Apache Flink: Checkpoint Subsystem Reflexion Analysis
FlinkForce

Rafael Dolores[1] **Hashir Jamil**[1] **Alex Arnold**[1] **Zachary Ross**[1]
Nabaa Gaziy[1] **James Le**[1] **Walid AlDari**[1] **Maaz Siddiqui**[1]
[1]Department of Electrical Engineering & Computer Science, York University
{rafd47,hashirj,alex290,zachross,Nabaagz,jamesqml,walidald,msiddiqi}@my.yorku.ca

Abstract

In this study we show the revision of previously extracted concrete architecture for the distributed stream platform Apache Flink, with a focus on the systems statefulness guaranteeing Checkpoint Subsystem. This follows the reflexion model technique and source sticky note technique to learn about system evolution by using git blame and developer logs. Moreover the whitepapers and implementation documentation is used to help reconcile the architectural discrepancy. Several dependencies are investigated in depth by looking at Confluence documentation and Jira issues. Finally, a revised conceptual architecture is presented. Throughout this report all sticky notes have the original pull request as an in-line hyperlink. As well, all diagrams are vectorized images that can be zoomed in on without loss of quality. The original diagrams were created using diagrams.net and the cloud file is provided here.

1 Introduction

Source code and system documentation discrepancies are very common, and more importantly, inevitable in large scale open-source software. Code authorship is also difficult to track over the course of an open-source systems evolution. Furthermore, conceptual architecture presented in system documentation often diverges from the architecture extracted via source code and object code analysis tools [1–3]. In this study we further analyse Apache Flink, a distributed stateful streaming framework for bounded and unbounded datastreams [4] and in particular is state snapshotting checkpoint mechanism [5]. We build upon previous architectural analysis performed at the conceptual level using the system documentation and at the concrete level using the project structure [6, 7]. The latter analysis was aided by using source code analysis by leveraging Understand and LSEdit [8, 9]. Both of these architectural views are considered to propose revisions and reconciliations for the description of the Apache Flink system.

This is accomplished by investigating the source code in Apache Flink release 1.17, specifically looking at interfile dependencies and naming conventions. Moreover, the commit logs, public conversations on GitHub Pull Requests, Confluence documentations, Jira issues and code comments are used to gain deeper insights. These developer conversations are used to generate source sticky notes to map development history, and a formal reflexion analysis is performed [10, 11]. This analysis is performed at three hierarchichal levels of Flink's architecture. First, the top level view, the runtime subsystem level view and the checkpoint subsystem level view. We then amend and reconcile the previously presented architectures with new proposed architectural diagrams. Then conclude with some comments on previously discussed use cases, overall limitations and the general lessons learned.

2 Methodology

2.1 Reflexion Model Analysis

2.1.1 Approach 1 (Manual)

For our reflexion analysis of Apache Flink we follow the process shown in Figure 1. First we compare our conceptual architecture diagram to our concrete diagram of Apache Flink and find the absences, convergences, and divergences. This allows us to understand which dependencies are diverging for the discrepancy analysis. We then explored the Apache Flink source code and found the package import lines for the dependencies. Using a VSCode extension GitLens we are able to find the commit that added the dependency [12]. From the commit we are then able to explore the related GitHub pull request and Jira tasks to find more information about who/what/when/why the dependency was added. Which we then use to create our divergence sticky notes.

2.1.2 Approach 2 (Systematic)

This approach can be seen in Figure 2. The idea in this approach is to use the command line interface for git and use this to go through commit logs by dumping these directly to the germinal to read as raw text. This methodology was dropped because it would require a script and it would be harder to read the code while navigating around the source files. This would have been very time consuming as there is no GUI to navigate the code with.

2.2 Architectural Revision

The conceptual architecture was revised by looking at the documentation again, using the source code heavily and lastly using the concrete archtiecture extraction results from Understand and LSEdit. The concrete architecture was not updated as we felt it was at a sufficient level of granularity.

3 Architecture Reflexion Analysis

In this section we will work our way through the architectural views of the Apache Flink Streaming framework. It is best to show the system by working top to bottom to show it as three distinct levels. These levels will have both conceptual and concrete models. The first view will be a top-level view of the major high level subsystems. The journey is meant to focus on the checkpoint subsystem. Hence, the next level that will be examined is the runtime subsystem as this contains the checkpoint subsystem. Finally, the checkpoint subsystem is examined in detail along with four distinct subsubsystems that it contains. These are the CheckpointCoordinator, Channel, Metadata and

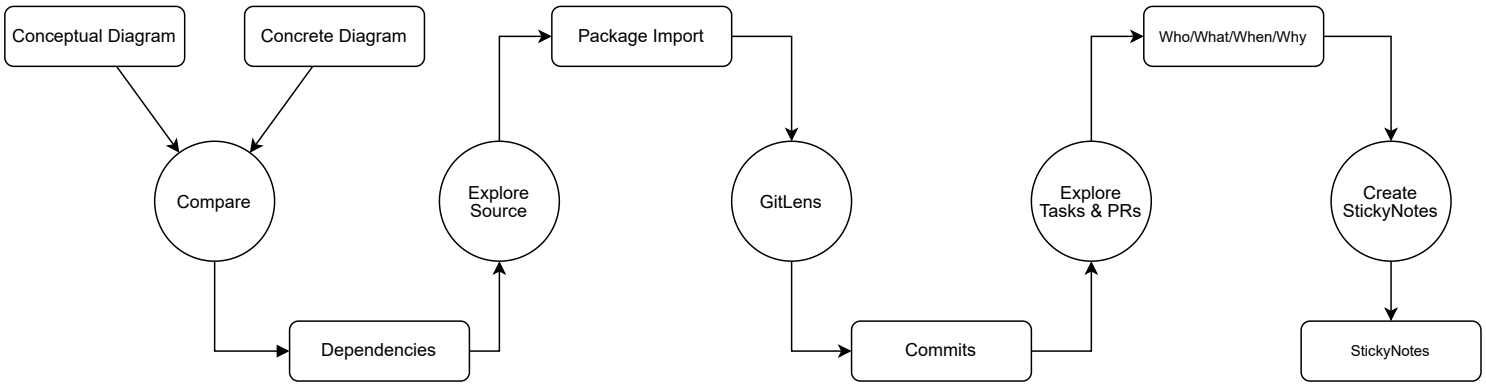

Fig. 1: Conceptual & concrete architecture revision methodology using GitLens to analyze Git logs for sticky note creation.

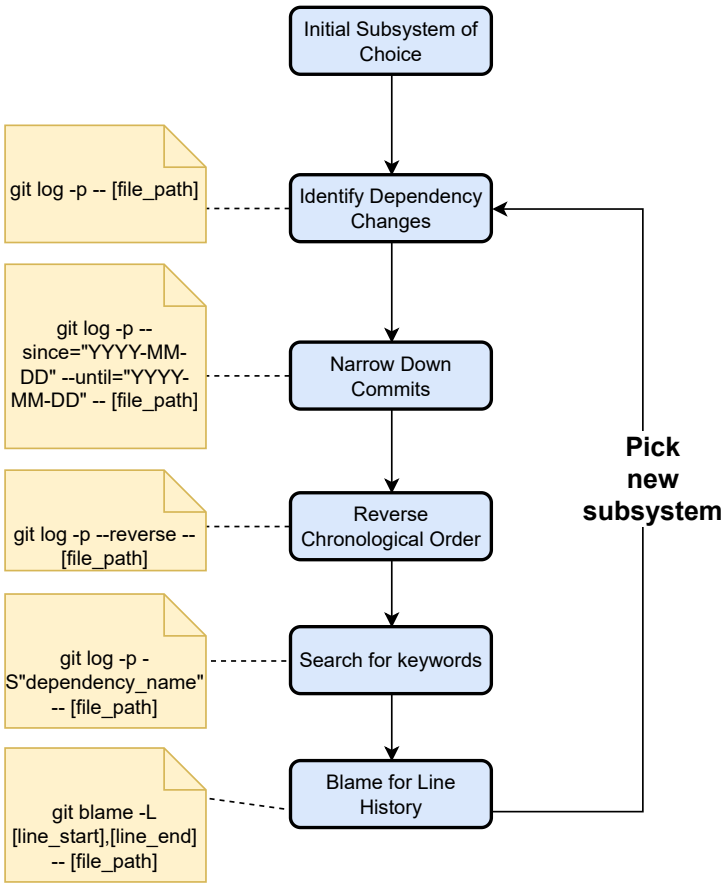

Fig. 2: Alternative proposed methodology for analyzing devlopment history.

Hooks. Each level will also present select dependencies for analysis as source sticky notes.

3.1 Top-Level Subsystems View

3.1.1 Conceptual Model

Within the conceptual architecture of Flink (As shown in Figure 3), the Flink Cluster stands out as a key component, providing the core runtime environment for executing Flink applications. It is composed of the Job Manager, Task Managers, and the essential infrastructure necessary for coordinating and executing jobs. This architecture is highly adaptable, supporting various deployment modes like standalone, YARN, or Kubernetes, thereby facilitating Flink's seamless integration with diverse cluster management and resource systems. The design of Flink includes specialized tools for native interaction with these systems, enhancing the efficiency of resource management.

