Group Name: BLESSERS

EECS 4314: Project Report A3

Apache Flink: Discrepancy Analysis Report

Authors

Noor Ahamed Sadique

Abeed Arefin

Kasemi Osaroedey

Kingsley Okon

Daoud Ali

Kamsi Idimogu

Table Of Contents

Executive Summary

This report provides a consolidated analysis of the discrepancies between the conceptual and operational architectures of Apache Flink, with a focus on its task manager component. It investigates the architectural discrepancies of Apache Flink's TaskManager, contrasting its conceptual framework with practical implementations. The report aims to provide a detailed understanding of Flink's design evolution for developers, architects, and researchers seeking to comprehend the intricacies of its operational architecture.
Through meticulous analysis of official documentation, developer insights, and a summarization of findings, the report highlights significant enhancements in the TaskManager's data processing capabilities and underscores Flink's adaptability in real-world applications. The study showcases Flink’s operational strengths and suggests areas for ongoing improvement within its architecture.

1 Introduction

This report presents a discrepancy analysis of Apache Flink, an open-source platform for scalable stream and batch data processing. The analysis involves a reflexion method, comparing Apache Flink's proposed conceptual architecture with its actual implemented architecture. This examination is crucial for identifying misalignments that could affect the system's performance, maintainability, or scalability. The report aims to align the conceptual

design with practical implementations, ensuring Flink's robustness and adaptability in the dynamic field of data processing.
The report's focus is on a detailed comparison between Flink's high-level conceptual architecture, which includes design patterns and interactions, and its concrete architecture as realized in the system. Key subsystems like the Core, Runtime, and TaskManager, along with their secondary components, are scrutinized. The analysis documents discrepancies, assesses their impacts, and provides recommendations for reconciling these differences. The ultimate objective is to refine Apache Flink's architecture for enhanced performance and future development.

2 Architectural Overview

In this section, we scrutinize the design and operational structure of Apache Flink, highlighting the theoretical and practical architectures. This analysis will serve as a foundation for understanding the system's architecture as originally planned and its subsequent implementation.

2.1 Conceptual Architecture

The conceptual architecture of Apache Flink, shown in Figure 1, acts as a foundational blueprint for the system's design. It illustrates the key principles and expected interactions among different components, providing a high-level view of the intended workflow and operational dynamics. This representation simplifies the complexities of the actual implementation for easier understanding [1].

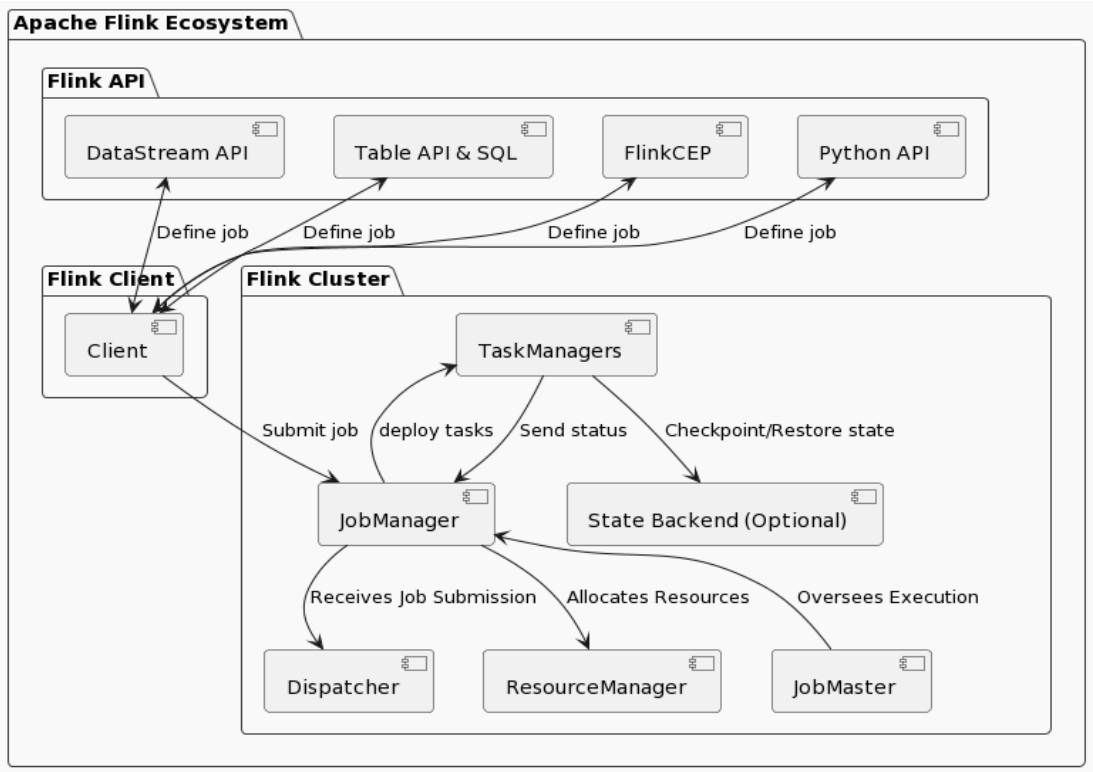

Figure 1: Conceptual Architecture of Apache Flink

Central to this architecture is the Flink API, which functions as the primary interface for users to define and submit jobs. This interface is a critical component, as it encapsulates the system's versatility and ease of use, offering various abstraction layers such as the DataStream/DataSet APIs for different types of data processing tasks. The API's design is intentional, reflecting a user-centric approach that prioritizes accessibility and flexibility in job definition and submission [1].

Beneath the API, the architecture delineates the interactions between the client and the cluster, illustrating a clear pathway for job execution. The JobManager plays a pivotal role within this structure, acting as the conductor that orchestrates the execution of jobs by delegating tasks to the TaskManagers. This delegation is based on a distributed computing model, where the TaskManagers are responsible for the actual execution of tasks, maintaining the state and checkpoints, and ensuring fault tolerance [1].

