Flink - Concrete Architecture

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

Presentation Outline

- JobManager overview/subsystems
- Derivation of concrete architecture
- Component interaction
- Code dive
- Reflexion analysis
- Lessons learned
- LSEdit demo

JobManager Overview

- Responsibilities related to the execution of the Flink application
- Converts JobGraph into an ExecutionGraph
- Scheduling tasks
- Checkpoint coordination
- Reacting to task events

ResourceManager

- Manages task slots within the TaskManagers.
- Allows for communication between the JobMaster and the TaskManager.

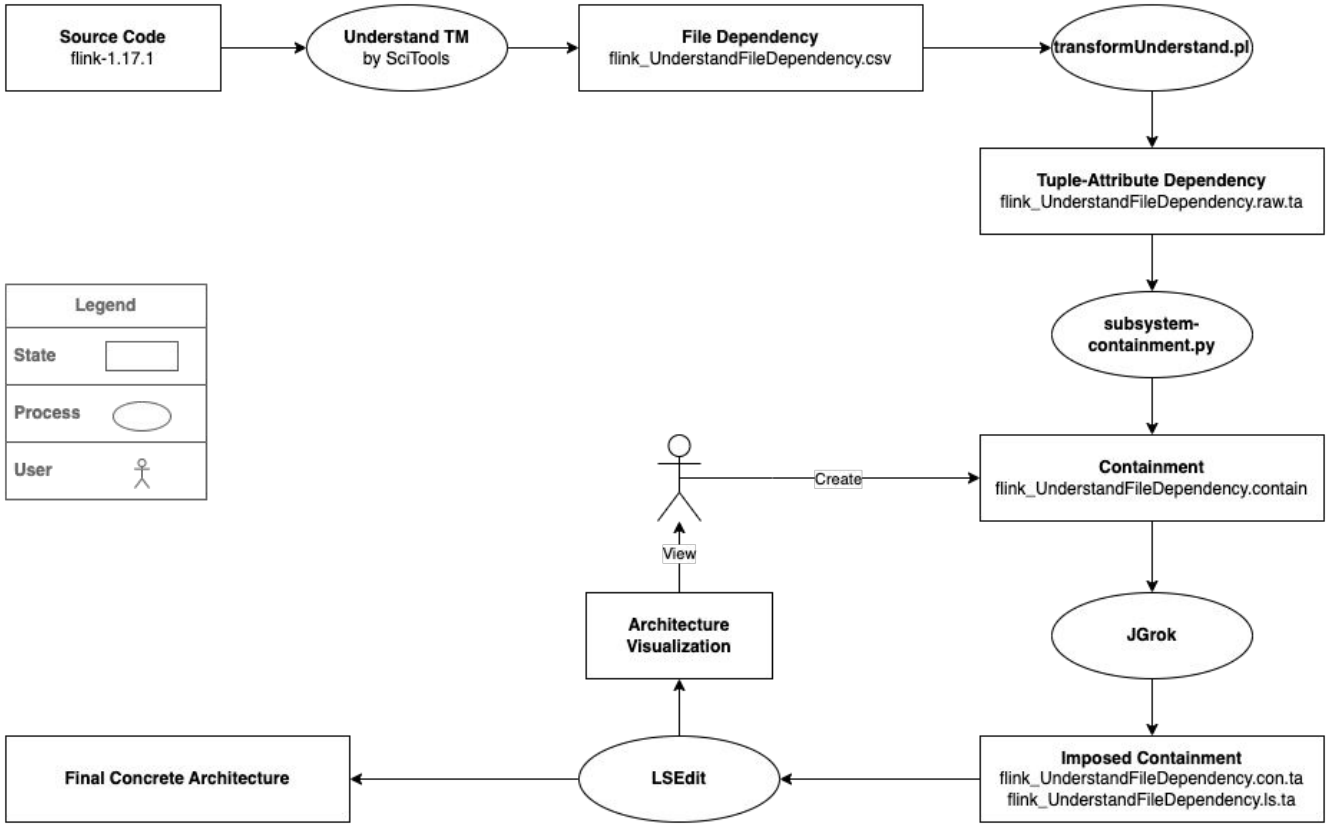
Dispatcher

- Gateway for job management
  - REST interface for executing applications

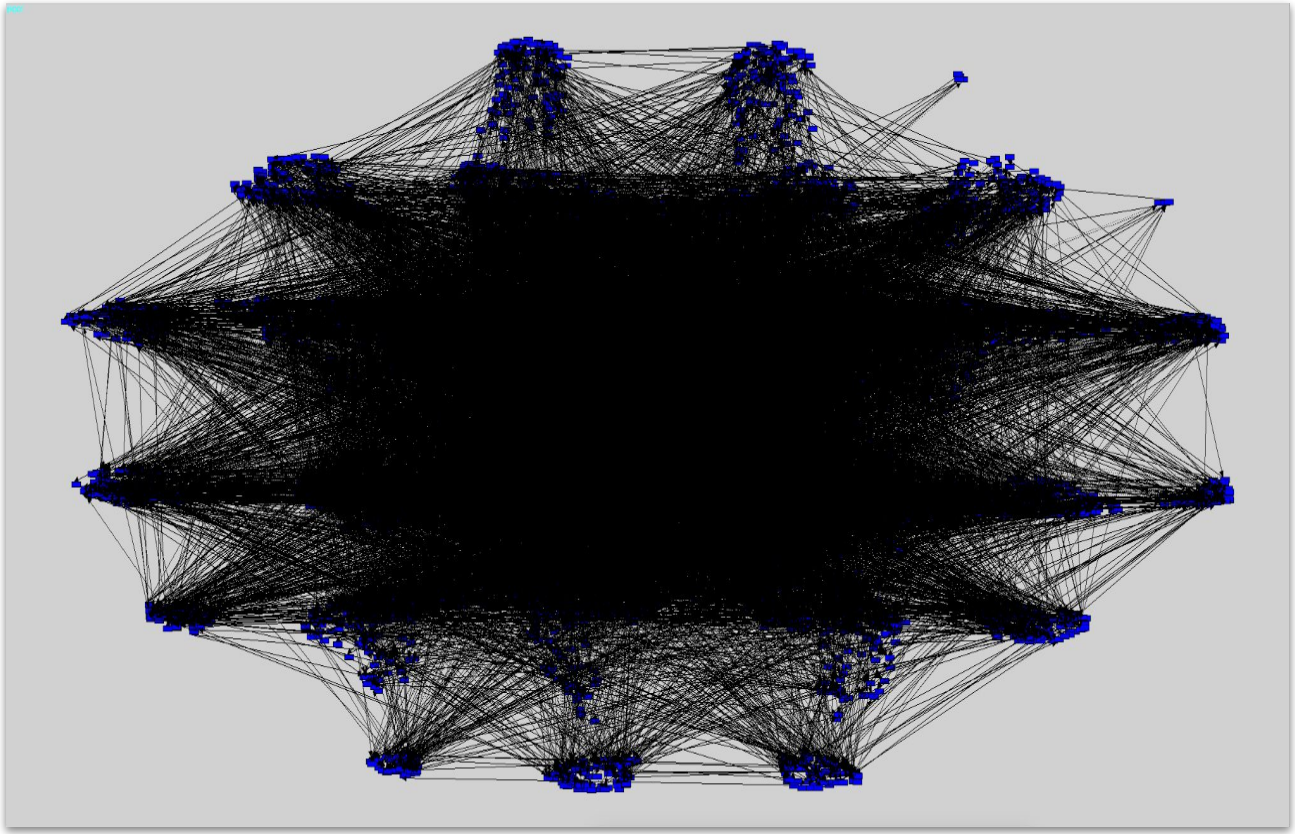
JobMaster

- Responsible for managing the execution of a single JobGraph

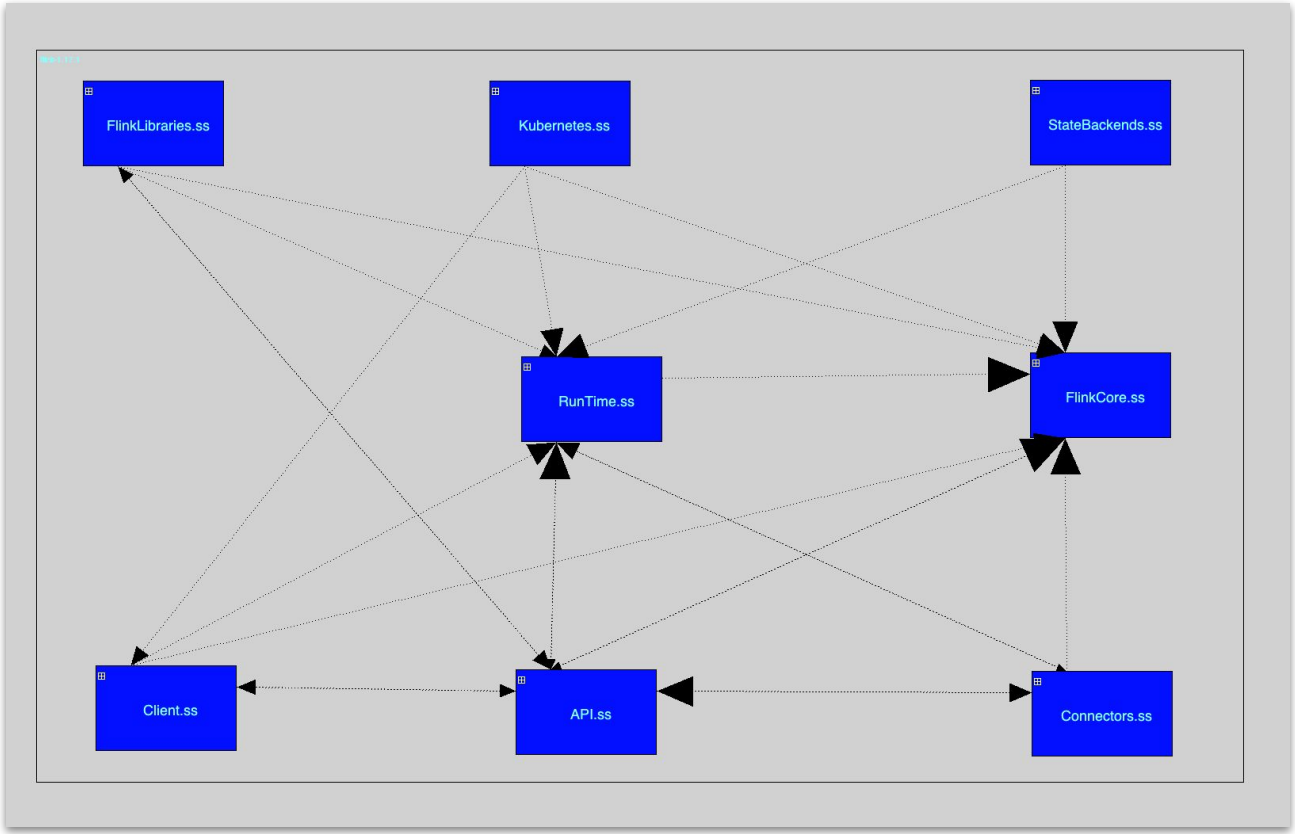
Concrete Architecture - Derivation Process



