EECS 4314

Assignment 3

Discrepancy Analysis Report - Apache Flink (v1.17.1)

November 17, 2023

The Debeggars

Alain Ballen - alain612@my.yorku.ca
Arjun Kaura - arjunka@my.yorku.ca
Davyd Zinkiv - dzinkiv@my.yorku.ca
Nilushanth Thiruchelvam - nilut@my.yorku.ca
Peter Lewyckyj - peterlew@my.yorku.ca
Xu Nan Shen - jshenxn@my.yorku.ca

Table of contents

Abstract

This report completes a discrepancy analysis of Apache Flink's conceptual and concrete architectures, as derived in our previous reports using Flink's documentation and dependency extraction software respectively. Using the Software Reflexion framework, we perform a discrepancy analysis of these architectures through a three-step iterative approach. Our investigation includes exploring Confluence documents, JavaDocs, source code analysis, and tracing git pull requests with corresponding Jira tickets. We highlight architectural discrepancies at both the top-level and sub-system level, focusing on the evolution of the JobManager. We discovered divergences related to the Scheduler and SlotPool subsystems, which were part of the JobMaster system in the concrete architecture. The ResourceManager and Dispatcher subsystems also exhibited an unexpected dependency, analyzed as a low-coupling dependency introduced for improved testability, code duplication, and performance. We discuss the influence of the FLIP-6 proposal on the current architecture and the introduction of the "Flink Master" concept in Flink 1.9. The report includes a revised Architecture diagram representing the current state of Apache Flink and the JobManager subsystem. The implications of these discrepancies on specific use cases and the overall system are discussed, along with the limitations of our findings and lessons learned.

1. Analysis Process

In our analysis, we employ the Software Reflexion Framework (Figure 1), following an iterative three-step approach—proposing, comparing, and investigating. Initially, we proposed conceptual and concrete architectures in our first and second reports. Comparing these, we identified absences, convergences, and differences between our initial understanding and the system's actual implementation. To bridge the gaps, we utilized various methods, referencing Flink's Confluence documents for development direction, JavaDoc for base classes and interfaces, and GitHub Blame for code history, pinpointing changes in dependencies through individual PRs. Comments and descriptions provided insight into the rationale behind these changes. For clarification, we consulted blog posts and Stack Overflow answers from experts and Apache Flink committers. After updating diagrams, we iterated through this process, enabling a thorough exploration of Apache Flink's architecture and an understanding of the rationale behind design decisions.