2.2 Concrete Architecture

The concrete architecture, as showcased in Figure 2, is a manifestation of the system's operational state. It is a detailed visual representation of the various components that make up the entirety of the Apache Flink ecosystem, illustrating not only the planned elements but also those that were added or modified as the system was brought to life.

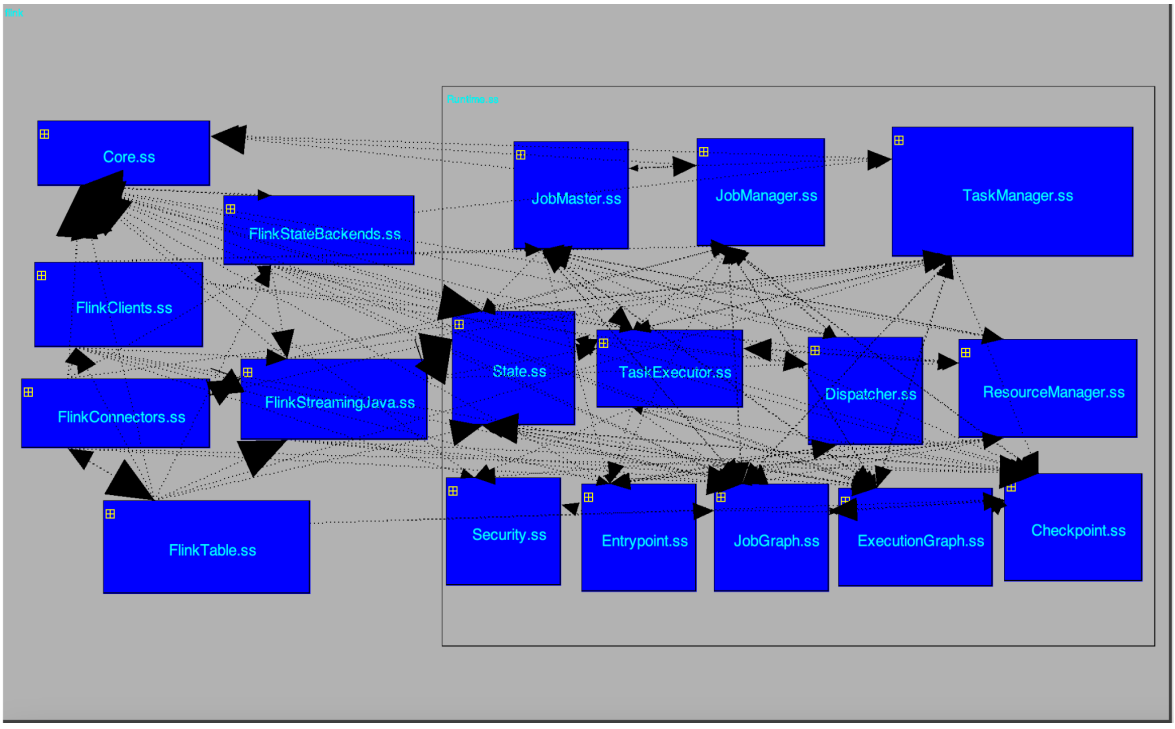

Figure 2: Concrete Architecture of Apache Flink

The concrete architecture of Apache Flink as shown in figure 2 represents a complex network of interrelated components, significantly extending beyond its initial conceptual framework to address practical needs like system resilience, scalability, and performance optimization. This expansion is a strategic adaptation to real-world data processing demands and the evolving technological landscape. Unlike the conceptual architecture's more abstract view, the concrete architecture delineates specific functionalities into distinct modules, such as StateBackend and Checkpointing, offering granular control over state management and enhancing fault tolerance [2]. It also sheds light on the intricate communication patterns between components like TaskExecutors and JobManagers, which are vital for task distribution and management in the system's distributed environment. This detailed visualization of component interactions highlights the importance of coordination in a distributed system, contributing to overall coherence and efficiency. Ultimately, the concrete architecture of Flink illustrates the transition from high-level concepts to a sophisticated, operational distributed data processing platform, embodying the iterative development process where theoretical models are refined through practical application and feedback [2].

3 Comparative Analysis and Proposed Modifications

3.1 Methodology

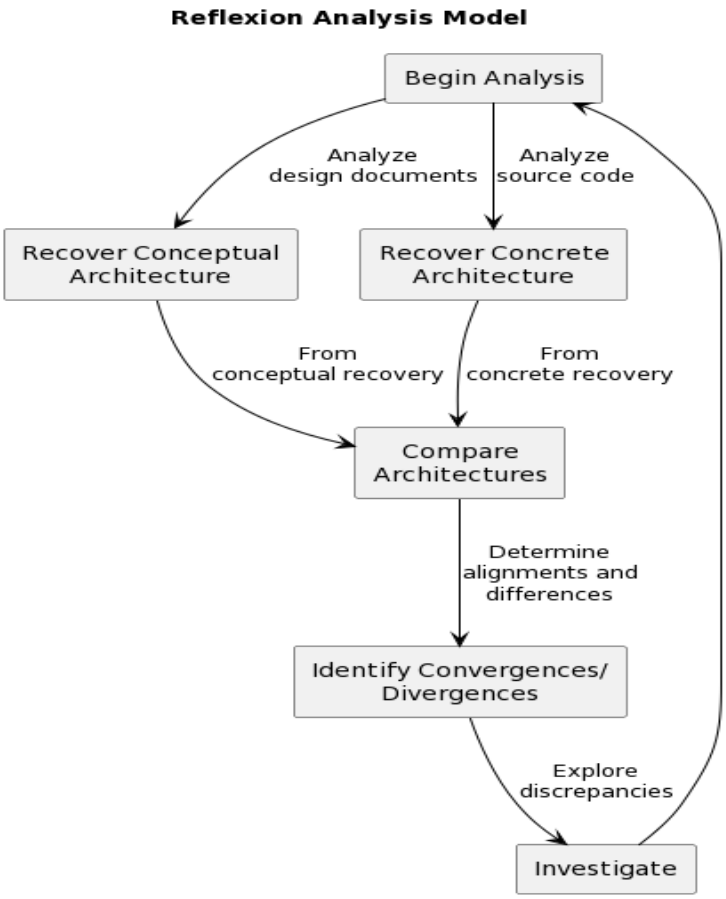

Figure 3: Reflexion Analysis Framework

The reflection analysis process for Apache Flink is a structured approach to evaluate the alignment between its conceptual architecture and concrete implementation. This methodology is vital for maintaining the system's robustness, scalability, and maintainability, particularly for complex systems like Apache Flink [3]. The process begins with mapping architectural components, establishing a direct correlation between high-level designs and their practical counterparts in the system. For example, the conceptual 'JobManager' is mapped to its concrete implementation, 'JobManager.ss', setting the stage for a thorough comparative analysis.

