**ANKARA YILDIRIM BEYAZIT UNIVERSITY**

**FACULTY OF ENGINEERING AND NATURAL SCIENCES**

**SENG 307**

**SOFTWARE ARCHITECTURE**

**FINAL REPORT**

**APACHE FLINK – ENIAC**

**DEPARTMENT OF SOFTWARE ENGINEERING**

**MUSA YÜKSEL- 21050911018**

**ROJDA SÜSLÜ- 21050911017**

**SENA DİLAN ÇAKIR- 21050911027**

TABLE OF CONTENTS

[TASK I: Application and Tool Description for Apache Flink 5](#_b6vcghjt186z)

[Introduction of Apache Flink 5](#_l0j14p2mqh7b)

[Purpose and Functionality of Apache Flink 5](#_bixsm9fkq1p)

[Apache Flink Core Features 6](#_9l3bsuh16mn1)

[Advantages of Apache Flink 6](#_7e0bugq0v0tl)

[Disadvantages of Apache Flink 7](#_hj8bp0b7jyjh)

[Intended Customers and Scenarios: 7](#_qlmij5ycn0sj)

[Who uses Apache Flink? 8](#_3ggm0p5948mx)

[Use Cases 11](#_vur6flflaaqu)

[Task II: Documenting the Prescriptive Architecture of Apache Flink 13](#_lg6pgw4endam)

[Architectural Style & Patterns 13](#_ngtg5p1b38mt)

[Key Components and Their Interactions 16](#_3wlecqesiaj1)

[Flink Architecture Diagram 19](#_81dmnliwnos9)

[Task III: Analyze the descriptive architecture 21](#_vme1lnmdaloy)

[Identify Architectural Styles 21](#_6m3augqtquo1)

[Layered Architecture Style 21](#_f5cdzq9rvqtv)

[Pipe and Filter Style 22](#_xwhsb97xpegs)

[Master-Slave Architecture Style 22](#_82vwnciczck9)

[Event-Driven Architecture Style (EDA) 22](#_vw5gg69ydtxm)

[Architectural Pattern 23](#_iz6wln1a2mlb)

[a. Master-Slave Pattern 23](#_rt514dkoqg0r)

[b.Model-View-Controller (MVC) Pattern 23](#_vyo6d1rt48r7)

[c. Microservices Architecture 24](#_kp34j5bzevm)

[Diagram Creation 24](#_ego2e8myo8u0)

[Evaluate the structure 24](#_uclfksf9vtgo)

[● Modularity 24](#_krcpcf2o4jt7)

[● Coupling 25](#_rdkmbggrg8vz)

[● Cohesion 28](#_prnzxzjxr9bk)

[The System’s Scalability, Reliability, and Performance 29](#_p113vyihzx1m)

[Layered Style 29](#_fo3i9rcozvt9)

[Pipe and Filter Style 29](#_x7k0zps9asja)

[Distributed Architecture Style 30](#_mizux04sb91y)

[Model-View-Controller (MVC) Pattern 30](#_70bz223r37m6)

[Master-Slave Pattern 30](#_xmogykvak8nc)

[Code Fragman And Code Examples 31](#_1q6enpibciuf)

[Tool Output: 33](#_ybpofas6won8)

[Task IV: Critique The Descriptive Architecture 34](#_3nwm7ium71gf)

[Identify Architectural Smells 34](#_ioi32qq69ahg)

[Reliability 34](#_6ew4cxwswlaw)

[Maintainability 36](#_mx8d5k110dso)

[Security 39](#_4lmjkpwkjt0w)

[Architectural Constraints 45](#_r8b0lajyhy5d)

[1. Performance Constraints 45](#_sza37aabjioc)

[2. Compatibility Constraints 46](#_fks6lhp261rh)

[3. Modularity Constraints 46](#_rj9mbt271mb)

[4. Security Constraints 46](#_tsirussf0y6s)

[5. Fault Tolerance Constraints 46](#_6cpwoogfabkv)

[6. Scalability Constraints 46](#_6n3bw7opyvlx)

[Suggest An Improved Architecture 47](#_jhp8qsvd76y6)

[1. Performance Improvements 47](#_hiqgeihl2lm9)

[2. Compatibility Improvements 47](#_8db2tipjxmiu)

[3. Modularity Improvements 47](#_2fcddiitosyr)

[4. Security Improvements 47](#_24vjh1o5uirb)

[5. Fault Tolerance Improvements 47](#_86zrd2spbroj)

[6. Scalability Improvements 47](#_5o3xqr9f6cts)

[Tool Output 48](#_94bpqgor2gun)

[References: 58](#_cbh6i55tm776)

# TASK I: Application and Tool Description for Apache Flink

## Introduction of Apache Flink

Apache Flink was initially developed in 2010 as a research project called "Stratosphere" at the Berlin Technical University and became a fully-fledged project under the Apache Software Foundation in 2015. Flink is a framework and distributed processing engine designed for stateful computations over unbounded and bounded data streams.

This application collects and analyzes data which came from social media platforms in real-time. Leveraging Flink's in-memory processing speed and scalability features, we can effectively track users' posts, likes, comments, and shares. The application helps identify popular content and trends, providing marketing managers and social media professionals with data-driven insights for decision-making. Flink’s ability to run in any cluster environment makes it ideal for meeting big data analytics requirements, enabling real-time analysis of social media interactions and a better understanding of user behavior.

## Purpose and Functionality of Apache Flink

**Real-Time Data Processing:** TheFlink is designed to handle continuous data streams, enabling real-time analytics and immediate responses to events.

**Batch Processing**: It can process large, finite datasets in batch mode, making it suitable for analyzing historical data.

**Stateful Computations**: Flink supports stateful processing, allowing applications to maintain and manage state information throughout their lifecycle. This enables complex logic beyond simple event transformations, where results depend on the history of received events.

**Event Time Processing:** The framework can accurately process events based on their timestamps, facilitating the handling of out-of-order or late-arriving data.

**Scalability**: Flink’s distributed architecture allows it to scale horizontally, accommodating increased data volumes efficiently.

**Integration with Various Data Sources:** It integrates seamlessly with multiple data sources and sinks, such as Apache Kafka, HDFS, and NoSQL databases, enhancing its flexibility.

**Fault Tolerance:** Flink offers strong fault tolerance mechanisms to ensure data consistency and prevent loss during processing.

**Complex Event Processing:** It enables complex event patterns to be detected and acted upon in real-time, making it suitable for applications like fraud detection or monitoring.

**Streaming Analytics:** Flink provides capabilities for running continuous queries over data streams, allowing for ongoing insights and monitoring.

**Data Analysis Applications:** It facilitates and demonstrates us extracting information and insights from data. Traditionally, this is done by querying limited datasets and re-running queries or modifying results to include new data. With Flink, analysis can be executed through continuous updates, stream queries, or real-time processing of incoming events, leading to constantly disseminated and updated results.

**Data Pipeline Applications:** Flink allows for the transformation and enrichment of data being moved from one repository to another. Traditionally, the extract-transform-load (ETL) model is executed in batches at regular intervals. Within the Apache Flink, the process can run continuously, moving data to its targets with low latency.

## Apache Flink Core Features

1. **Real-Time and Batch Processing**: Can process both continuous data streams (unbounded) and specific datasets (bounded).
2. **Stateful Processing**: Allows complex business logic applications with state management based on the history of events.
3. **High Performance**: Provides low-latency data processing with in-memory computation, allowing for fast execution of operations.
4. **Event Time Processing**: Enables processing of events using timestamps, effectively handling late-arriving data.
5. **Advanced Fault Tolerance**: Prevents data loss and ensures application continuity with checkpoints and automatic recovery mechanisms.
6. **Horizontal Scalability**: Applications can be executed in parallel across thousands of tasks and distributed across multiple machines.
7. **Diverse Connectors**: Offers integration with popular data sources and sinks like Apache Kafka, Amazon Kinesis, and HDFS.
8. **Multiple Programming Language Support**: Provides flexibility by allowing development in languages such as Java, Scala, and Python.
9. **Different Abstraction Levels**: Offers high-level APIs (Stream SQL, Table API) and lower-level controls (DataStream API, ProcessFunction API) for user control.

### Advantages of Apache Flink

1. Real-Time Analytics
2. Exactly Once State Consistency
3. Data Transformation and Enrichment
4. Real-Time Analytics
5. Continuous Processing
6. Advanced Out-of-Order Data Management
7. Stream and batch processing
8. Real-time data processing
9. Fault-tolerant
10. Easily scalable
11. Handle Large-scale data processing
12. Easy integration with other big data tools

### Disadvantages of Apache Flink

1. **Complex Setup and Configuration:** Setting up and configuring Flink can be complicated and may require specialized configurations. Ensuring stable operation on distributed systems can be challenging without precise tuning.
2. **High Resource Consumption:** Flink is optimized for low-latency, high-throughput data processing, which can lead to high CPU and memory consumption, especially when working with large datasets.
3. **Difficult Debugging Process:** Debugging in Flink, especially in distributed environments, can be more complex compared to some other data processing tools. Tracking and resolving errors may require significant expertise.
4. **Limited Support and Documentation:** Flink has a smaller user base compared to more popular data processing tools like Spark, which can make it harder to find documentation or community support for certain issues.

## Intended Customers and Scenarios:

Apache Flink is designed for a diverse range of professionals and industries that require real-time data processing and large-scale data analysis. Key users include big data engineers, data scientists, and software developers who work with streaming data pipelines and need to derive insights from massive datasets in real time.

Flink is especially valuable for industries that rely on continuous data streams and quick decision-making, such as:

1. **Retail and E-Commerce:** With real-time processing, Flink supports personalized recommendations, inventory management, and customer behavior analysis. This helps e-commerce platforms and retail businesses enhance customer experience by delivering customized offers and adjusting stock based on demand trends.
2. **Healthcare and Life Sciences:** Healthcare providers and research facilities use Flink to analyze data from patient monitoring devices, medical imaging, and other real-time sources to improve patient outcomes and optimize medical research. For instance, Flink can help monitor patient vitals in real time, alerting healthcare teams to any critical changes.
3. **Financial Technology (FinTech):** In the finance sector, real-time data analysis is crucial for applications such as fraud detection, risk assessment, and high-frequency trading. Flink’s low-latency processing enables financial institutions to react instantly to market changes and secure transactions.
4. **Internet of Things (IoT):** Flink’s ability to handle data streams makes it ideal for IoT applications, where devices generate vast amounts of data continuously. Industries like manufacturing, smart cities, and logistics leverage Flink to monitor machine performance, optimize energy usage, and track assets in real time.
5. **Telecommunications:** Telecom companies use Flink to monitor network performance, enhance service quality, and detect anomalies in real time. This helps ensure network reliability and allows for rapid troubleshooting of issues, which is essential for maintaining high service standards.

### Who uses Apache Flink?

**NortonLifeLock**

NortonLifeLock is a global cybersecurity and internet privacy company that offers services to millions of customers for device security, and identity and online privacy for home and family.  
  
 NortonLifeLock offers a VPN product as a freemium service to users. Thus they need to enforce usage limits in real time to stop freemium users from using the service when their usage is over the limit. The challenge for NortonLifeLock is to do this in a reliable and affordable fashion.  
  
 NortonLifeLock simplified the implementation of user and device-level aggregation by adopting Apache Flink.



**Samsung SmartThings**

As an independent subsidiary of Samsung, SmartThings is one of the leading IoT ecosystems in the world, creating the most effortless way for anyone to create a smart home.  
  
 Samsung SmartThings were running into issues like having the resources reserved for individual applications. This caused a delay and performance degradation while processing data. It eventually led them to high costly overhead at maintaining workloads in operations. They had to re-architect the data platform.  
  
 They moved from Apache Spark to Apache Flink.



**BT Group**

BT Group is the UK’s leading telecommunications and network provider and a leading provider of global communications services and solutions, serving customers in 180 countries. Its principal activities in the UK include the provision of fixed voice, mobile, broadband, and TV (including Sport), and a range of products and services over converged fixed and mobile networks to consumer, business, and public sector customers.  
  
 BT needed a service-monitoring application to support the rollout of Digital Voice, its new consumer product enabling high-definition voice calling over its UK broadband network.  
  
 BT built an event-driven analytics service using Apache Flink, to ingest, process, and visualize service data.



**Autodesk**

Autodesk, a leading provider of 3D design and engineering software, wants to do more than create and deliver software. It also wants to ensure its millions of global users have the best experience running that software. Autodesk makes software for people who make things. They serve 200+ million customers. They needed to eliminate silos to find and fix customer issues faster.

