project subsystem as shown in purple. Additionally, several new dependencies were identified in the concrete model (purple arrows). For example, instead of one-way dependencies, HDFS has mutual dependencies with YARN and Common, and MapReduce has a mutual dependency with YARN. These interactions indicate stronger relationships that cannot be analysed based on just documentation, which explains why we found no mutual dependencies in the conceptual model. The red elements in Figure 3 are the subsystems and dependencies that we intentionally excluded when constructing Figure 2. Overall, the concrete architecture is more complex, from the added functionalities and relationships.

## 2.3) Rationale

The main reasons for discrepancies between the conceptual and concrete architectures come from how each model is created. Conceptual diagrams are derived just based on documentation which often abstracts functionalities and subsystem interactions that are not obvious, leading to unexpected dependencies.

One example is the Tools-Project subsystem. Although it was absent in the conceptual architecture, it appears in the concrete architecture because it contains a collection of tools that share configuration handling used across subsystems, which is necessary for configuration [2]. Another observation was finding a mutual dependency between HDFS and YARN. This occurs because YARN allows data access applications to interact with HDFS, creating a mutual interaction [1]. Another discrepancy involves the relationship between YARN and MapReduce. YARN manages MapReduce job scheduling and coordination using the ApplicationMaster, executing tasks through NodeManagers and working with the ResourceManager [3]. Lastly , all dependencies that involve Tools-Project were also new since this subsystem was not present in the original architecture.

### 2.4) Fix/Revision

To refine our conceptual architecture, several updates were made to refine the model. First, we added the missing Hadoop-Tools subsystem. We also added the missing dependencies that appeared through the source code analysis to ensure the model accurately reflected the interactions. We also intentionally made exclusions to certain subsystems / dependencies. Subsystems such as build-tools, client-modules, project, pom.xml, assemblies, cloud-storage, maven-plugins, project-dist, minicluster, and dist were not included. Including them would hinder clarity and make it harder to focus on the subsystems in focus. Additionally, we used dashed lines to highlight where the concrete and conceptual model diverged, to keep the model abstract and readable. Overall, Figure 2 shows the concrete architecture our group developed after making all the revisions/fixes.

## 3) Mapreduce Analysis

This section revisits the MapReduce subsystem by comparing our initial conceptual architecture with the extracted concrete architecture from Hadoop 3.4.2. We focus on 2 main aspects (See Appendix iii for further analysis):

- Evolution of the control plane + Layering differences
- Shared task runtime subsystem

### 3.1) Evolution of the control plane + Layering differences

#### 3.1.1 Conceptual Architecture (A1)

In our conceptual architecture, the control flow of MapReduce was modeled using the JobTracker/TaskTracker approach (See figure 4). Under this design, the JobTracker accepts jobs, performs input splitting, assigns tasks to TaskTrackers, and monitors each task's progress. TaskTrackers in turn execute map or reduce tasks and report their status back to the

JobTracker. In MRv1, there was no explicit pipeline, shared runtime, and no representation of where resource management or storage actually happened.