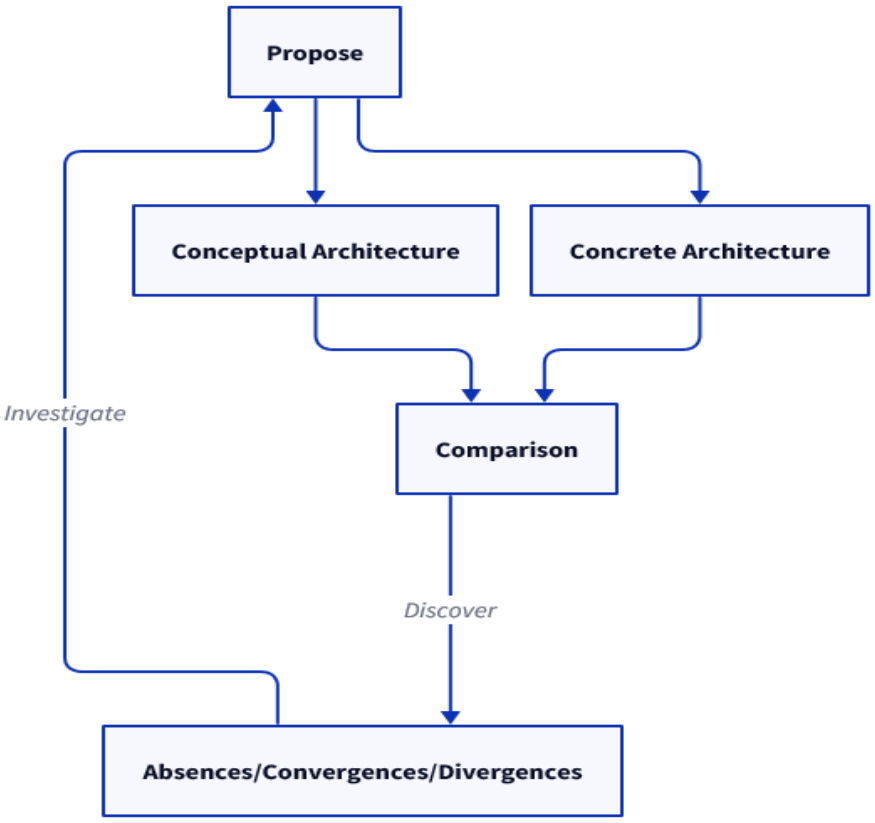

Figure 1: *Software Reflexion Framework used to investigate discrepancies.*

2. Comparing Conceptual and Concrete Architectures

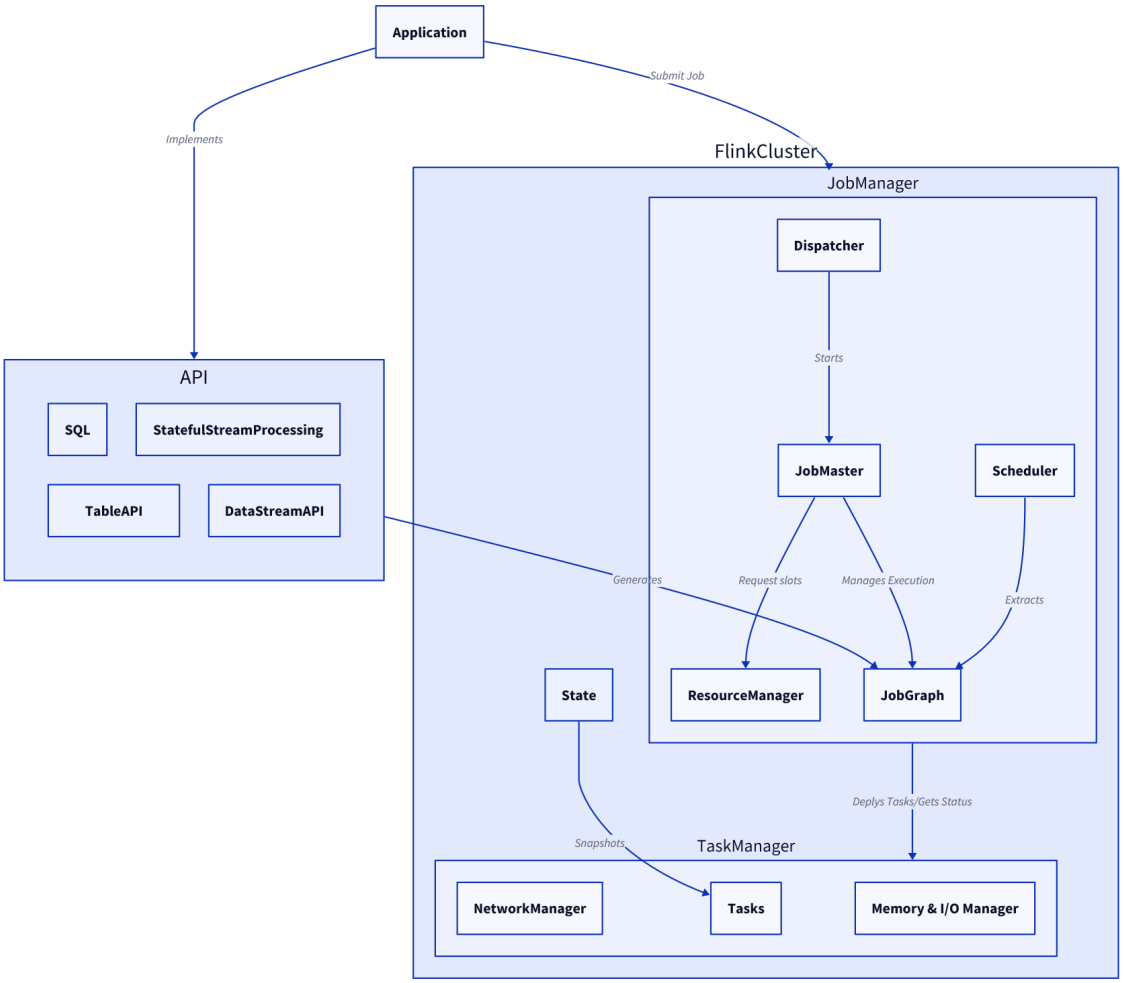

Figure 2: *Conceptual Architecture based on A1 (Full Resolution)*

By comparing the conceptual architecture of Apache Flink that was proposed in the Conceptual Architecture Report (report link), to the Concrete Architecture that was derived (report link) from Flink's source code, we arrived at the reflexion model (Figure 3) that displays the major differences between the conceptual architecture and the implementation in v1.17.1 of Apache Flink.

