Apache Flink Concrete Architecture: 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 project we extract the concrete architecture of Apache Flink using the Scientific Toolworks, Inc. software Understand and the open source LSEdit tool. We present a novel pipeline for this extraction process to remove symbolic linking and yield an accurate concrete architecture showing proposed subsystems and subsystem interactions. We then explain the functionality of these subsystems with a focus on the Checkpoint subsystem. We also show the architectural styles and design patterns confirmed to be present.

1 Introduction

This project builds on the previous work done by our team, in which a conceptual architecture for Apache Flink was proposed. Conceptual architectures are based on documentation and research papers about the software system in question. It is a common problem that these sources are not up to date or accurate representations of the true architecture. For this reason researchers need to use tools that search through project directories to associate and label subsystems based on inter-file dependencies.

1.1 Research Goal

Our goal is to take the jgrok lse-edit software extraction process and propose a novel pipeline to give results all in one step. This entails creating a modern scripting workflow for reflexion analysis.

2 Concrete Extraction & Derivation Process

2.1 Overview

To extract the concrete architecture for Apache Flink we follow the process shown in Figure 1. First we extract the relations from the Flink source code using Understand [1]. This provides the relations in `.csv` format. We then use the provided `transformUnderstand.pl` script to convert it to the `.raw.ta` format. Next we use custom python scripts to generate a .contain file. Details on our custom scripts are discussed in Section 2.2. The `.contain` file along with the `.raw.ta` file is then used by the provided `createContainment.bat` script to generate the `.con.ta` and `.ls.ta` files which are used by lsEdit.

2.2 Custom scripts

2.2.1 Subsystem Tree (system_structure.json)

As the subsystems are defined in a hierarchical structure (via the source code directory), we made a JSON file that represents the tree structure of the Flink system, where the root node refers to the whole system. We defined the JSON externally because in order to arrive at a final concrete architecture, many iterations were needed as new relations and components were discovered. Having the JSON file made it so that anyone could edit the structure and re-run the pipeline without having to know exactly how the scripts work.

Each node within this tree is a JSON object which has name, pattern, and children attributes. The name refers to the name of the subsystem we are trying to impose/render in lsEdit. Rather than using something like glob, we simply check if a directory contains the pattern and if so, match it. This means that root patterns will also match children that may be in lower subsystems, and so it is important that we match the subsystems from leaves up to the node via depth first search. The children attribute simply represents the children of that node and is an array of objects. If children are not defined, it is assumed to be a leaf node.

2.3 Symbolic Linking

When talking with the professor we were told that it would be an issue if any symbolic linking existed in the project, and would throw errors when we attempted to process it with JGrok. Thus, we wrote a script `find_symlinks.py` which recursively walks the directory and ensures that no links exist. After running this on the Flink system no links were found, so we proceeded with that assumption.

2.3.1 .contain Generation (gen_contain.py)

To generate the containment file, we first read in the `.raw.ta` file provided to us. It is also possible to load the code into Understand, extract the dependencies into `.csv`, and then use the given perl script to transform that into a `raw.ta` file. We opted to use the provided `raw.ta` file to remove any potential errors in that process. Once we read the `raw.ta` file, we look at the lines beginning with `$INSTANCE` as these are the concrete entities in our system (cLinks represent the dependency relations). For each of these lines, we perform the subsystem matching process. This is a depth first search implemented recursively which matches the patterns from the leaves up to root. If the instance does not match any pattern it is simply ignored. If it does match a directory pattern, we impose the subsystem containment by adding `contain {subsystem name} {contained file}` to the containment file.

2.3.2 .raw.ta Cleanup (process_ta.py)

To finish up, lsEdit will throw errors if there are files in the `.raw.ta` file that do not have any containment specified by the `.contain file`. Thus, we use `process_ta.py` to remove any unneeded lines from the `.raw.ta` file. This is done by checking both the concrete instances and links. If the concrete instances don't

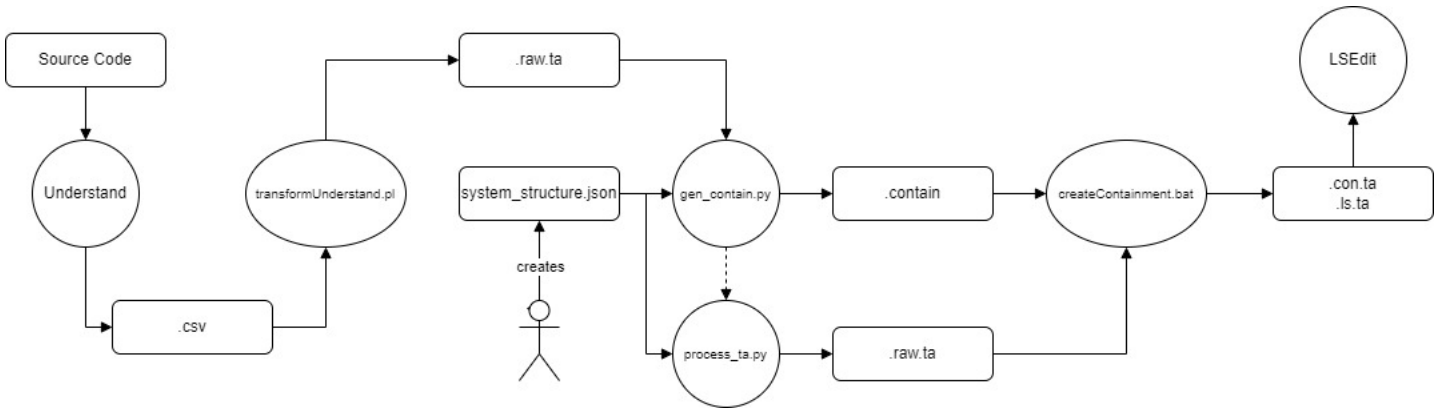

Fig. 1: Concrete architecture extraction flow chart

match any patterns, they are removed. Likewise if caller of the linking relation does not match any of the patterns being imposed, it should be ignored too. As we have added new subsystems, instances of them must also be added to the `.raw.ta` file. This is done by adding a new line to the top of the file in the form of `$INSTANCE {subsystem name} cSubSystem`.

2.4 Pipeline

After running the process several times during iteration, we realized that this process should be automated to improve our ability to iterate and share the code amongst the team. For this reason, `pipeline.py` was added. The pipeline file cleans up any previously generated files, copies in a fresh `.raw.ta` file (the one provided to us), and then runs the containment generation and cleanup processes, followed by imposing the containment with JGrok and lastly running lsEdit. This way we had a single command that can be ran from the CLI to do the whole process as the subsystem structure JSON is modified.

3 High-Level Concrete Architecture

The top level concrete architecture can be seen in the high-level LSEdit view

3.1 Core.ss

The Core.ss module represents the fundamental building blocks and utilities upon which the Flink system is built [2]. This core module encapsulates the primary functionalities required for distributed stream and batch data processing, and many other core functionalities of Flink. The module comprises of essential utility classes, enums, and interfaces such as SourceEvent.java, which is used for event-based source operations, and plays a crucial role in managing data input from various sources into Flink. Components outside the core.ss module often rely on its foundational structures and utilities to implement and extend Flink's capabilities.

3.2 FlinkClient.ss

FlinkClient.ss serves as a pivotal intermediary, bridging the gap between end-users and the Flink cluster [3]. At its core, this module is responsible for simplifying and facilitating user interactions with the cluster. More importantly, there exists key components dedicated to the translation of Flink's data structures, ensuring that user-defined pipelines are appropriately converted for runtime execution. This aspect of the module is integral to the preparation and initialization phases, enabling user workflows to be understood and executed by the Flink system. By abstracting these complexities, users are liberated from the detailed mechanics, allowing them to concentrate on their primary data flow tasks.

A notable feature of FlinkClient.ss is its adaptability to various Flink cluster configurations. Whether addressing the needs of expansive production setups or more constrained local development environments, the module showcases versatility. This adaptability ensures a smooth experience for users, irrespective of the scale and nature of their cluster environment. Furthermore, by leveraging HTTP REST requests, the module introduces a layer of compatibility with contemporary web systems, further broadening its utility and integration potential.

From a usability standpoint, the module stands out by offering a suite of user-friendly interfaces, including command-line tools. These tools are meticulously designed to ensure that even those with limited familiarity with Flink's internals can effectively interact with and control their Flink jobs. In essence, the module's primary objective is to encapsulate the inherent complexities of Flink, presenting users with an accessible and efficient platform to navigate and operate within the Flink ecosystem.