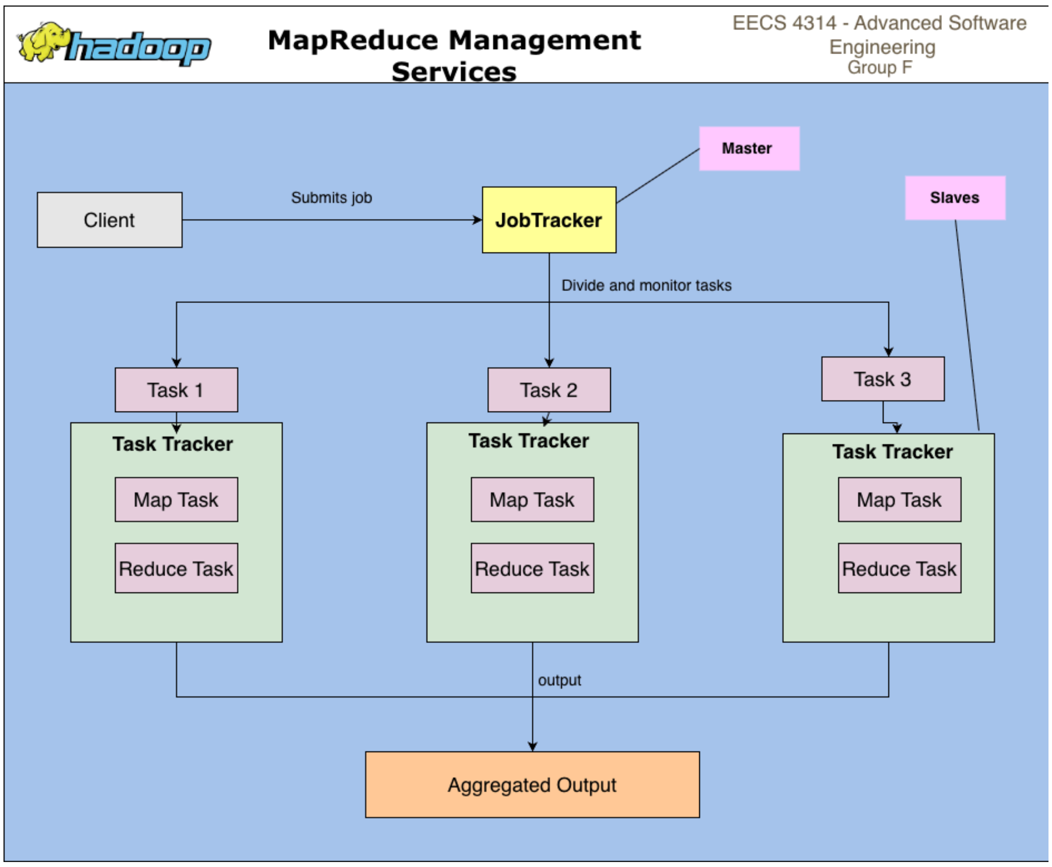


Figure 4: Map Reduce previous conceptual architecture diagram

### 3.1.2 Concrete Architecture (A2)

The architecture extracted in Assignment 2 from Hadoop 3.4.2 revealed two major discrepancies:

a) Real Processing Pipeline: Instead of the JobTracker view, A2 showed MapReduce implemented as a multi-stage processing pipeline, including:

- MapTask
- Shuffle subsystem
- ReduceTask
- Input/OutputFormat, etc

b) YARN/HDFS Based Control and Infrastructure: A2 showed that modern MapReduce does not run under JobTracker/TaskTracker. Instead, it relies on the MRv2 (YARN-based) execution model, introduced in Hadoop 0.23 / Hadoop 2.0 [4, 6]:

- JobClient submits jobs
- ApplicationMaster (MRAppMaster) coordinates each job

- YARN ResourceManager allocates cluster resources & NodeManagers launch containers
- HDFS provides input splits, intermediate storage, and final output

### 3.1.3 Rationale

Hadoop engineers intentionally redesigned MapReduce between 2008-2011 under the project MAPREDUCE-279 (“MapReduce 2.0”) See Appendix iii.i for detailed history. The redesign occurred because the original JobTracker design couldn’t keep up as Hadoop clusters grew. It became a single point of failure because it handled everything (scheduling, monitoring, and resource management). At the same time, Hadoop needed tighter resource isolation and this moved towards a container-based architecture under YARN.