In the comparative analysis phase, the methodology involves a detailed examination of both architectures, focusing on identifying convergences (alignments) and discrepancies between them. Discrepancies are categorized into absences (missing conceptual components in the concrete architecture) and divergences (deviations in structure or function from the conceptual design). A color-coding scheme is employed to visually differentiate these elements, with green boxes indicating alignments and red highlighting discrepancies. This visual tool is crucial for quickly spotting areas needing further analysis.

The analysis is conducted at two critical levels: the top-level subsystem and the detailed design level. The top-level subsystem includes essential components like JobManager, TaskManagers, Dispatcher, ResourceManager and the API and Client interfaces, focusing on task orchestration and resource management. The detailed design level delves into the intricacies of how system components support high-level functionalities. This includes examining State Backend, JobMaster and various API subsystems which are crucial for the system’s overall performance. This two-tiered approach ensures a comprehensive assessment of Apache Flink’s architecture from a macro and micro perspective.

3.2 Top-Level Architecture

Comparative Analysis
An in-depth comparative analysis of Apache Flink's architecture, visualized in Figure 3 with a color-coding scheme, enables us to methodically examine the system’s evolution from its conceptual blueprint to its concrete implementation. This diagrammatic approach is integral to our understanding of how the theoretical design has translated into the operational framework. Green boxes are used to highlight components that are consistent across both conceptual and concrete architectures, showing where the actual system matches the original plan. Red marks are used to highlight components that were not identified in the conceptual architecture, highlighting a gap in our conceptual architecture research or the system's evolution beyond its initial design to meet real-world needs.

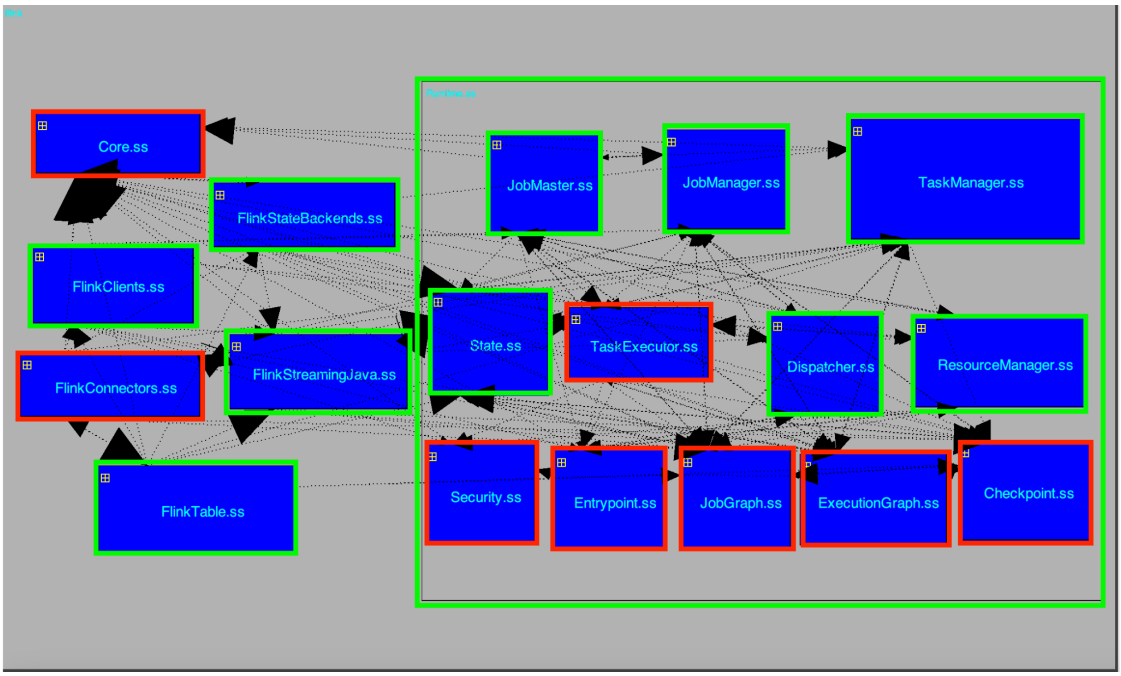

Figure 4: Comparative Architecture with Discrepancy Highlighting

Mapping Architectural Components:
The mapping process is the first step in our analysis, establishing a direct correlation between the conceptual elements and their concrete counterparts. It sets the stage for a foundational comparison, which is critical in assessing the fidelity of the implementation to the original design intent.

Comparative Analysis:
Following the mapping, we delve into a comparative analysis that uncovers the following:
Convergences (Green): In Apache Flink's architecture, Figures 5 through 8 illustrate a remarkable alignment between the system's conceptual design and its actual implementation, showcasing successful convergences in key components. Figure 5 highlights the effective realization of the JobMaster and JobManager roles in the system, as mirrored in the JobMaster.ss and JobManager.ss components, indicating that the initial design accurately captured the essentials of task coordination and management. Figure 6 shows a similar convergence between the conceptual Flink Cluster and its practical counterpart, Runtime.ss, suggesting that the foundational structure of the cluster was well-conceived and effectively implemented. In Figure 7, the alignment between the Flink Client/Table API and FlinkClients.ss/FlinkTable.ss demonstrates a successful translation of user interface and API design to practical application. Lastly, Figure 8's convergence between the State Backend and FlinkStateBackends.ss & State.ss reflects the accurate implementation of state management mechanisms, crucial for Flink's stream processing capabilities. These convergences underscore the effectiveness of Flink's architectural planning in meeting its operational goals and maintaining the integrity of its design principles.