3.3 FlinkConnector.ss

Apache Flink offers a module system for its connectors, allowing it to interface with a variety of external systems for data input and output [4]. The FlinkConnector.ss serves as the foundation, providing essential classes and interfaces for other connector implementations. Such a structure ensures that the foundational utilities and functionalities, which are common across different connectors, are available centrally.

FlinkConnector.ss demonstrates Flink's capabilities in handling both data sources and sinks efficiently. The architecture is designed to manage asynchronous operations, ensuring advanced non-blocking data transactions. With a clear emphasis on both data ingestion and dispatch, Flink ensures seamless interaction between

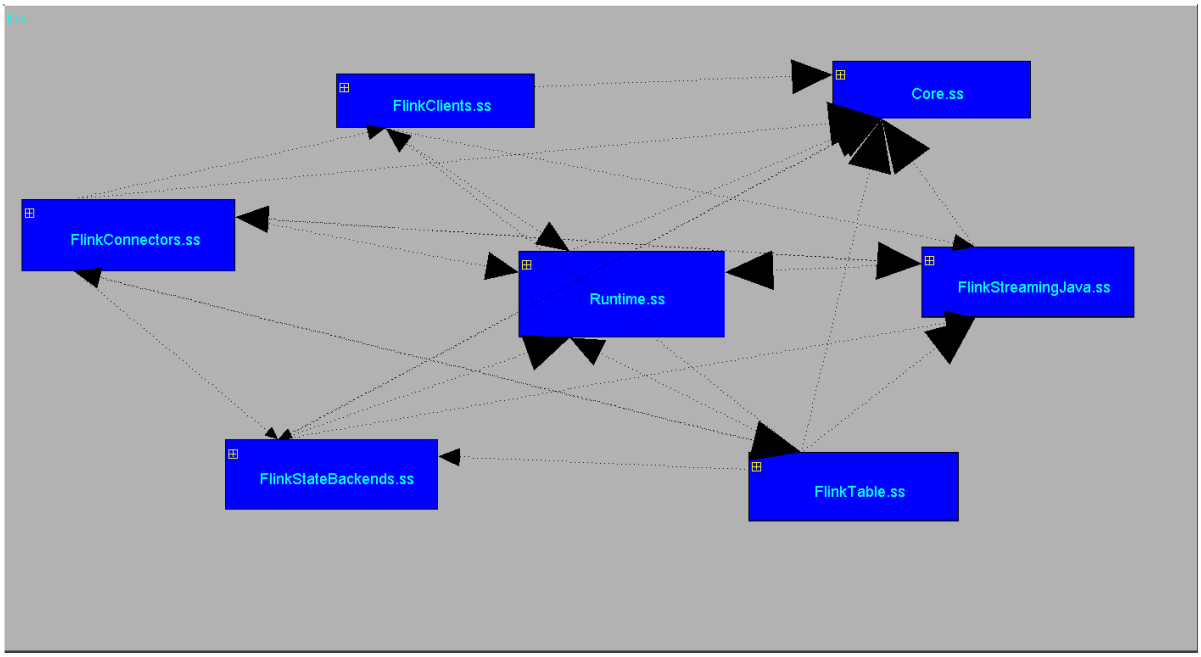

Fig. 2: A top level view of the concrete subsystems resulting from the extraction process

its core system and external data systems through dedicated functionalities for sources and sinks.

FlinkConnector.ss highlights Flink's capabilities in efficiently managing interactions with various data sources and sinks. In this framework, sources are the systems from which Flink ingests data, encompassing databases, message queues, and real-time data streams. Flink's structured architecture facilitates consistent data intake from these sources. Conversely, sinks denote the endpoints to which Flink directs the processed data. This can involve transferring results to cloud storage, updating databases, or relaying notifications. Flink's design ensures a reliable delivery of outputs to these specified destinations.

Additionally, with its efficiency at handling asynchronous operations, Flink's focus on both sources and sinks allows for sophisticated non-blocking data transactions. This translates to a processing system that effectively manages data intake and assures dependable data output, enabling smooth interactions with external data systems.

In the broader context of the Flink ecosystem, connectors are important because after data is ingested from an external system using a source connector, it traverses through Flink's Directed Acyclic Graph (DAG) for processing [5]. Once processed, the data can be dispatched to external systems using sink connectors. The foundational components within the base connector module facilitate this seamless data flow, abstracting intricacies from specific connector implementations and ensuring efficient interaction with the core Flink system.

3.4 FlinkStateBackend.ss

The Apache Flink project includes a top-level subsystem named FlinkStateBackend.ss. This module focuses on state management, which is a cornerstone of stream processing. Within Flink, state backends delineate the methodology and location for state storage and retrieval. They are tailored to meet diverse requirements, such as scalability, resilience against failures, and data persistence. The module houses a plethora of classes and interfaces, each crafting the behavior of varied state backends, including, but not limited to, RocksDB and heap memory.

RocksDB is a high-performance embedded database for key-value data. It's essentially a persistent and embeddable key-value store, originally developed by Facebook. One of the standout features of RocksDB is its ability to handle large amounts of data and sustained read and write operations, making it suitable for applications requiring efficient storage and retrieval mechanisms. In the context of Flink, RocksDB serves as one of the state backends. When Flink jobs require large state sizes that can't be held entirely in memory, the system resorts to the RocksDB state backend. It allows the state to spill over to disk, ensuring that the application doesn't run out of memory, while still maintaining relatively fast access times. This combination of in-memory and on-disk storage grants Flink the flexibility to handle vast, stateful computations without compromising on performance.

Zooming out to view the Flink system, the role of state backend is very important. Their task is to guarantee the dependable recovery of stream processing jobs post any potential failures. The demands of stateful stream processing are manifold - it necessitates a system that's efficient, scalable, and unfailingly reliable in state management. The flink-state-backends module is Flink's response to these challenges. As data courses through Flink, it's intermittently transformed, necessitating the continuous management, update, and storage of its state. By maintaining this state, Flink can offer advanced features such as exactly-once processing guarantees, adept event time processing, and sophisticated windowing capabilities. All these solidify the state backend's standing as an indispensable component in the Flink framework.

3.5 FlinkStreamingJava.ss

Apache Flink is renowned for its ability to process large-scale data streams in real-time, and the streaming subsystem is central to this capability. FlinkStreamingJava.ss is designed to handle continuous data flows, unlike traditional batch processing systems which operate on finite data sets. The primary goal of this component is to process these streams with low latency and high throughput, ensuring timely insights from data [6].

At the heart of FlinkStreamingJava.ss is its support for stateful operations. This means that Flink can maintain and update information across different events in a stream, allowing for more complex and meaningful analyses. For instance, FlinkStreamingJava.ss can track the running average of a data stream or identify trends over sliding windows of time. It incorporates event time processing, which ensures that events are processed in the order they occurred, irrespective of when they arrive. This is crucial for applications where out-of-order data can lead to incorrect results. Furthermore, FlinkStreamingJava.ss supports exactly once processing semantics, ensuring data consistency and reliability, even in the face of potential failures [7].

FlinkStreamingJava.ss seamlessly integrates with the broader Flink ecosystem. The stateful operations in this component leverage the state backends provided by the flink-state-backends module, enabling efficient state management, and ensuring fault tolerance. The streaming data can also be partitioned, windowed, and joined with other streams or batch data sets, making Flink a truly unified data processing system. The results from the stream processing can be pushed to various sinks, be it databases, dashboards, or other storage systems. Additionally, this module can ingest data from a myriad of sources, from traditional message queues like Apache Kafka to file systems, ensuring flexibility and adaptability to various data architectures.

3.6 FlinkTable.ss

FlinkTable.ss is a core component of the Apache Flink ecosystem that focuses on structured data processing. It provides users with table abstractions and functionalities, enabling them to work with both streaming and batch data using SQL queries and table operations, rather than dealing directly with data streams or datasets [8].

This subsystem integrates very easily with SQL capabilities with Flink's powerful stream and batch processing features. By offering SQL interfaces and table APIs, users can perform complex transformations, aggregations, and computations on data, like how they'd interact with a traditional relational database. This abstracts away the complexities of stream processing and allows for more intuitive and declarative data operation.

FlinkTable.ss is interconnected with the broader Flink architecture. While Flink's core provides the foundational capabilities for stateful stream processing and scalable data computations, the FlinkTable.ss extends these capabilities to offer structured data processing. By doing so, it bridges the gap between traditional SQL-based data analytics and modern stream processing, ensuring that users can leverage both paradigms effectively within the Flink environment.