Central to the Flink architecture are the Job Managers, which orchestrate Flink job execution. They include several key components: the Dispatcher, acting as a REST API gateway for job submissions and initiating isolated JobManager instances for each job; the ResourceManager, which manages physical resources, allocates tasks, and optimizes resource utilization; and the JobMaster, responsible for task scheduling, state management, and fault recovery. Together, these components ensure isolated, efficient, and reliable execution of each job.

Task Managers function as worker nodes, executing sub-tasks and processing data streams within Flink's distributed setup. They manage concurrent task execution in multiple slots, handle data buffering, and facilitate data shuffling for optimized processing. The State Backend is pivotal for fault tolerance, managing job states and supporting checkpointing for data recovery and computational resumption post-failure. This system ensures that Flink maintains data consistency and reliability, minimizing failure impacts and ensuring quick recovery for continuous and accurate data processing.

3.1.2 Concrete Model

Apache Flink's architecture is defined by several interconnected subsystems (shown in Figure 4), each contributing uniquely to its data processing effectiveness. The Core module is essential, pro-

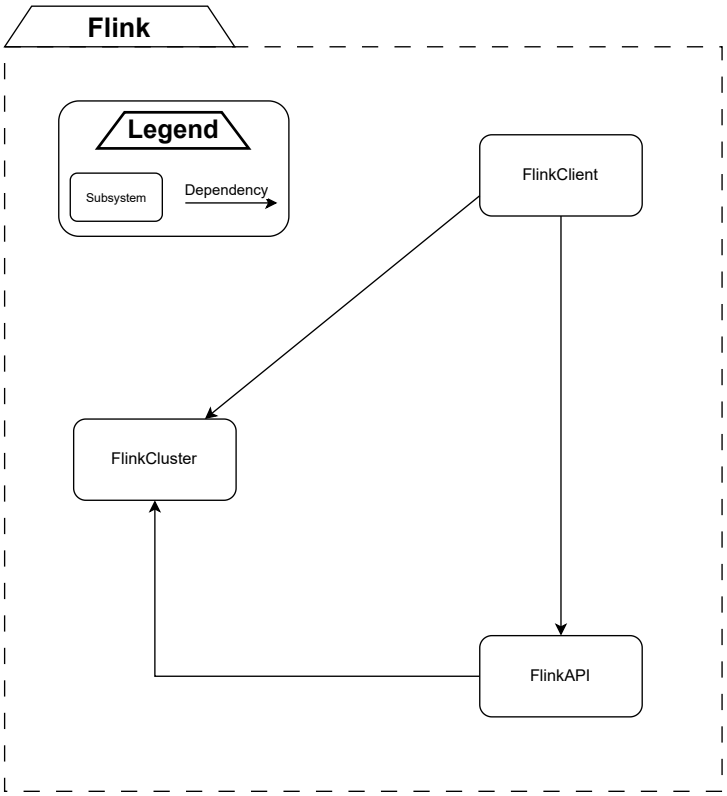

Fig. 3: Top-level view conceptual architecture for Apache Flink.

viding the basic utilities and classes for both stream and batch data processing. It lays the foundation for the rest of the system, allowing for further expansion and enhancement of Flink's functionalities.

The FlinkClient module serves as an intermediary between users and the Flink cluster. It transforms complex user-defined pipelines into executable formats and provides accessible interfaces, such as command-line tools. This module's versatility in managing various cluster configurations and integration with web systems via HTTP REST requests is key to facilitating user interaction and workflow management.

FlinkConnector and FlinkTable play crucial roles in interfacing with external systems and structured data processing, respectively. FlinkConnector manages data sources and sinks, ensuring efficient data flow through the Directed Acyclic Graph (DAG), while FlinkTable allows for SQL queries and table operations, integrating traditional SQL analytics with stream processing. The Runtime module is central to the lifecycle management of Flink jobs, focusing on resource allocation, fault tolerance, and the orchestration of various components from job submission to execution.

Complementing these, the FlinkStreaming module addresses the needs of real-time data processing with continuous data streams. It is designed for low-latency and high-throughput processing, vital for real-time analytics and decision-making applications. The FlinkStateBackend subsystem, with its state backends like RocksDB and heap memory, is essential for scalable, resilient stream processing. This includes supporting reliable job recovery and features like exactly-once processing guarantees, underscoring the system's capability for comprehensive data processing.

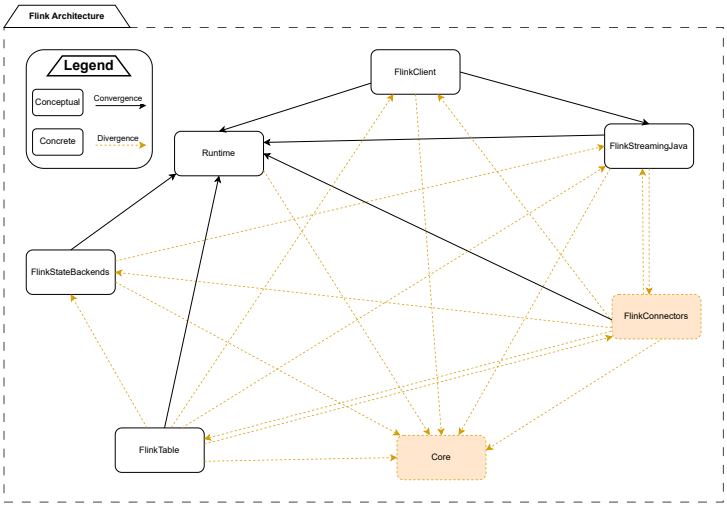

Fig. 4: Top-level view cocnrete architecture for Apache Flink.

3.1.3 Subsystem Interactions

In Flink's conceptual architecture, the interaction between the Job Manager and Task Managers is pivotal for job execution. The Job Manager, comprising the Dispatcher, ResourceManager, and JobMaster, serves as the orchestration hub. The Dispatcher initiates job execution by creating JobManager instances for each submitted job. The ResourceManager then allocates physical resources and tasks to the Task Managers, which are the worker nodes responsible for executing the sub-tasks of the job. The JobMaster coordinates these tasks, ensuring they are scheduled and executed efficiently. This orchestrated interaction ensures that each job is processed reliably, and resources are utilized optimally.

A critical aspect of Flink's operation is the management of data streams and job states. Task Managers handle the processing of data streams, executing tasks concurrently and managing data shuffling between tasks for optimized processing. Concurrently, the State Backend plays a crucial role in maintaining the state of these jobs. It manages job states and facilitates checkpointing for data recovery, which is essential for fault tolerance. The interaction between Task Managers and the State Backend ensures that data processing is not only efficient but also reliable, with a strong mechanism for recovery in case of failures.

Flink's conceptual architecture highlights various deployment modes (like standalone, YARN, or Kubernetes), which demands seamless integration and coordination between different subsystems. The Job Manager's components ensure this integration by providing tools and interfaces for efficient resource management and job execution across diverse environments. This coordination ensures that Flink can adapt to different operational requirements, maintaining efficiency and reliability regardless of the deployment scenario. The integration of these subsystems and their interactions form the backbone of Flink's capability to process large-scale data streams effectively and reliably.

In Flink's concrete architecture, the Core, FlinkClient, and FlinkConnector modules work in unison to manage data processing workflows. The Core module lays the groundwork with essential utilities for stream and batch data processing. Building on this,

the FlinkClient module acts as an intermediary, transforming user-defined pipelines into executable formats and streamlining the user interface with tools like command-line interfaces. It adeptly handles various cluster configurations and integrates with web systems via HTTP REST requests. The FlinkConnector then comes into play, facilitating the connection to external data systems. It manages the crucial data flow through Flink's Directed Acyclic Graph (DAG), handling data sources and sinks. This interaction ensures a smooth transition from user input to data processing and output delivery, emphasizing the system's adaptability and user-centric design.

Flink's data processing capabilities are further expanded by the FlinkConnector, FlinkTable, and FlinkStreaming modules. The FlinkConnector facilitates the integration with external data systems, managing data sources and sinks, and ensuring efficient data transactions. FlinkTable allows for structured data processing using SQL, merging traditional database operations with Flink's streaming capabilities, enabling intuitive data analytics. FlinkStreaming, a key component in real-time data processing, is essential for handling continuous data flows, emphasizing Flink's strength in processing large-scale streams with low latency and high throughput. These modules work in concert to manage the flow of data through Flink's Directed Acyclic Graph (DAG), underlining the system's versatility in data handling.