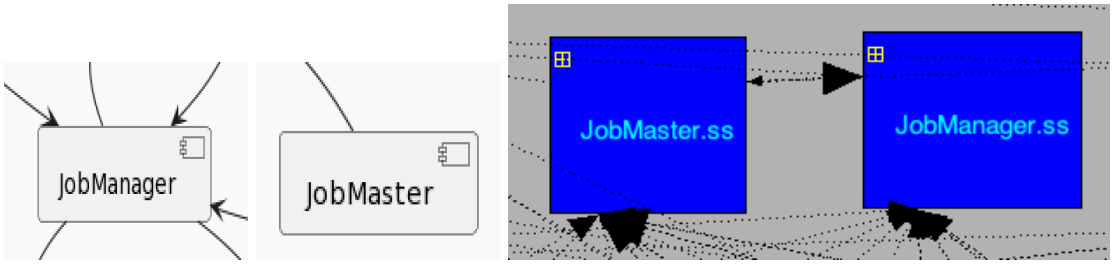

Figure 5: Convergence - JobMaster/JobManager & JobMaster.ss/JobManager.ss

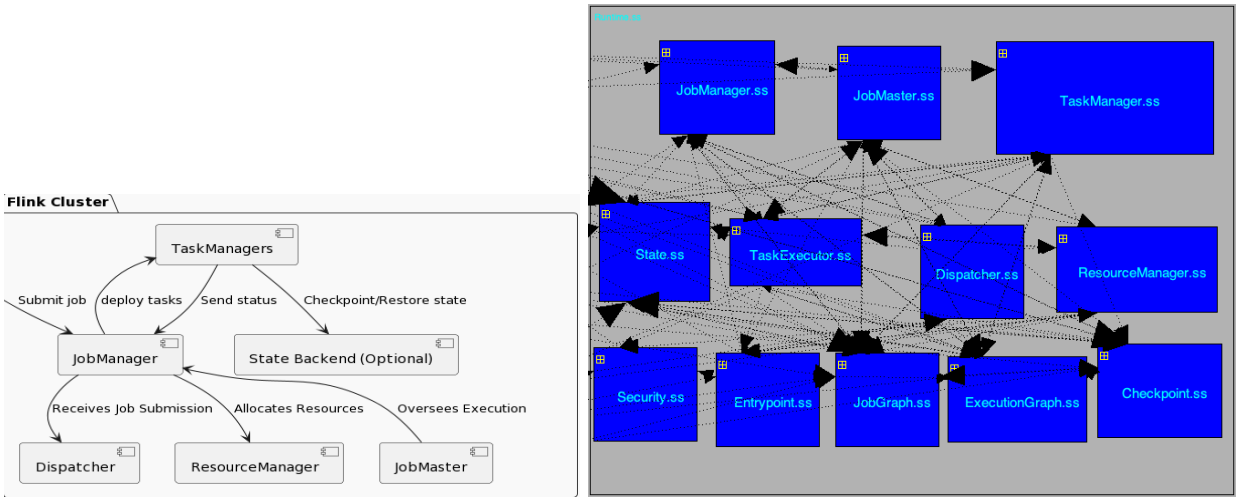

Figure 6: Convergence - Flink Cluster & Runtime.ss

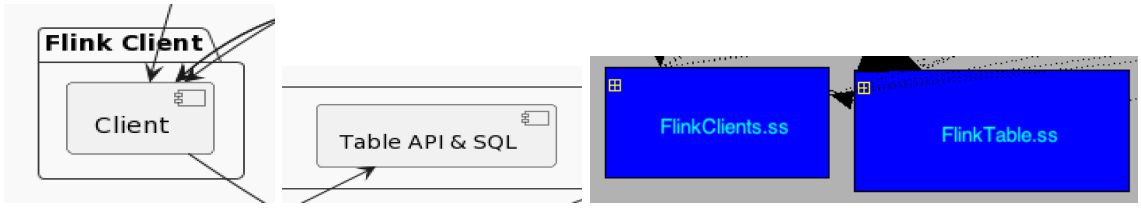

Figure 7: Convergence - Flink Client/Table API & FlinkClients.ss/FlinkTable.ss

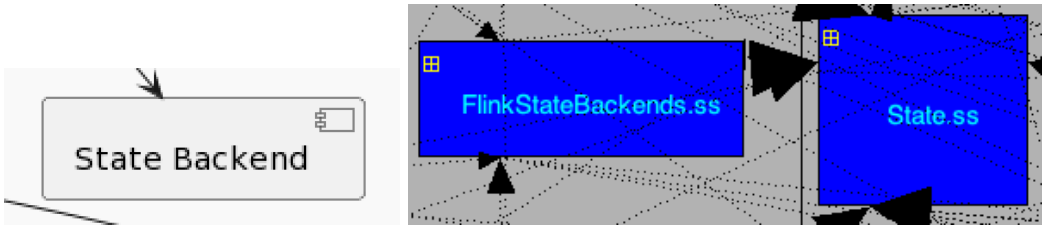

Figure 8: Convergence - State Backend & FlinkStateBackends.ss & State.ss

Divergences (Red): In Apache Flink's architecture, notable divergences from the conceptual design to the actual implementation enhance its functionality and efficiency. A key example is the interaction between the JobManager and TaskManager, which, contrary to the direct approach initially envisioned, is facilitated through the JobMaster. This intermediary layer adds complexity but significantly improves scalability and fault tolerance by managing task allocation and coordination more effectively. Another divergence is observed in the role of the

TaskManager. Rather than merely executing tasks, the TaskManager, through its TaskExecutor component, actively manages task execution. This includes handling resource allocation, thread management, and overseeing the task lifecycle. These adaptations, while adding to the system's complexity, are crucial for handling larger-scale workloads and ensuring robust fault tolerance, demonstrating a practical evolution from the original architectural design to a more sophisticated and capable system.