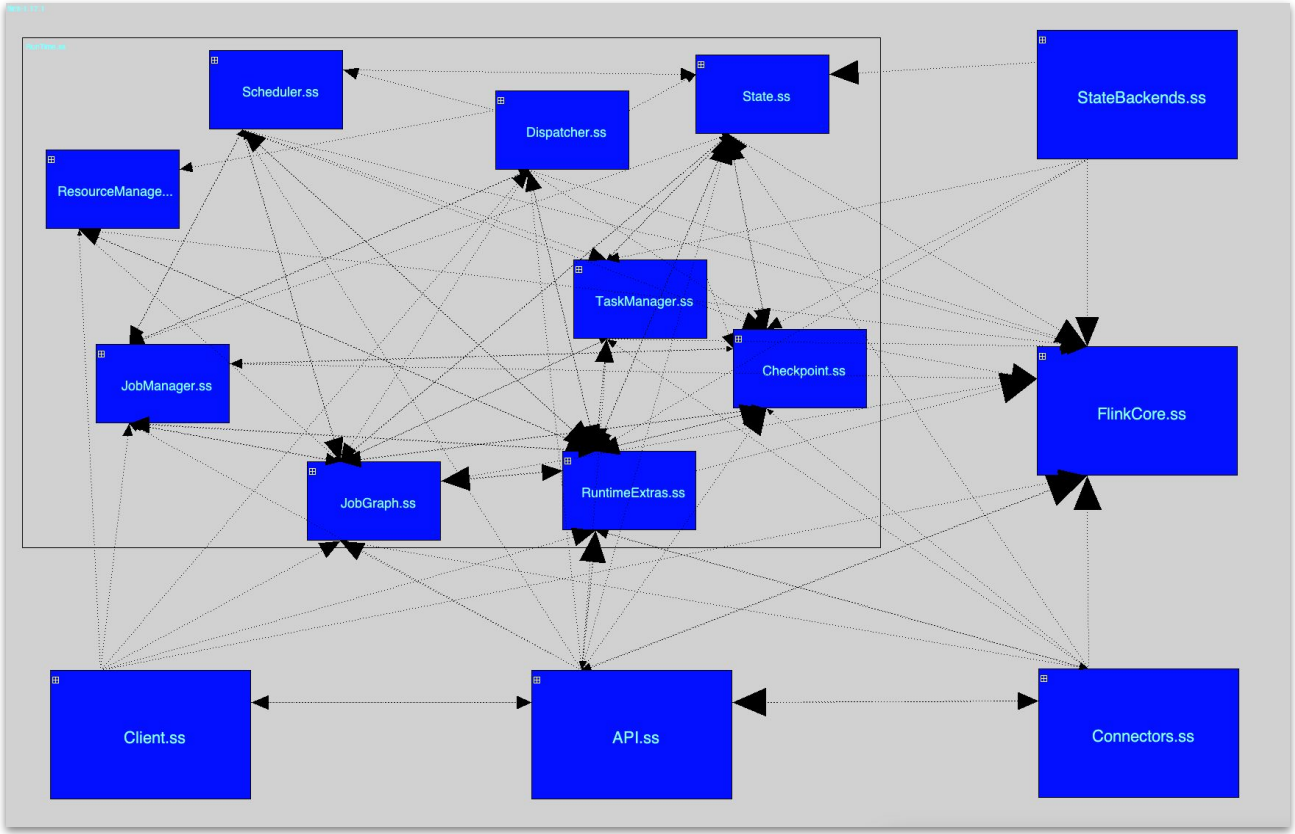
Concrete Architecture - 1st Iteration



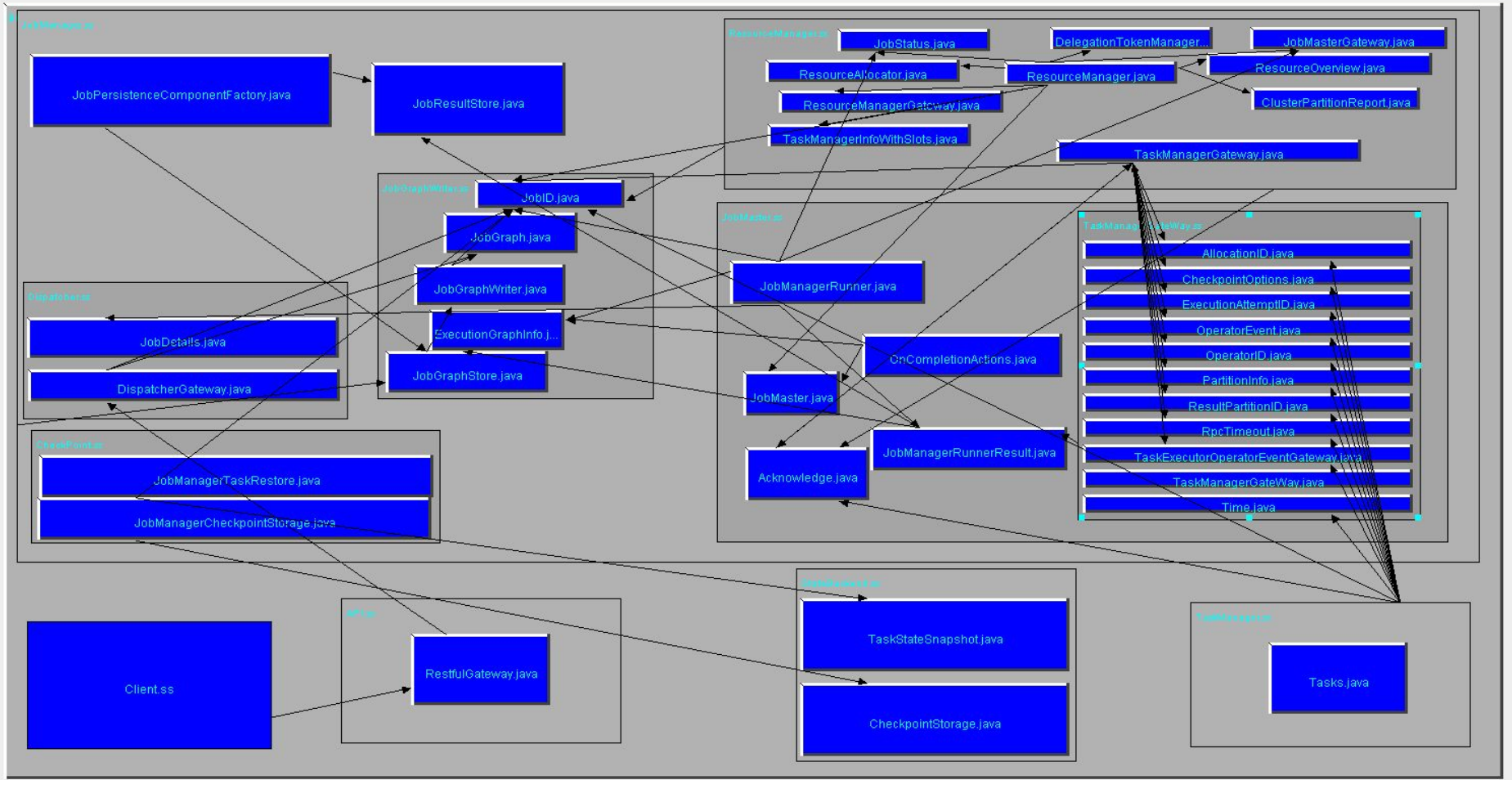
Concrete Architecture - 2nd Iteration



Concrete Architecture - 3rd Iteration



Concrete Architecture - 4th Iteration

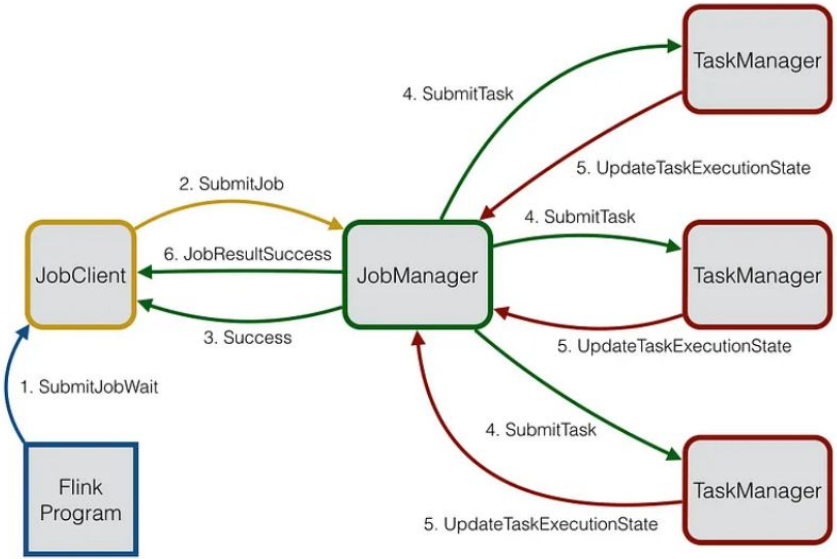


Architectural Style: Pipe and Filter

JobManager **coordinates** the distributed execution of Flink Applications

It decides when to schedule the next task (or set of tasks), **reacts** to finished tasks or execution failures

There are a lot of dependencies, but they are implementations of abstract interfaces, data structures, and generic protocols that allow each subsystem to operate independently.