As displayed in the diagram (Figure 3), there were no absences. The convergences show that the overall flow of Flink's runtime cluster is consistent between the conceptual and concrete architectures. From a top-level subsystem point of view, the flow of Flink application implementing API to submit jobs to the JobManager which in turn deploy the jobs in the form of tasks to a cluster of TaskManagers are implemented accordingly without any unexpected dependencies.

Several divergences were discovered and analyzed to show the main differences in the architectures that are related to the JobManager and its top-level subsystems, which will be discussed in the following sections.

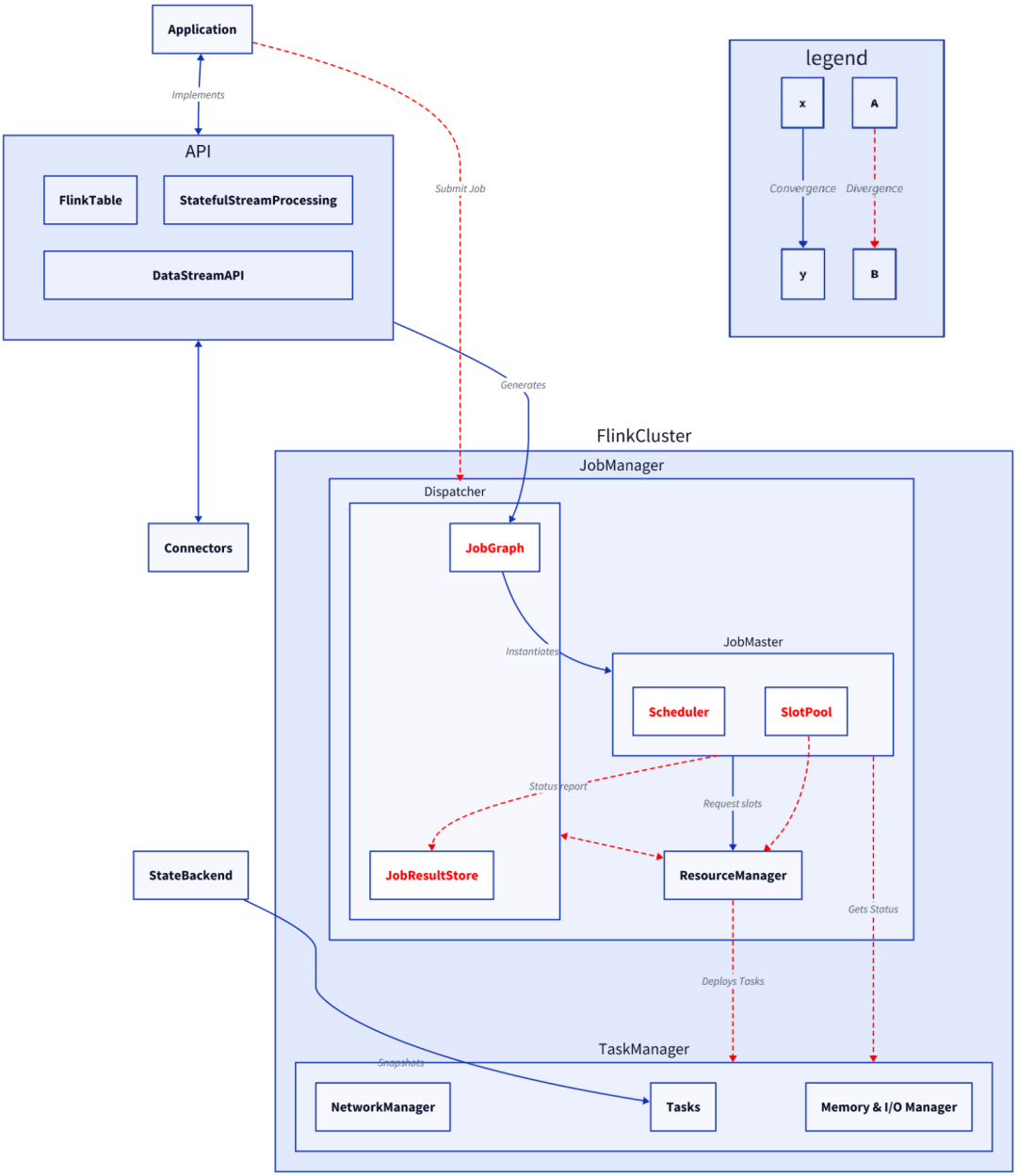

Figure 3: *Reflexion based on A2 Concrete Architecture (Full Resolution)*

3. Divergences

3.1. JobGraph/JobResultStore

As Figure 3 highlights, there are two divergences concerning two subsystems of the dispatcher, JobGraph and JobResultStore. The JobGraph subsystem, consisting mainly of components such as JobGraphWriter and JobGraphStore, is responsible for the storing and processing of received JobGraphs. In the conceptual proposal, it was identified as a top-level subsystem of JobManager, but the concrete recovery shows that it is in fact implemented under the dispatcher.