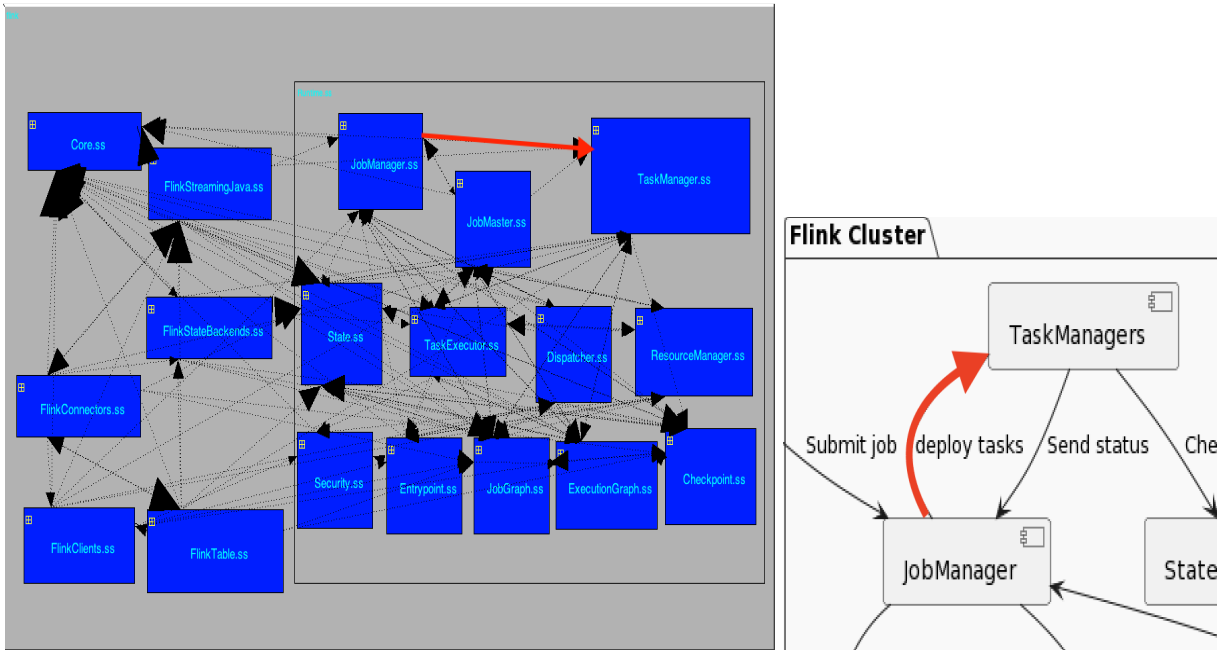

Figure 9: Divergence - The interaction between JobManager & TaskManager is facilitated through the JobMaster

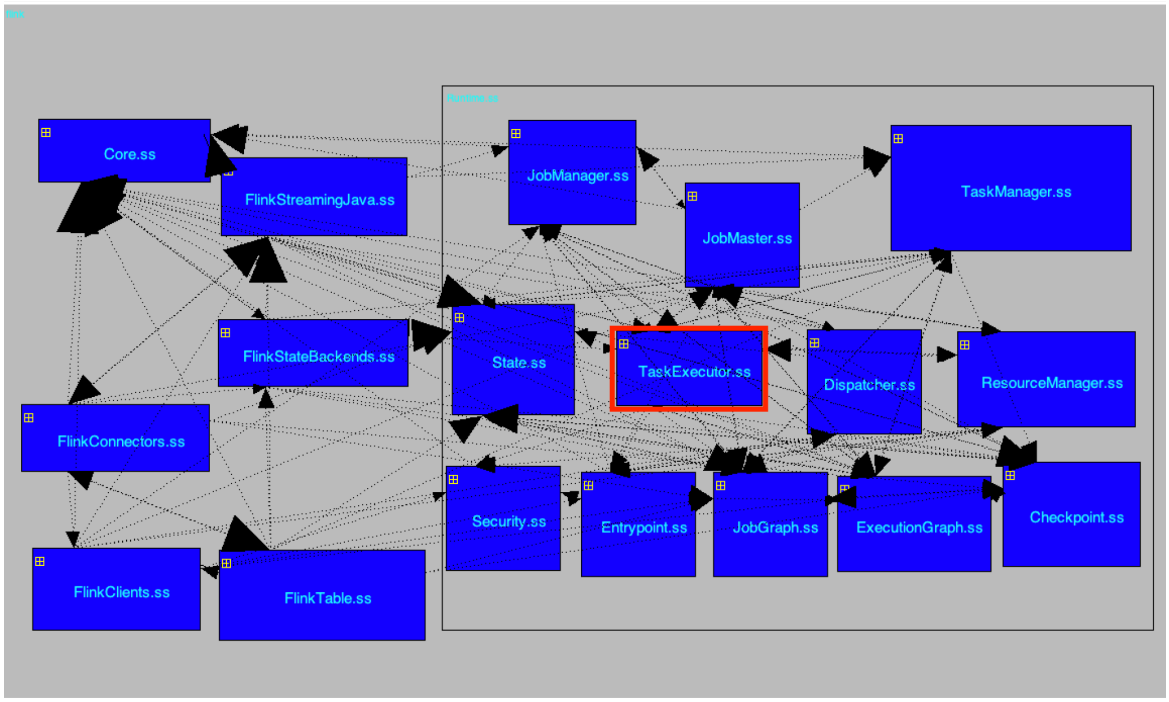

Figure 10: Divergence - The Task Manager executes tasks and manages task execution through the TaskExecutor.ss

Absences: There were no significant absences found during the comparative analysis which is common for large scale software systems.

Proposed Modifications

To harmonize the conceptual and concrete architectures of Apache Flink, several key modifications can enhance clarity and cohesion. Firstly, the concrete architecture should explicitly incorporate the high-level design patterns of the conceptual architecture, like the Repository and Facade patterns. This would bridge the gap between overarching design principles and the nitty-gritty of component functionalities, aiding developers in understanding how high-level design translates into system operations.