Flink Job Execution Process

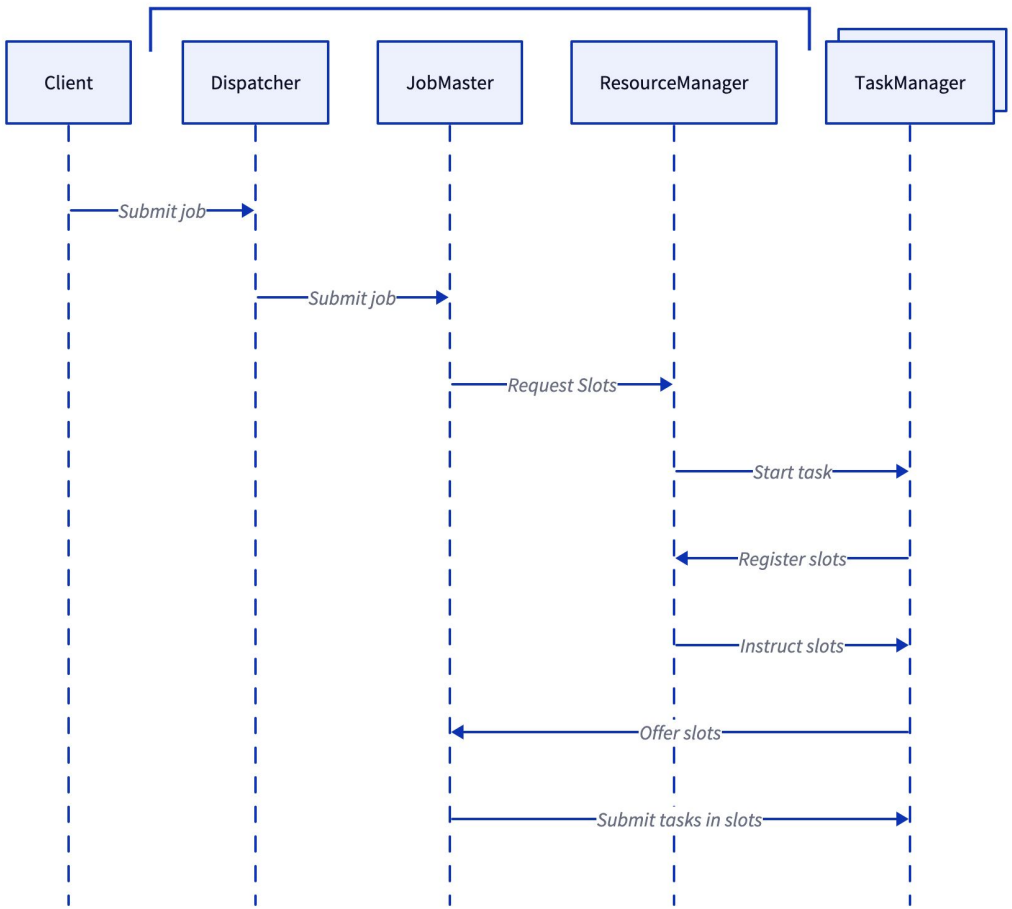
Design Patterns

Adapter (based on implementation, there is adapter for different sizes of clusters, as well as for different data sources)

Façade (JobManager provides gateways between dispatcher and client/api, and between ResourceManager/JobMaster and TaskManagers)

Master-Slave (JobMaster to ResourceManager to TaskManagers relation)

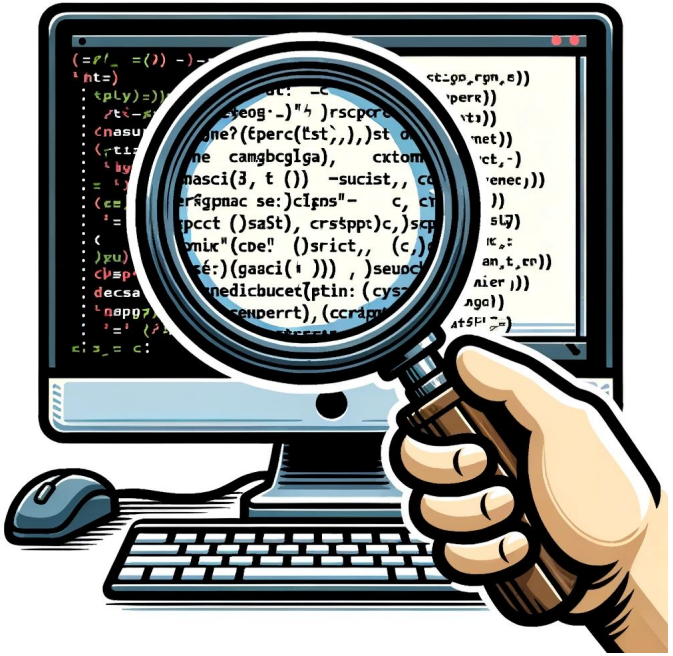
Job Manager Components



JobManager Source Code Deep Dive

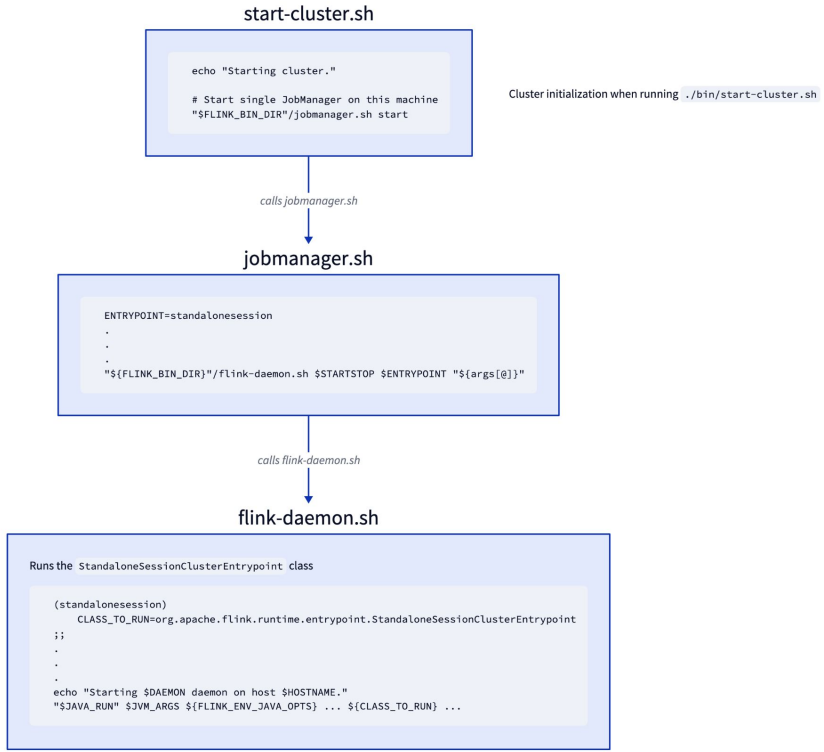
Investigating two scenarios:

- Flink cluster startup/initialization
- Flink job submission



Flink cluster startup/initialization

Diagram for `./bin/start-cluster.sh`



Flink cluster startup/initialization continued

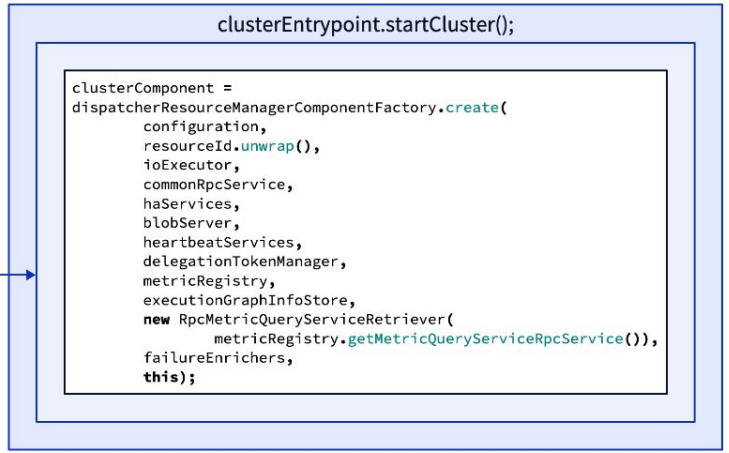
StandaloneSessionClusterEntrypoint.java

```
StandaloneSessionClusterEntrypoint entrypoint = new StandaloneSessionClusterEntrypoint(configuration);
ClusterEntrypoint.runClusterEntrypoint(entrypoint);
```

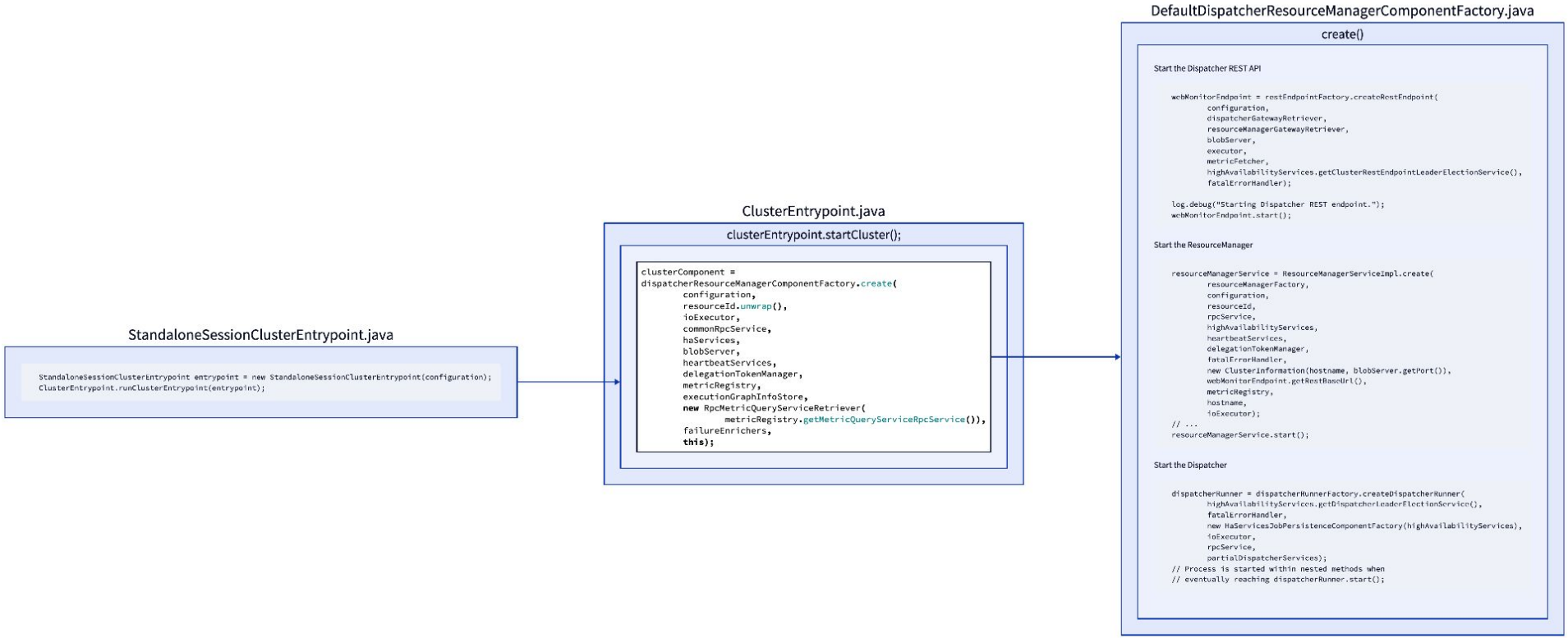
Flink cluster startup/initialization continued

StandaloneSessionClusterEntrypoint.java

```
StandaloneSessionClusterEntrypoint entrypoint = new StandaloneSessionClusterEntrypoint(configuration);
ClusterEntrypoint.runClusterEntrypoint(entrypoint);
```