3.7 Runtime.ss

Runtime.ss is the heart of Flink's stream processing capabilities. It manages the lifecycle of a Flink job, from submission to execution and termination, ensuring efficient resource allocation, fault tolerance, and coordination between various components. The following sections will dive into the most important subsystems housed within Runtime.ss, providing a detailed insight into their functionality and significance in the larger Flink ecosystem [9, 10].

3.7.1 Checkpoint.ss

The Checkpoint.ss subsystem in Apache Flink is pivotal for ensuring fault tolerance. Its primary role is to periodically capture the state of streaming applications, providing a mechanism for recovery in the event of failures. This feature is vital as it guarantees that data processing can continue seamlessly without data loss or inconsistencies, irrespective of machine or software failures. To achieve this, Checkpoint.ss integrates with the State.ss subsystem for proficient state management and is coordinated by the JobManager.ss, which triggers and restores checkpoints. By leveraging consistent snapshots, Checkpoint.ss also enables features like stateful stream processing and savepoints. Savepoints provide a way to version the state of a job, allowing for upgrades, rollbacks, and job modifications, thereby enhancing the overall operability and flexibility of stream processing tasks within Flink. This subsystem will be discussed in more detail in the upcoming sections.

3.7.2 Dispatcher.ss & Entrypoint.ss

The duo of Dispatcher.ss and Entrypoint.ss establishes the foundational layer for Flink's job execution. Specifically, Dispatcher.ss manages job submissions and acts as the primary contact point for client interactions. In contrast, Entrypoint.ss serves as the initial interface for launching a Flink cluster. Their significance lies in initiating and overseeing Flink jobs. To ensure smooth operation, Dispatcher.ss forwards job requests to the JobManager.ss and liaises with ResourceManager.ss for resource allocation. Moreover, these subsystems are essential for scaling and adaptability. As jobs get submitted, Dispatcher.ss efficiently routes them, ensuring job isolation and concurrent execution. Meanwhile, Entrypoint.ss enables various deployment modes of Flink, ensuring that Flink seamlessly integrates with diverse infrastructure setups, whether it's standalone clusters, Kubernetes, or other container orchestrations.

3.7.3 JobGraph.ss & ExecutionGraph.ss

JobGraph.ss and ExecutionGraph.ss represent the structure and execution plan of a Flink job, respectively. JobGraph.ss provides a high-level, abstracted view detailing the logical layout of a job, defining its structure, and interconnecting tasks. On the other hand, ExecutionGraph.ss breaks this down into a comprehensive execution strategy, detailing task parallelism, job stages, and dependency relations. These components are paramount, as they ensure the job's blueprints are clearly defined and that its execution is optimally strategized for performance and resource utilization. The transformation from JobGraph.ss to ExecutionGraph.ss is a critical phase in Flink's job lifecycle. Managed by the JobManager.ss, this process includes

optimizations, task chaining, and parallelism inference. Once transformed, the ExecutionGraph.ss serves as the guiding structure for job execution, with tasks distributed and coordinated across various TaskManager.ss instances, ensuring the job's efficient and timely completion.

3.7.4 JobManager.ss & JobMaster.ss

The coordination and execution of Flink jobs are seamlessly managed by the JobManager.ss and JobMaster.ss subsystems. The JobManager.ss functions as the nerve center for Flink job executions, orchestrating various components, monitoring job progress, and mitigating any possible failures. Its responsibilities encompass ensuring correct job execution, optimizing resource allocation, and managing potential errors or failures. In a more granular capacity, the JobMaster.ss caters to individual Flink jobs, overseeing their initialization, task distribution, checkpointing, and eventual termination. It acts as a guardian for each job, ensuring its success by addressing challenges specific to that job's lifecycle. The synergy between the two is evident in their collaborative operations: while JobManager.ss maintains a holistic view and interfaces with the TaskManager.ss for task execution and the ResourceManager.ss for optimal resource provisioning, the JobMaster.ss zooms into the specifics, supervising each phase of individual jobs and coordinating with other subsystems, such as Checkpoint.ss, to ensure job resilience and reliability.

3.7.5 ResourceManager.ss

Resource management forms the bedrock of Flink's robust processing capabilities, and this critical responsibility is adeptly managed by the ResourceManager.ss subsystem. At its core, ResourceManager.ss ensures that every Flink job gets the right number of computational resources – be it memory, CPU, or other infrastructural necessities. Such resource provisioning ensures that jobs run efficiently, without undue resource contention or wastage. Its power lies in its ability to dynamically adjust resource allocation based on the evolving demands of active jobs and the fluctuating availability of resources. This is achieved through its intricate coordination mechanisms: while it interfaces with JobManager.ss to comprehend the nuances of upcoming job requirements, it simultaneously remains in sync with TaskManager.ss to get real-time insights into resource availability and usage patterns. This dual-channel communication ensures that resources are judiciously allocated, reused, or released, upholding Flink's commitment to high-performance stream processing.

3.7.6 State.ss & Security.ss

The combination of State.ss and Security.ss ensures consistency and security in Flink's operations. The State.ss subsystem manages stateful operations in Flink, pivotal for ensuring a consistent stream processing experience. It facilitates the maintenance and retrieval of state information across tasks, making it an indispensable tool in the stream processing arsenal. Any interruptions or failures can be gracefully managed without sacrificing data integrity, thanks to the state management capabilities of this subsystem.

Simultaneously, the Security.ss subsystem ensures the integrity and confidentiality of the system. It acts as a bulwark against unauthorized access and potential threats, reinforcing the walls of the Flink ecosystem. By implementing robust authentication, authorization, and encryption protocols, Flink guarantees that data remains inaccessible to prying eyes and is transmitted securely across nodes. Moreover, it plays an important role in ensuring that task execution environments are insulated from potential vulnerabilities, providing peace of mind to operators and developers alike.

Their synergistic operation is evident in multiple facets of Flink's workflow. State.ss, for instance, is heavily involved during checkpointing processes, closely interacting with Checkpoint.ss to capture and restore the state of stream computations. On the other hand, Security.ss casts its protective net over all Flink subsystems, ensuring that from data ingestion to computation and output, every step is securely managed and shielded from potential threats.

3.7.7 TaskManager.ss & TaskExecutor.ss

The actual computation and data processing within Flink are anchored by the TaskManager.ss and TaskExecutor.ss subsystems. TaskManager.ss executes tasks, manages memory, and oversees local resources, ensuring efficient task execution. In tandem, TaskExecutor.ss serves as the runtime container for these tasks, providing an isolated environment replete with necessary dependencies and configurations. Together, they form the backbone of Flink's distributed processing capabilities, tackling massive data volumes and complex computations with finesse.

Their operations are intricately linked with other key subsystems. They receive tasks and directives from the JobManager.ss, coordinate with ResourceManager.ss for resource provisioning, and relay critical feedback on job progress, outcomes, and any encountered challenges. This synergy ensures Flink's ecosystem remains robust, responsive, and resilient to a myriad of processing demands.

4 Subsystems

4.1 DataStream

This API is tailored for real-time stream applications, facilitating the handling and analysis of continuous data streams. It allows for operations on data streams, including filtering, state updates, window definitions, and aggregation. These data streams can originate from diverse sources like message queues, socket streams, and files. The results can be directed to output files or standard output [11].

4.2 DataSet

Employed for batch processing, it conducts parallel and distributed processing of static datasets, proving its significance in offline and non-streaming data processing tasks. It executes transformations on datasets, including filtering, mapping, joining, and grouping. These datasets are initially generated from specific sources, like reading files or from local collections [12].

4.3 Table API & SQL

Designed for processing relational data, this subsystem simplifies data transformation and task analysis by employing SQL-like queries. Flink optimizes these queries and translates them into DataStream and DataSet operations, providing a seamless experience. It allows for composing queries using relational operators like selection,

filtering, and joins in an intuitive manner. Flink's SQL support is based on Apache Calcite, ensuring adherence to the SQL standard. Whether working with streaming data or datasets, both the Table API and SQL interface offer consistent semantics and yield the same results. These interfaces harmoniously integrate with Flink's DataStream API, enabling easy transitions between them and libraries built on top of them [13].