Secondly, enriching the conceptual architecture with more explicit interaction details can ensure a better alignment with the concrete implementation. Standardizing terminology across both documents is crucial to avoid confusion and ensure that each term and component is consistently understood, regardless of the level of abstraction.

Lastly, developing a functional map that connects the conceptual layers to the concrete subsystems would illustrate the direct relationship between planning and execution stages. Additionally, instituting a feedback loop would allow for the conceptual architecture to be dynamically refined based on practical insights from the implementation, ensuring both documents evolve together in line with system changes and requirements.

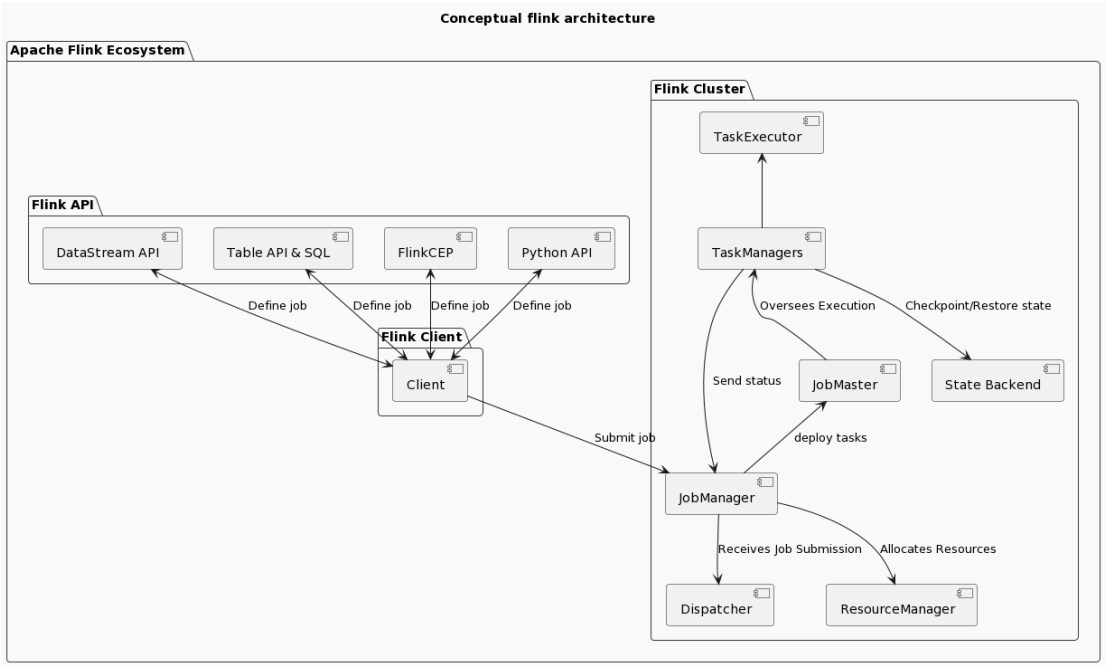

Figure 11: Modified Conceptual Architecture

Tangible Benefits of Proposed Modifications:

Apache Flink can achieve better operational efficiency by aligning the conceptual and concrete architectures, especially in memory and task management. Thus it enhances the system

performance. It can also lead to faster data processing. Modifications that cater to scaling issues, such as dynamic memory allocations, can significantly improve the system's ability to handle larger data sets more efficiently.

4 Subsystem Level Architecture: Task Manager In-Depth Discrepancy Analysis

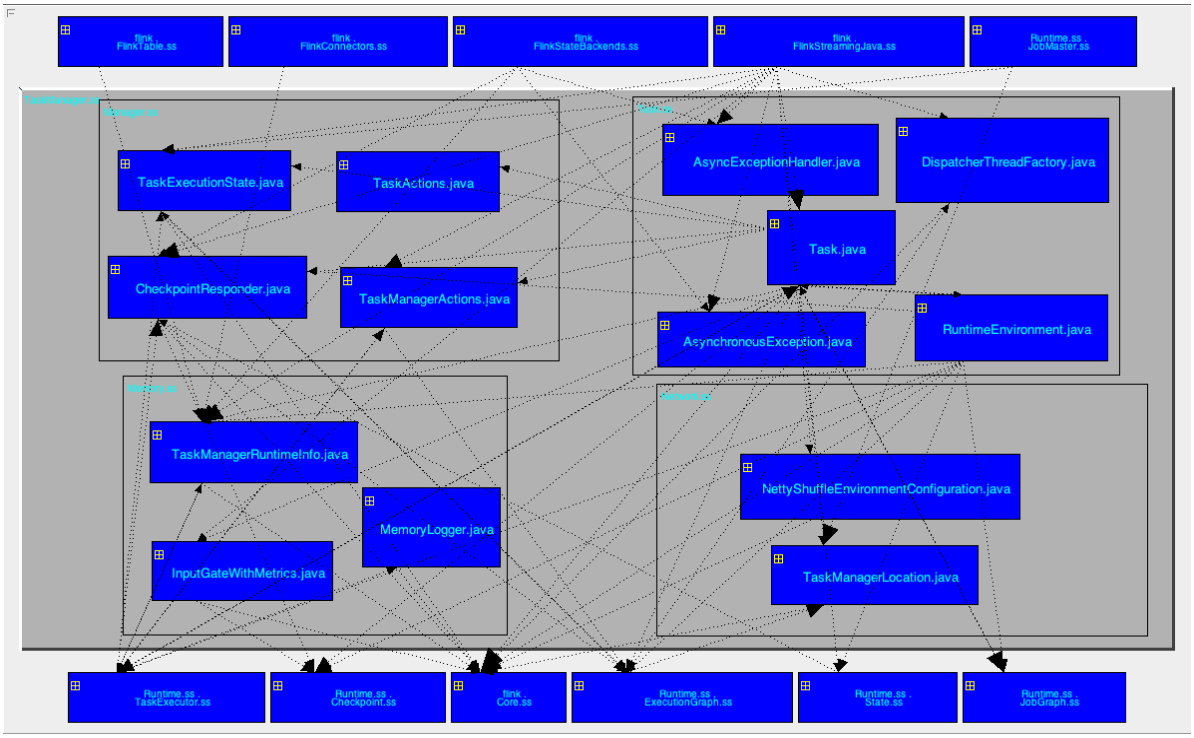

Figure 12: Task Manager Concrete Architecture

The TaskManager subsystem is crucial for the operation of Apache Flink, as it manages task execution, maintains the life cycle of the tasks, and communicates with other components for job orchestration. The TaskManager is also responsible for memory management, network input/output, and interacting with the persistent state backend [2].