They wanted a consistent way to collect and measure metrics with a small operations team without escalating costs or creating data lock-in.  


**NHL**

The National Hockey League is the second-oldest of the four major professional team sports leagues in North America. Today, the NHL consists of 32 Member Clubs, each reflecting the League’s international makeup, with players from more than 20 countries represented on team rosters.  
  
 NHL was facing several technical challenges like determining the features required and modeling methods to predict an event that has a large amount of uncertainty, and determining how to use streaming PPT sensor data to identify where a face-off is occurring, the players involved, and the probability of each player winning the face-off, all within hundreds of milliseconds.  
  
 Leveraging Apache Flink, NHL was able not just to predict the winner of a face-off, but to build a foundation for solving a number of similar problems in a real-time and cost-efficient way.



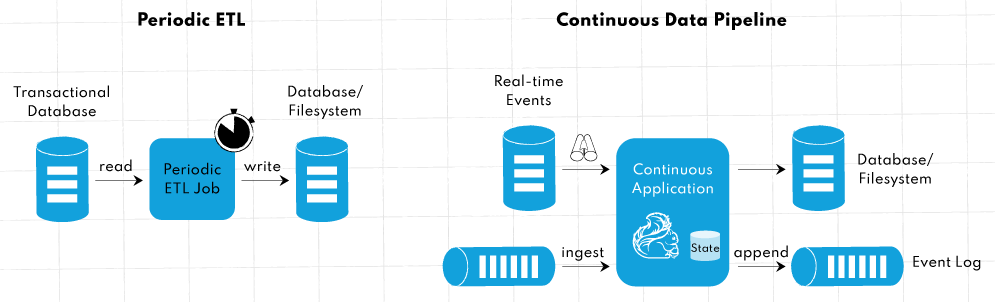
**Poshmark**

Poshmark is a leading social marketplace for new and secondhand style for women, men, kids, pets, home, and more. Their community of more than 80 million people across the US, Canada, Australia, and India is shaping the future of shopping to be simple, social, and sustainable.  
  
 Poshmark has been focusing on achieving top-line growth through personalization and enhancing user experience. The initial approach of using batch processing for personalization and security did not meet expectations for customer experience improvement.  
  
 Poshmark designed real-time personalization using real-time data enrichment with Apache Flink.

## Use Cases

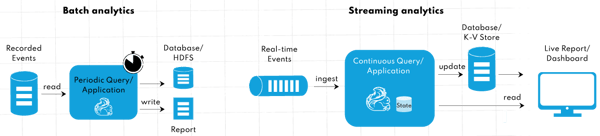
Flink can be used for a variety of use cases, such as:

* **Event-driven applications:**These are stateful applications that ingest events from one or more event streams and react to incoming events by triggering computations, state updates, or external actions. Examples include fraud detection, online gaming, and social network analysis.
* **Stream and batch analytics:** All of these are analytical jobs that extract information and insight from raw data. Flink supports both traditional batch queries on bounded data sets and real-time, continuous queries from unbounded, live data streams. Examples include sentiment analysis, anomaly detection, and machine learning.
* **Data pipelines and ETL:**The processes that convert and move data between storage systems. Flink can read and write data from various sources and sinks, and perform transformations such as filtering, aggregating, joining, and enriching. Examples include data ingestion, data cleansing, and data integration.



**Periodic ETL vs. continuous data pipeline**

**2.Real-time data analytics:** Flink is a true streaming engine with very low processing latencies that is ideal and useful for processing data in near real-time, making it an excellent tool for monitoring and triggering actions or alerts (e.g., ad-hoc analysis of live data in various industries, customer experience monitoring, large-scale graph analysis, and network intrusion detection).



Batch and real-time processing with Flink

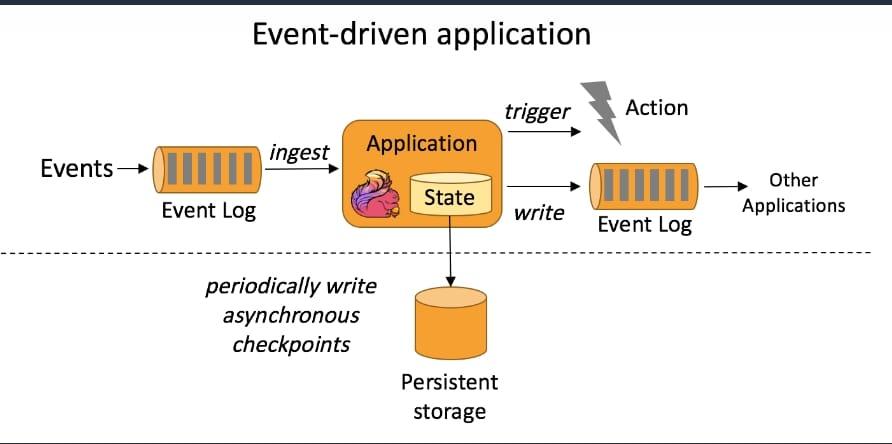
* **Machine learning:** FlinkML provides a library of distributed machine learning algorithms that run on top of the DataSet API, allowing developers to train models quickly with large datasets. FlinkML enables integration with other deep learning frameworks for more complex AI solutions.
* **Graph processing:** Gelly is a versatile graph processing and analysis library that runs on top of the DataSet API, providing graph computations.

# Task II: Documenting the Prescriptive Architecture of Apache Flink

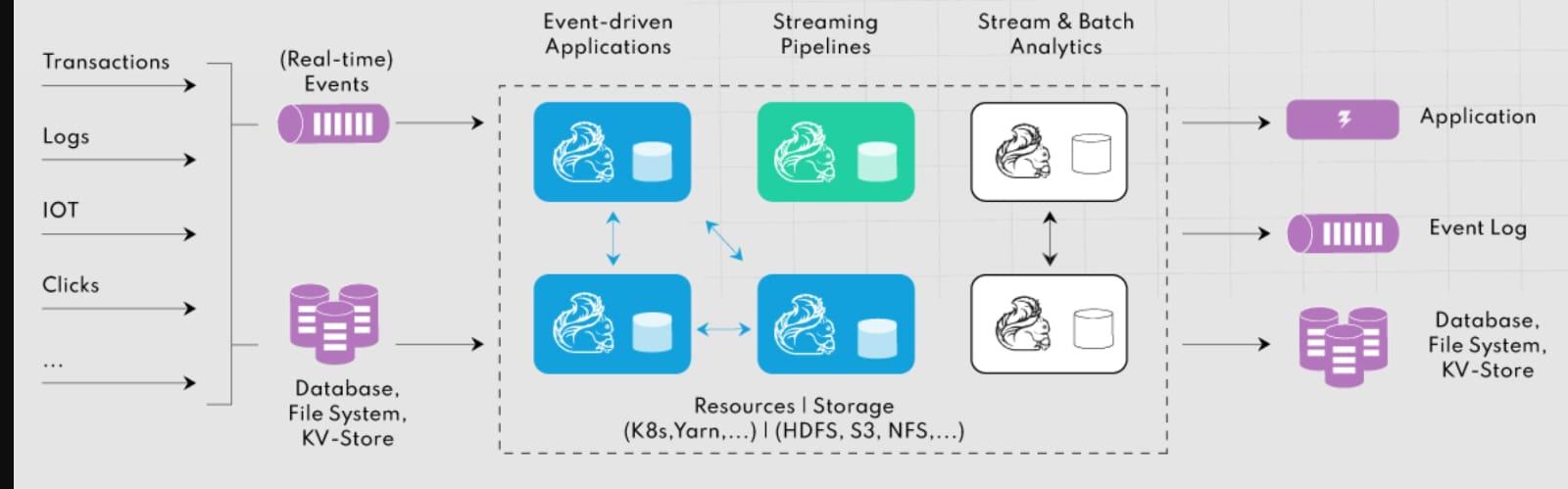
## **Architectural Style & Patterns**

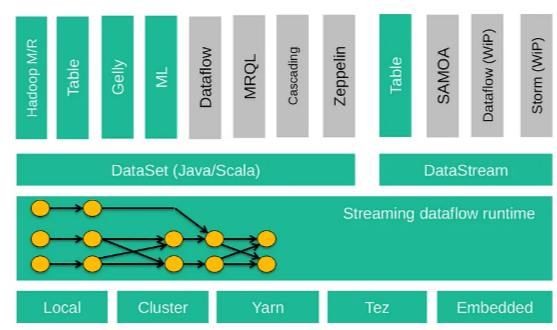
Apache Flink follows a **Distributed Stream Processing Architecture** and employs several architectural styles and design patterns:

* **Event-Driven Architecture (EDA)**: It processes real-time data by reacting to a continuous flow of events.

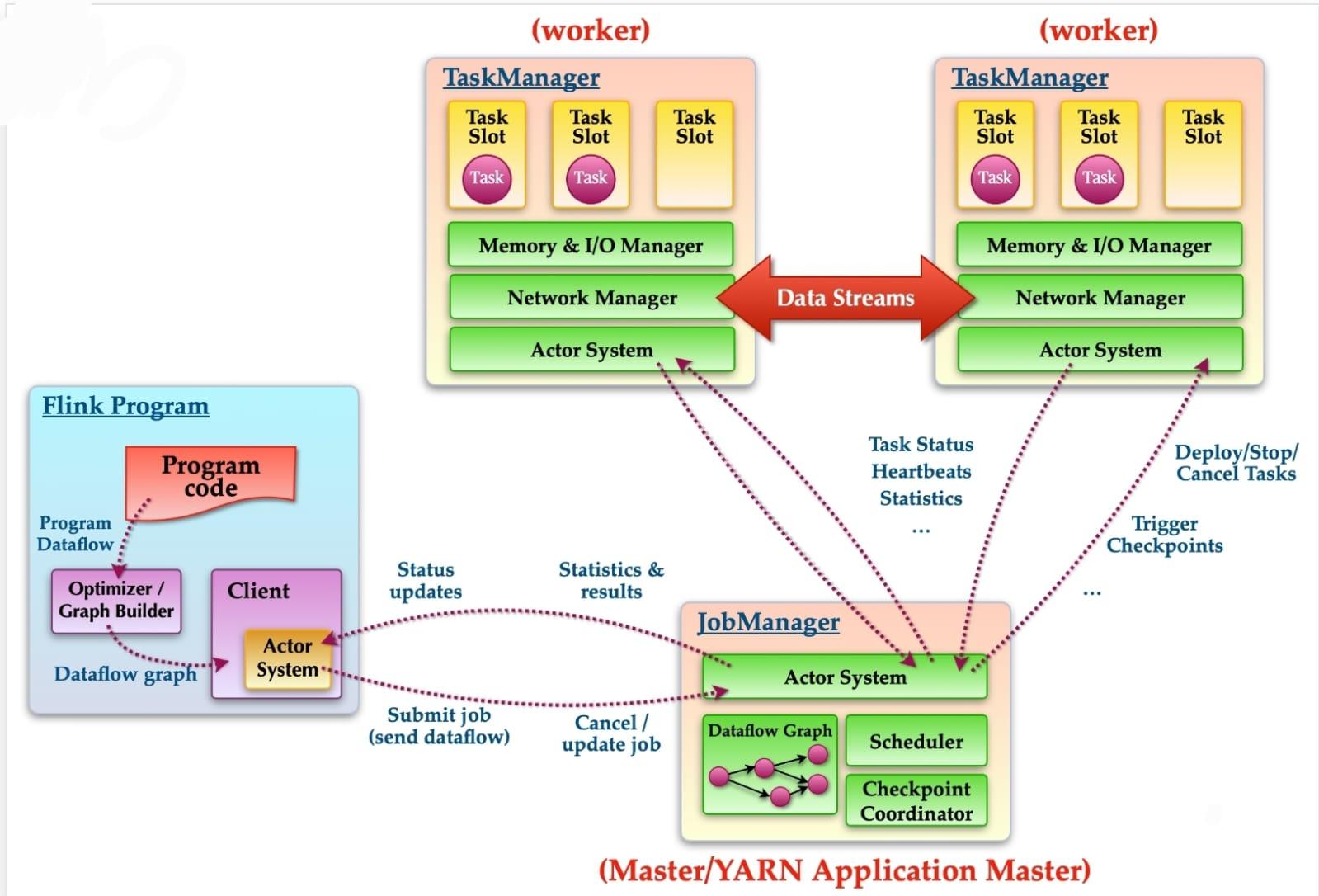


* **Micro-batching**: Although Flink processes data streams natively, therefore it can perform micro-batching, where data is grouped into small batches before being processed.
* **Pipeline Pattern**: For distributed data flow processing, jobs are represented as pipelines where tasks are executed in a sequence.