4.4 Checkpoint.ss

Checkpoints provide a critical function by ensuring data integrity and fault tolerance, enabling applications to bounce back from failures and maintain a consistent state. They empower Flink to restore both state and stream positions, preserving the application's behavior as if it were running without interruptions. In essence, a checkpoint acts as an automatically captured snapshot by Flink, serving the primary purpose of facilitating recovery from faults. These checkpoints can be incremental in nature and are designed for swift restoration, enhancing the system's resilience [14]

First and foremost, fault tolerance is achieved through state snapshots. When a checkpoint is initiated as directed by the task manager, it prompts all data sources to record their offsets and inject checkpoint barriers with assigned numbers into their data streams. These barriers traverse the job graph, marking the segments of the stream both before and after each checkpoint [14].

Checkpoint 'n' captures the state of each operator, reflecting their status after processing all events up to checkpoint barrier 'n' while excluding any events that follow. As each operator in the job graph encounters these barriers, it diligently records its state. Flink's state backend employs a copy-on-write mechanism, ensuring uninterrupted stream processing while older state versions are asynchronously snapshot. It's only after these snapshots are durably persisted that the older state versions become eligible for garbage collection [14]. Looking deeper into the checkpoint subsystem there are 4 distinct subsubsystems to discuss. These can be seen in the checkpoint focused view.

4.4.1 CheckpointCoordinator.ss

The checkpoint coordinator subsystem plays a crucial role in ensuring fault tolerance and data consistency within distributed stream processing applications. This subsystem employs distributed snapshotting, a technique where the system captures the state of all the involved components, including JobManager, TaskManager, JobMaster, and StateBackend, at a specific point in time. In the event of a fault state, where a failure occurs within the system, processing is halted temporarily. During this pause, the checkpoint coordinator orchestrates the process of taking a snapshot of the application's state. This snapshot includes the states of all the distributed components, capturing a consistent view of the data and the processing progress. Once the snapshot is successfully taken, the system can then resume processing from the last consistent state, ensuring that no data is lost and the application continues its execution seamlessly. This mechanism is vital for maintaining the reliability and fault tolerance of Apache Flink applications, allowing them to recover gracefully from failures and ensuring the continuity of data processing tasks [15].

4.4.2 Metadata.ss

The metadata subsubsystem plays a pivotal role by managing the accessory data associated with checkpoints. This metadata includes crucial information about the storage location, status, and the (de)serialization process of checkpoints within the system. The metadata subsystem acts as a repository of essential details, enabling efficient tracking and management of checkpoints. It facilitates interactions between various components, such as the CheckpointCoordinator, JobManager, and TaskManager. The CheckpointCoordinator relies on metadata to coordinate the checkpointing process, ensuring the timely creation and persistence of checkpoints. JobManager utilizes this metadata to monitor the status and progress of checkpoints, enabling it to make informed decisions about job execution. TaskManagers, on the other hand, use the metadata to access and retrieve checkpoint data during recovery processes. By providing a centralized mechanism for storing and accessing checkpoint-related information, the metadata subsubsystem enhances the overall reliability and fault tolerance of Apache Flink applications, enabling seamless recovery and consistent data processing [16].

4.4.3 Hooks.ss

The hooks subsubsystem in the Apache Flink checkpoint subsystem serves a critical function by providing a mechanism for executing custom actions associated with the beginning of checkpoints. This functionality allows developers to define specific actions that should be triggered when a checkpoint process starts. These custom actions can be tailored to meet various application requirements, such as managing external resources, updating metadata, or executing specific logic relevant to the checkpointing process. The hooks subsystem interacts closely with key components including the CheckpointCoordinator, TaskManager, and TaskExecutor. When a checkpoint is initiated, the CheckpointCoordinator communicates with the hooks subsystem, triggering any predefined custom actions associated with the beginning of the checkpoint. This interaction ensures that developers have the flexibility to integrate their own logic seamlessly into the checkpointing process, enhancing the extensibility of Apache Flink and enabling the implementation of sophisticated and customized checkpointing behaviors tailored to specific use cases [17].

4.4.4 Channel.ss

The channels subsubsystem in the Apache Flink checkpoint subsystem plays a crucial role in ensuring the consistency and fault tolerance of data in transit within distributed stream processing applications. This subsystem is responsible for creating a snapshot of the data that is currently in transit between different components of the system. This includes the sender and receiver information associated with these data streams. By capturing this snapshot, the channels subsubsystem enables the system to resume the streams from the exact point where they were interrupted during a checkpoint. Interacting closely with the CheckpointCoordinator, the channels subsystem helps in coordinating the process of capturing these snapshots efficiently. Additionally, it interacts with the State component, which represents the application's state, and the TaskManager, which oversees the execution of tasks within the Flink cluster. By ensuring the integrity of data in transit and facilitating the seamless resumption of

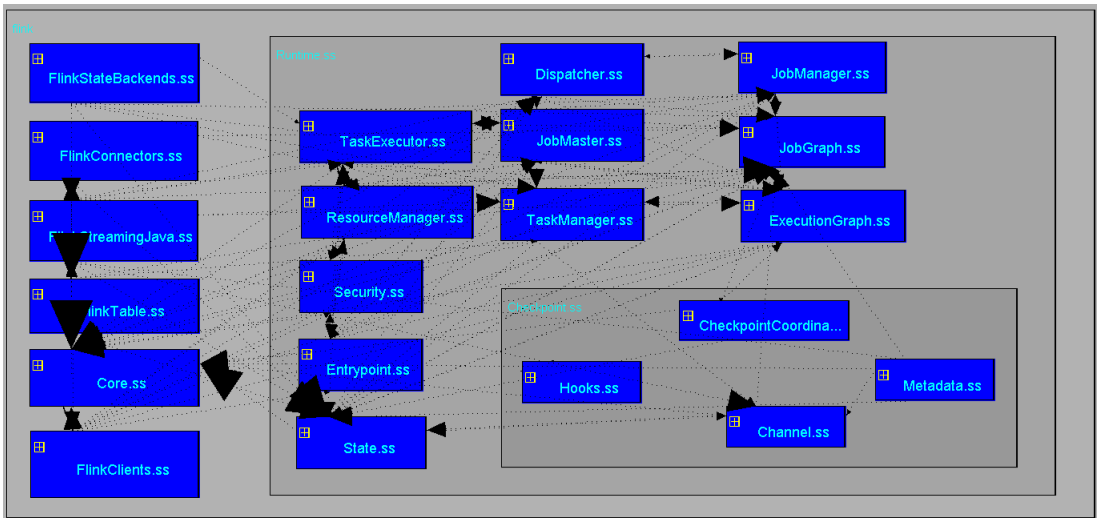

Fig. 3: Architecture view with checkpoint subsystem focused

streams, the channels subsubsystem contributes significantly to the overall reliability and fault tolerance of Apache Flink applications [18].

4.5 Cluster Management

It oversees the connection between Flink and different cluster managers to handle resource allocation, job submission, and the scaling of Flink applications. The specific cluster manager in use is responsible for creating a dedicated cluster for each submitted job, making it exclusively accessible to that job [10].

4.6 Connectors

Flink applications have the capability to both read from and write to diverse external systems using connectors. They are equipped to handle multiple data formats, ensuring that data can be encoded and decoded to align with Flink's data structures. This subsystem manages the smooth integration with a broad spectrum of data sources and destinations, encompassing Apache Pulsar, Apache Kafka, databases, and filesystems. These connectors streamline the process of ingesting and outputting data, with options available for both DataStream and Table API/SQL [19].

4.7 Checkpoint Subsystem Interactions

4.7.1 Runtime.ss

The runtime subsystem in Apache Flink interacts with checkpointing by taking on key responsibilities in the checkpointing process. These responsibilities include triggering checkpoint creation, coordinating with operators, inserting synchronization barriers into the data stream, serializing and persisting data, collecting acknowledgments, and facilitating recovery. This interaction is essential for ensuring fault tolerance and maintaining data consistency within Flink applications [20].

4.7.2 Corse.ss - State Management & Fault Tolerance

Checkpointing in Flink closely interacts with the state management subsystem to ensure data integrity and fault tolerance. It coordinates checkpoint creation, manages the serialization and storage of operator and intermediate data states, and facilitates the recovery process in case of failures. This collaboration is vital for maintaining application consistency and resilience [20].

4.7.3 FlintClient.ss

The Flink client in Apache Flink indirectly interacts with the checkpointing mechanism, allowing users to configure, monitor, and control checkpointing behavior in their Flink jobs. It enables users to set checkpoint-related parameters, offers monitoring capabilities, supports manual save point initiation, and manages recovery in case of job failures. The Flink client ensures that checkpointing aligns with user requirements and facilitates a smooth recovery process.