### 3.1.4 Fix/Revision

Since A1 and A2 differs a lot, we update the conceptual architecture:

➢ Adopt the Pipeline Format Revealed in A2 as shown in Figure 5: It shows MapReduce as a pipeline of cooperating subsystems, eg, Input → Map → Shuffle → Reduce → Output

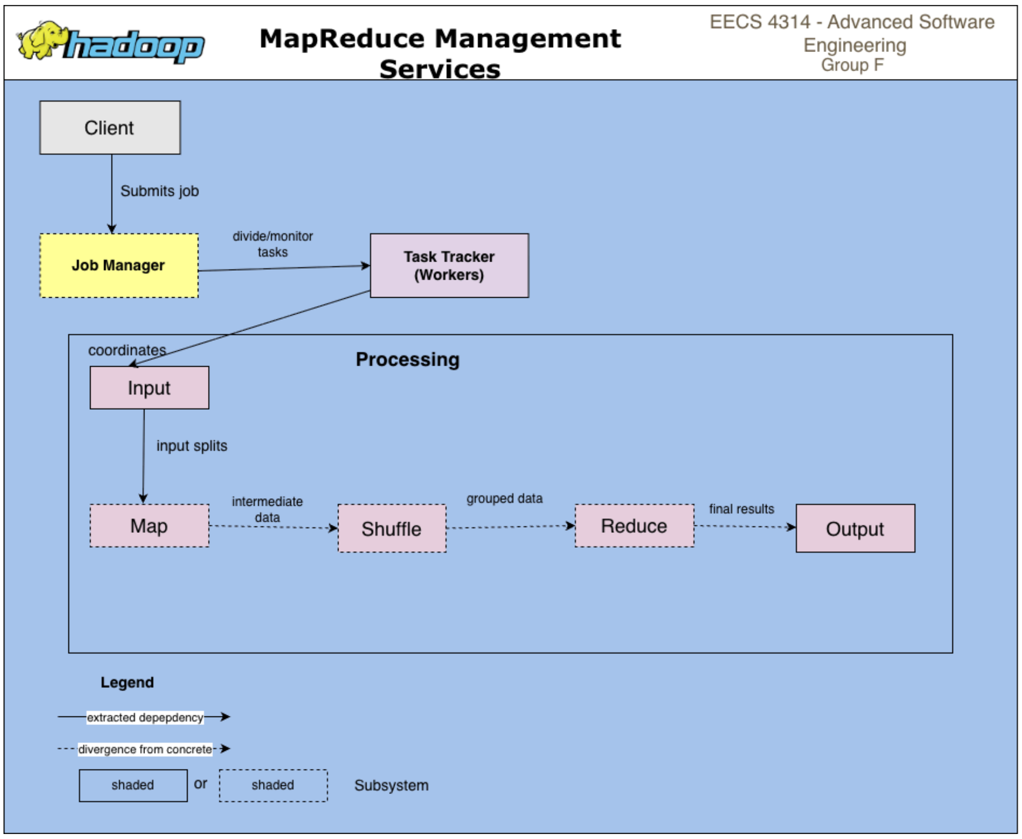


Figure 5: Map Reduce revised conceptual architecture diagram 1

➢ We add Architectural Layers/subsystems as shown in figure 6:
  - Client

- Processing Layer eg Map → Shuffle → Reduce pipeline
- Simplified YARN + HDFS subsystems