The Runtime and FlinkStateBackend modules are central to Flink's operational management and resilience. The Runtime module oversees the entire lifecycle of Flink jobs, from submission to execution, ensuring efficient resource allocation and fault tolerance. This coordination is crucial for managing complex data processing tasks and maintaining system integrity. The FlinkStateBackend subsystem, with state backends like RocksDB and heap memory, is vital for scalable and resilient stream processing. It supports dependable job recovery and features like exactly-once processing guarantees, playing a critical role in maintaining data consistency and reliability across various processing scenarios. The interplay of these modules demonstrates the sophisticated and cohesive nature of Flink's architecture, showcasing its comprehensive abilities in large-scale data processing.

3.1.4 Reflexion Analysis

3.1.4.1 Absences

In the reflexion analysis of the conceptual versus concrete architecture of Apache Flink, one can observe a unique alignment between the two models without any clear absences. Notably, all dependencies present in the conceptual framework find their counterparts within the concrete architecture. What is particularly interesting is that some singular conceptual components correspond to a combination of multiple concrete modules. As indicated in the image, the conceptual 'FlinkAPI' aligns with both 'FlinkTable' and 'FlinkStreamingJava' in the concrete model, meaning that 'FlinkAPI' is an abstraction representing these two distinct but related functionalities. Similarly, the 'FlinkCluster' in the conceptual model corresponds to the 'Runtime' component in the concrete architecture, implying that the 'Runtime' encompasses the cluster's operational aspects.

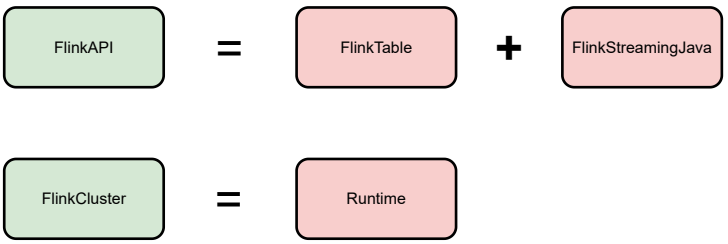

Fig. 5: Divergence between the conceptual & the concrete models at the top-level view of Apache Flink subsystems.

3.1.4.2 Convergences

In Apache Flink's architecture, the convergences between the conceptual and concrete models are marked by the consistent presence and interdependence of key components. The FlinkClient is central to both models, acting as the gateway for users, facilitating job submission and cluster interaction. It abstracts the intricacies of job configurations, serving as the user-friendly face of the system. Parallel to this, the Runtime, known conceptually as the FlinkCluster, is the engine room where job execution, task scheduling, and state management occur, ensuring efficient and fault-tolerant operations.

The interplay between these components is crucial; the FlinkClient relies on the Runtime to carry out its directives, creating a direct line from user interaction to core system functions. This dependency is a testament to the system's unified architecture, where user commands initiated via the FlinkClient are executed by the Runtime. The FlinkAPI, which represents the programming interface in the conceptual model, depends on the FlinkCluster for execution. In the concrete model, this translates into the FlinkClient necessitating the combined capabilities of FlinkTable and FlinkStreamingJava, highlighting a smooth operational flow from the user's high-level commands to the system's backend processing.

This cohesive dependency structure illustrates a system where the user's interface through the FlinkClient and FlinkAPI is seamlessly supported by the system's execution layers, represented by the Runtime. This strategic design ensures that the user's experience is consistently supported by robust and reliable system operations, demonstrating the architectural integrity of Apache Flink from conceptualization to practical implementation.

3.1.4.3 Divergences

Divergences in Apache Flink's architecture highlight dependencies that are present in the concrete implementation but not in the conceptual framework. To track and understand these differences, 'sticky notes' serve as a critical annotation tool, detailing the nature and rationale of modifications and additions to the software's structure. Through the "4 W's"—which dependencies are introduced in the concrete model, who is responsible for these changes, when they were incorporated, and why they were necessary—developers can attain a comprehensive understanding of the architectural evolution. This aforementioned approach will be utilized in the following subsections below where key divergence instances are highlighted in detail.

3.1.4.4 Dependency: FlinkStateBackend $\xrightarrow{\text{Uses}}$ FlinkStreaming

The integration of FlinkStreaming with FlinkStateBackend, illustrated in Table 1, was specifically designed to enhance the performance of stateful stream processing by adopting an asynchronous approach to RocksDB file operations. This meant that instead of performing file copies synchronously—which could stall the main thread and degrade performance—these tasks were shifted to execute in parallel, thereby allowing the main processing thread to continue uninterrupted. By decoupling the state management tasks from the primary computation thread, Flink made significant strides in reducing processing delays, particularly during state snapshotting and recovery processes. This method of asynchronous file handling is especially beneficial during the checkpointing phase, where quick and efficient state snapshots are essential for fault tolerance and smooth operation in distributed streaming applications.

By moving away from a synchronous model, Flink's ability to handle stateful computations scales more effectively, accommodating the demands of high-volume, low-latency applications. This change also means that the system can dynamically adjust to the workload, as the sizing of state data can now be deferred until it's operationally necessary, thanks to the getSize() method's dynamic invocation. Consequently, the system is better prepared to handle varying checkpoint sizes, which are often not predictable at the time of job initialization but can be determined in real-time, leading to more effective resource utilization and improved overall performance of the streaming application. This strategic enhancement of FlinkStreaming underscores Flink's commitment to providing robust, state-of-the-art stream processing capabilities.

Which	Package org.apache.flink.statebackend Depends on Package org.apache.flink.streaming
Who	Committer: Aljoscha Krettek
When	PR [FLINK-3359] committed on Feb. 11, 2016
Why	The dependency between FlinkStateBackend and FlinkStreaming was introduced to improve the efficiency of state management operations. Specifically, this was done by making the copying of RocksDB files asynchronous, which means that these operations could be performed without blocking the main thread of execution. This change was crucial for optimizing checkpointing performance, as the state size determination, previously done during object construction, could now be delayed until required by the getSize() method.

Table 1: State-Backend depends on FlinkStreaming sticky note.

3.1.4.5 Dependency: FlinkStateBackend $\xrightarrow{\text{Uses}}$ Core

This dependency is illustrated in Table 2. The FlinkStateBackend was structurally reorganized to bring about a more coherent and modularized testing framework within Flink's ecosystem. This reformation was prompted by the introduction of the flink-test-utils-junit project, which aimed to consolidate common test utilities like the TestLogger and retry rules, previously scattered across various projects. By centralizing these utilities, they became readily accessible to other projects without necessitating a dependency on the entire flink-core test JAR. Concurrently, the flink-test-utils-parent project was created to encapsulate both flink-test-utils-junit and flink-test-utils projects. This not only simplified the dependency graph but also enabled a clearer segregation of test scopes and dependencies. Moreover, this restructuring facilitated the movement of certain test cases between projects and allowed for the inlining of simple utility functions to further streamline the test JAR dependencies. In essence, this consolidation fostered an environment where FlinkStateBackend could depend on FlinkCore in a more organized fashion, enhancing maintainability and clarity in the testing domain, and reinforcing the integrity of Flink's test infrastructure.

Which	Package org.apache.flink.statebackend Depends on Package org.apache.flink.core
Who	Committer: Stephan Ewen
When	PR [FLINK-3359] committed on Jun. 10, 2016
Why	The FlinkStateBackend underwent a structural reorganization to streamline Flink's testing framework. By introducing the flink-test-utils-junit project, essential test utilities such as TestLogger and retry rules were centralized, making them easily available across projects without the need for a full flink-core test JAR dependency.

Table 2: State-Backend depends on FlinkCore sticky note.

3.1.4.6 Dependency: FlinkTable $\xrightarrow{\text{Uses}}$ FlinkClient

The interdependency between FlinkTable and FlinkClient, illustrated in Table 3, was strategically forged to equip the FlinkClient with enhanced capabilities, specifically the parsing of Python command-line options. This integration was imperative to accommodate the evolving needs of FlinkTable, which serves as a comprehensive table API for both batch and streaming operations within Flink. The objective was to facilitate a more robust support system for Python UDFs within the SQL Client environment, a feature increasingly sought by data engineers and scientists.

With the integration of Python configuration parsing, developers now have the agility to pass various Python-specific settings directly through the CLI when initializing SQL Client sessions. This advancement enables a seamless fusion of Python UDFs into SQL workflows, ensuring that these functions are efficiently managed and executed within Flink's operational context. The symbiosis of FlinkTable and FlinkClient thus empowers users to exploit the full potential of Python within Flink, unlocking new paradigms of data transformation and analysis.

3.1.4.7 Dependency: FlinkTable $\xrightarrow{\text{Uses}}$ FlinkStateBackend

The FlinkTable module has been updated to use FlinkStateBackends directly for unit testing (Shown in Table 4), a move prompted by the removal of Scala dependencies from the flink-streaming-java and

Which	Package org.apache.flink.table Depends on Package org.apache.flink.client
Who	Committer: Wei Zhong Reviewer: Diane Fu
When	PR [FLINK-17612] committed on May 11, 2020
Why	The dependency between FlinkTable and FlinkClient was established to enhance FlinkClient with the capability to parse Python command-line options. This was necessary because FlinkTable, which provides a unified table API for batch and stream processing, needed to support configurations for Python UDFs (User-Defined Functions) within the SQL Client environment.