4.7.4 FlinkConnector.ss

Flink connectors in Apache Flink collaborate with the checkpointing mechanism to secure data consistency and fault tolerance. They are pivotal in upholding data integrity and ensuring that data ingestion and output align seamlessly with checkpointed states, thereby enabling exactly once processing and dependable data handling. This partnership is fundamental for data integrity across both streaming and batch processing scenarios [21, 22].

4.7.5 FlinkTable.ss

Flink's Table API and SQL interact indirectly with the checkpointing mechanism through the Flink runtime and state management subsystems. Declarative queries defined in the Table API and SQL are translated into lower-level operations by the runtime, which includes state management and barrier insertion to ensure that checkpointing is consistent and aligned with the data stream. This interaction guarantees exactly-once semantics and reliable data process-

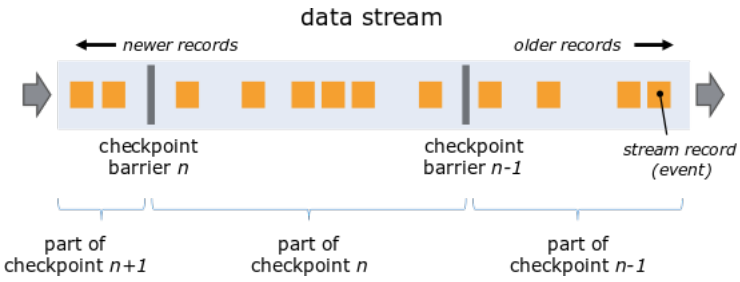

Fig. 4: Checkpointing subsystem capturing state snapshots of each operator [14]

ing while enabling users to express their data processing logic using declarative queries [23].

5 Architectural Styles

5.1 Repository

Utilize figure 4 when referring to components in this section. In Apache Flink, the repository architectural style is best illustrated by its state management subsystem. Specifically, the State.ss subsystem acts as a central repository that manages stateful operations, ensuring a consistent stream processing experience. This centralized design simplifies interactions and data consistency, as various subsystems interact with a unified state repository rather than disparate state sources. Components, such as the Checkpoint.ss, play a pivotal role in this architecture. They access and modify this repository to capture the state periodically, enabling fault tolerance and recovery mechanisms. Additionally, by consolidating state management in this manner, Flink efficiently handles distributed data processing scenarios, ensuring that state information remains coherent and synchronized across different tasks and nodes in the system.

5.2 Pipe & Filter

Utilize figure 5 when referring to components in this section. The pipe & filter architectural style is exemplified through its data flow and processing components. As data streams move through the system, subsystems such as TaskManager.ss and TaskExecutor.ss play a pivotal role. They function similar to filters, processing data and ensuring it undergoes the necessary transformations before moving to the next stage. For instance, ExecutionGraph.ss defines the parallel execution strategy, serving as a road-map for how data should flow and be processed. As data travels, it moves through these "filters," each of which performs a specific function, whether it's managing state with State.ss or handling checkpoints with Checkpoint.ss. This modularity ensures that each subsystem can focus on its dedicated function, optimizing performance and ensuring data integrity throughout the process.

5.3 Layered

Utilize figure 4 when referring to components in this section. Apache Flink is designed with a structured approach that can be seen as layers. At the top, Entrypoint.ss and Dispatcher.ss are the gatekeepers, handling job submissions and overall management. They're the first point of contact for any job coming in. Following this, ResourceManager.ss plays a pivotal role, ensuring each job gets the resources it needs to run smoothly. This is crucial because resources, like memory or CPU, can be limited. Then, the ExecutionGraph.ss and JobGraph.ss come into the picture, providing a clear road-map for how the job should proceed. They break down the job into smaller tasks and set the order for execution. At the ground level, State.ss maintains a consistent data environment, and Security.ss ensures all operations are carried out safely. This layered approach not only organizes tasks but also ensures that each layer can focus on its specific responsibility, leading to efficient execution and easier troubleshooting.

5.4 Client-Server

Utilize figure 4 when referring to components in this section. The client-server architectural style emerges clearly in the communication dynamics among its subsystems. Dispatcher.ss, acting as a server, stands central to this design, fielding requests and dispatching responses. Beyond the direct user interfaces like the graphical UI or the command-line tool, other subsystems also adopt the client role at times. For example, TaskManager.ss often operates as a client, communicating with JobManager.ss to obtain execution instructions or relay task status updates. Similarly, ResourceManager.ss might request the JobManager.ss for job specifics to ensure proper resource allocation. This bidirectional interaction pattern, with certain components sometimes acting as clients and at other times as servers, underpins the flexible and efficient orchestration of tasks in Flink.

5.5 Implicit Invocation

Utilize figure 5 when referring to components in this section, despite that figure being designed for pipe-and-filter, it's components can be used for this architecture style. Apache Flink utilizes the implicit invocation architectural style extensively to maintain modularity and decouple its various components. By doing so, the system remains agile and can react to different events efficiently. When TaskManager.ss completes a specific task or encounters an error, it doesn't directly command other components but rather sends a signal or notification. In response, components like JobManager.ss, which have registered an interest in such events, take appropriate actions without the need for direct invocation.

5.6 Distributed System

Utilize figure 4 when referring to components in this section. Apache Flink is inherently designed for distributed processing [10]. The division and coordination between the JobManager.ss, ResourceManager.ss, and multiple TaskManager.ss instances exemplify this. Each TaskManager.ss can run on a separate machine, collaboratively processing data in parallel. This distributed architecture also supports dynamic scaling [24], allowing for the addition or removal of TaskManager.ss instances based on the workload. This elasticity ensures that Flink can adapt to varying data volumes and computational needs. Moreover, the ResourceManager.ss plays a pivotal role in this distributed setup by managing resources and ensuring that each TaskManager.ss has the necessary resources to execute its tasks. The interplay between these components, combined

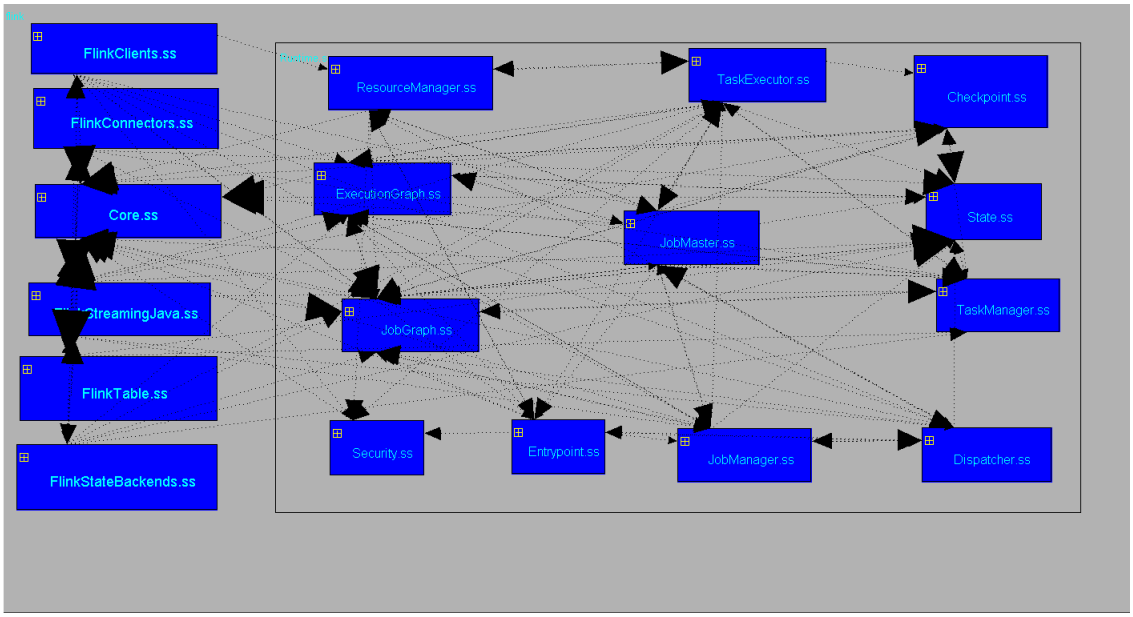

Fig. 5: Concrete architecture view of Runtime.ss

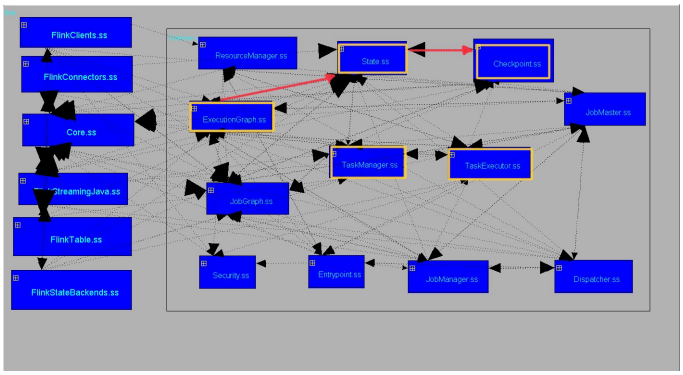

Fig. 6: A representation of Pipe and filter architecture

with Flink's robust fault-tolerance mechanisms, ensures that even in a distributed environment, data processing remains consistent and reliable. The design emphasizes the significance of distributing data and computation across multiple nodes, optimizing for both speed and resilience.