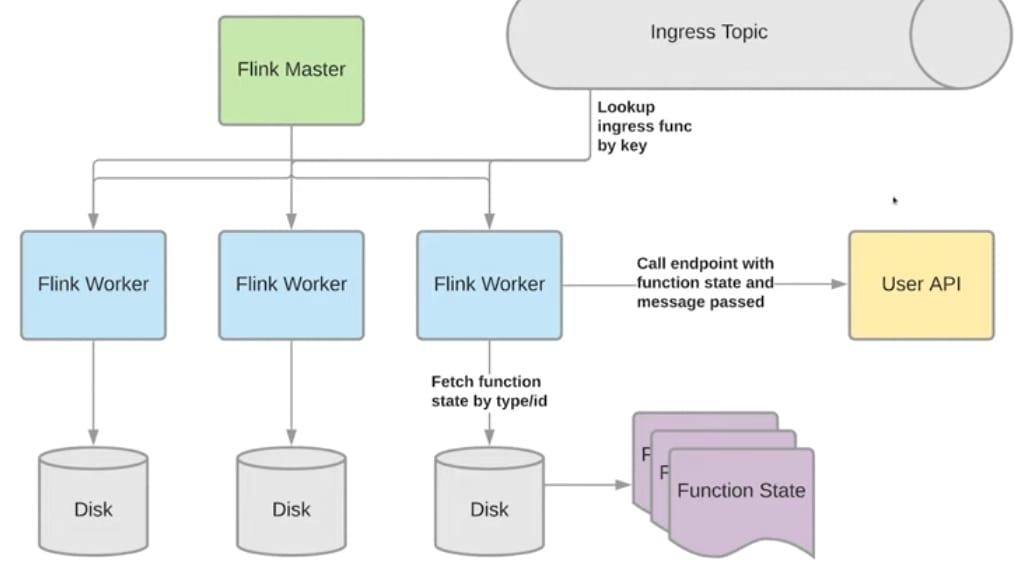


* **Layered Architecture:** Apache Flink has a layered software stack like an OSI model. Each layer abstract the details of the bottom layers and make it’s easier for the end-user to program application.

* **Master-Slave Pattern**:It Involves a centralized master node (Job Manager) controlling worker nodes (Task Managers).

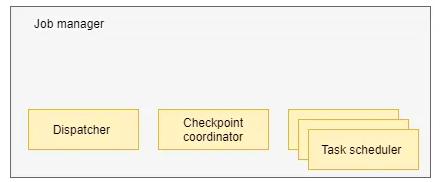


* **Actor Model**: Job Manager and Task Managers communicate via messages using the actor model for fault-tolerance and recovery



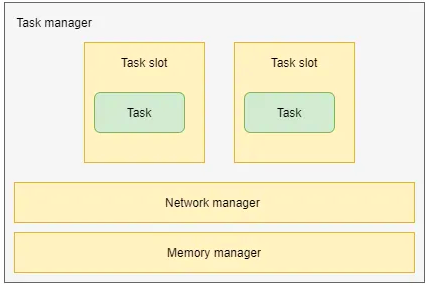
## Key Components and Their Interactions

**1. Job Manager**

* **Role**: The master node that coordinates the execution of jobs. It schedules tasks, monitors resource allocation, and handles fault tolerance.
* **Interaction**: It interacts with Task Managers and Resource Managers to allocate resources and restart tasks in case of failure.

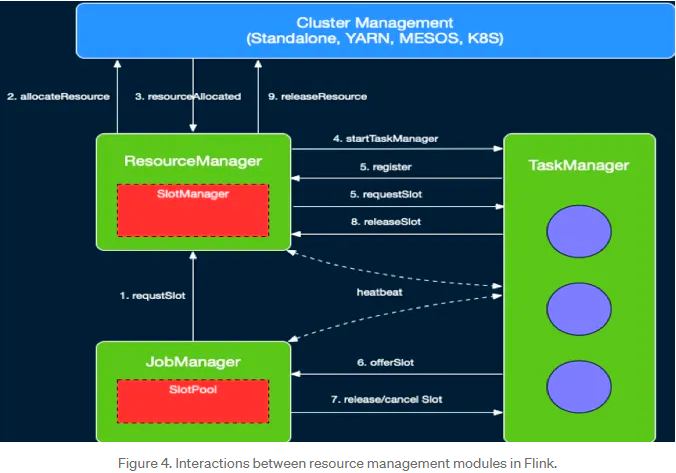
**2. Task Manager**

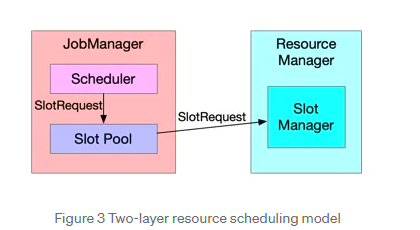
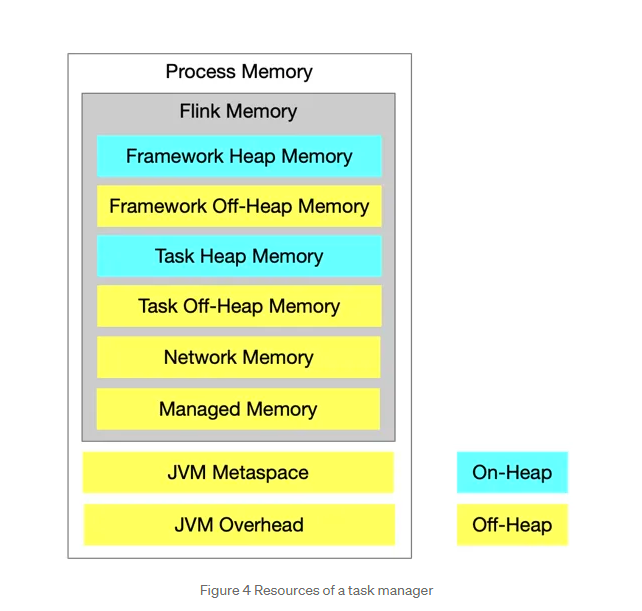
* **Role**: These are the worker nodes that execute the actual tasks. Each Task Manager has several slots where tasks are executed in parallel.
* **Interaction**: Task Managers receive tasks from the Job Manager and execute them in a distributed manner.



**3. Resource Manager**

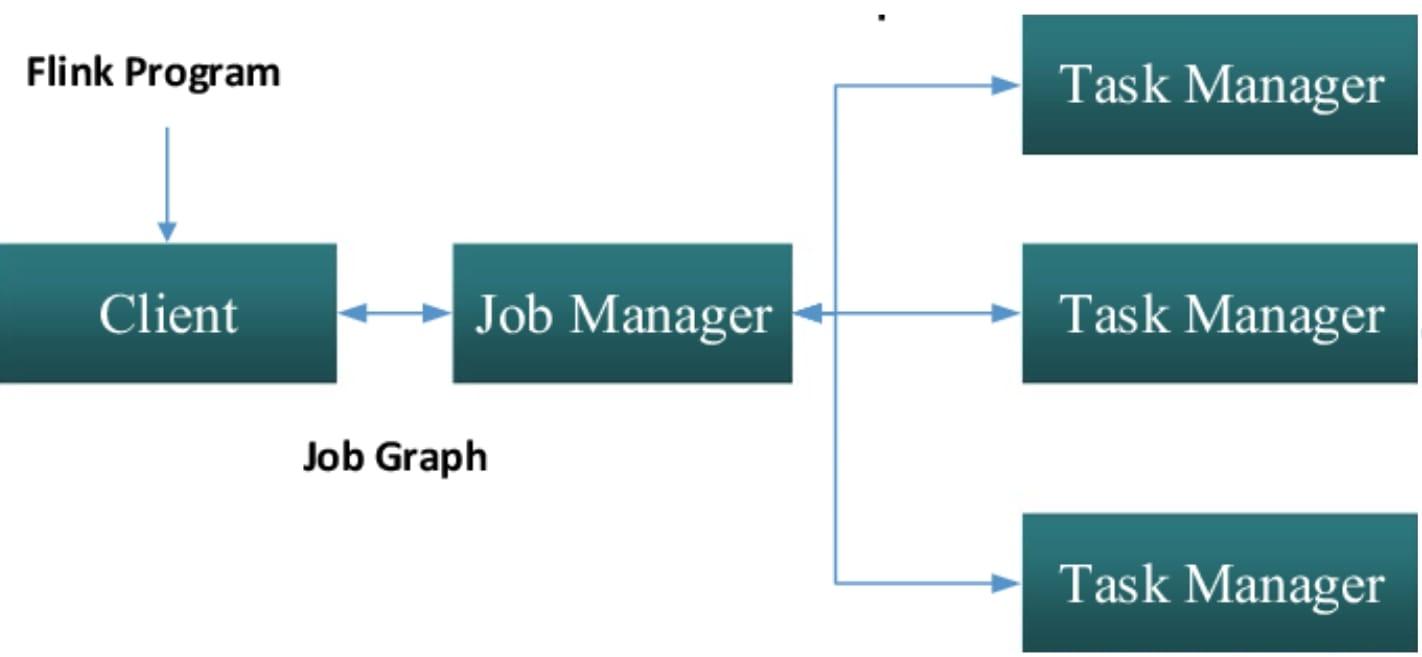
* **Role**: Manages resource allocation across different Task Managers.
* **Interaction**: It collaborates with the Job Manager to provision resources for distributed tasks.



****

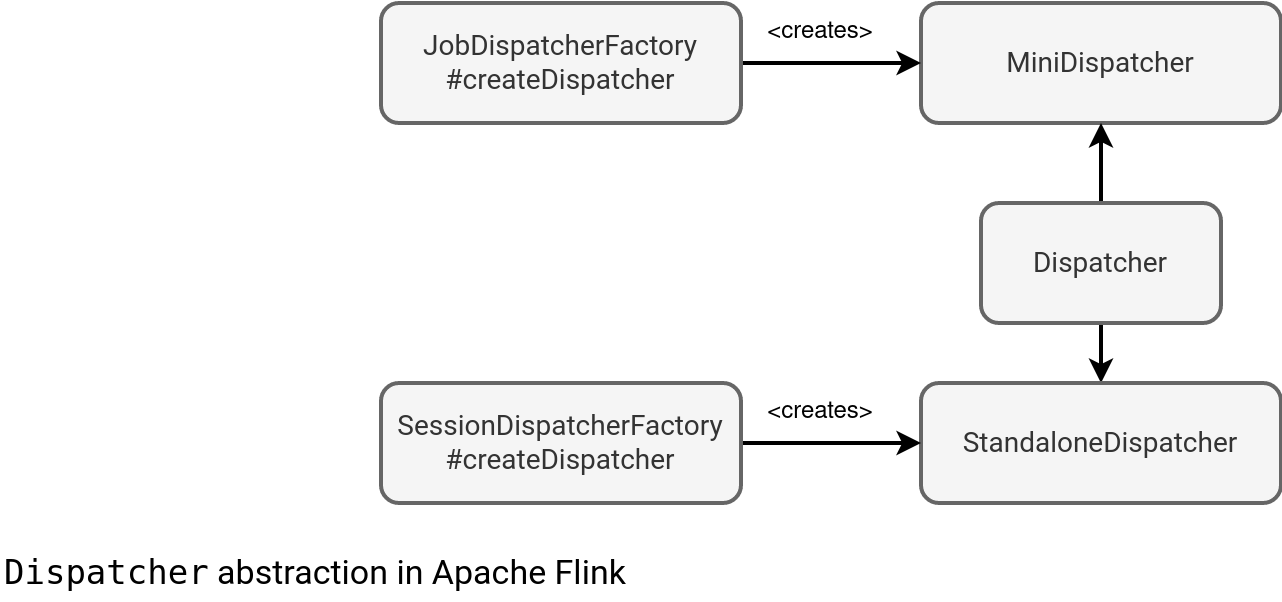
**4. Client**

* **Role**: In that part The client submits jobs to the Job Manager. The client can be an API or CLI that allows users to interact with the Flink cluster.
* **Interaction**: Once the job is submitted, the client communicates with the Job Manager, which takes over execution.