Table 3: FlinkTable depends on FlinkClient sticky note.

flink-cep modules. This integration is critical for verifying the reliability of FlinkTable's state management, particularly for streaming processes where data states are continuously updated. The inclusion of the RocksDB state backend within FlinkStateBackends ensures that state management is both efficient and effective, capitalizing on RocksDB's high performance and durability in dynamic data environments.

This update signifies a meticulous and proactive step taken by the Flink development team to maintain system integrity and performance post-updates. It reflects a commitment to quality, ensuring that the state management capabilities of FlinkTable remain robust and resilient in the face of architectural changes. The deliberate testing and integration process aims to prevent any disruptions or regressions in functionality, thereby preserving the user experience and trust in Flink's streaming capabilities.

Which	Package org.apache.flink.table Depends on Package org.apache.flink.statebackend
Who	Committer: Chesnay Schepler Reviewer: Matthias Pol
When	PR [FLINK-24018] committed on Oct. 25, 2021
Why	The FlinkTable module now integrates FlinkStateBackends, particularly the RocksDB backend, for unit testing to ensure robust state management, especially post-removal of Scala dependencies from flink-streaming-java and flink-cep. This revision guarantees the reliability of data handling during streaming without the Scala components.

Table 4: FlinkTable depends on State-Backend sticky note.

3.1.4.8 Dependency: FlinkTable $\xrightarrow{\text{Uses}}$ FlinkConnectors

The dependency for this section is illustrated in Table 5.The FlinkTable module has been refined to expose option classes for connectors and formats as part of its public API, aligning with the evolving Table & SQL API's design ethos. This move was carried out to ensure compatibility with the new TableDescriptor and FormatDescriptor APIs, facilitating users in discovering and utilizing these options with greater ease. The changes included relocating connector options for the HBase connector, along with the entirety of the datagen, blackhole, and print connector options, into a more consistent package structure. Additionally, the pull request involved correcting annotations to properly reflect the status of API methods as either @Internal or @PublicEvolving, thereby clarifying their intended usage and stability guarantees to developers.

Which	Package org.apache.flink.table Depends on Package org.apache.flink.connectors
Who	Committer: Ingo Burk Reviewer: Timo Walther
When	PR [FLINK-23192] committed on Aug. 4, 2021
Why	The FlinkTable module now publicly exposes connector and format option classes, enhancing API usability and compatibility with new TableDescriptor and FormatDescriptor APIs. This update consolidates connector options into a unified package and clarifies API method annotations for developer guidance.

Table 5: FlinkTable depends on FlinkConnectors sticky note.

3.1.4.9 Dependency: FlinkTable $\xrightarrow{\text{Uses}}$ FlinkStreamingJava

The dependency from FlinkTable to FlinkStreaming, illustrated in Table 6, was enhanced to support DataStreamTable, marking a significant advancement in the Table & SQL API capabilities of Apache Flink. This enhancement included the addition of Java and Scala stream translators, the introduction of DataStream rules for operations like calc and scan, and the creation of a robust testing suite including streaming test utilities. Furthermore, support for streaming union operations was added, expanding the API's functionality to include streaming-specific operations.

Additionally, the change incorporated a shift in the code generation from the calc rule to the calc node, streamlining the process and removing unnecessary rules. This refactoring effort also aimed to unify code between dataset and DataStream translation, reducing redundancy and improving the overall codebase maintainability. By making these changes, FlinkTable significantly increased its streaming data handling capabilities, allowing for more complex and efficient data processing pipelines within the Flink ecosystem.

3.1.4.10 Dependency: FlinkTable $\xrightarrow{\text{Uses}}$ Core

The evolution of FlinkTable continued with its divergence towards FlinkCore to bolster the SQL Client's capabilities (Shown in Table 7), as detailed in the implementation of FLIP-24. The goal was to enhance the user experience with streaming SQL by introducing a set of foundational features. These features included a new CLI component for configuration file reading, providing pre-registered table sources and job parameters, alongside an executor for pre-flight in-

Which	Package org.apache.flink.table Depends on Package org.apache.flink.streaming
Who	Committer: Vasiliki Kalvari Reviewer: Timo Walther
When	PR [FLINK-3547] committed on Mar 7, 2016
Why	The FlinkTable to FlinkStreaming dependency was updated to include DataStreamTable support, enhancing Flink's SQL API with stream translators, DataStream rules, and streaming test utilities. It also added union operations and streamlined code generation, improving maintainability and data processing efficiency.

Table 6: FlinkTable depends on FlinkStreaming sticky note.

formation retrieval and CLI SQL parsing for commands like SHOW TABLES and DESCRIBE TABLE.

Furthermore, the scope of FlinkTable was expanded to support streaming append queries and their submission to the executor. This allowed for executing SQL statements stored in local files and handling result collection directly on the CLI side, streamlining the process for users. The integration of such features marked a significant step in the convergence of FlinkTable and FlinkCore, reflecting a deliberate move to create a more intuitive and powerful interface for Flink's streaming SQL capabilities.

Which	Package org.apache.flink.table Depends on Package org.apache.flink.core
Who	Committer: Timo Walther Reviewer: Fabian Hueske
When	PR [FLINK-8607] committed on Dec. 7, 2017
Why	FlinkTable evolved to integrate with FlinkCore, enhancing the SQL Client per FLIP-24's plan. This update introduced essential features such as a new CLI component for reading configuration files, streamlined execution of SQL commands, and support for streaming queries. These improvements simplified user interactions with Flink's streaming SQL and marked a significant step towards a more user-friendly and powerful system.

Table 7: Flink table depends on Flink core sticky note.

3.1.4.11 Dependency: FlinkStreaming.java $\xrightarrow{\text{Uses}}$ Core

FlinkStreaming.java utilizes Flink Core as illustrated in the sticky note in Table 8. In this dependency, the FlinkStreaming.java component utilizes the Path and Input Split classes from Core to enable reading text files in distributed file systems. This commit is part of a PR dedicated to fixing a bug with ByteBuffer wrapping that results in an IndexOutOfBounds exception. This is a crucial low-level feature in Apache when handling unbounded data streams as they are packed into bytes that need buffers.

Which	Package org.apache.flink.streaming Depends on Package org.apache.flink.core
Who	Committer: szape
When	PR [FLINK-1327] Commited On Dec 15, 2014
Why	Streaming imported functionalities from core to help with reading text files in distributed file systems. Two main classes were imported, Path and InputSplit.

Table 8: Flink Streaming depends on Flink Core sticky note.

3.1.4.12 Dependency: FlinkStreaming.java $\xrightarrow{\text{Uses}}$ Connector

The dependency between streaming and connector exists, as illustrated in the sticky note in Table 9, because streaming utilized a sink existing in connector. StreamingFileSink was a sink that placed input elements into files that were categorized into buckets. That sink however was programmed to work with batch data stream inputs, therefore the commit above aims to deprecate that and replace it with a new sink labelled "FileSink" which aims to support both bounded and unbounded datastreams.

This change is vital as it helps expand the functionalities of the sink in Connector to support both batch and continuous data streams further empowering the capabilities of apache flink.

Which	Package org.apache.flink.streaming Depends on Package org.apache.flink.connector
Who	Committer: Martjin Visser
When	PR [FLINK-27188] Committed on Jul 22, 2022
Why	Deprecates StreamingFileSink and replaces it with FileSink to support bounded and unbounded data streams.

Table 9: Flink Streaming depends on FlinkConnector sticky note

3.2 Runtime Subsystem Level View

In this section the runtime subsystem is reviewed and analysed. The discrepancies between our initial conceptual model and the uncovered concrete model are analysed in detail.

3.2.1 Conceptual Model vs Concrete Model

When comparing our conceptual model (Figure 6) with the concrete model (Figure 7) of Apache Flink's architecture its noticed that there are many divergences. These are inferred from mapping on the LSEdit recovery and confirmed by investigating source code. It's also noticed that some subsystems we defined in our conceptual model are absorbed into other subsystems in the concrete model as shown in Figure 8.