6 Design Patterns

6.1 Factory Design Pattern

Apache Flink's architecture embodies the Factory pattern, a design principle that offers a way to encapsulate the creation logic of objects, allowing the system to remain flexible and scalable. This pattern is particularly evident in classes such as StreamExecutionEnvironment where the Factory pattern allows for the creation of different execution environments that are suitable for various deployment scenarios, such as local, remote, or cluster execution. Similarly, the instantiation of the ResourceManager is managed through factory-based methods, which can dynamically provide the appropriate resource management strategy depending on the deployment configuration. By delegating the instantiation process to specialized Factory classes or methods, Flink ensures that the system can handle diverse operational contexts and resource landscapes, while keeping the instantiation process consistent and maintainable. The use of the Factory pattern is instrumental in abstracting the complexity involved in the creation of objects, particularly in a system designed for high-performance distributed processing where adaptability to different runtime conditions is crucial.

6.2 Adapter Design Pattern

The Adapter pattern finds its application in the Connectors subsystem of Apache Flink, where it plays a critical role in bridging communication between Flink and external systems. This design pattern allows Flink to adapt to the interfaces of various systems like Hadoop, Kafka, and others without changing the core logic of the Connectors. By using Adapters, Flink can effectively 'speak' the protocol of the external system, ensuring that data can be read from and written to these systems in a way that appears native to Flink. For instance, a Hadoop connector might use an Adapter to interact with the Hadoop Distributed File System (HDFS) API, translating Flink's data structures and operations into the appropriate HDFS calls. This approach not only simplifies the process of extending Flink's capabilities to new external systems but also allows for easier maintenance and updates to the Connectors, as changes in the external system's API may only require modifications in the Adapter layer, rather than a complete overhaul of the Connector code. The Adapter pattern thus provides a versatile and robust means for integration, ensuring that Flink remains at the forefront of big data processing by easily accommodating diverse data sources and sinks.

6.3 Singleton Pattern

During our in-depth analysis of Apache Flink's underlying architecture, we've identified the implementation of the Singleton design pattern as a cornerstone in its structural integrity. This design paradigm ensures the limitation of class instantiation to just one object. This is prominently showcased in the JobManager component of Flink. The JobManager plays a pivotal role in orchestrating job execution within a Flink cluster, and its functionality is hinged on the Singleton pattern. By confining the JobManager to a solitary instance, Flink fortifies the uniformity and coherence of its job management processes. Such an approach is instrumental in averting the chaos and discrepancy that could stem from the presence of multiple instances of the JobManager. The Singleton's role is, therefore, integral to the durability and steadfastness of the job management mechanism in Flink, solidifying its capability to execute, oversee, and handle jobs with a level of consistency and predictability that is essential for reliable operations.

6.4 Observer Pattern

Within Apache Flink's sophisticated architecture, the Observer pattern plays a pivotal role, particularly within its event-driven runtime system. In this context, operators, acting as observers, are designed to respond to the dynamics of incoming data streams—these are the subjects to which they are attuned. Whenever there's an update to the data stream, such as the arrival of new data, it triggers a notification to all the connected operators. In response, these operators then process the newly arrived data.

This pattern is critical as it allows for a substantial decoupling between the data streams and their corresponding operators. This decoupling grants a heightened level of flexibility and scalability to Flink's runtime operations. The incorporation of the Observer pattern is, therefore, not merely a design choice but a strategic enhancement, equipping Flink with the ability to perform real-time, event-driven processing across extensive data streams.

6.5 Decorator Pattern

Looking through the structure of Flink provides some genuine understanding of how the Decorator pattern is used in practice. When it comes to giving objects additional characteristics precisely when you need them, this pattern is crucial. And the way Flink addresses the problem of serialization and deserialization is a great example of it in action. Let's break it down: This fundamental Serializer, which works well for basic data types, is where Flink starts. But Flink's approach is really clever when it comes to scaling up to more complex data structures like arrays, lists, or maps. Rather than stopping at the essentials, it imbues each core serializer with extra capabilities (the Decorators) that enable it to process increasingly complicated data kinds. This approach of stacking up functions means Flink stays agile and capable when it comes to a wide range of data types, keeping its internals uncluttered. Imagine having a toolkit where there's a spot for every tool and you know exactly where to find it—that's what this is like. Maintenance becomes straightforward. And let's not overlook how choosing Decorators sidesteps the hassles of overusing subclasses, which is the traditional route for adding new features.

7 Concrete Architecture vs. Conceptual Architecture

Utilize figure 6 when similarities or differences are discussed. Yellow boxes in the figure correspond to sections that match in both conceptual and concrete graph. Meanwhile, red boxes correspond to elements missing in the conceptual and present in the concrete.

7.1 Absences

In the comparison between the concrete and conceptual architectures, certain components stand out due to their presence in the concrete model and absence in the conceptual design. The Core.ss component, for instance, lacks representation in the conceptual architecture. Similarly, Checkpointing.ss, which is integral for stateful computations, is absent from the conceptual visuals. The Entrypoint.ss serves as the initial phase for executing Flink but doesn't appear in the conceptual framework. Another essential component, Jobgraph.ss, represents the submitted job yet is missing from the conceptual overview. Furthermore, the Executiongraph.ss, vital for delineating the task execution process, is also not depicted in the conceptual design. Lastly, Security.ss, which caters to the platform's security protocols and measures, is not incorporated into the conceptual layout. These absences can be attributed to the depth of abstraction the team chose to adopt. For the sake of maintaining a high-level perspective, the functionalities of these missing components are abstracted and are assumed to be encapsulated within the "Flink cluster" of the conceptual architecture.

7.2 Convergences

The Runtime.ss and Flink Cluster are equivalent in their architectural roles, serving as the system's backbone. Both components mirror the role of a cluster in distributed computing, acting as a central hub that orchestrates primary functionalities and facilitates component interaction. Similarly, the incorporation of FlinkStateBackend.ss and Statebackend in the architecture points to their equivalence, emphasizing the importance of state management. This is important for tasks such as stream processing and real-time analytics, ensuring data consistency and resilience.

The FlinkStreamingJava.ss and Datastream are also analogous in their intent and functionalities. While FlinkStreamingJava.ss is geared towards Java-based streaming capabilities, Datastream specializes in the efficient real-time flow and processing of data. Additionally, FlinkClient.ss and Client are directly comparable, with both acting as the primary interfaces for user-system interaction, spanning tasks from job submission to result retrieval.

The significance of the highlighted Runtime subsystems — TaskManager.ss, Dispatcher.ss, JobMaster.ss, JobManager.ss, and FlinkStateBackends.ss — is evident in both architectures, pointing to their functional equivalence. Their consistent presence underscores their vital roles in task management, job dispatching, state management, and holistic system orchestration.