By looking at the code history of DispatcherLeaderProcess.java[1], we were able to identify the rationale behind the implementation. A change was implemented in October 2019 by Till Rohrmann to address the issue where individual dispatchers lose track of corresponding JobGraph processes. Under the corresponding JIRA ticket[2], Till implements the leader process to encapsulate the running of JobGraph processes along with their corresponding dispatchers, resulting in the architecture that is observed in Figure 3. This dependency strengthens persistence of JobGraph statuses and will likely remain as the implemented architecture in the long-run.

On a similar note, but introduced more recently, the JobResultStore subsystem was discovered as a top-level subsystem under Dispatcher and has dependencies with the JobMaster. Under "Introduce JobResultStore" ticket[3] (Changes by Matthias Pohl, in December 2021), the interface is described to be an in-memory implementation of a process that allows dispatchers to be able to confirm the status of globally terminated tasks. The ticket also states the rationale, that because the component is closely related to the JobGraphWriter, it is to be initialized along with the writer and integrated into the Dispatcher.
Although motivated by bug reports, both divergences and their rationale show the direction of consolidation of JobGraph-related components and processes into the dispatcher process. This architectural choice strengthens the coupling of individual JobGraphs with their dispatcher services, eliminating potential bugs resulting from loss of persistence.

3.2. Scheduler/SlotPool

Another set of related divergences that was discovered as shown in Figure 3 is the scheduler and the slotpool subsystems. In the conceptual proposal, the SlotPool subsystem was not shown and the scheduler is proposed as a top-level subsystem of JobManager. The concrete recovery shows both as subsystems of the JobMaster system.

The new feature epic (Redesign Flink Scheduling)[4], resolved in January 2021, provides insight into the design and the current implementation of scheduler and SlotPool. The scheduler is implemented to consolidate responsibility of scheduling, previously shared across different components, under the jobmaster process. The purpose of this change is to improve performance by limiting the amount of run-time coordination that is necessary between different components. As a result of this change, new dependencies are introduced. The scheduler is initialized under individual JobMasters, and based on the given JobGraph, schedules accordingly and sends requests to the SlotPool.

A sub-task of this epic, extract scheduling-related code from SlotPool[5], outlines the updated responsibilities of SlotPool, which is to obtain, hold and release slots in interaction with the ResourceManager. By looking into JobMaster.java[6], there is a clear flow of creation, usage and maintenance of the scheduler and SlotPool subsystems.

With a clear rationale to improve performance and remove bottlenecks, the dependencies that can be observed in Figure 3 are purposeful implementations to achieve the responsibilities of JobMaster.

3.3. ResourceManager and Dispatcher

In the concrete architectural recovery, the unexpected dependency between Dispatcher and ResourceManager was analyzed by looking at the source code dependencies. From a conceptual level, the two subsystems should not interact with or depend on each other to a high degree as their responsibilities do not intersect with each other.

Based on its JavaDoc[7], DispatcherResourceManagerComponent is a point of dependency that was discovered to be a "Component which starts a Dispatcher, ResourceManager and WebMonitorEndpoint in the same process". From there, "Make ClusterEntrypoint more modular"[8] further confirms the rationale behind this divergence: the component was introduced in September 2018 by Till Rohrmann to implement the Compositional start-up of different components as an improvement for testability, code duplication, performance etc.

We also found additional points of dependencies between ResourceManager and Dispatcher through the Dispatcher's use of ResourceManagerGateway.Java[9]. Similarly, the implementation is to address potential future needs of improved web maintenance of the ResourceManager through existing interfaces of Dispatcher, without any impact on the two subsystems core functionalities and processes.

The rationale and concrete implementation show that this dependency is one of low coupling. The dependencies do not impact each system's performance or other aspects of concern.

3.4 JobManager

The remaining divergences in Figure 3 relating to the JobManager, including direct job submission to the dispatcher (instead of JobManager), and the divergences found amongst ResourceManager, JobMaster and TaskManagers concerning task allocations and tracking, will be addressed in the following section in detail.