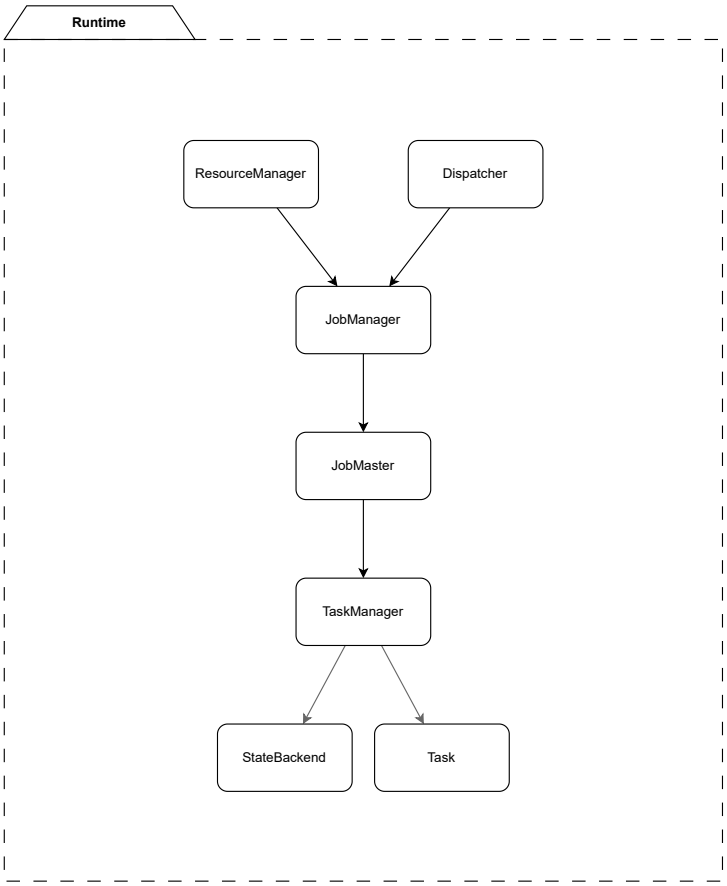

Fig. 6: Conceptual model of Apache Flink-Runtime

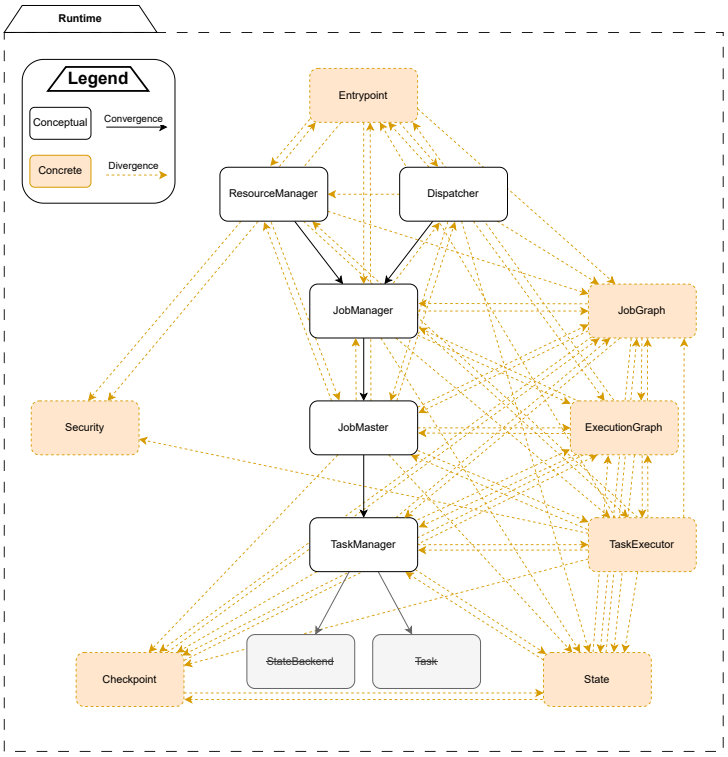

Fig. 7: Concrete model of Apache Flink-Runtime.

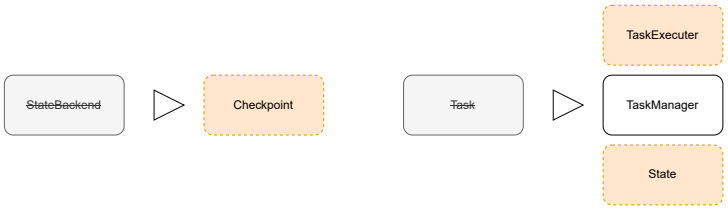

Fig. 8: Converging concerns of Apache Flink-Runtime.

3.2.2 Reflexion Analysis

3.2.2.1 Dependency: JobMaster $\xrightarrow{\text{Uses}}$ JobManager

The dependency between JobMaster and JobManager was created as shown in Table 10. To ensure task coordination, fault tolerance management, and collective contribution to efficient execution and monitoring of jobs within the ditributed computing framework.

Which	Package org.apache.flink.runtime.jobmaster Depends on Package org.apache.flink.runtime.jobmanager
Who	Committer: Kurt Young Reviewer: Stephan Ewen
When	PR [FLINK-4408] committed on Sept 8th, 2016
Why	JobMasterRunner was introduced by JobManager and ExecutionGraph set up was done.

Table 10: JobMaster depends on JobManager sticky note.

3.2.2.2 Dependency: JobManager $\xrightarrow{\text{Uses}}$ Dispatcher

The JobManager was refactored which caused a divergence towards the Dispatcher to provide a common resource cleaner as shown in Table 11. This refactor was part of the FLIP-6 project.

Which	Package org.apache.flink.runtime.jobmanager Depends on Package org.apache.flink.runtime.dispatcher
Who	Committer: Matthias Pohl Reviewer: David Moravek
When	PR [FLINK-8234] committed on Dec. 15th, 2021
Why	`JobGraphWriter.java` was refactored to extend and implement `LocallyCleanableResource.java` and `GloballyCleanableResource.java`. The job-specific cleanup of the different resources were combined to provide a common resource cleaner taking care of the actual cleanup of all resources.

Table 11: JobManager depends on Dispatcher sticky note.

3.2.2.3 Dependency: ResourceManager $\xrightarrow{\text{Uses}}$ JobMaster

The dependency between ResourceMAnager and JobMaster as shown in Table 12. To ensure efficient job-specific resources, dynamic scaling, fault tolerance, and overall job lifecycle management. Ensuring optimal performance and adaptability.

Which	Package org.apache.flink.runtime.resourcemanager Depends on Package org.apache.flink.runtime.jobMaster
Who	Committer: Till Rohrmann
When	PR [FLINK-7506] committed on Aug 24th, 2017
Why	To enable JobMaster automatic fencing of all messages. JobMasterId is introduced as a token to improve safety when passing multiple fencing tokens around.

Table 12: ResourceManager depends on JobMaster sticky note.

3.2.2.4 Dependency: JobMaster $\xrightarrow{\text{Uses}}$ ResourceManager

The dependency between JobMaster and ResourceManager as shown in Table 13. This addition was part of the FLIP-6 project. This dependency establishes robust foundation for resource allocation, dynamic scaling, fault tolerance, and global resource management, to ensure synchronized and efficient job execution across the cluster.

Which	Package org.apache.flink.runtime.JobMaster Depends on Package org.apache.flink.runtime.ResourceManager
Who	Committer: Till Rohrmann Reviewer: Stephan Ewen
When	PR [FLINK-4529] committed on Aug 29th, 2016
Why	TaskExecutor, JobMaster, and ResourceManager are out of rpc packager. Now they are contained in packages on runtime level

Table 13: JobMaster depends on ResourceManager sticky note.

3.2.2.5 Dependency: JobMaster $\xrightarrow{\text{Uses}}$ Dispatcher

The dependency between JobMaster and Dispatcher was created as shown in Table 14. The JobExecutionResults needed to be cached in the Dispatcher after the JobManagerRunner was finished. This addition was part of the FLIP-6 project.

3.2.2.6 Dependency: Dispatcher $\xrightarrow{\text{Uses}}$ JobMaster

The dependency between Dispatcher and JobMaster was created as shown in Table 15. The Dispatcher was initially created to handle job submissions and listing of jobs. This addition was also part of the FLIP-6 project.

Which	Package org.apache.flink.runtime.jobmaster Depends on Package org.apache.flink.runtime.dispatcher
Who	Committer: GJL Reviewer: Till Rohrmann
When	PR [FLINK-8234] committed on Dec. 19th, 2017
Why	`JobResult.java` was created and used `Dispatcher.java`. In order to serve the JobExecutionResults the Dispatcher caches them after the JobManagerRunner has finished.

Table 14: JobMaster depends on Dispatcher sticky note.

Which	Package org.apache.flink.runtime.dispatcher Depends on Package org.apache.flink.runtime.jobmaster
Who	Committer: Till Rohrmann Reviewer: Zhihong Yu & Chesnay Schepler
When	PR [FLINK-7103] committed on July 4th, 2017
Why	`Dispatcher.java` was created and used `JobManagerRunner.java`. The dispatcher is responsible for receiving job submissions, persisting the JobGraphs, spawning JobManagerRunner to execute the jobs and recovering the jobs in case of a master failure.

Table 15: Dispatcher depends on JobMaster sticky note.

3.3 Checkpoint Subsystem Level View

In this section, the checkpoint subsystem, which is a subsystem of the runtime subsystem, is reviewed and analysed at a conceptual level and concrete level. We present three high-quality figures as reference models for this subsystem as a concrete structure. This is followed up by interactions with a focus on outward and inward dependencies. Finally, we analyse the discrepancies between our initial conceptual model and the uncovered concrete model.