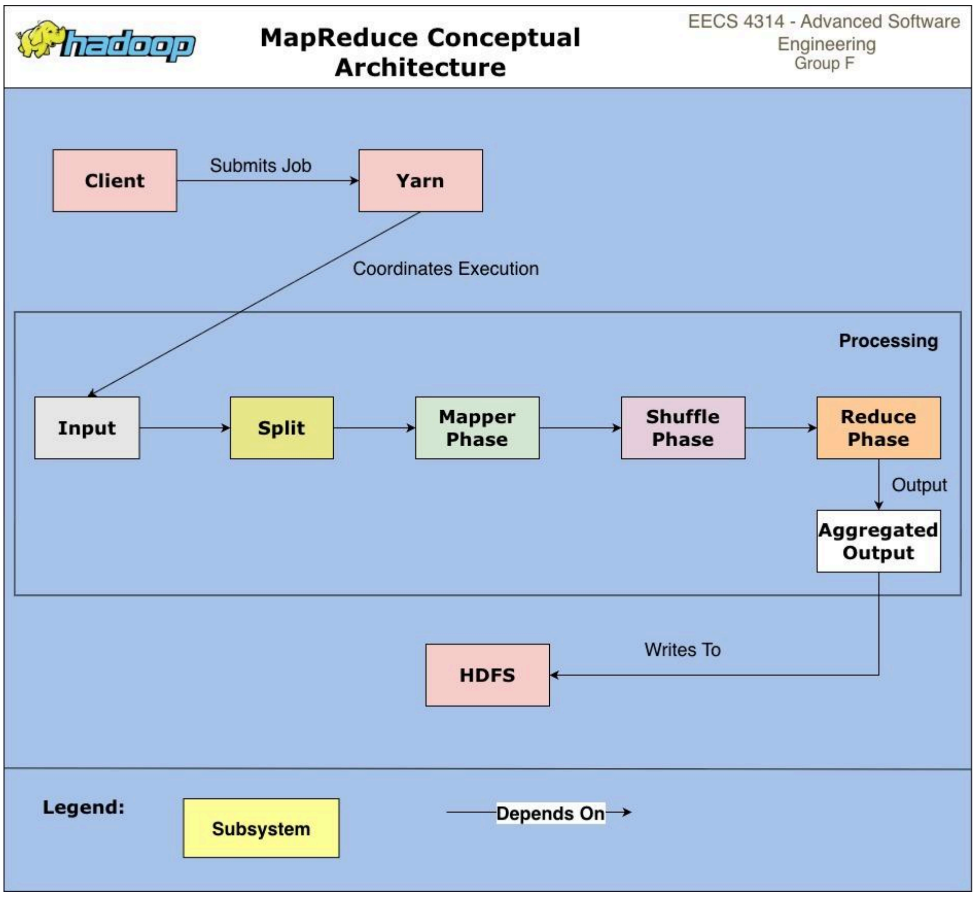


Figure 6: Map Reduce revised conceptual architecture diagram 2

## 3.2) Shared task runtime subsystem

### 3.2.1 Conceptual Architecture (A1)

The A1 model treats Map and Reduce tasks as independent concepts, ie. Each task runs on a TaskTracker, produces data, and reports back. However, there is no mention of a shared runtime layer.

### 3.2.2 Concrete Architecture (A2)

The architecture we extracted in Assignment 2 showed that instead of each task handling its own execution details, Hadoop actually includes a dedicated Task Runtime subsystem. This subsystem is made up of core components like Task, TaskAttempt, TaskEvent, and TaskAttemptEvent, and it manages the work that keeps a distributed job running smoothly. It handles retries when tasks fail, reports progress back to the ApplicationMaster, and manages the coordination with YARN containers.

### 3.2.3 Rationale

Fault tolerance is one of the most complex aspects of distributed computing and the conceptual model leaves this out because showing detailed state machines (as in A2) would overwhelm the high-level explanation of MapReduce. However, in reality, a shared runtime is necessary because failures are common across large clusters, and the concrete architecture centralizes this logic to simplify code and improve reliability [8, 9].

### 3.2.4 Fix/Revision

We update the concrete diagram to introduce a shared execution layer as seen in Figure 7