Conceptual vs. Concrete Architecture in TaskManager.ss:
The conceptual architecture represents the TaskManager as a monolithic block, handling multiple responsibilities without detailing the internal components and their interactions. In contrast, the concrete architecture diagrams show that the TaskManager is composed of multiple smaller components, each with a specific role.

Discrepancies Noted in the TaskManager.ss Subsystem:
In Apache Flink, the TaskManager.ss's actual architecture reveals differences from the original design, showing optimization progress. Memory is actually managed by Memory.ss, not

TaskManager.ss. Task execution is by TaskExecutor.ss, not TaskManager.ss, for distributed efficiency. Network operations are by Network.ss, not abstractly by TaskManager.ss, indicating a distributed system. State.ss manages state, not the TaskManager.ss as planned. JobGraph.ss and ExecutionGraph.ss handle job and task coordination, not the simplistic original concept. Checkpoint.ss emphasizes fault tolerance not detailed before. Manager.ss indicates structured TaskManager interactions unlike the direct conceptual ones. These changes mark a move to a more intricate but effective architecture.

Proposed Modifications
Conceptual modifications to enhance these subsystems could include a unified configuration interface to streamline setup and maintenance, and a modular logging mechanism for greater flexibility. A unified interface would simplify system configuration, reducing complexity and the likelihood of errors, while a modular logging system would allow for customized logging solutions that can be tailored to specific operational needs, increasing the system's adaptability.

A concrete modification could be implementing dynamic memory allocation to optimize resource usage. This adaptive approach would allocate memory based on current workload demands, ensuring efficient utilization of resources, potentially reducing costs, and avoiding bottlenecks. Additionally, introducing task lifecycle plugins would offer extensibility by allowing custom actions to be triggered at different stages of the task lifecycle, enhancing the system's flexibility to handle various operational scenarios.

The justification for these modifications lies in the expected improvements to system efficiency, performance, and maintainability. A unified configuration interface would mitigate configuration complexities, while a modular logging system would cater to diverse deployment needs. Dynamic memory allocation would make resource management more efficient, and task lifecycle plugins would encourage community engagement and innovation, potentially leading to broader functionality and improved task management within Apache Flink.

5 Rationale Behind Discrepancies

5.1 Performance Considerations

The concrete architecture of Apache Flink, particularly in the Task Manager and Memory Manager subsystems, has been significantly influenced by performance optimization. The conceptual design aimed at managing tasks and memory more broadly. However, the high throughput and low latency requirements of real-world usage required improvements that were not entirely accounted for in the conceptual design. These include techniques that were added to improve performance but diverged from the original architectural ideals, such as memory segmentation and dynamic task slot allocation.

5.2 Scaling Challenges

Due to the wide range of cases in which Apache Flink is used—from big business systems to small-scale clusters—its design has had to adjust to meet a variety of scalability needs. Differences in the concrete architecture resulted from the conceptual architecture's failure to properly account for these scaling issues. For instance, to address the scaling requirements—which were not made clear in the initial design—extra components were added to the job Manager subsystem in order to facilitate effective resource management and job allocation.

5.3 Iterative Development using Feedback Loops

Feedback loops and incremental improvements have been integral to the development of the subsystems. This has caused its architecture to be continuously improved and adjusted in response to user feedback and real-world difficulties. Although these incremental improvements have improved the system's usefulness and robustness, they have unavoidably caused deviations from the initial conceptual architecture.

Since Apache Flink is an open-source project, it has developed as a result of community contributions, which frequently result in innovations and modifications not included in the initial plan. As new features and enhancements are incorporated into the system, this collaborative development model may lead to differences since it reflects the community's combined input and knowledge rather than a single, predetermined architectural plan.

A few instances of the differences caused by this collaborative model are illustrated below with the help of 4W's method. These records of changes have been retrieved from the official GitHub repository of Apache Flink:
[4]

WHAT	This change involved moving the CheckpointListener from the 'flink-runtime' module to the 'flink-core' module in Apache Flink. The change affected 45 files, with 195 additions and 64 deletions.
WHO	Becket Qin
WHEN	November 5, 2020
WHY	This was done most likely to improve the modularity and architectural coherence of Apache Flink. By moving the CheckpointListener to flink-core, it aligns more closely with the core functionalities of Flink, possibly making it more accessible or logically placed for other components that depend on it. They decided to not immediately delete the CheckpointListener from 'flink-runtime' to maintain backward compatibility.

[5]

WHAT	The change involved moving Executors and ExecutorThreadFactory from their previous location(org.apache.flink.runtime) to flink-core(org.apache.flink.util). The change impacted 105 files with 116 additions and 112 deletions, indicating a substantial modification in the project structure.
WHO	Chesnay Schepler
WHEN	June 23, 2021
WHY	This centralized the Executors and ThreadFactory functionalities within the core module of Apache Flink. This also potentially improved access, maintainability, and logical organization. By positioning these components in flink-core, they became more integral to the framework.

[6]

WHAT	This change involved renaming the state "RECOVERING" to "INITIALIZING" in the runtime module of Apache Flink. This change impacted 23 files within the project, with an equal number of 72 additions and 72 deletions.
WHO	Andrey Kalashnikov (Author) and Dawid Wysakowicz (Commiter)
WHEN	April 13, 2021
WHY	Renaming from "RECOVERING" to "INITIALIZING" is done to emphasize this state within the Flink runtime, perhaps to better convey the initialization activities rather than the recovery aspect.