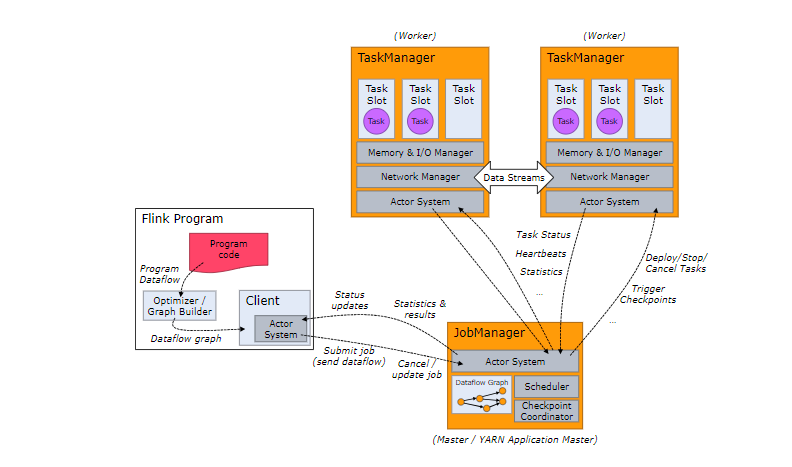
**5. Dispatcher**

* **Role**: A lightweight component that forwards jobs to the Job Manager for execution.
* **Interaction**: The dispatcher is responsible for distributing jobs to the correct Job Manager in case of a multi-tenant setup.



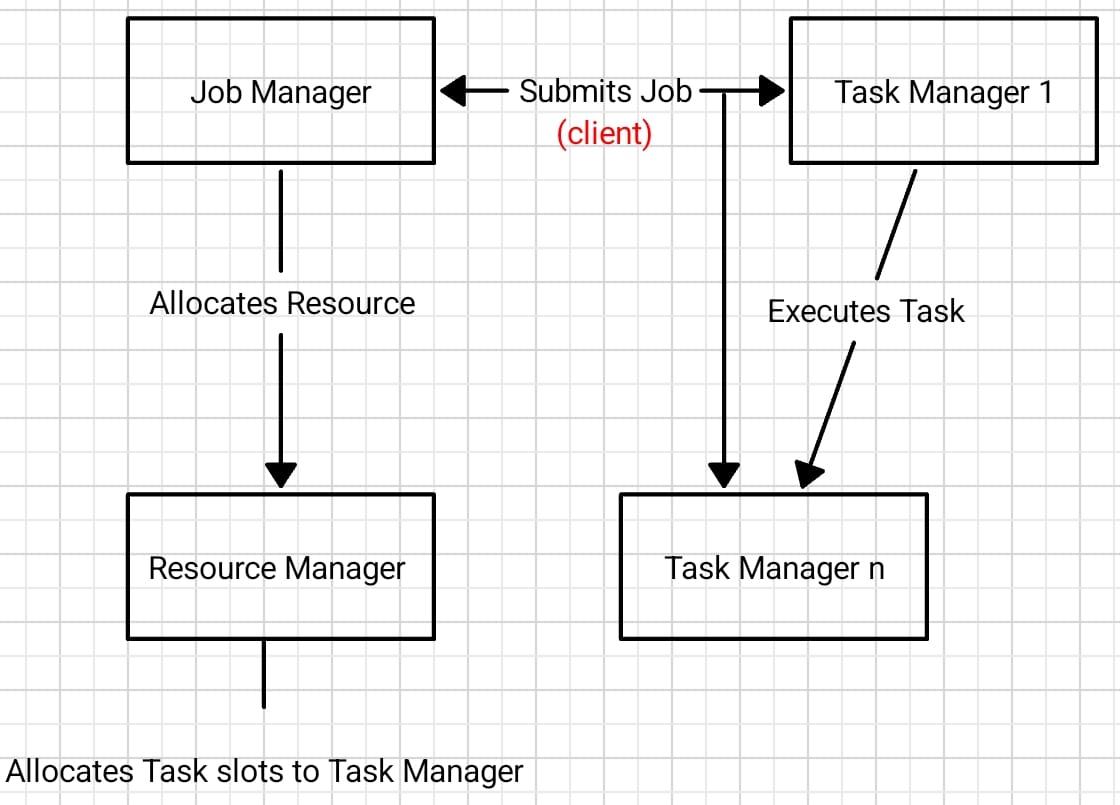
**6. Checkpointing and State Backend**

* **Role**: It ensures fault-tolerance by persisting state at regular intervals. If a failure occurs, Flink can restore the state and continue processing.
* **Interaction**: The Task Managers periodically and regularly snapshot the state to a state backend (e.g., HDFS, RocksDB), which is coordinated by the Job Manager.



## Flink Architecture Diagram

Below is a simplified diagram illustrating the components and their interactions:



**Claims of the Intended Architecture**

Claims of the Intended Architecture The architecture of Apache Flink seeks to achieve a number of important scalability and performance objectives:

**Scalability:**

The master-slave paradigm allows Flink to scale horizontally. The task is split up across several slots in each Task Manager, which are dispersed throughout nodes.

The system can scale to millions of events per second and is built to handle massive data streams.

**Fault Tolerance:** Flink can recover from task failures without losing data thanks to checkpointing and state recovery.

Event duplication and data loss during processing are prevented by the exactly-once processing semantics.

**High Throughput & Low Latency:** Apache Flink handles streams with little latency in real time. The architecture's distributed execution and pipelined processing are intended to maximize data throughput.

# Task III: Analyze the descriptive architecture

## Identify Architectural Styles

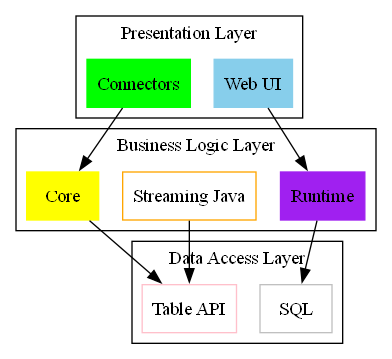
We used a code analysis tool like SonarQube to identify Apache Flink's architectural styles. We examined the dependencies and interactions between components. We visualized the diagrams using Graphviz.

### Layered Architecture Style

Each layer is dependent on a lower layer. For example, Presentation Layer (flink-connector, flink-web-ui) depends on Business Logic Layer.

Components in the Business Logic Layer (flink-core, flink-runtime) depend on the Data Access Layer (flink-table, flink-sql).

* Presentation Layer: Modules that interact with the user. (flink-connectors and flink-web-ui)
* Business Logic Layer: Modules that manage business logic. (flink-core,flink-streaming-java and flink-runtime)
* Data Access Layer: Data access and processing. (flink-table-api and flink-sql)



### Pipe and Filter Style

To convey data, each component depends on the one behind it. For instance, the data is processed by flink-core before being sent to flink-runtime. For data processing, components like flink-core and flink-runtime, respectively, create a sequential workflow.



### Master-Slave Architecture Style

* A Master component governs several Slave components in a Master-Slave architecture. The Slave components do these tasks while the Master oversees them.
* The master component, such as flink-core, controls the system and oversees tasks.
* The instructions that the Master gives them determine how the slave components (such as flink-runtime and flink-streaming-java) behave.



### 

### Event-Driven Architecture Style (EDA)

* SourceFunction: Since our Flink code contained continuous data flows, we can argue that this is an illustration of event-driven architecture. Every item that is retrieved from data sources is treated as an event by SourceFunction, and actions are carried out on these events.
* A function called ProcessFunction analyzes each event once it is received and generates the outcomes.
* Emit Events (From Event Source to Event Router): The Event Router receives events from the Event Source and uses them to determine what should be done.
* Route Events (From Event Router to Event Processor): Events are routed to the relevant Event Processor by the Event Router. The kind of event and the necessary processing determine this routing.
* Event Processor to Event Sink Process Output:

## Architectural Pattern

### a. Master-Slave Pattern

We believe that there is a "Master-Slave" model in Apache Flink's design.  
The Master node includes the main parts like the JobManager and the system for allocating resources. The Slave nodes, which are the TaskManagers, are responsible for running the tasks. The Master node breaks the work into smaller tasks and sends them to the Slaves, which helps to run things faster, use resources better, and reduce processing time. We figured this out by looking at how the JobManager and TaskManager work together in the code, especially in the functions JobManager.scheduleTasks() and TaskManager.executeTask().

### b.Model-View-Controller (MVC) Pattern

We believe that the Model-View-Controller is a high-level pattern for Apache Flink.  
When we look at the Layered Style, we see that it fits well with the MVC pattern.

* The Presentation Layer corresponds to the Display ı mean view. This is shown by the web-based monitoring parts in Flink, like the JobManagerUI and TaskManagerUI, which show job statuses and task execution in real-time.
* The Business Logic Layer corresponds to the interaction that means controller. In Flink, this is handled by the JobManager module, which controls job execution and manages resources. Important functions like JobManager.handleJobSubmission() and JobManager.submitJob() are part of the interaction between the user and the system.
* The Data Access Layer corresponds to the Model. In Flink, the TaskManager is in charge of actual and real data processing and computation, with functions like TaskManager.executeTask() or TaskManager.reportStatus() being important in the processing.

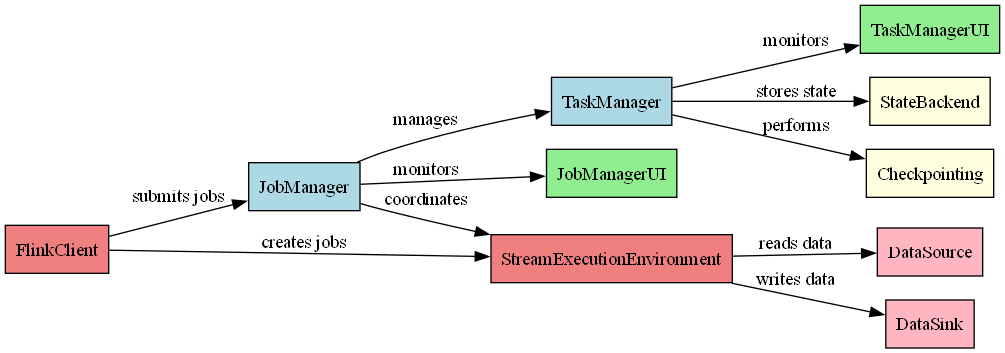
### c. **Microservices Architecture**

Flink works well with a microservices architecture because it manages big data processing through independent modules and services.

**Features**: The application is made up of small,tiny and independent services. Each service performs its own function and can be updated separately.

**In Flink**: Apache Flink has a structure with modularity, which allows different components to work independently. For example, the **JobManager** and **TaskManager** modules can be seen as separate services. Additionally, Flink's **StateBackend** and **Checkpointing** mechanisms help each service maintain its state and work independently