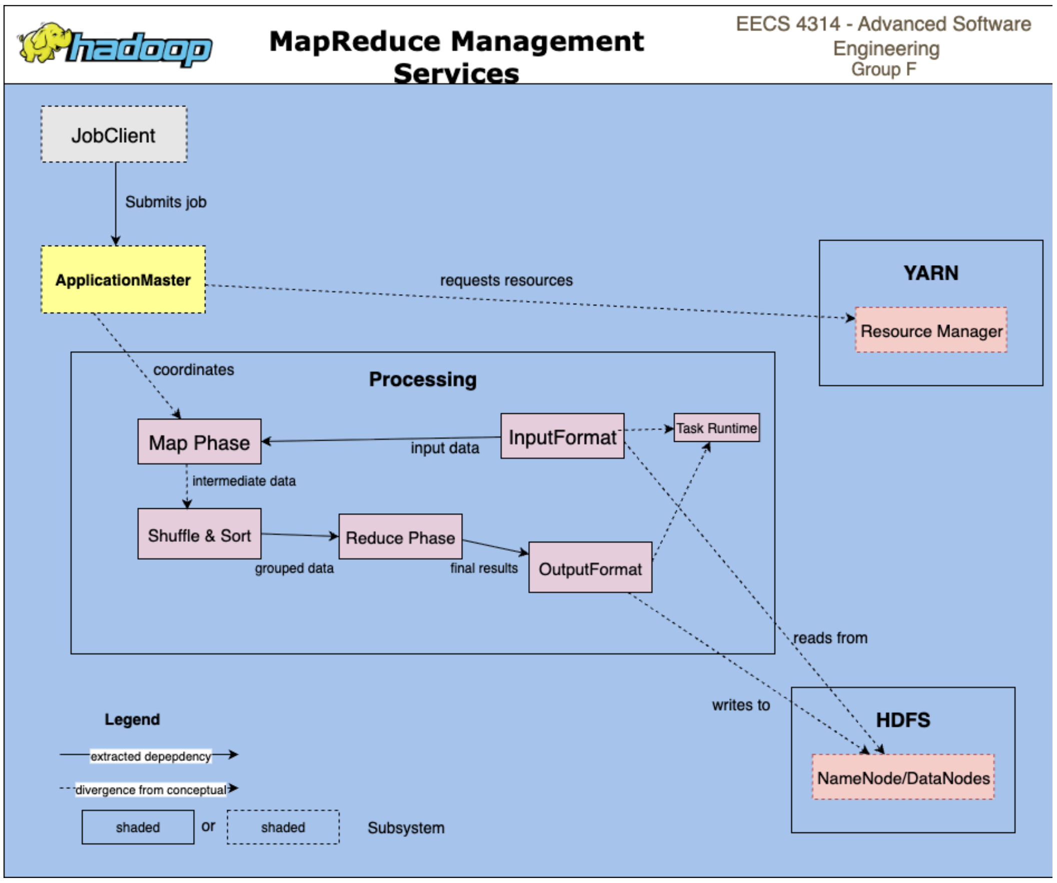


Figure 7: Map Reduce revised concrete architecture diagram

## 4) Sequence Diagram

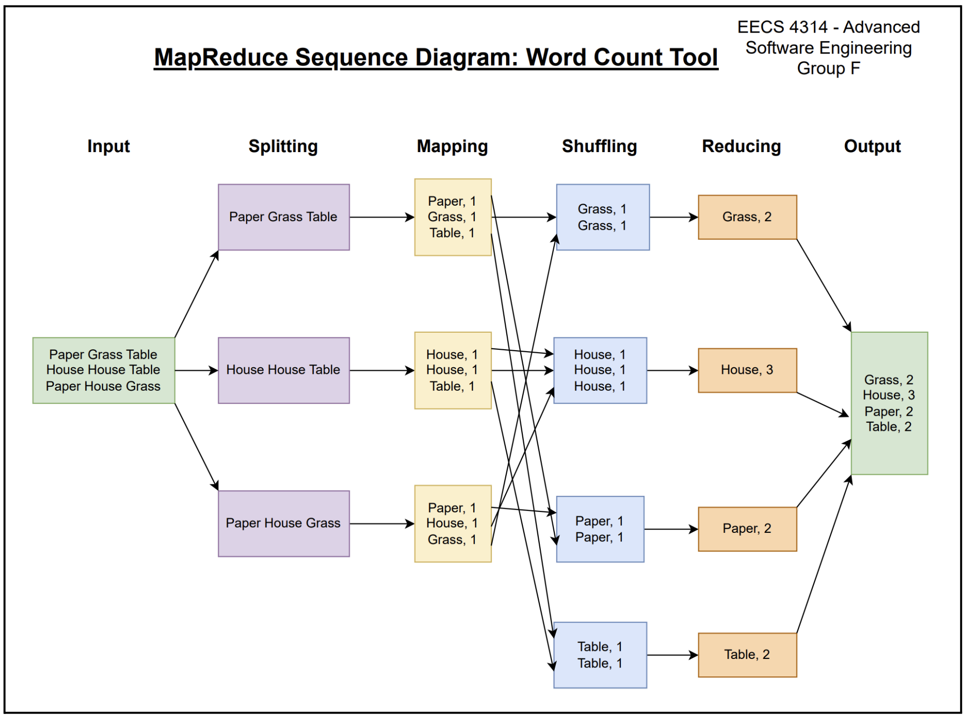

MapReduce Sequence Diagram: Word Count Tool
EECS 4314 - Advanced Software Engineering Group F
Input
Splitting
Mapping
Shuffling
Reducing
Output
Paper Grass Table House House Table Paper House Grass
Paper Grass Table
House House Table
Paper House Grass
Paper, 1 Grass, 1 Table, 1
House, 1 House, 1 Table, 1
Paper, 1 House, 1 Grass, 1
Grass, 1 Grass, 1
House, 1 House, 1 House, 1
Paper, 1 Paper, 1
Table, 1 Table, 1
Grass, 2
House, 3
Paper, 2
Table, 2
Grass, 2 House, 3 Paper, 2 Table, 2

Figure 8: MapReduce sequence diagram for a word count tool

No changes were made to the concrete-level sequence diagram since it already reflects the MapReduce workflow from the conceptual model. As shown in Figure 8, the diagram includes the core MapReduce phases in the same order as the conceptual model. Additionally, it matches the level of abstraction in the conceptual model by focusing on how data moves and transforms through each phase.

## 5) Concurrency and Development Teams

Concurrency influences the MapReduce architecture and reveals some minor differences between the conceptual and concrete architectures. In the conceptual architecture, all map tasks run in parallel across cluster nodes, and reduce tasks start only after all map tasks are complete. Additionally, YARN handles task distribution for MapReduce jobs and allocates cluster resources. In the concrete architecture, MapReduce schedules independent map and reduce tasks to run concurrently across worker nodes, while coordinating with YARN's ResourceManager and NodeManagers. The system also manages the shuffle phase and sorts intermediate outputs before reducers begin, ensuring tasks run correctly in parallel [12].