[7]

WHAT	This change involved moving the ‘SecurityManager’ component from its previous location to the ‘flink-core’ module. This affected 10 files, with 13 additions and 13 deletions, indicating a moderate update to the project's structure.
WHO	Chesnay Schepler
WHEN	June 23, 2021
WHY	The relocation of SecurityManager to ‘flink-core’ likely aimed to centralize security-related functionalities within the core module of Apache Flink. Moving key components like SecurityManager to the core module can be done to enhance accessibility, maintainability, and logical organization of security-related features.

6 Impact of Discrepancies on System Operation

Complexity and Abstraction of the System

The Conceptual Architecture of Apache Flink provides a high- abstraction of how the data flows between the systems and how they interact with others on an abstracted level with the upside of having less complexity as the low-level execution is not shown. On the other hand, the concrete architecture of the Task Manager subsystem shows little to no abstraction which exhibits more complexity of the architecture. It is responsible for managing the complex details of low-level execution.

Impact on Functionality : Although abstraction can allow Users to give a good overview of how the system works on a high-level, challenges can be faced when Users need control or visualization of the low-level execution that takes place in the Task Manager subsystem.

Impact on Performance: Even though abstraction provides less complexity, it can also introduce overhead which impacts the performance in scenarios where low-level optimizations are crucial

Impact on Scalability: Abstraction does simplify the development but may cause issues when dealing with scalability as optimizations and fine-tuning on large data sets are crucial

Checkpointing Overhead

The Conceptual Architecture of Apache Flink provides the checkpointing for fault tolerance at a high level while the Concrete Architecture of Task Manager provides the coordination and execution of checkpoint. Balancing the trade-off between the fault-tolerance and frequency of checkpointing is crucial.

The discrepancy between the two architectures may lead challenges to balancing the frequency of checkpointing. If the coordination and the execution mechanisms in the Task Manager bring forth a significant amount of overhead, this can cause a significant impact in the overall efficiency of the systems operation especially in scenarios where low-latency processing is crucial.

Task Parallelism

The Conceptual Architecture of Apache Flink provides the checkpointing for fault tolerance at a high level while the Concrete Architecture of Task Manager provides the coordination and execution of checkpoint. Balancing the trade-off between the fault-tolerance and frequency of checkpointing is crucial.
The discrepancy between the two architectures may lead challenges to balancing the frequency of checkpointing. If the coordination and the execution mechanisms in the Task Manager bring forth a significant amount of overhead, this can cause a significant impact in the overall

efficiency of the systems operation especially in scenarios where low-latency processing is crucial.

Considerations

Better Documentation - We can consider having better documentation on both the conceptual architecture of Apache Flink and the concrete architecture of Task Manager. This can ensure that the users are have a deeper understand of the pros and cons of each level of architecture that is present in Apache Flink

Optimization and Fine-Tuning: Having a guidelines of the different ways of optimization and fine-tuning can enable the users to understand which parameters can affect their use-case and adjust it accordingly

Feedback: Having a tool that records the feedback of the users that identifies their frustrations can allow the project to have new modifications regarding the architecture

Best Practises Guideline: Since Apache Flink is open source and contributed by the community, having a guideline for best practises, the do’s and don'ts can massively increase the quality and can further improve architecture.

Monitoring: Increasing the capabilities of monitoring and diagnostics can allow the users using the system to deeply understand any flaws or optimization that can be added in the application and improving the architecture

7 Conclusion

This reflection analysis has not only mapped the conceptual design onto the concrete implementation but has also scrutinized the gaps to extract meaningful insights into the architectural evolution of Apache Flink. The transition from a conceptual to a concrete architecture underscores a transformation from a simplified, monolithic design to a sophisticated, distributed, and modular system architecture.

This transformation reflects a broader trend in software engineering, where distributed systems are increasingly favored for their scalability, resilience, and flexibility. The findings of this report highlight the critical importance of maintaining an iterative approach to design, where architectures are not static blueprints but living structures that must adapt to new insights, emerging requirements, and technological advancements.

For Apache Flink, and similar systems, the ongoing success hinges on the alignment of documentation with the actual architecture, ensuring that stakeholders have a clear and accurate understanding of the system's capabilities. This alignment is not a one-time effort but a continuous process that must be woven into the fabric of the development lifecycle, ensuring

that the system's architecture can evolve and adapt while remaining comprehensible and accessible to all users.

8 References

[1]. Sadique, N. A., Arefin, A., Osaroedey, K., Okon, K., Ali, D., & Idimogu, K. (2023). Apache Flink: Conceptual Architecture Recovery. Unpublished manuscript, EECS 4314, York University.
[2]. Sadique, N. A., Arefin, A., Osaroedey, K., Okon, K., Ali, D., & Idimogu, K. (2023). Apache Flink: Concrete Architecture Recovery. Unpublished manuscript, EECS 4314, York University.
[3]. Murphy, G. C., Notkin, D., & Sullivan, K. J. (2001). Software reflexion models: Bridging the gap between design and implementation. IEEE Transactions on Software Engineering, 27(4), 364-380.
[4] Apache Flink. (2020, November 5). Commit 6bf7d77 [Commit to the repository Apache Flink]. GitHub. https://github.com/apache/flink/commit/6bf7d77a76afa1c705c86adf46fb8732841b9dd8
[5]. Apache Flink. (2021, June 3). Commit 52609d3 [Commit to the repository Apache Flink]. GitHub. https://github.com/apache/flink/commit/52609d3f018d6402990460d34131efd7815bc61a
[6]. Apache Flink. (2021, April 13). Commit b387928 [Commit to the repository Apache Flink]. GitHub. https://github.com/apache/flink/commit/b3879280fa60f11c874af042b81ff5ac4672a3eb
[7]. Apache Flink. (2021, June 23). Commit c08bef1 [Commit to the repository Apache Flink]. GitHub. https://github.com/apache/flink/commit/c08bef1f7913eb1c416c278354fd62b82b172549