## Diagram Creation

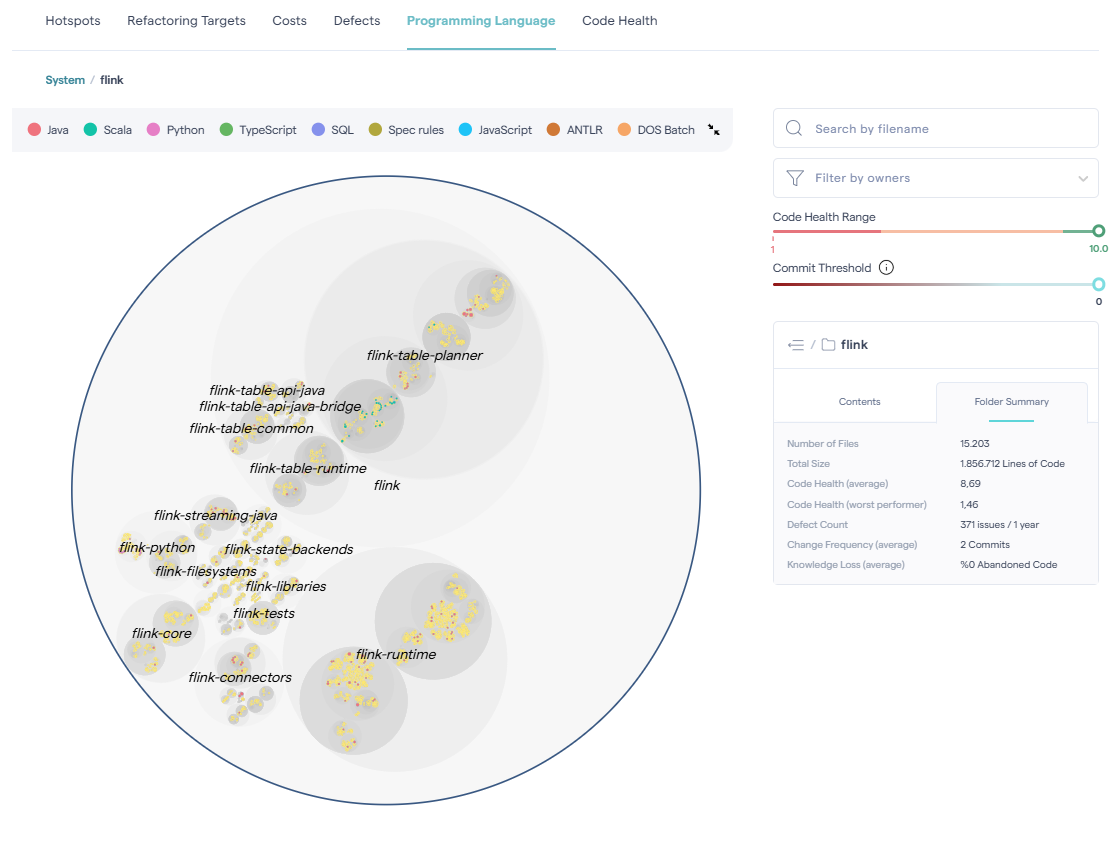


## Evaluate the structure

### Modularity

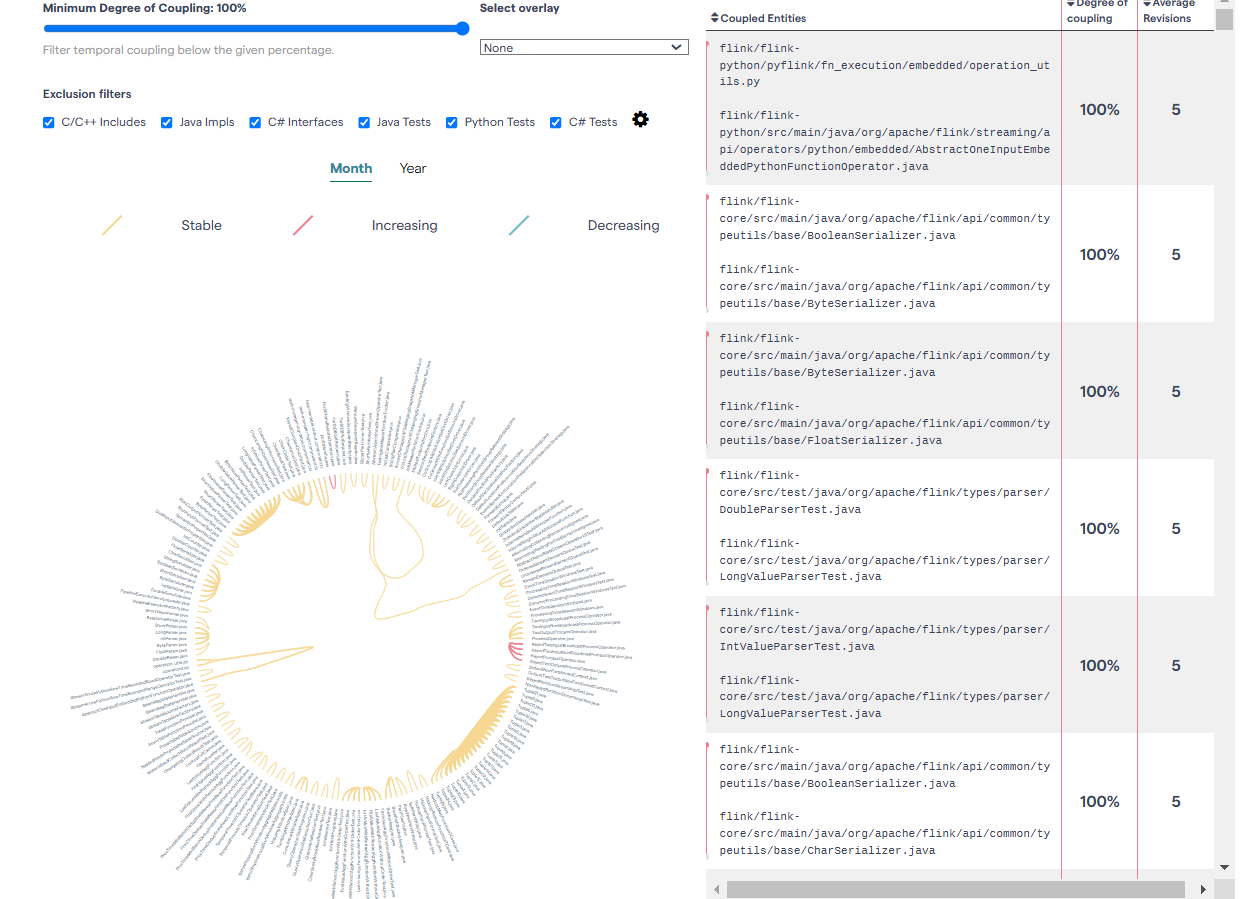
Different modules (e.g. flink-core, flink-table) are clearly separated in the image. This shows that Flink has a modular structure in general.

However, the dense amount of hotspots within modules suggests that some modules require more attention and there may be modularity problems in these areas.

This visually shows that the modularity of the Flink project is generally well assembled, but the hotspot parts in some modules can be marked by modularity issues.

### Coupling

Coupling describes how a component is dependent or not dependent on the other component.



In this chart and the list on the right, "Degree of Coupling" is stated as one hundred percent(100%). This means that the listed files are highly dependent on each other and are often changed at the same time. This indicates that there are high dependencies between modules or files.

The list on the right shows highly linked files. For instance:

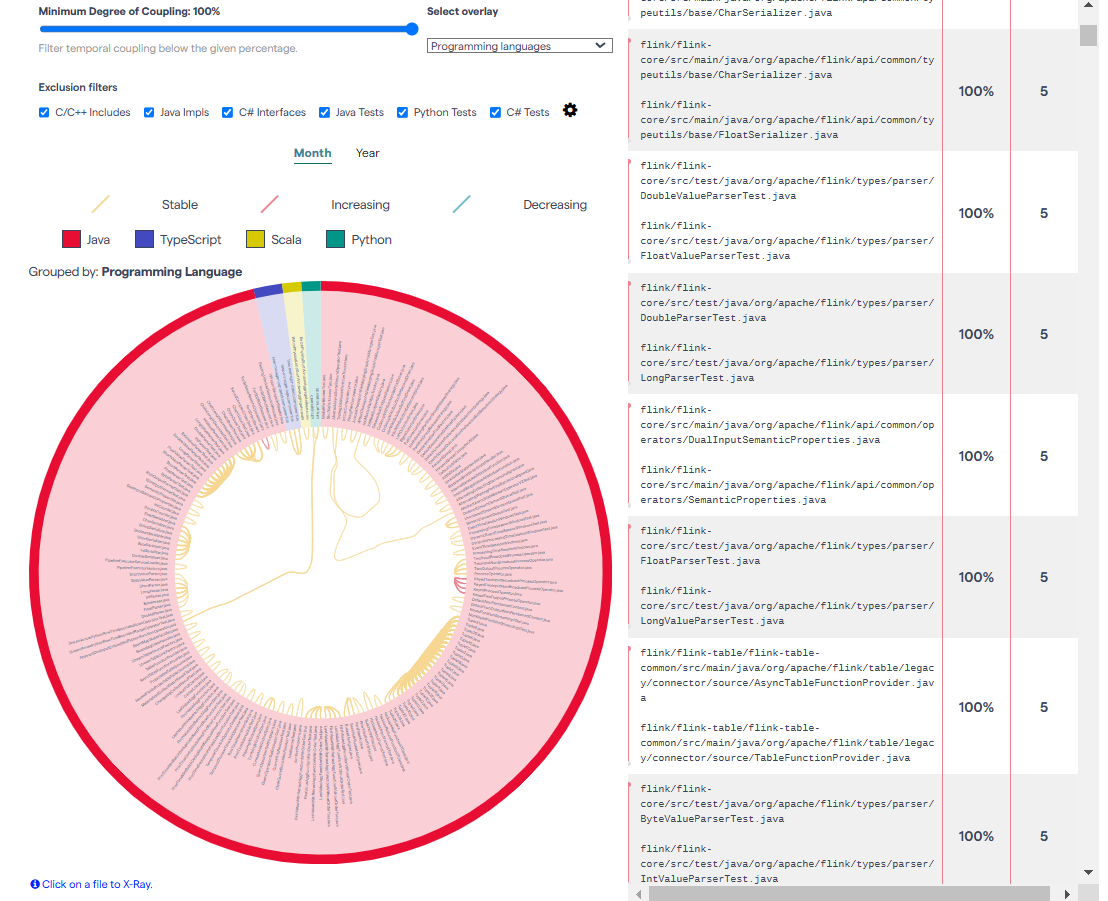
Classes such as (BooleanSerializer.java, ByteSerializer.java, FloatSerializer.java )are components that require modification together.

The fact that these files are related indicates that these components have to work together or are tightly interconnected.

This is an indication of low modularity or high coupling.



Coupled entities created by CodeScene.



When we look at this image, we can see the coupling among different programming languages and different row color has meaning different meaning.

### Cohesion

Cohesion refers to how closely related the functions within a module or class are.

## The System’s Scalability, Reliability, and Performance

### Layered Style

1. Scalability  
   The "connector" components and other modules, such as "sql" and "ui-test," can scale independently. This separation allows the system to handle increased loads by allocating resources to specific layers.
2. Reliability  
   Proper handling error mechanisms between the "connector" layer and external systems ensure that failures in external dependencies do not propagate to the entire system, increasing overall reliability.
3. Performance  
   Optimizations in the "sql" component such as indexing and query processing, improve data retrieval speeds, which directly enhances system performance.

### Pipe and Filter Style

1. Scalability  
   The system's components can be distributed across between multiple nodes, enabling better utilization of computational resources and scaling horizontally.
2. Reliability  
   Each filter operates independently, allowing updates or evaluations without affecting the entire system. This modular approach enhances the system's fault tolerance.
3. Performance  
   The pipe-and-filter style is well-suited for parallel data processing, which improves throughput and reduces processing latency.

### Distributed Architecture Style

1. Scalability  
   Components can be replicated and deployed on multiple nodes, supporting load balancing and fault tolerance. This design allows for easy scaling as demand increases.
2. Reliability  
   The distributed nature ensures that if one node fails, other nodes can clearly continue operations, minimizing system downtime and ensuring high availability.
3. Performance  
   Tasks are parallelized across multiple nodes, reducing bottlenecks and enhancing processing speeds. Each node handles a specific task, ensuring efficient resource utilization.

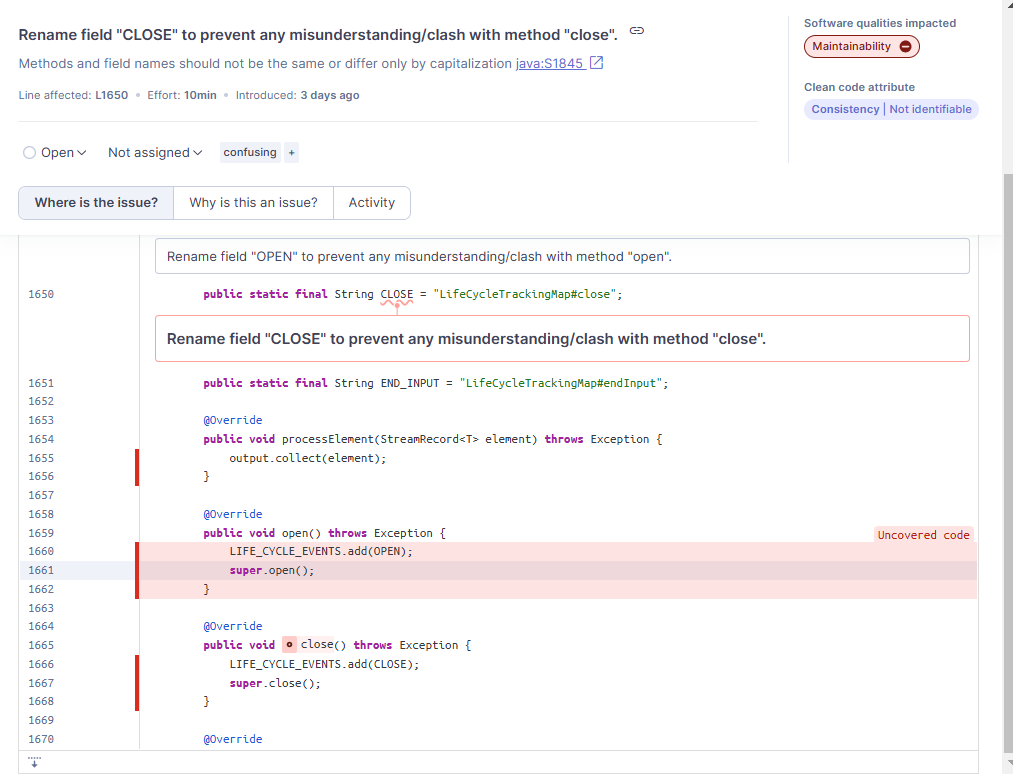
### Model-View-Controller (MVC) Pattern

1. Scalability  
   A load balancer can be integrated with the view layer to handle increased user interactions, enabling horizontal scaling of the user interface.
2. Reliability  
   The model layer can implement ACID-compliant databases, ensuring data consistency and reliability during transactions.
3. Performance  
   Indexing in the model layer speeds up database queries, while separating concerns across the MVC layers ensures optimized and responsive performance.