The MapReduce architecture guides how development teams collaborate and organize their work. The conceptual architecture highlights modularity and clear subsystem boundaries, allowing developer teams to work independently. In the concrete architecture, this modularity is achieved through separate components with defined APIs, which enables parallel development without conflicts. Complex parts of MapReduce, such as the shuffle and sort phase, are isolated as separate modules, allowing specialized teams to maintain or extend these parts without affecting other areas of the system [12].

## 6) Limitations of our findings

Our analysis had a few important limitations, the first of which was that we focused on the static code structure, and not on any of the runtime behaviour. This is because we primarily relied on the documentation and on source code extraction in order to construct our diagrams, making it so that the dynamic portion of hadoop is left out. While this does not discount the correctness of our diagrams, it instead just provides a static view of the system that ignores the complexities of a dynamic runtime environment.

Another key limitation was for the Mapreduce analysis, we left out features that weren't directly related to core Mapreduce logic such as security and cloud tools. We did this in order to maintain a manageable scope for our assignment, as the entire Mapreduce system is very complex. This results in our architectural view being more focused on the Mapreduce core logic, rather than the entire deployment ready Mapreduce we see in use.

Lastly, the use of diagrams that are based on static code results in the relations between YARN and HDFS being heavily simplified. As both YARN and HDFS are dynamic systems that have complex real time communication, their behaviour can not be accurately depicted in our diagrams. This simply means that although our diagram is good at showcasing the core Mapreduce logic, it provides an abstracted view of how resources and data are managed within the system.