Flink cluster startup/initialization continued

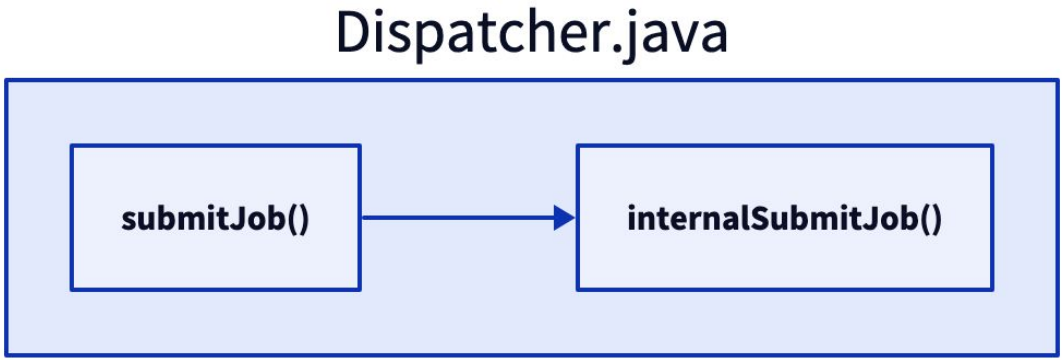


Observations

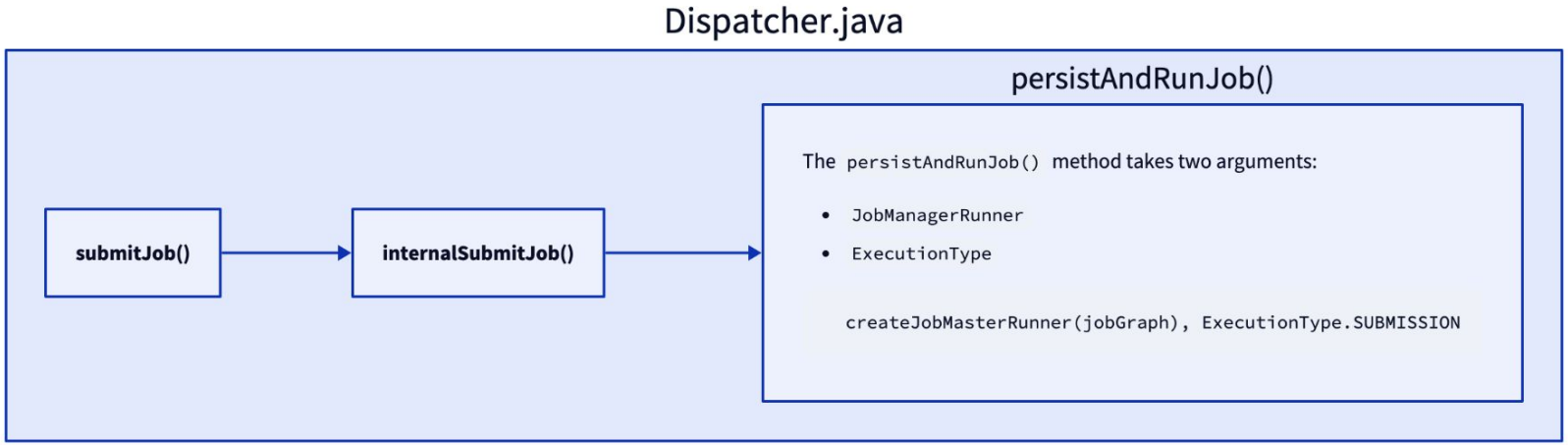
- The ResourceManager and Dispatcher are loosely coupled.
  - Don’t have a direct dependency.
  - One can fail without affecting the other.
- Lots of factory patterns being used within the source code
  - clusterComponent = dispatcherResourceManagerComponentFactory.create(...)
- First encounter of asynchronous methods/patterns
  - clusterComponent.getShutDownFuture().whenComplete(....)
    - CompletableFuture<ApplicationStatus>
- The JobMaster is not initialized.

Flink job submission

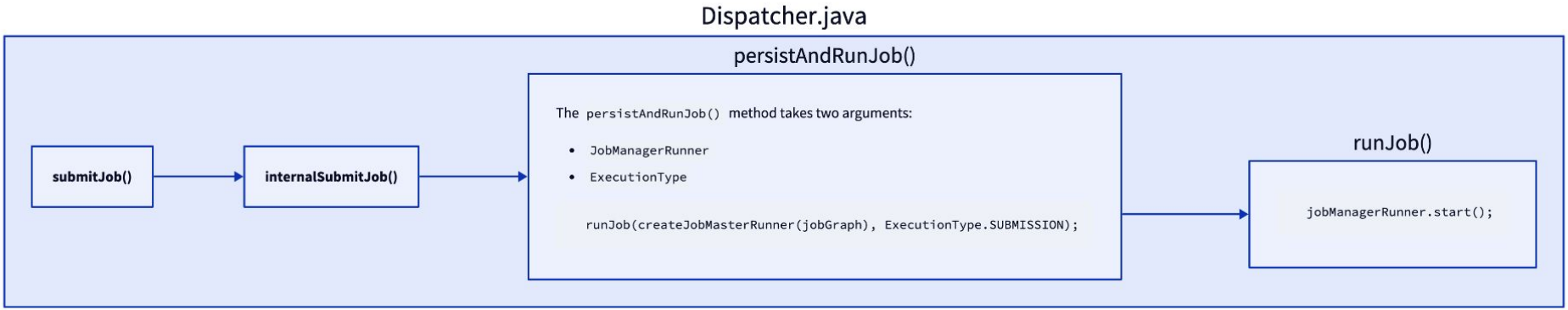
Flink job submission



Flink job submission



Flink job submission



Observations

- Once again, a lot of asynchronous patterns
  - CompletableFuture<CleanupJobState>
  - CompletableFuture<JobManagerRunnerResult>
- Once again, a lot of factory patterns
  - jobManagerRunnerFactory.createJobManagerRunner(...)
- Use of ‘runners’ (JobManagerRunner, JobMasterRunner)
  - Abstractions used to manage the lifecycle and execution of components
- JobMaster initialized on job submission
- Inconsistency between JobManager and JobMaster
- Use of ‘leaders’ for fault tolerance

```
@Nonnull private final LeaderRetrievalService dispatcherLeaderRetrievalService;
@Nonnull private final LeaderRetrievalService resourceManagerRetrievalService;
```

Changes to the JobManager (FLIP-6)

Single Job JobManager

The most important change is that the **JobManager handles only a single job**. The JobManager will be created with a JobGraph and will be destroyed after the job execution is finished. This model more naturally maps what happens with most jobs anyways.