### Master-Slave Pattern

1. Scalability  
   Worker nodes can scale horizontally to handle additional tasks. However, the single master node can become a bottleneck, limiting overall scalability.
2. Reliability  
   Multiple worker nodes ensure that failures in one node have minimal impact on the system. However, vulnerabilities in the master node can disrupt the entire system, and maintaining consistency across worker nodes can be challenging.
3. Performance  
   Parallel processing across worker nodes enhances performance. However, the single master node introduces a potential bottleneck, limiting task distribution efficiency.

## Code Fragman And Code Examples

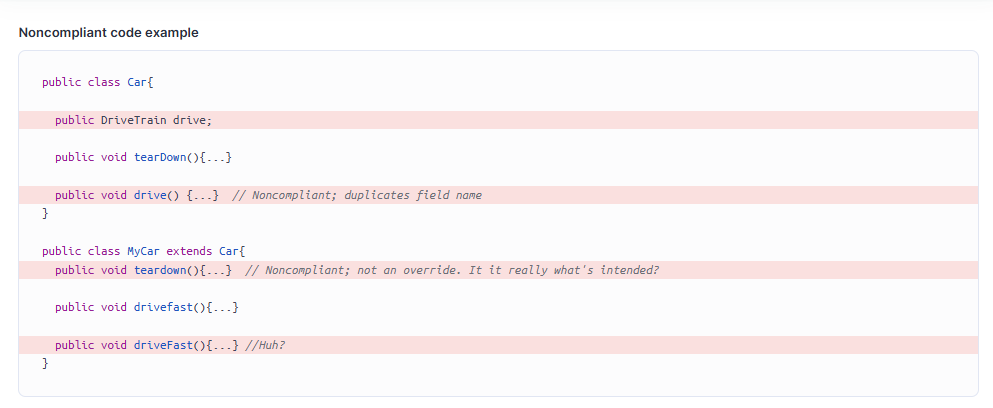


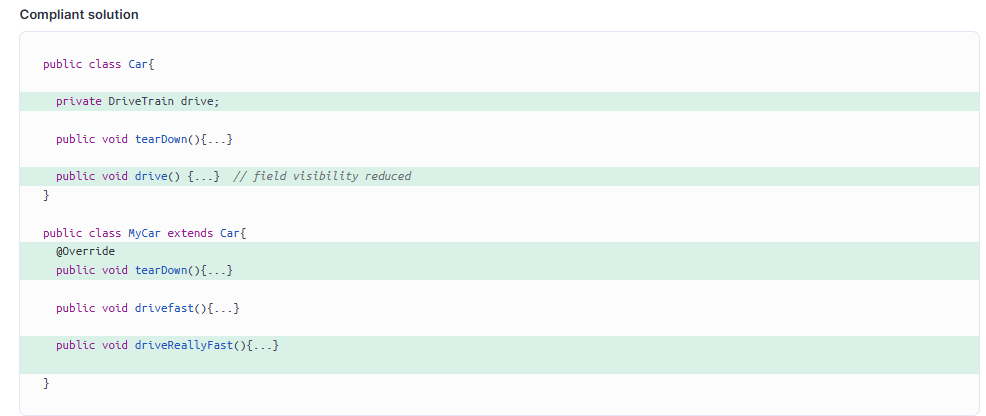
As we see ,this rule raises an issue when there is a method and a field in a class with names that differ only by capitalization.

If we look at the set of methods in a class, including superclass methods, and finding two methods or fields that differ only by capitalization is confusing to users of the class. It is similarly confusing to have a method and a field which differ only in capitalization or a method and a field with exactly the same name and visibility.

In the case of methods, it may have been a mistake on the part of the original developer, who intended to override a superclass method, but instead added a new method with nearly the same name.

However, this situation simply indicates poor naming. Method names should be action-oriented, and thus contain a verb, which is unlikely in the case where both a method and a member have the same name (with or without capitalization differences). However, renaming a public method could be disruptive to callers. Therefore renaming the member is the recommended action.

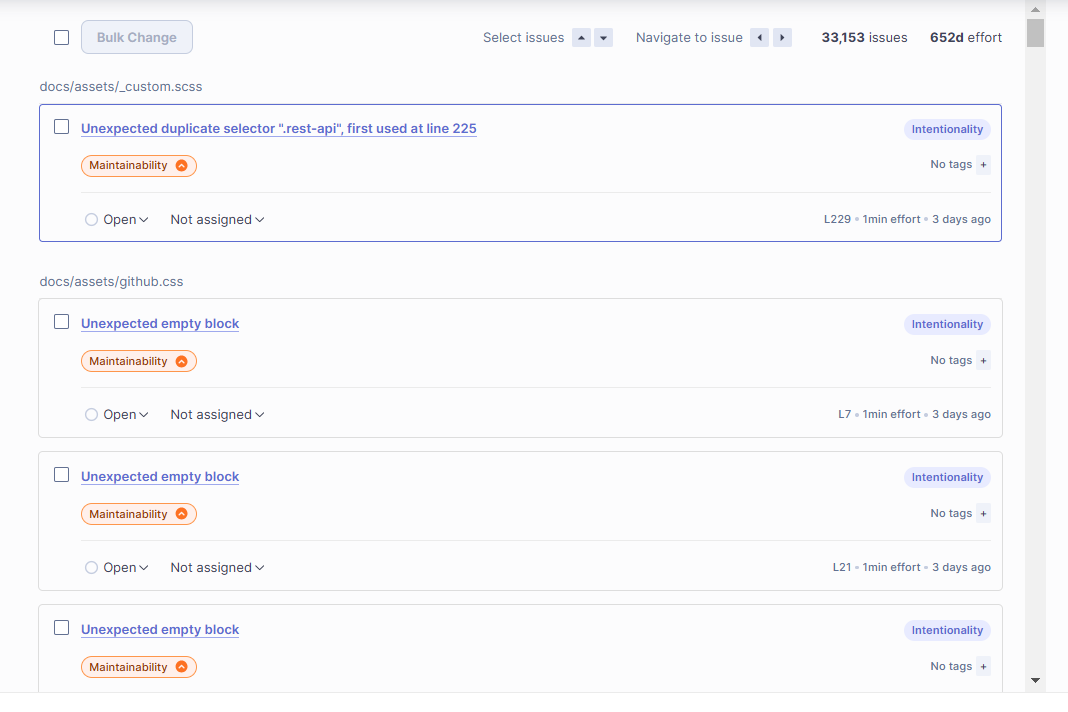


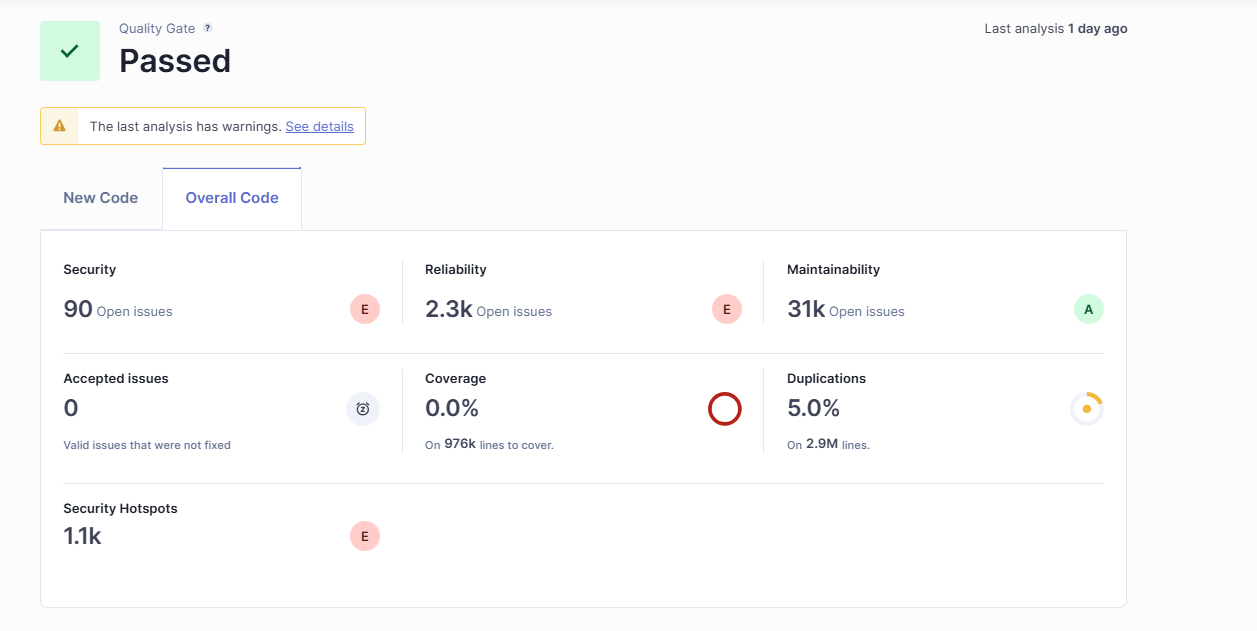


## Tool Output:

The output of the analysis done using SonarQube is given below. It includes numerical data such as lines of code, number of functions, and number of files. Additionally, it shows test results like maintainability, security, duplication, and coverage.







# Task IV: Critique The Descriptive Architecture

## Identify Architectural Smells

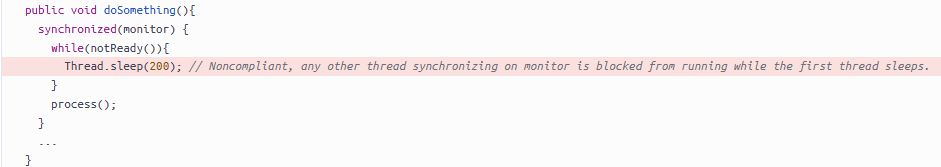
### Reliability

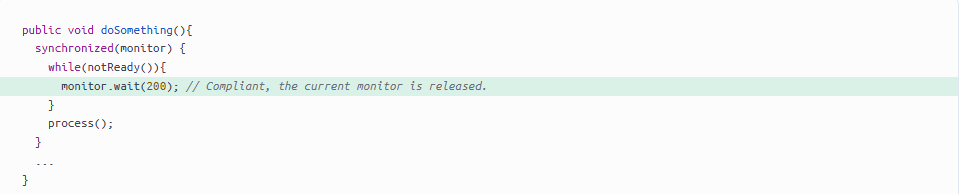
* **Replace the call to "Thread.sleep(...)" with a call to "wait(...)".**



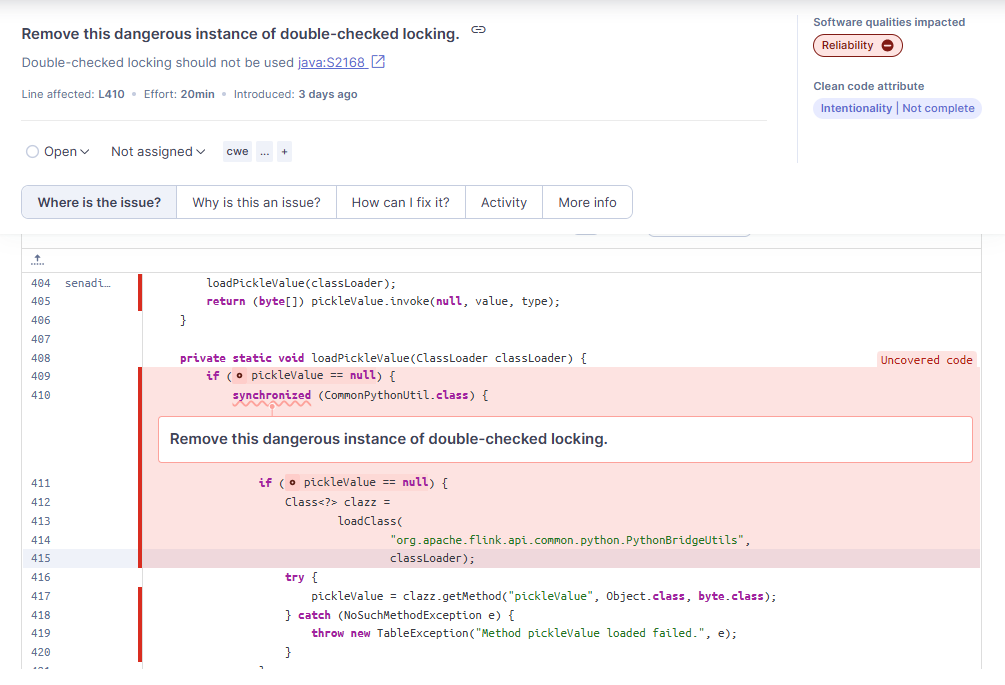
In a multithreaded environment, a thread may need to wait for a particular condition to become true. One way of pausing execution in Java is Thread.sleep(…​).If a thread that holds a lock calls Thread.sleep(…​), no other thread can acquire said lock. This can lead to performance and scalability issues, in the worst case leading to deadlocks.