4. The Evolution of The JobManager

4.1 JobManager History

The history and evolution of the Flink JobManager can explain some architecture discrepancies and a lot of the inconsistency between the references to a JobManager, JobMaster, and even a Flink Master. This history of renaming and responsibility reassignment creates confusion when it comes to the JobManager especially when

resources discussing Flink are often not caught up with the latest changes (including parts of the Flink documentation itself).

Version 1.0 of Flink contained a JobManager process which itself was responsible for task scheduling, resource management, communicating with TaskMangaers, and more. This was found in a JobManager.scala file which has since been deleted. This was a much simpler architecture where the JobManager only had a few dependencies and interacted directly with the TaskManagers. Despite the JobManager evolving beyond this design, the high-level responsibility of the JobManager component remains the same.

FLIP-6 was a proposal to redesign the JobManager and break it down into smaller components with a better separation of concerns. This would allow a cluster to be more fault-tolerant and configurable. The following are the key changes in the proposal:

- JobManager handles only a single job
- ResourceManager lives across jobs and JobManagers
- ResourceManager must be able to fail without interfering with the execution of current jobs
- TaskManagers are in communication with both the ResourceManager and JobManager
- The dispatcher accepts job submissions from clients.

The proposed changes are very similar to the Flink architecture today which shows how influential this proposal was. The primary difference is that the JobManager in the proposal is still a process which is complemented by the Dispatcher and ResourceManager while in the current implementation, the JobManager process is replaced by a JobMaster.

***Figure 4**: The components of the JobManager as proposed by FLIP-6*

Flink 1.9 introduced the concept of a "Flink Master" which encapsulates the Dispatcher, ResourceManager, and JobManager. This version of Flink is also missing the original JobManager.scala file which indicates that the new implementation of the JobManager is different enough from the original to warrant discarding the old one.

***Figure 5**: The components of the JobManager (in this version, the "Flink Master") as of Flink 1.9*

In Flink 1.11 onwards, the Flink Master was renamed back to the JobManager and the JobManager component became today's JobMaster. This is most similar to the version we see in Flink 1.17.

***Figure 6**: The components of the JobManager as of Flink 1.11 onwards*

4.2 The Current State of The JobManager

Unfortunately, this rapid evolution of the JobManger's architecture means that much of the existing resources on Flink became outdated, or that they simply refer to an older Flink architecture. The Flink 1.0 documentation and Flink 1.17 documentation both use the following diagram to explain the architecture:

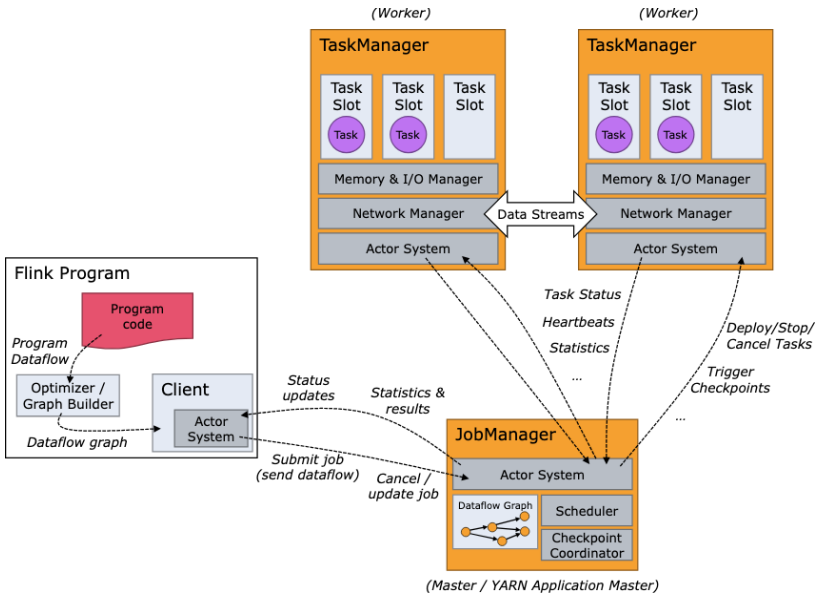

***Figure 7**: Architecture diagram included in both Flink's version 1.0 and 1.17 documentation*

The codebase itself still has inconsistencies with the naming between JobManager and JobMaster. This is the result of the continual iteration of the JobManager which unfortunately can unnecessarily confuse developers who are new to the Flink project.

Although the distinction between these different implementations is clearer now, it can remain confusing for anyone new to Flink, as we were when writing the original report for the conceptual architecture. Despite the responsibilities of the JobManager and most of the components remaining accurate, the distinction between the JobManager and JobMaster and especially the role of the newer JobMaster should be emphasized.