7.3 Divergences

Divergences between the conceptual and concrete architectures are evident in the representation and interactions of specific components. One of the primary differences is seen in the representation of

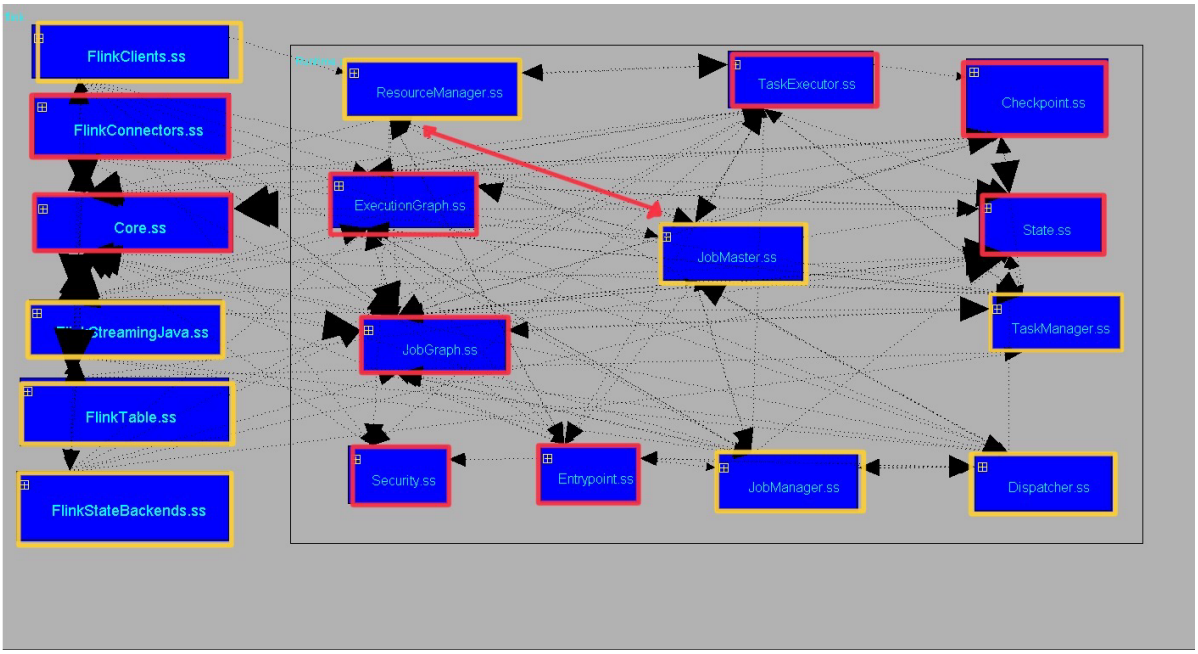

Fig. 7: Conceptual vs Concrete: Yellow boxes highlight similarities between the graphs, red boxes highlight differences or missing elements

the State Backend. In the concrete architecture, the State Backend is integrated within the Runtime. However, in the conceptual representation, it stands as an external entity. Another notable difference is in the interactions of the ResourceManager. In the concrete design, the ResourceManager communicates directly with the JobMaster. This is in stark contrast to the conceptual layout where the ResourceManager's interaction is with the JobManager. Furthermore, the communication pattern of the JobMaster in the concrete architecture is more intricate than its conceptual counterpart. In the concrete version, the JobMaster interacts with both the TaskManager and the TaskExecutor. In comparison, the conceptual design shows the JobMaster communicating solely with an entity labeled 'Task.' The communication flow further diverges when observing the JobManager. In the concrete architecture, there's a direct line of interaction between the JobManager and the TaskExecutor, an aspect missing from the conceptual diagram.

Additionally, the "Flink APIs" in the conceptual layout lack a direct channel of communication with the TaskExecutor. Their interaction is primarily with the TaskManager, which differs from the concrete representation. Lastly, an evident divergence is seen in the bidirectional communication between the TaskExecutor and the TaskManager in the concrete design. This suggests a two-way flow of information, contrasting the unidirectional flow depicted in the conceptual architecture.

8 Checkpoint Use Case

Apache Flink's checkpointing mechanism plays a pivotal role in ensuring fault tolerance and data consistency in distributed data processing applications. Various diagram types, including Use Case Diagrams, Sequence Diagrams, and State Diagrams, are instrumental in depicting different facets of checkpointing. Use Case Diagrams provide a high-level overview of how and why checkpointing is essential in specific scenarios. They outline the main use cases, such as "Stream Processing" and "State Consistency," highlighting the broad context in which checkpointing is applied and underscoring its significance in terms of ensuring data reliability and application resilience. Sequence Diagrams offer a more detailed, step-by-step representation of how checkpointing functions in practice. They reveal the dynamic behavior of checkpointing by illustrating the flow of events during the process, showing the interactions and communication between components like data sources, job managers, task managers, and state backends. State Diagrams, meanwhile, present the checkpointing process in terms of states, transitions, and system behavior. States such as "Running," "Triggering Checkpoint," "Failed," and "Completed" are depicted, along with transitions between these states. These diagrams help us grasp the overall flow and lifecycle of checkpointing, demonstrating how the system responds to various events, including checkpoint triggers and failures, and how it recovers from those failures.

8.1 Use-Case Diagram

The use case diagram illustrates the checkpointing process in Apache Flink, which is integral for event-driven applications widely employed in industries like finance, healthcare, and transportation. These applications excel in handling data streams at scale and responding to individual events or aggregated event data, often within specified time windows. The diagram captures how Flink's components, such as "Stateful Computations" and "Process Streams," enable these applications to recognize patterns and respond promptly by triggering computations, maintaining states, and taking external actions while ensuring fault tolerance and reliable state management.

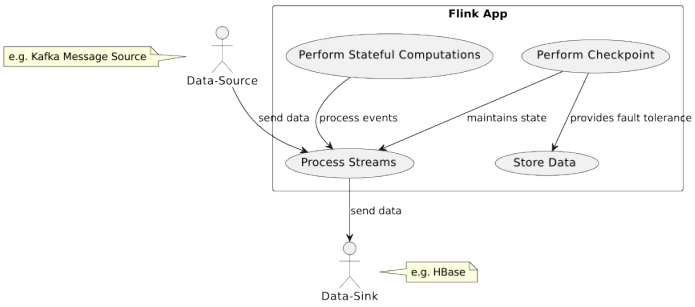

Fig. 8: Checkpointing use-case diagram with Kafka source and HBase sink

8.2 Sequence Diagram

The sequence diagram provides a representation of the interactions and steps involved in the Apache Flink checkpointing process, which is a critical aspect of maintaining the consistency and fault tolerance of stateful stream processing applications. The process begins with Core.ss triggering the checkpoint, which is then coordinated by CheckpointCoordinator.ss. Before checkpoint execution, custom pre-checkpoint hooks are executed in Hooks.ss, and metadata about the checkpoint is collected in MetaData.ss. Data transfer is facilitated by Channel.ss to ensure that the state data is captured and transferred correctly. Post-checkpoint hooks are then executed by Hooks.ss. The checkpoint trigger is acknowledged, and data transfer is confirmed. CheckpointCoordinator.ss notifies MetaData.ss of the checkpoint's completion. Furthermore, JobMaster.ss plays a key role in triggering checkpointing, and the job structure is defined in JobGraph.ss. The execution graph is initialized in ExecutionGraph.ss, and the checkpoint coordinator registers it. The actual execution of the checkpoint is carried out by TaskExecutor.ss, with state capture and restoration handled by State.ss. The successful state restoration is confirmed, and the completion of checkpoint data transfer is acknowledged. This sequence diagram outlines a simplified yet essential overview of the complex interactions and the order of execution for various components involved in the Flink checkpointing process, which is vital for maintaining the reliability and consistency of stateful stream processing applications.

8.3 State Diagram

The state diagram illustrates the Apache Flink checkpointing mechanism, a critical component in distributed stream processing systems. In this diagram, there are several states and transitions that depict the various phases of Flink's checkpointing process. It begins in the "Running" state, representing the initial phase where the Flink job is actively processing data. When checkpointing is triggered ("StartCheckpoint"), the system moves to "TriggeringCheckpoint," initiating the checkpoint process. Subsequently, it enters the "TakingCheckpoint" state, where Flink captures the state of the running job. After taking the checkpoint, it transitions to "SavingCheckpoint," indicating the process of persistently storing the checkpoint data. If the checkpoint process is successful, it proceeds to "Completed." However, if any step in the process fails, the system enters the "Failed" state, requiring recovery. Ultimately, when the job completes successfully, it reaches the "Completed" state. Apache Flink's checkpointing mechanism plays a vital role in ensuring fault tolerance and data consistency in distributed stream processing workflows, making it a fundamental feature for stream processing systems.

Use Case Diagrams, Sequence Diagrams, and State Diagrams collaboratively provide a comprehensive understanding of checkpointing in Apache Flink. They encompass its high-level context, specific use cases, intricate mechanics, and state transitions, collectively enabling a holistic view of this critical mechanism in distributed data processing applications.

9 Limitations

The architecture presented covers much of the design space of Apache Flink but it is not fully exhaustive of every single subsystem and sub-subsystem. Moreover, the focus is on the checkpoint subsystem and its interactions with other subsystems. When taking this into consideration, it is important to note that there may be alternative ways to group and arrange this if looking at the system from a general lens or from the lens of another subsystem. There are also divergences in the concrete architecture if other builds of Flink are used as the source. Flink has had many iterations over its several years of development. This study is accurate for version 1.17 as that is the source we used. Lastly, LSEdit does not offer fully modernized architectural diagrams. These diagrams are correct but may not be accepted for formal publication. The work around would be to use them as a template to make new diagrams from a modern diagram tool.