## 7) Lessons Learned

One of the most important things we learnt in this assignment is that software architecture is dynamic. Due to our initial conceptual architecture being based on older ideas in Mapreduce v1, we initially saw very drastic changes after extracting the concrete architecture as it was based on Mapreduce v2. We then had to make another conceptual model based on the v2 model, however the entire process taught us a lot about how software adapts to address different needs. The changes from v1 to v2 were largely due to scalability requirements, reflecting the importance of the need to iterate on the design process in order to meet new demands.

Performing the actual reflection analysis itself was also very valuable, as we learned a few key lessons. We noticed that divergences were much more common than absences, a trend that matches what we learned in class. This made a lot of sense after performing the analysis, as the conceptual model in general abstracts away many of the details in order to get a high level view of the system. We also learnt that the divergences themselves were often not mistakes, but were instead necessary adaptations to the code that allowed the system to function as expected.

Lastly, the whole process as a whole taught us a lot about conceptual architecture vs concrete architecture. We learnt that conceptual architecture focuses on clarity and is helpful when trying to understand how a system actually works at the high level. The main drawback is the abstraction of detail this results in, but this is exactly where the concrete architecture is more helpful, as they expose the real dependencies and interactions that are required in order for the system to function. This is great at helping you understand the implementation of the high level architecture, and is an important part in being able to learn a system.

## 8) Conclusions

This report performed a discrepancy analysis on Apache Hadoop and Hadoop Mapreduce. We compared the conceptual and concrete diagram made from assignments one and two, and identified the absences and divergences at the top level. By analysing these discrepancies, we were able to learn about the rationale behind the concrete design decisions, allowing us to better understand the system as a whole. We also examined how concurrency is handled, and took another look at a use case by using a sequence diagram in order to understand real world applications.

Overall, performing the reflection analysis left us with a much better understanding of both Hadoop and Mapreduce. Both Hadoop and Mapreduce ended up being much more interconnected than we initially expected, mainly due to the discovery of unexpected dependencies between subsystems that are needed in order for the system to function, which created a whole web of interaction that was not visible at the high level.

While the assignments one and two were really good at helping us understand how the systems work, this part really emphasised the "why", and allowed us to gain a deeper appreciation of decision decisions behind large software applications. Performing all three assignments revealed that software development is a dynamic process, as real world applications have to  change and adapt in order to meet scalability and performance demands.

## 9) References

[1] A. Lambda, "Understanding basics of HDFS and YARN," *Cloudera Community*, Aug. 30, 2018. [Online]. Available: https://community.cloudera.com/t5/Community-Articles/Understanding-basics-of-HDFS-and-YARN/ta-p/248860

[2] The Apache Software Foundation, "Hadoop Common Commands Manual," *Apache Hadoop Documentation*. [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/CommandsManual.html

[3] The Apache Software Foundation, "Hadoop YARN: Architecture," *Apache Hadoop Documentation*. [Online]. Available:
https://hadoop.apache.org/docs/r2.7.1/hadoop-yarn/hadoop-yarn-site/

[4] Apache JIRA, "MAPREDUCE-279: Map-Reduce 2.0," *Apache Issue Tracker*. [Online]. Available: https://issues.apache.org/jira/browse/MAPREDUCE-279

[5] The Apache Software Foundation, "Hadoop 0.23.0 Release Notes," *Apache Hadoop Release Notes*. [Online]. Available:
https://github.com/apache/hadoop/blob/8b564569f18906a11f91ab3f55a467edc36e9adf/hadoop-common-project/hadoop-common/src/site/markdown/release/0.23.0/RELEASENOTES.0.23.0.md