3.3.1 Conceptual Model

At the conceptual level, the checkpoint subsystem was initially visualized as a behavioural component implicitly present in Apache Flink. In other words as a programming paradigm used throughout the framework to maintain state snapshots. This was an oversight as the documentation [13] does not show checkpoint components as specific subsystems. Concrete architectural recovery and examination of the source code shows otherwise. Moreover, re-examination of the Flink whitepapers also shows otherwise [4, 5]. Provided are two conceptual architectural diagrams from these whitepapers. Figure 9 shows the interaction of the checkpointing subsystem with runtime components. The TaskManager subsystem executes the storage of snapshots of the physical tasks into the file systems specifically for checkpointing. Figure 10 shows the process of snapshotting occurring at regular intervals over the processing of a data stream.

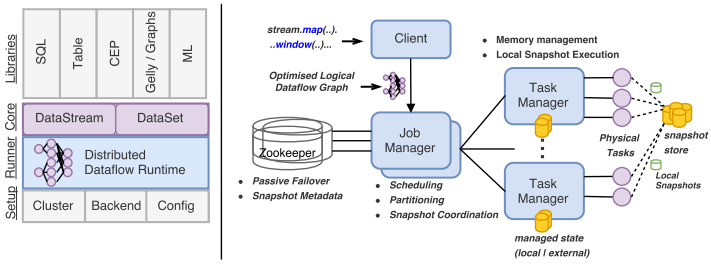

Fig. 9: Apache Flink conceptual view of original system and state snapshot process [5].

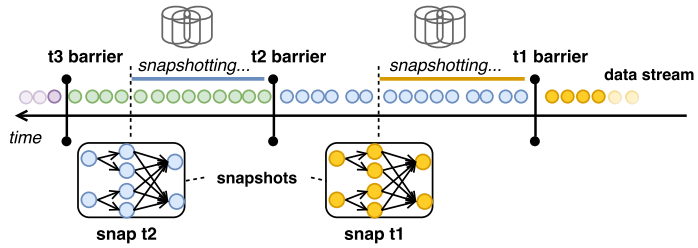

Fig. 10: Apache Flink dataflow of checkpoint storage in an asynchronous context [4].

3.3.2 Concrete Model

For the concrete model the majority of the dependencies of the checkpoint subsystem are within the runtime subsystem. However, there are still 4 total discovered interactions at the top level. These are inferred from mapping on the LSEdit recovery and confirmed by investigating source code. This view can be seen in Figure 11. Figure 12 shows the checkpoint subsystem with an open view of its subsubsystems and the various dependencies the other runtime components have on the checkpoint system. Figure 13.

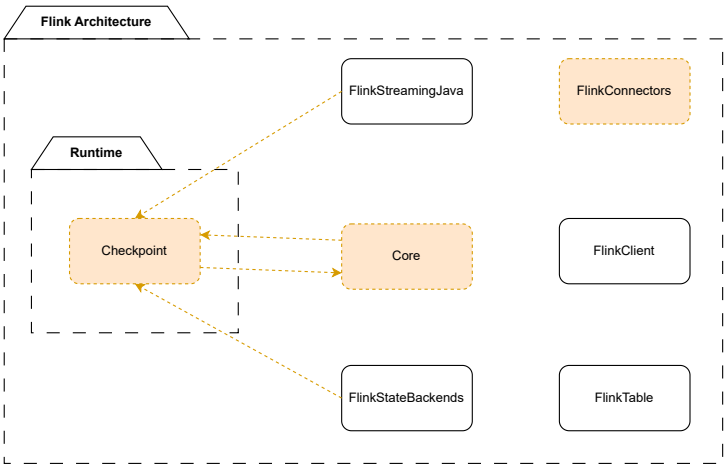

Fig. 11: Apache Flink top-level subsystem dependencies in relation to the Checkpoint Subsystem.

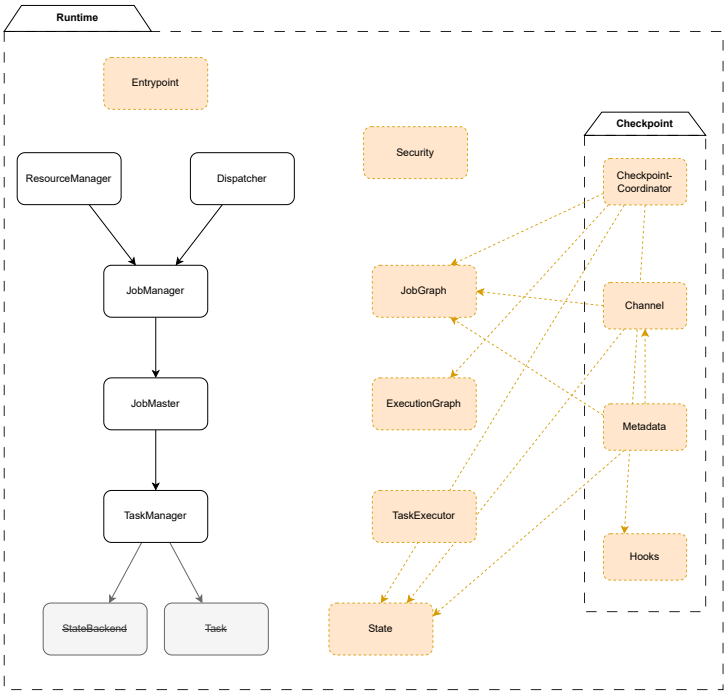

Fig. 12: Apache Flink runtime-level outward dependencies for Checkpoint Subsystem and its subsubsystems.

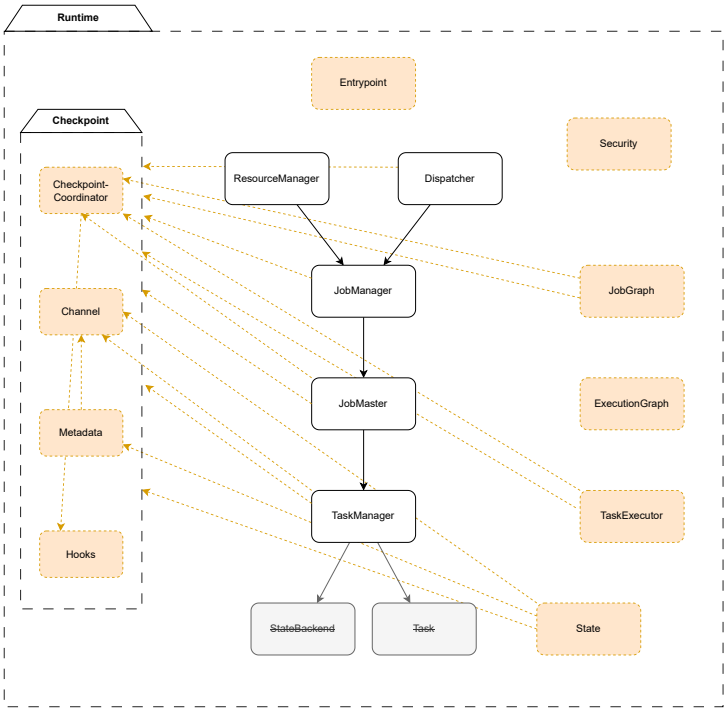

Fig. 13: Apache flink runtime-level inward dependencies for other subsystems on Checkpoint Subsystem and its subsubsystems.

3.3.3 Reflexion Analysis

The key mismatch between the original conceptual and the concrete models was that the checkpoint subsystem was not present in

the conceptual model. Therefore the focus will be on divergences as all interactions with the checkpoint system are non existent when using the conceptual model. The following sticky notes will show several dependencies by the checkpoint subsystem as well as upon the checkpoint subsystem. There are several key dependencies for the checkpoint subsystem and what follows is not meant to be exhaustive by any means. The dependencies will all be listed and described with sticky notes in order of commit time. Lastly, a revised conceptual architecture including checkpoint will be presented near the end.

3.3.3.1 Dependency: ExecutionGraph $\xrightarrow{\text{Uses}}$ CheckpointCoordinator

This dependency is shown as a sticky note in Table 16. This is one of the earliest design decisions in Flink. This was an early integration of the checkpoint coordinator and the execution graph as concrete classes. Both were created in this pull request. Moreover, this allowed the checkpointing system to coordinate better with the execution graph.

Which	Package org.apache.flink.runtime.executiongraph Depends on Package org.apache.flink.runtime.checkpoint
Who	Committer/Merger: Stephen Ewan (regular contributor) Reviewer: None
When	PR [FLINK-1953] committed May 5, 2015
Why	`ExecutionGraph.java` `CheckpointCoordinator.java`. Introdces an abstraction to separate tasks that start checkpoint and need to acknowledge checkpoints.

Table 16: ExecutionGraph Subsystem's dependency on CheckpointCoordinator Subsubsystem sticky note.