Cross-job functionality is handled by other components that wrap and create JobManagers. This leads to a better separation of concerns, and a more modular composability for various cluster managers.

https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=65147077

Flink / FLINK-4344

Implement new JobManager

Details

Description

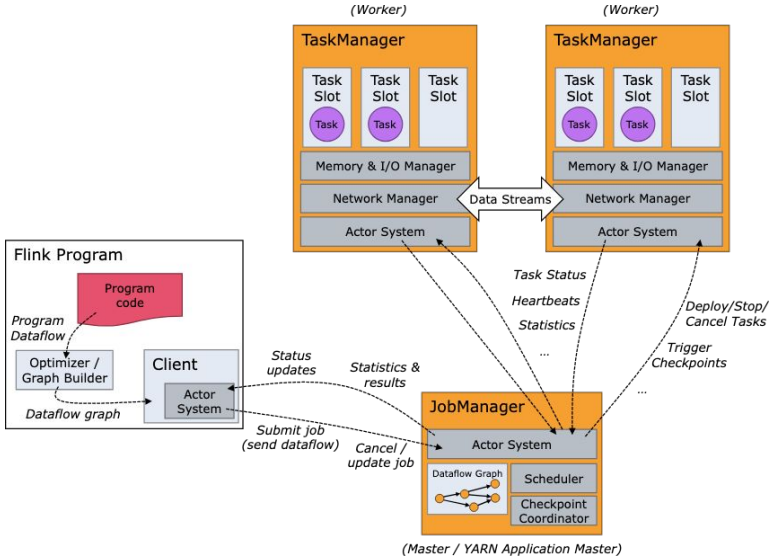
This is the parent issue for the efforts to implement the `JobManager` changes based on FLIP-6 (https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=65147077)

Because of the breadth of changes, we should implement a new version of the `JobManager` (let's call it `JobMaster`) rather than updating the current `JobManager`. That will allow us to keep a working master branch.
At the point when the new cluster management is on par with the current implementation, we will drop the old `JobManager` and rename the `JobMaster` to `JobManager`.

https://issues.apache.org/jira/browse/FLINK-4344

Flink Documentation vs. Code Implementation

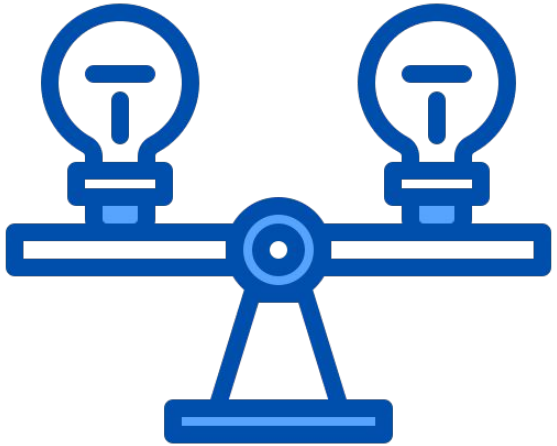
- Documentation explains two main processes
  - "*The Flink runtime consists of **two types of processes**: a JobManager and one or more TaskManagers*"
- Some parts of the documentation have not changed for 7+ years



Reflexion analysis: (comparison to the conceptual architecture and modifications)

SIMILARITIES

- The overall Pipe-and-filter architecture flow from converting the JobGraph into an ExecutionGraph, scheduling tasks, resource allocation and managing task events.



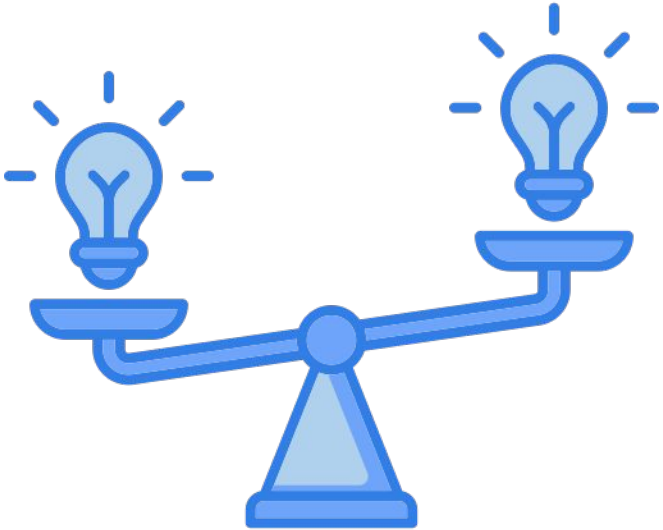
Reflexion analysis: (comparison to the conceptual architecture and modifications)

Notable unexpected dependencies were discovered, such as Dispatcher dependency with the checkpointing subsystems to support new API.

RATIONALE FOR DIFFERENCES

- Possible additional responsibilities or optimizations in the real world architecture which were not described in the conceptual architecture.

- High availability deployments using Zookeeper or Kubernetes, resulting in extra redundancy in the real-world implementation.



Reflexion analysis: (comparison to the conceptual architecture and modifications)

POSSIBLE MODIFICATIONS

- The conceptual architecture could benefit from more details on high availability deployments.

- If there are real-world optimizations the conceptual architecture can include them for a more comprehensive understanding

Lessons Learned

Lesson 1: Large software system requires a significant amount of effort to be put into pre-processing of source code/dependencies in order to retrieve a helpful Concrete Architecture. Sometimes human judgement is necessary.

Lesson 2: It was very helpful to complete the Conceptual architecture study first. The process allows for reflection on certain developmental choices.

Lesson 3: In Flink's case, the overall architectural designs remain consistent. However, it is clear that in some cases implementations differ from Architectural principles and ideas.

Lesson 4: In large softwares such as Flink, subsystems have large amount of dependencies. To present a clear overview of the architecture. It is crucial to discern important modules under the correct subsystems. Understanding development history and roadmap is important.

Thank you!

- The formal derivation process was followed to extract concrete architecture
- The overall architecture mainly follows conceptual architecture, Pipe-and-filter.
- Additional dependencies were discovered with other architectural concepts and design patterns.
- Actual source code has discrepancies with conceptual architecture and documentation.

Questions?