10 Lessons Learned

Certainly! In the realm of software architecture, the relationship between conceptual and concrete architecture is akin to a blueprint transforming into a fully constructed building. Conceptual architecture provides the abstract vision, outlining the system's high-level components and their interactions, while concrete architecture delves into the intricate details, addressing specific technologies, platforms, and implementation strategies. The meticulous organization of a project's directory structure significantly impacts the efficiency of data extraction. A well-structured directory simplifies the extraction process, ensuring data is obtained from the correct sources and seamlessly integrated into the system. Defining subsystems within a project is a nuanced task, demanding a profound understanding of the project's intricacies. It requires establishing clear boundaries between subsystems to maintain modularity and simplify maintenance, fostering collaboration among developers. In the context of data processing, the significance of checkpointing in Apache Flink cannot be overstated. Checkpointing stands as a cornerstone feature, safeguarding data integrity in stream processing applications. By creating consistent snapshots of the application's state, Apache Flink's checkpointing functionality ensures fault tolerance and recovery in the face of failures, guaranteeing that data processing tasks can resume seamlessly from a consistent state after unforeseen disruptions. These nuanced aspects underscore the complexity and precision required in the realm of software architecture and data process-

Fig. 9: Checkpointing sequence diagram showing synchornous and asynchronous workflow with lifelines

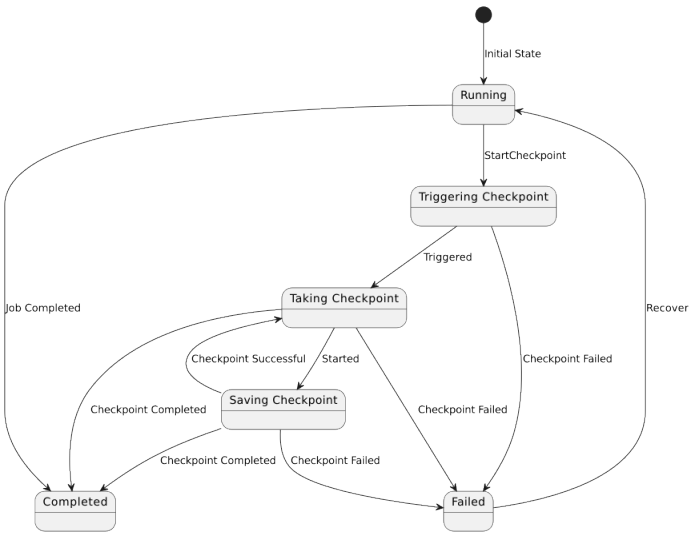

Fig. 10: Checkpointing state diagram showing transitions and failures

ing, where attention to both conceptual frameworks and concrete implementation details is paramount for building robust, reliable, and scalable systems.

11 Conclusion

In conclusion, we have presented a novel workflow using existing tools to extract the concrete architecture of a large scale open-source distributed system. This extraction process was used to map and create diagrams for Apache Flink and its subsystems. We elucidated, examined and showed various views of the architecture. First, the top level view was shown and then we examined the runtime subsystem and its internal components. We finished this tour of Flink by ending at the checkpoint subsystem. We then presented the various architectural patterns and design patterns. The concrete architecture was compared to the conceptual architecture presented in previous works. There were some divergences observed and documented but also several similarities. Finally, a use case involving checkpointing was shown with various UML diagram.

References

[1] “Understand: The Software Developer’s Multi-Tool.” [Online]. Available: https://scitools.com/

[2] “flink/flink-core at 1.17 · apache/flink.” [Online]. Available: https://github.com/apache/flink/tree/release-1.17/flink-core

[3] “flink/flink-clients at 1.17 · apache/flink.” [Online]. Available: https://github.com/apache/flink/tree/release-1.17/flink-clients

[4] “flink/flink-connectors at 1.17 · apache/flink.” [Online]. Available: https://github.com/apache/flink/tree/release-1.17/flink-connectors

[5] “Apache Flink 1.2 Documentation: Dataflow Programming Model.” [Online]. Available: https://nightlies.apache.org/flink/flink-docs-release-1.2/concepts/programming-model.html

[6] “flink/flink-streaming-java at 1.17 · apache/flink.” [Online]. Available: https://github.com/apache/flink/tree/release-1.17/flink-streaming-java

[7] “An Overview of End-to-End Exactly-Once Processing in Apache Flink (with Apache Kafka, too!),” Feb. 2018, section: posts. [Online]. Available: https://flink.apache.org/2018/02/28/an-overview-of-end-to-end-exactly-once-processing-in-apache-flink-w

[8] “flink/flink-table at 1.17 · apache/flink.” [Online]. Available: https://github.com/apache/flink/tree/release-1.17/flink-table

[9] “flink/flink-runtime at 1.17 · apache/flink.” [Online]. Available: https://github.com/apache/flink/tree/release-1.17/flink-runtime

[10] “Flink Architecture,” section: docs. [Online]. Available: //nightlies.apache.org/flink/flink-docs-master/docs/concepts/flink-architecture/

[11] “Apache Flink 1.2 Documentation: Flink DataStream API Programming Guide.” [Online]. Available: https://nightlies.apache.org/flink/flink-docs-release-1.2/dev/datastream_api.html#:~:text=DataStream%20programs%20in%20Flink%20are,%2C%20socket%20streams%2C%20files

[12] “Apache Flink 1.0.3 Documentation: Flink DataSet API Programming Guide.” [Online]. Available: https://nightlies.apache.org/flink/flink-docs-release-1.0/apis/batch/index.html#:~:text=DataSet%20programs%20in%20Flink%20are,%2C%20or%20from%20local%20collections

[13] “Overview,” section: docs. [Online]. Available: //nightlies.apache.org/flink/flink-docs-master/docs/dev/table/overview/

[14] “Fault Tolerance,” section: docs. [Online]. Available: //nightlies.apache.org/flink/flink-docs-master/docs/learn-flink/fault_tolerance/

[15] “flink/flink-runtime/src/main/java/org/apache/flink/runtime/checkpoint/Cl at release-1.17 · apache/flink.” [Online]. Available: https://github.com/apache/flink/blob/release-1.17/flink-runtime/src/main/java/org/apache/flink/runtime/checkpoint/CheckpointCoordinator.java

[16] “flink/flink-runtime/src/main/java/org/apache/flink/runtime/checkpoint/m at release-1.17 · apache/flink.” [Online]. Available: https://github.com/apache/flink/tree/release-1.17/flink-runtime/src/main/java/org/apache/flink/runtime/checkpoint/metadata

[17] “flink/flink-runtime/src/main/java/org/apache/flink/runtime/checkpoint/ho at release-1.17 · apache/flink.” [Online]. Available: https://github.com/apache/flink/tree/release-1.17/flink-runtime/src/main/java/org/apache/flink/runtime/checkpoint/hooks

[18] “flink/flink-runtime/src/main/java/org/apache/flink/runtime/checkpoint/ch at release-1.17 · apache/flink.” [Online]. Available: https://github.com/apache/flink/tree/release-1.17/flink-runtime/src/main/java/org/apache/flink/runtime/checkpoint/channel

[19] “Connectors and Formats,” section: docs. [Online]. Available: //nightlies.apache.org/flink/flink-docs-release-1.16/docs/dev/configuration/connector/

[20] “Apache Flink 1.2 Documentation: Distributed Runtime Environment.” [Online]. Available: https://nightlies.apache.org/flink/flink-docs-release-1.2/concepts/runtime.html#:~:text=The%20Flink%20runtime%20consists%20of,coordinate%20recovery%20on%20failures%2C%20etc

[21] “Connectors.” [Online]. Available: //nightlies.apache.org/flink/flink-docs-release-1.14/docs/dev/python/table/python_table_api_connectors/

[22] “Fault Tolerance Guarantees,” section: docs. [Online]. Available: //nightlies.apache.org/flink/flink-docs-master/docs/connectors/datastream/guarantees/

[23] “DataStream API Integration,” section: docs. [Online]. Available: //nightlies.apache.org/flink/flink-docs-master/docs/dev/table/data_stream_api/

[24] “Scaling Flink automatically with Reactive Mode,” May 2021, section: posts. [Online]. Available: https://flink.apache.org/2021/05/06/scaling-flink-automatically-with-reactive-mode/