4.3 Revised Component Interaction Diagram

The diagram showcasing the interaction between the JobManager components on the first report is inaccurate due to the JobMaster being labelled as the JobManager. Another change that would help make the JobManager's role more clear is to outline which components are part of the JobManager which are the Dispatcher, ResourceManager, and JobMaster.

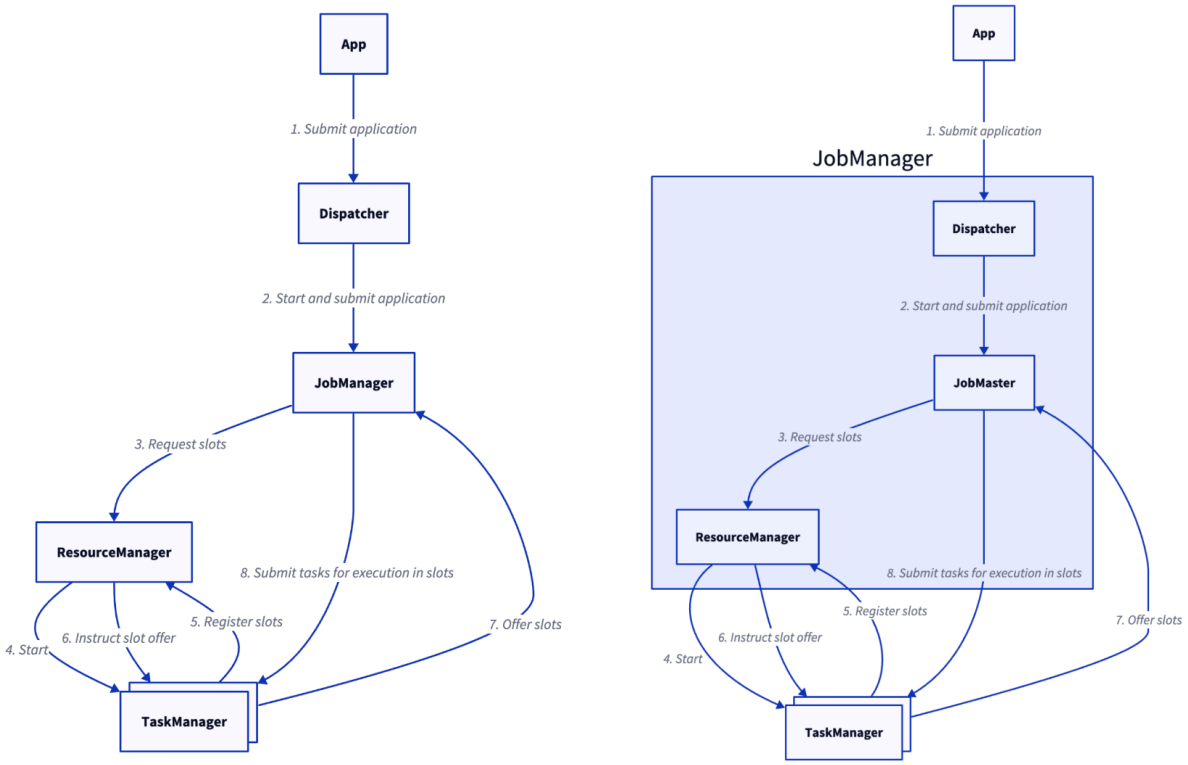

***Figure 8**: Comparison of the Flink JobManager component interaction flow diagram used in the conceptual report (left - Diagram code) and the revised version (right - Diagram code).*

4.4 Final Concrete Architecture Revision

Based on the analysis of the divergences in Section 3 and this section, the revised architecture diagram of Apache Flink is shown in Figure 9.