3.3.3.2 Dependency: Metadata $\xrightarrow{\text{Uses}}$ JobGraph

This dependency is show on Table 17. In Flink, the operators are stateful and need to be snapshotted as part of checkpoints. This dependency exists to allow the Metadata serialization and deserialization system to obtain the ID of Operators to be. This is a step related to creating and parsing metadata for the checkpointing process so that its correctness is ensured.

3.3.3.3 Dependency: Metadata $\xrightarrow{\text{Uses}}$ Channel

This dependency is in Table 18. There are two types of checkpoints in Flink: aligned and unaligned. For unaligned checkpoints the channel state as well as operator state needs to be stored. In this change that was addressed and this dependency was a part of the pull request. Recall that in flink a channel is the result of an operator.

3.3.3.4 Dependency: Channel $\xrightarrow{\text{Uses}}$ JobGraph

This dependency is shown in Table 19. This dependency was introduced during the implementation of an API design to persist channel state. Specifically, the implementation focused on enabling the

Which	org.apache.flink.runtime.checkpoint.metadata Depends on org.apache.flink.runtime.jobgraph
Who	Committer/Merger: Stephen Ewan (regular contributor) Reviewer: Jiangjie (Becket) Qin (regular contributor)
When	PR [FLINK-16177] committed on March 5, 2020
Why	`MetadataV3Serializer.java` `OperatorID.java`. Integration of Operator with checkpointing triggering & committing as operators are also stateful & need to be snapshotted in the checkpoints. The serializer uses the OperatorID class to mediate the snapshotting of the associated operator

Table 17: Checkpoint Metadata Subsubsystem's MetadataV3Serializer's dependency on the Jobgraph Subsystem's OperatorID sticky note.

Which	org.apache.flink.runtime.checkpoint.metadata Depends on org.apache.flink.runtime.checkpoint.channel
Who	Committer/Merger: Roman Khachatryan (regular contributor) Reviewer: Piotr Nowojski (PMC Committee)
When	PR [FLINK-16513] committed on March 23, 2020
Why	`ChannelStateHandleSerializer.java` `InputChannelInfo.java`. Integration of channel subsubsystem and metadata subsystem. This dependencies is involved in exchange of information regarding channel states through serialization and deserializaiton of metadata associated with the channels.

Table 18: Checkpoint Channel Subsubsystem's ChannelStatehandleSerializer dependency on the Channel Subsubsystem's InputChannelInfo sticky note.

persistence of channel state for unaligned checkpoints within Flink. Unaligned checkpoints represent an important feature, allowing tasks to maintain processing continuity while a checkpoint is in progress. Refactoring was also involved to enhance the overall structure and efficiency of the codebase.

3.3.3.5 Dependency: Metadata $\xrightarrow{\text{Uses}}$ Channel

This dependency is the sticky note in Table 20 This is a change to improve checkpoint metrics displayed on the Flink web UI. The change allows for an update of checkpoint statistics even if a checkpoint fails. This is a quality of life improvement to enhance the resilience of the checkpoint guarantee provided by flink.

3.3.3.6 Dependency: StateBackends $\xrightarrow{\text{Uses}}$ Checkpoint

This final dependency is shown in Table 21. introduced a change that exposes an API for the control of the binary format for the save-

Which	org.apache.flink.runtime.checkpoint.channel Depends on org.apache.flink.runtime.jobgraph
Who	Committer: Roman Khachatryan Reviewer: zhijiang
When	PR [FLINK-16744] commited on March 29, 2020
Why	`ChannelStateWriteRequest.java` `JobVertexID.java`. Implement channel state persistence for unaligned checkpoints. Unaligned checkpoints are a feature in Flink that allows tasks to continue processing while a checkpoint is ongoing.

Table 19: Dependency of Channel Subsubsystem's ChannelStateWriteRequest on JobGraphs's Subsystem's JobVertexID sticky note.

Which	org.apache.flink.streaming.runtime.tasks Depends on org.apache.flink.runtime.checkpoint.channel
Who	Committer/Merger: Roman Khachatryan (regular contributor) Reviewer: Piotr Nowojski (PMC Committee)
When	PR [FLINK-19462] committed on Jan 25, 2020
Why	`AsyncCheckpointRunnable.java` `CheckpointMetrics.java`. This change was to improve the investigation of checkpoint failures by creating metrics even in the state of checkpoint failure.

Table 20: Flink Streaming subsystem AsyncCheckpointRunnable dependency on the Checkpoint subsystem's CheckpointMetrics sticky note.

point. The native binary of the state backend is what is stored and this is a quality of life improvement that allows for the faster restoration of save points.

Which	org.apache.flink.contrib.streaming.state.snapshot Depends on org.apache.flink.runtime.checkpoint
Who	Committer: Dawid Wysakowicz Reviewer: Anton Kalashnikov
When	PR [FLINK-25744] Support native savepoints committed Jan 31, 2022
Why	`CheckpointCoordinator.java` `SavepointFormatType.java`. CheckpointCoordinator has a methods with references to the SavepointFormatType object that specifies the type of binary format a snapshot is stored as. This is involved in a overall workflow to improve the speed of recovery.

Table 21: Dependency of the StateBackends' RocksDB subsubsytem on Checkpoint Subsystem sticky note.

4 Revised Conceptual Architecture

In the revision of the conceptual architecture, the subsystems were grouped and abstracted as larger subsystems based on overall grouping in the flow of events that take place in data streaming. This revision can be seen in Figure 14. The FlinkClient is the mediator from requests from applications implementing Flink. It enables the setup of the environment and submits jobs as well as sources streams. The FlinkStreaming sends its data streams to the FlinkConnectors, which streams to the FlinkTable APIs and the Dataset. These facilitate task execution for the Task subsystem. On the Runtime side jobs are mediated by the job manager and make their way in through the dispatcher. JobManager is grouped with the ResourceManager, JobMaster, and Dispatches. These assign tasks to the TaskManager. The FlinkStreaming subsystem uses the checkpoint subsystem to collect checkpoint metrics. Finally data is stored in the StateBackend and the Checkpoint subsystem manages the storage and deletion of state snapshots in the StateBackend.

5 Use Case Description

In this use case diagram of Figure 15, we've delineated a data processing design that synergizes Apache Kafka with Apache Flink, proposed at governing and analyzing real-time data, like interactions generated by a game player. Kafka serves as the base for message brokering, catching player events and expediting analytical queries. Concurrently, Flink is assigned with ingesting this data, regulating it by user-specific identifiers or query attributes, and enforcing essential data transformations and analyses.

Flink's design incorporates a sturdy state management and checkpointing system, analogous to version control in software development, that ensures data processing continuity and fault tolerance. These state snapshots are meticulously maintained in disk storage, assuring against data loss in case of system failures. Post-processing, Flink circulates the assembled data back into Kafka, that could then be passed down for downstream processing or real-time data visualization.

Using the state diagram in Figure 16, we see that rather than being a distinct architectural element, the checkpoint subsystem was seen as implicit element within Apache Flink in the conceptual model that served as the basis for the initial state diagram. The subsystem was eventually discovered to interact more deeply than originally illustrated with multiple runtime components, and hence did not fully reflect the documented architecture or the actual source code. These misunderstandings are fixed in the new state diagram, which includes a thorough overview of the interactions of the checkpoint subsystem and highlights its crucial place in the design. The discrepancy analysis phase, which compared the original understanding with the actual implementation and documentation, produced a more accurate and refined representation of the Flink's architecture, which is directly responsible for this adjustment. A detailed representation of the checkpoint subsystem is included in the updated state diagram, demonstrating how it is more than just a background operation but rather a sophisticated system that interacts with several different parts including the TaskManager, JobManager, and StateBackend. This degree of depth was attained by looking closely at source code and actual architectural components, which resulted in a thorough comprehension of the dependencies and interactions within the subsystem.