#### **Noncompliant code example**



**Compliant solution**

* Remove this dangerous instance of double-checked locking.

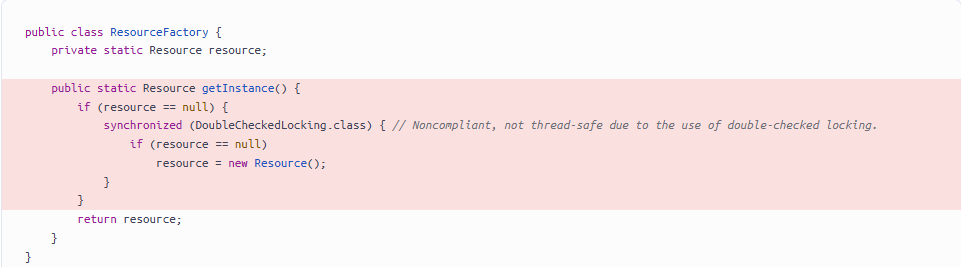


Double-checked locking is the practice of checking a lazy-initialized object’s state both before and after a synchronized block is entered to determine whether to initialize the object. In early JVM versions, synchronizing entire methods was not performant, which sometimes caused this practice to be used in its place.

Apart from float and int types, this practice does not work reliably in a platform-independent manner without additional synchronization of mutable instances. Using double-checked locking for the lazy initialization of any other type of primitive or mutable object risks a second thread using an uninitialized or partially initialized member while the first thread is still creating it. The results can be unexpected, potentially even causing the application to crash.

#### 

#### **Noncompliant code example**

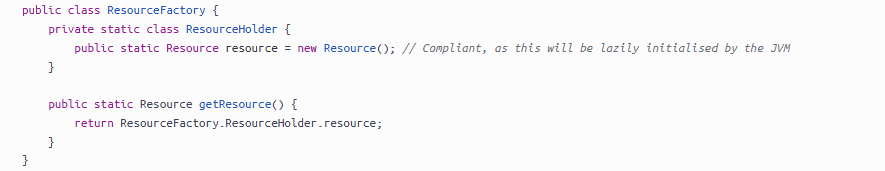


#### **Compliant solution**



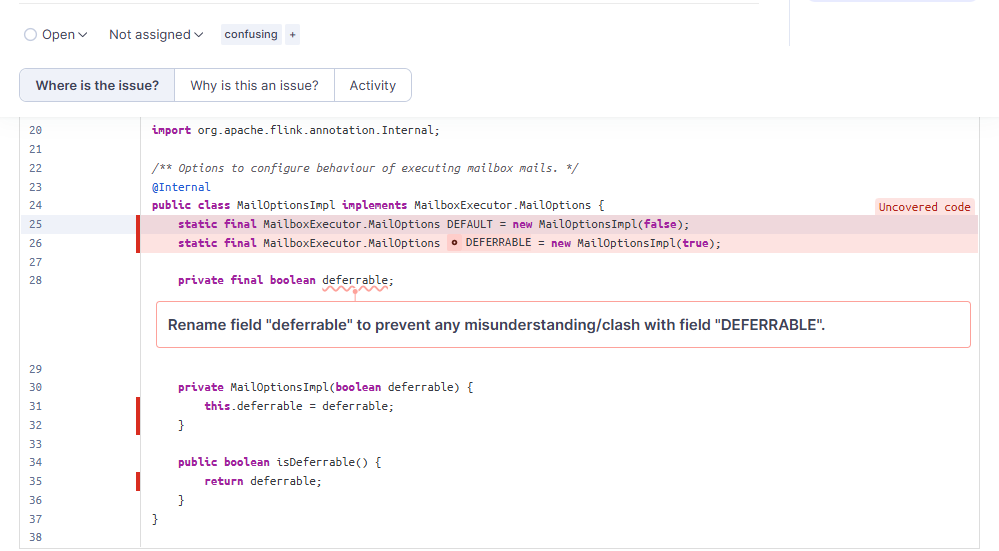
#### **Compliant solution**

Alternatively, a static inner class can be used. However, this solution is less explicit in its intention and hence should be used with care.



### Maintainability

# Rename field "deferrable" to prevent any misunderstanding/clash with field "DEFERRABLE".



This rule raises an issue when there is a method and a field in a class with names that differ only by capitalization.

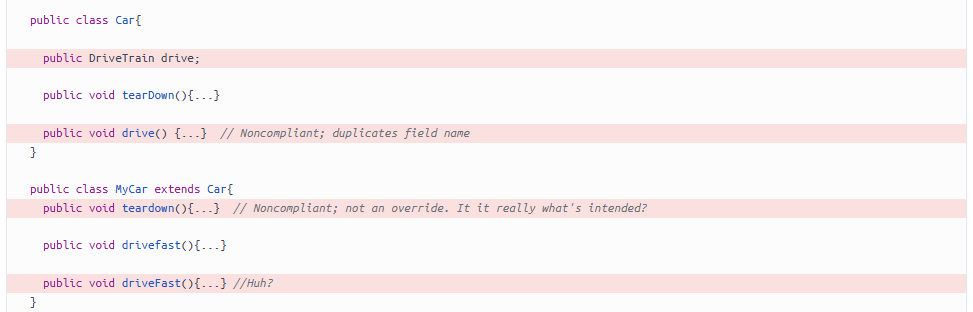
Then if we look at the set of methods in a class, including superclass methods, and finding two methods or fields that differ only by capitalization is confusing to users of the class. It is similarly confusing

to have a method and a field which differ only in capitalization or a method and a field with exactly the same name and visibility.

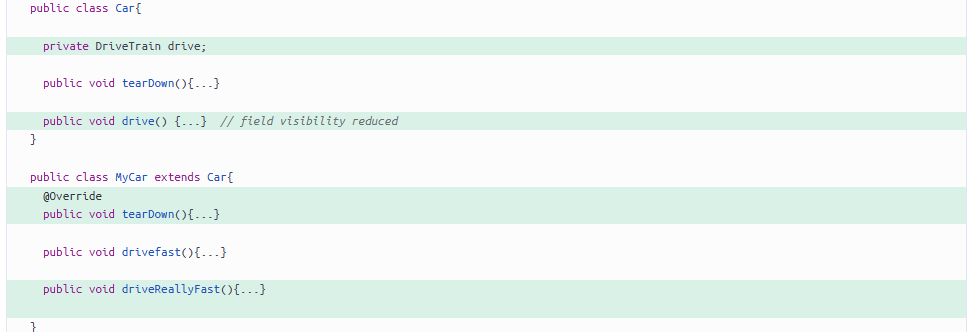
In the case of these methods, it may have been a mistake on the part of the original developer, who intended to override a superclass method, but instead added a new method with nearly the same name.

On the other hand, this situation simply indicates poor naming. Method names should be action-oriented, and thus contain a verb, which is unlikely in the case where both a method and a member have the same name (with or without capitalization differences). However, renaming a public method could be disruptive to callers. Therefore renaming the member is the recommended action.

#### **Noncompliant code example**

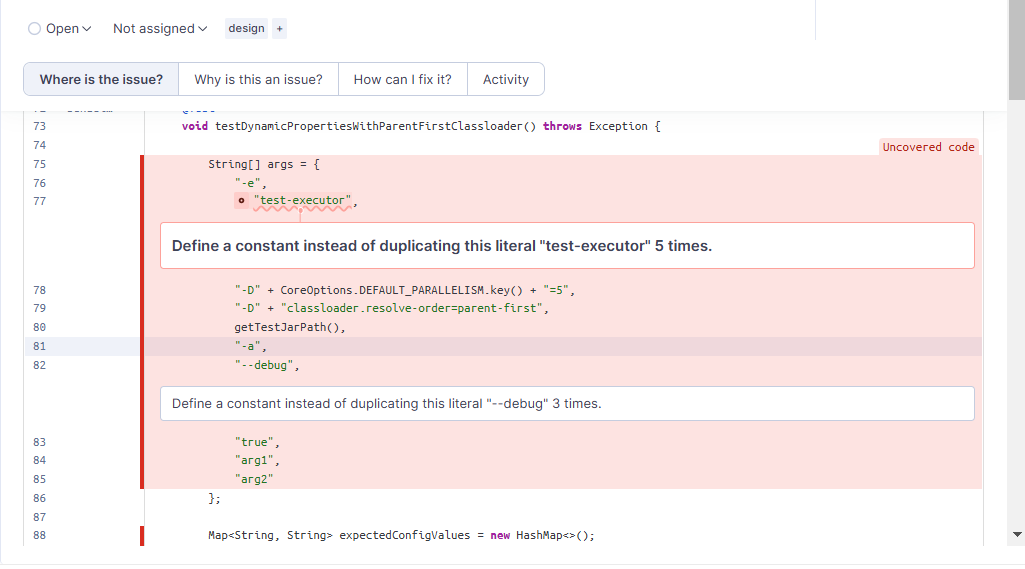


#### **Compliant solution**



# 

# **Define a constant instead of duplicating this literal "test-executor" 5 times.**



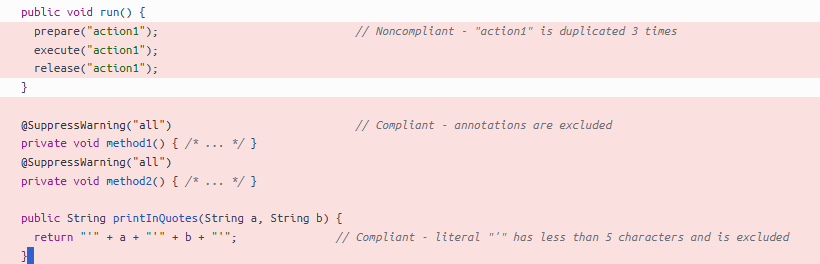
Duplicated string literals make the process of refactoring complex and error-prone, as any change would need to be propagated on all occurrences.

### **Exceptions**

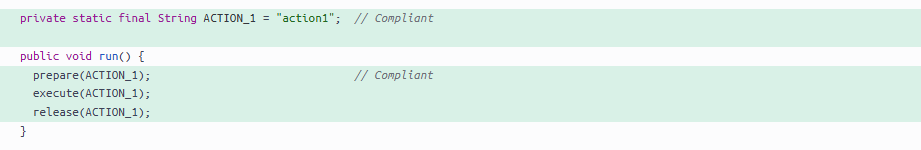
To prevent generating some false-positives, literals having less than 5 characters are excluded.

#### **Noncompliant code example**

With the default threshold of 3:

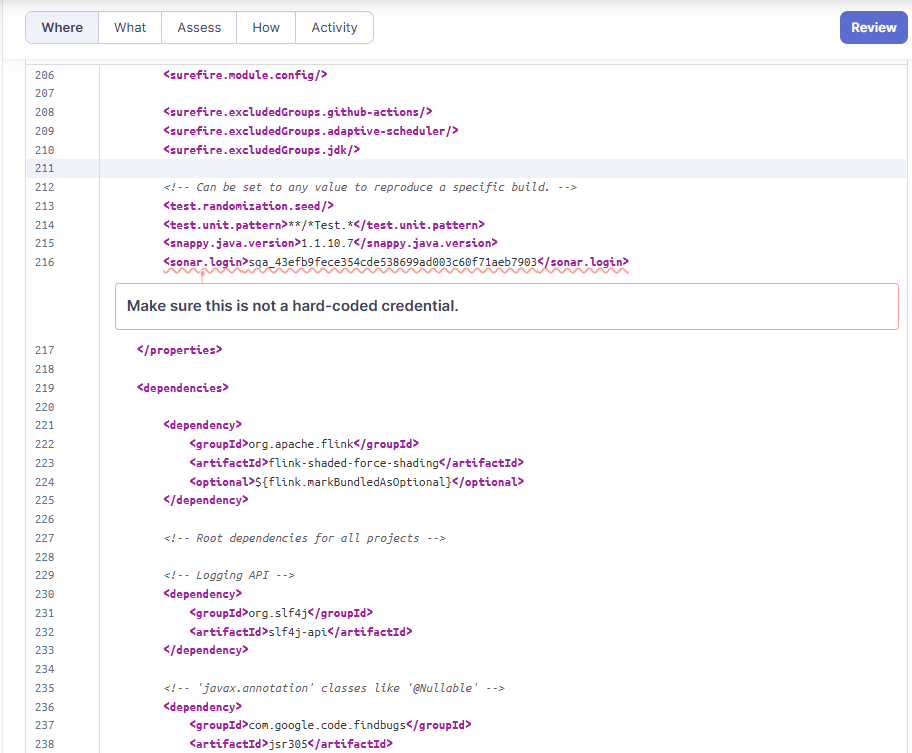


#### **Compliant solution**



### Security

# **Make sure this is not a hard-coded credential.**



# 

# 

# Because it is easy to extract strings from an application source code or binary, credentials should not be hard-coded. This is particularly true for applications that are distributed or that are open-source.