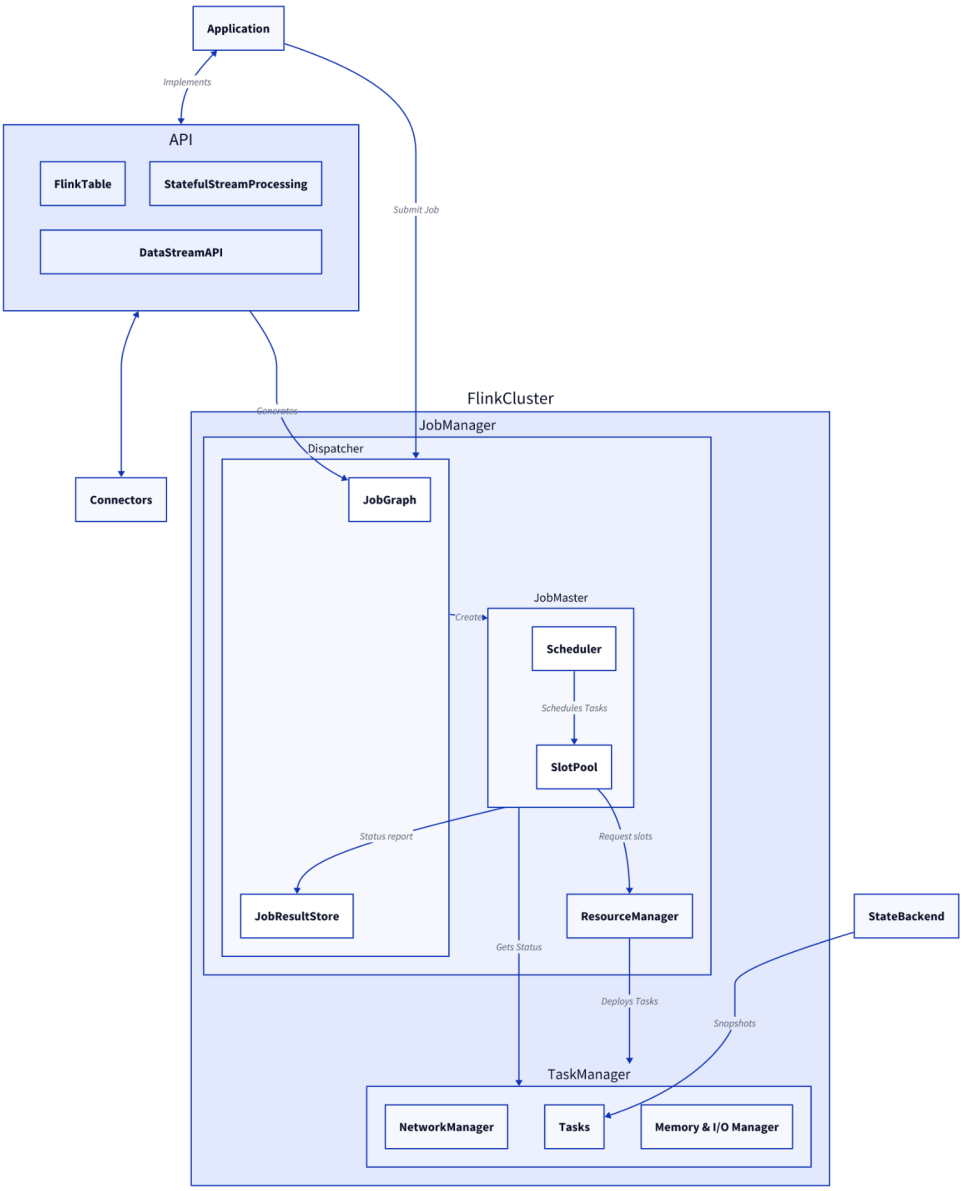

Figure 9: *Architecture Diagram (Post divergence analysis) (Full Resolution)*

5. Use Cases

Upon examination of the previously documented use case, no modifications were made to the identified actors involved and the objective of the use cases. Instead, revisions were made to how each use case interacts with each component of the JobManager. The focal point of these revisions is regarding the discrepancies between the concrete and conceptual architectures.

Use case one as stated previously, is about scaling large amounts of data for ecommerce websites to create a personalized shopping experience for users. Originally stated, the JobManager processes the data and delegates the workload to the TaskManager. However, it is actually the JobMaster that performs this task. Since the JobManager itself is composed of the Dispatcher, ResourceManager, and the JobMaster.

Use case two refers to online payment fraud detection for online banking and financial systems. The revision made to this use case is that the dispatcher submits an application to the Jobmaster instead of the JobManager and will distribute the analyzed tasks to the TaskManager. As already mentioned it is because the JobMaster is the process that performs this task as it is a component of the JobManager.

6. Implications

The release notes and development notes do not provide a full description of relevant changes. This affects how future changes are made and how developers handle changes. Having robust notes means the architecture is usually communicated over easily. It is also implied that the code base itself is more confusing and difficult to use. There is a greater barrier of entry for new developers as a result and it is more difficult to make changes and maintain software as a developer. This contributes to the gap between architecture. Additionally, based on analysis, we found that the JobManager maintains the same responsibilities across the architectures. The overall function remains consistent across based on our analysis.

7. Limitations

Following are the limitations of the completed discrepancy analysis and the resulting findings.