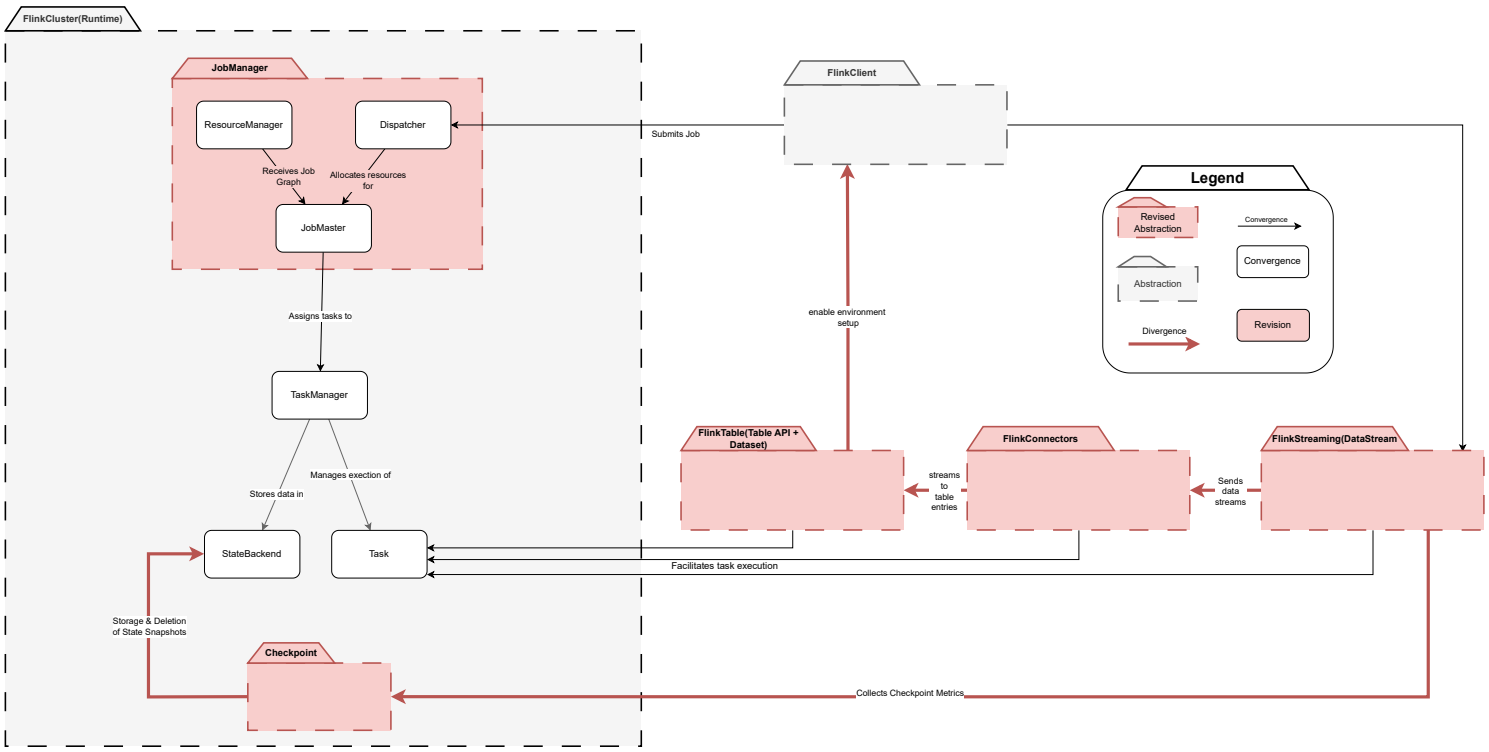

Fig. 14: Revised Conceptual Architecture for overall system.

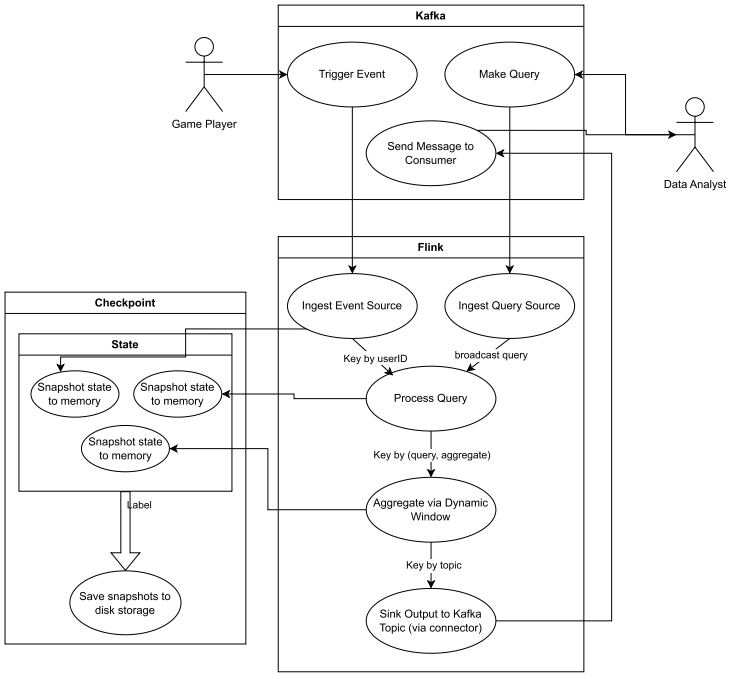

Fig. 15: Data Analytics Checkpointing Use Case Diagram Update.

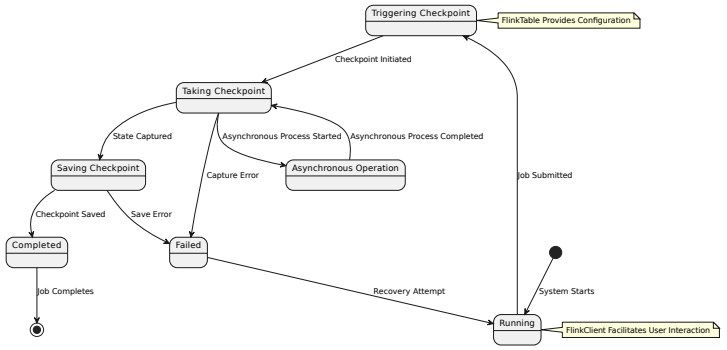

Fig. 16: Data Analytics Checkpointing State Diagram.

6 Limitations

Some pull requests are documented inaccurately, specifically in instances where a new dependency was introduced without proper mention. This oversight has led to confusion in pull request conversations between contributors and reviewers. Additionally, certain subsystems exhibit highly complex directory structures, making them challenging to parse and read. Given the extensive number of files and an even greater multitude of dependencies within the codebase, it is plausible that not all dependencies have been adequately mapped. This complexity introduces a potential risk, as overlooking dependencies could result in unintended consequences and hinder the overall understanding of the codebase. Finally, transient dependencies are not efficiently mapped; specifically, the transient dependencies involving the Checkpoint Subsystem.

7 Lessons Learned

The runtime concrete architecture exhibited significant divergences from its conceptual counterpart, with variations being notably more frequent than outright absences. In navigating this intricate landscape, tools such as GitKraken and GitLens prove to be invaluable, offering efficient means of analyzing authorship and attributing changes. Delving into the fabric of project evolution, perusing conversations within commit logs, and exploring details in Jira issues become essential practices. Additionally, tracing the associated identifier codes provides a nuanced understanding of the design decisions interwoven into the development process. These tools and methodologies collectively empower a more comprehensive comprehension of the intricate interplay between conceptualization and realization in the runtime architecture.

8 Conclusion

In conclusion, a revised conceptual architecture has been presented in accordance to the results found in the reflexion analysis of Apache Flink's architectural recovery. There are many models that highlight what the system does. However, the system evolves faster than it can be tracked and for this reason there are many possible ways to model it.

References

[1] D. Gopstein, J. Iannacone, Y. Yan, L. DeLong, Y. Zhuang, M. K.-C. Yeh, and J. Cappos, "Understanding misunderstandings in source code," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, Aug. 2017, pp. 129–139. [Online]. Available: https://dl.acm.org/doi/10.1145/3106237.3106264

[2] G. Avelino, L. Passos, A. Hora, and M. T. Valente, "Measuring and analyzing code authorship in 1+118 open source projects," *Science of Computer Programming*, vol. 176, pp. 14–32, May 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642318300388

[3] J. B. Tran and R. C. Holt, "Forward and reverse repair of software architecture," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, ser. CASCON '99. Mississauga, Ontario, Canada: IBM Press, Nov. 1999, p. 12.

[4] P. Carbone, A. Katsifodimos, Kth, S. Sweden, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink™: Stream and Batch Processing in a Single Engine," *IEEE Data Engineering Bulletin*, vol. 38, Jan. 2015.

[5] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in Apache Flink®: consistent stateful distributed stream processing," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1718–1729, Aug. 2017. [Online]. Available: https://dl.acm.org/doi/10.14778/3137765.3137777

[6] [Online]. Available: https://github.com/Anselmo21/EECS4314-FlinkForce/tree/main/Assignment_1

[7] [Online]. Available: https://github.com/Anselmo21/EECS4314-FlinkForce/tree/main/Assignment_2

[8] "Understand: The Software Developer's Multi-Tool." [Online]. Available: https://scitools.com/

[9] "The Landscape Editor LSEdit." [Online]. Available: https://www.swag.uwaterloo.ca/lsedit/

[10] A. Hassan and R. Holt, "Using development history sticky notes to understand software architecture," in *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.*, Jun. 2004, pp. 183–192, iSSN: 1092-8138. [Online]. Available: https://ieeexplore.ieee.org/document/1311060

[11] G. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: bridging the gap between design and implementation," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 364–380, Apr. 2001, conference Name: IEEE Transactions on Software Engineering. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/917525

[12] "Gitlens." [Online]. Available: https://marketplace.visualstudio.com/items?itemName=eamodio.gitlens

[13] "Overview," section: docs. [Online]. Available: //nightlies.apache.org/flink/flink-docs-release-1.17/docs/dev/table/sql-gateway/overview/