[6] The Apache Software Foundation, "Hadoop 0.23.0 Release Notes (Detailed Version)," *Apache Hadoop Release Notes*. [Online]. Available:
https://github.com/apache/hadoop/blob/8b564569f18906a11f91ab3f55a467edc36e9adf/hadoop-common-project/hadoop-common/src/site/markdown/release/0.23.0/RELEASENOTES.0.23.0.md?plain=1#L834

[7] The Apache Software Foundation, "The Apache Software Foundation Announces Apache Hadoop v2.2.0," *Apache News*. [Online]. Available:
https://news.apache.org/foundation/entry/the_apache_software_foundation_announces48

[8] Apache JIRA, "MAPREDUCE-6492: Shuffle Optimization and Fixes," *Apache Issue Tracker*. [Online]. Available: https://issues.apache.org/jira/browse/MAPREDUCE-6492

[9] Apache JIRA, "MAPREDUCE-5043: Improvements to Input/Output Format Handling," *Apache Issue Tracker*. [Online]. Available:
https://issues.apache.org/jira/browse/MAPREDUCE-5043

[10] Apache JIRA, "MAPREDUCE-4049: Shuffle plugin and generic shuffle service," Apache Issue Tracker. [Online]. Available:
https://issues.apache.org/jira/browse/MAPREDUCE-4049

[11] Maven Central Repository, "org.apache.hadoop:
hadoop-mapreduce-client-shuffle," MVNRepository. [Online]. Available:
https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-mapreduce-client-shuffle

[12] 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

## 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).

### iii) MapReduce Analysis Contd.

#### Hadoop from MRv1 to MRv2 Details:

- The central architectural change explaining our A1 vs A2 discrepancy is documented in Apache JIRA Issue: MAPREDUCE-279 (“MapReduce 2.0”) [4].
  - “The fundamental idea of MRv2 is to split up the two major functionalities of the JobTracker into separate daemons.” Source: Github Hadoop 0.23.0 Release Notes (Line 836) [6]
- Development began January 2008, resolved August 2011 [4]
- Introduced formally in Hadoop 0.23.0, production-ready in Hadoop 2.2.0 (2013)
  - January 2, 2008 - Issue Created [4]

    - 2011 - Hadoop 0.23.0 released (first release including YARN + MRv2) [4, 5]
    - October 2013 - Hadoop 2.2.0 released (first stable production Hadoop 2.x) Announcement [7]
- Led by core Hadoop engineers including Arun C. Murthy, Vinod Kumar Vavilapalli, and others [4]
- Motivated by scalability constraints, fault isolation, and the need for Hadoop to support more than MapReduce [6]

## Other Findings from MapReduce Discrepancy Analysis:

**Shuffle Subsystem + Pluggable Input/OutputFormat**

Conceptual Architecture (A1)

In the conceptual model from Assignment 1, Shuffle was a single internal step between Map and Reduce, and input splitting and output writing were shown as fixed, uniform processes controlled by the JobTracker and executed by TaskTrackers.

Concrete Architecture (A2)

The concrete architecture extracted from Hadoop 3.4.2 shows that Shuffle is implemented as a dedicated subsystem (hadoop-mapreduce-client-shuffle). It operates independently of MapTask and ReduceTask. Second, Hadoop does not use a fixed input or output mechanism. Instead, it relies on pluggable I/O subsystems (InputFormat and OutputFormat) which determine how data is split, parsed, and written. This supports different formats (text, sequence files, logs, database exports) and allows MapReduce to adapt to different workloads.

Rationale

Shuffle is one of the most critical parts of MapReduce, and including its full mechanics in the conceptual diagram would overwhelm the reader. Similarly, pluggable I/O formats do not change the high-level algorithm, so they are often omitted from introductory models. In contrast, the concrete architecture reflects the actual engineering where shuffle requires optimization and fault handling, and pluggable Input/Output formats are very essential for supporting large-scale enterprise data [10, 11].

Fix/Revision

The Shuffle stage is shown as a standalone subsystem just like in Figure 7. Likewise, we show InputFormat and OutputFormat in the diagram.