The effectiveness of our analysis and the accuracy of the conclusions that we have drawn at this stage is heavily limited by the quality of developer logs and comments on Github, Jira and/or Confluence. The incompleteness and or vagueness of such details and context contribute to potential inaccuracies and incompleteness of the discrepancies that were discovered, and the reflexion that was completed.

As Apache Flink is an open-source project, the constant changes and evolutions that the source code goes through, whether it is part of an improvement epic, to develop a new feature or to implement bug fixes, means that there are ongoing alterations to the architecture at the version that is under analysis, v1.17.1. These continuous changes necessitate additional human judgement to discern the impact of said changes on the overall architecture, and whether the impacts will persist in future versions of the software.

The size of Apache Flink limits the possible scope of analysis. Given limited time, we focused on the main system of concern of this report, JobManager. Discrepancy analysis in other subsystems is limited.

8. Learned Lessons

- The rationale behind differences between Conceptual architecture and concrete architecture must be understood clearly from multiple perspectives (4 W's). The rationale dictates the scale of impact on the architectural design of the software.
- Difference analysis, investigation and creating new conceptual/concrete architecture diagrams should be a process that is iterated multiple times, especially for large software projects, to present a clearer depiction of the software architecture.
- For large software projects, open-source ones especially, it may be incredibly difficult, if not completely impossible, to adhere to a strict conceptual architecture. Hence, it is important to keep documentation accessible and up-to-date.
- Understanding the general direction of software development is a great first step to difference analysis. Confluence helped us a lot.

Conclusion

Throughout this report we have completed the Apache Flink's discrepancy analysis in great detail through the evolution of the Flink JobManager and its transformation through various versions, highlighting the significant changes from a single JobManager.scala file in version 1.0, to the introduction of ResourceManager and Dispatcher components in later versions.

The complexities and challenges faced in understanding the conceptual and concrete architectures of Apache Flink, emphasizing the importance of clear documentation and rationale in software development are thoroughly examined. This is particularly crucial for large, open-source projects like Apache Flink, where constant changes and a diverse range of contributors can make understanding the system's architecture challenging.

There are several valuable lessons through this analysis made by our team. First, the necessity to understand the rationale behind architectural differences from multiple perspectives. Second, the iterative process required for analyzing and updating architecture diagrams for large software projects. Third, the challenges in adhering to a strict conceptual architecture in open-source projects and the importance of keeping documentation up-to-date. Finally, we recognized the significance of understanding the general direction of software development, which is essential for effective analysis.

In conclusion, our discussion into Apache Flink's architecture and the discrepancies between its conceptual and concrete aspects has provided us with a deeper understanding of the complexities involved in managing large-scale software projects. It's a reminder of the importance of clear communication, documentation, and the adaptability required in the software development field.

References

URLs used within the report:

1. https://github.com/apache/flink/blob/e656e17352f66ab7deef3ad1f10dc8db363f4499/flink-runtime/src/main/java/org/apache/flink/runtime/dispatcher/runner/DispatcherLeaderProcess.java
2. https://issues.apache.org/jira/browse/FLINK-11843
3. https://issues.apache.org/jira/browse/FLINK-25430
4. https://issues.apache.org/jira/browse/FLINK-10429
5. https://issues.apache.org/jira/browse/FLINK-10431
6. https://github.com/apache/flink/blob/e656e17352f66ab7deef3ad1f10dc8db363f4499/flink-runtime/src/main/java/org/apache/flink/runtime/jobmaster/JobMaster.java
7. https://nightlies.apache.org/flink/flink-docs-release-1.17/api/java/org/apache/flink/runtime/entrypoint/component/DispatcherResourceManagerComponent.html
8. https://issues.apache.org/jira/browse/FLINK-10411
9. https://nightlies.apache.org/flink/flink-docs-master/api/java/org/apache/flink/runtime/resourcemanager/ResourceManagerGateway.html

Additional resources used in research for this report:

Zhu, Z. (n.d.). *Flink course series (3): Flink runtime architecture*. Alibaba Cloud. https://www.alibabacloud.com/blog/flink-course-series-3-flink-runtime-architecture_597984

Rohrmann, T. (2017, April 14). *Flinkforward SF 2017: Till rohrmann- Redesigning Flink's distributed architecture*. YouTube. https://www.youtube.com/watch?v=4B1Dd2qYDGQ

Flip-6: Flink deployment and process model - standalone, yarn, Mesos, kubernetes, etc.. FLIP-6: Flink Deployment and Process Model - Standalone, Yarn, Mesos, Kubernetes, etc. - Apache Flink - Apache Software Foundation. (n.d.). https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=65147077