# In the past, it has led to the following vulnerabilities:

# [CVE-2019-13466](http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-13466)

# [CVE-2018-15389](http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-15389)

# Credentials should be stored outside of the code in a configuration file, a database, or a management service for secrets . It’s recommended to customize the configuration of this rule with additional credential words such as "oauthToken", "secret", etc.

## **Sensitive Code Example**



## Recommended Secure Coding Practices

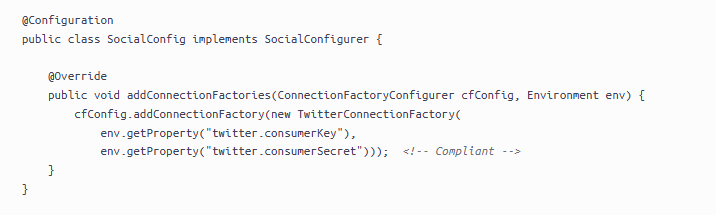
# Store the credentials in a configuration file that is not pushed to the code repository.

# Store the credentials in a database.

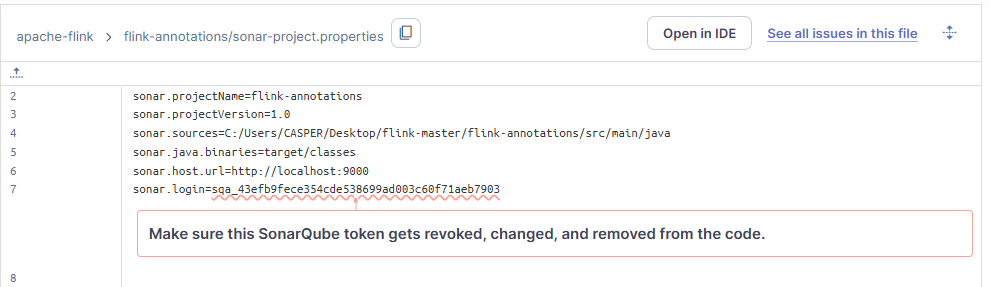
# Use your cloud provider’s service for managing secrets.

# If a password has been disclosed through the source code: change it.

## **Compliant Solution**



# **Make sure this SonarQube token gets revoked, changed, and removed from the code**

Secret leaks often occur when a sensitive piece of authentication data is stored with the source code of an application. Considering the source code is intended to be deployed across multiple assets, including source code repositories or application hosting servers, the secrets might get exposed to an unintended audience.

In generally most cases, trust boundaries are violated when a secret is exposed in a source code repository or an uncontrolled deployment environment. Unintended people who don’t need to know the secret might get access to it. They might then be able to use it to gain unwanted access to associated services or resources.

The trust issue can be more or less severe depending on the people’s role and entitlement.

### **What is the potential impact?**

A SonarQube token is a unique key that serves as an authentication mechanism for accessing the SonarQube platform’s APIs. It is used to securely authenticate and authorize external tools or services to interact with SonarQube.

Tokens are typically generated for specific users or applications and can be configured with different levels of access permissions. By using a token, external tools or services can perform actions such as analyzing code, retrieving analysis results, creating projects, or managing quality profiles within SonarQube.

If a SonarQube token leaks to an unintended audience, it can pose a security risk to the SonarQube instance and the associated projects. Attackers may use the leaked token to gain unauthorized access to the SonarQube instance. They can potentially view sensitive information, modify project settings, or perform other dangerous actions.

Additionally, attackers with access to a token can modify code analysis results. This can lead to false positives or negatives in the analysis, compromising the accuracy and reliability of the platform.

**Revoke the secret**

Revoke any leaked secrets and remove them from the application source code.

Before revoking the secret, ensure that no other applications or processes are using it. Other usages of the secret will also be impacted when the secret is revoked.

**Analyze recent secret use**

When available, analyze authentication logs to identify any unintended or malicious use of the secret since its disclosure date. Doing this will allow determining if an attacker took advantage of the leaked secret and to what extent.

This operation should be part of a global incident response process.

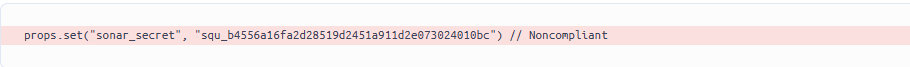
The SonarQube audit log can be downloaded from the product web interface and can be used to audit the malicious use of the compromised key. This feature is available starting with SonarQube Enterprise Edition.

**Use a secret vault**

A secret vault should be used to generate and store the new secret. This will ensure the secret’s security and prevent any further unexpected disclosure.

Depending on the development platform and the leaked secret type, multiple solutions are currently available.

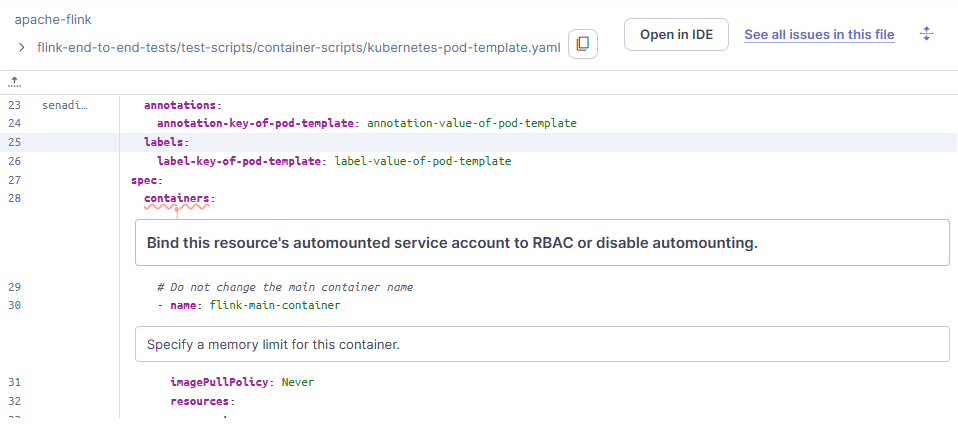
#### **Noncompliant code example**



#### **Compliant solution**



* **Bind this resource's automounted service account to RBAC or disable automounting**

****

### What is the potential impact?

#### Unauthorized Access

If a pod with a mounted service account gets compromised, an attacker could potentially use the token to interact with the Kubernetes API, possibly leading to unauthorized access to other resources in the cluster.

#### Privilege Escalation

Service account tokens are often bound with roles that have extensive permissions. If these tokens are exposed, it could lead to privilege escalation where an attacker gains higher-level permissions than intended.

#### Data Breach

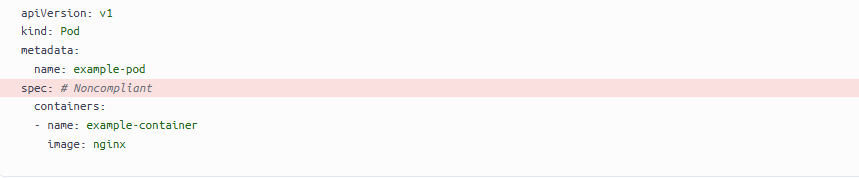
Service account tokens can be used to access sensitive data stored in the Kubernetes cluster. If these tokens are compromised, it could lead to a data breach.

#### Denial of Service

An attacker with access to a service account token could potentially overload the Kubernetes API server by sending a large number of requests, leading to a Denial of Service (DoS) attack.

#### **Noncompliant code example**

In this example, the service account token is mounted in the pod example-pod by default, but is unnecessary for the pod and its service(s) to function correctly.



In this example, the service account token is mounted in the pod example-pod and is necessary, for example because it allows a third-party service to authenticate with the Kubernetes API. However, no specific permissions are granted to the service account:



#### 

#### 

#### **Compliant solution**

### 

### 

### 

### 

### 

### **How does this work?**

The essential part of the solution is to make sure that permissions within the cluster are constructed in a way that minimizes the risk of unauthorized access.

To do so, it follows a least-privilege approach.

1. If the service account token is unnecessary for the pod to function, disable automounting.
2. If the service account token is required, ensure that the service account has the least amount of permissions necessary to perform its function.

Additionally, service account token automounting can be disabled directly from the service account specification file.

## Architectural Constraints

### 1. Performance Constraints

* Low Latency: The system must process data in real-time with delays measured in milliseconds. For example, in financial trading, transactions must happen within milliseconds to ensure efficiency.
* High Throughput: The system should handle millions of events per second. For instance, a social media platform needs to process user interactions instantly.
* Resource Usage: CPU, memory, and disk usage on processing nodes should stay below a specific limit. This ensures efficient resource usage and keeps costs under control.

### 2. Compatibility Constraints

* Java and Scala Compatibility: Flink works only with certain JDK versions (e.g., JDK 11 or 17). While many applications use the latest versions, Flink supports only these two to ensure stability and performance.
* Libraries and Dependencies: Flink must be compatible with specific versions of libraries like Netty and Akka. For example, using the correct version of Netty ensures smooth network communication.
* Deployment Platforms: Flink should work with platforms like Kubernetes, Docker, and Hadoop YARN. This makes it easier to deploy and manage in different environments.

### 3. Modularity Constraints

* Independent Modules: Flink’s modules (API, Runtime, Connector) should have minimal dependencies on each other. This allows easier development and testing of each module.
* Plug-and-Play: Adding new connectors or extensions should require minimal code changes. For example, adding a new data source should not need major changes to the existing code.

### 4. Security Constraints

* Authorization: User access controls and permissions must be in place. For example, only authorized users should be able to perform certain actions.
* Data Encryption: Data must be encrypted during transfer and storage using secure protocols like TLS or SSL. This protects data from unauthorized access.
* Audit Logging: Logs for actions and errors should be traceable and stored securely. This helps track and analyze events in the system.

### 5. Fault Tolerance Constraints

* Checkpoint Duration: The checkpoint process must complete within a set time (e.g., 1-2 seconds). This ensures quick system recovery.
* Fault Tolerance: Data should be replicated across nodes so that if one fails, the process can continue without interruption.
* State Recovery: The system should quickly restore its state after an error, ensuring it can return to its previous condition.

### 

### 6. Scalability Constraints

* Horizontal Scalability: New nodes should be added to the Flink cluster as needed to handle increasing workloads.
* Dynamic Load Management: Resources should be allocated automatically when processing load increases. This optimizes system performance.

## Suggest An Improved Architecture

### 1. Performance Improvementsv

* Use processing time instead of event time to make data processing faster.
* Prefer data locality to process data closer to where it is stored, reducing network delays.
* Increase task parallelism to use more threads and process more tasks at the same time.
* Use RocksDB or other high-performance storage to handle more events.
* Make resource allocation dynamic and balance the load between nodes.
* Improve memory usage by using off-heap memory for better efficiency.

### 2. Compatibility Improvements

* Support more JDK versions by optimizing Flink’s build and runtime configurations.
* Update APIs to work better with the latest versions of Scala.
* Use a dependency management tool like Maven or Gradle to handle library versions more effectively.
* Provide alternative versions or fallback mechanisms for incompatible libraries.
* Develop Flink Kubernetes Operators further to make integration with platforms like Kubernetes easier.

### 3. Modularity Improvements

* Reduce dependencies between modules so each module can run independently (*microservices approach*).
* Create a plugin architecture to add new connectors or data sources without changing the existing system.

### 4. Security Improvements

* Strengthen access control with detailed user roles and permission levels.
* Use TLS/SSL for secure internal communication in Flink.
* Support hardware-based encryption to improve performance while securing data.
* Store logs securely and centrally by integrating with an external log management system.

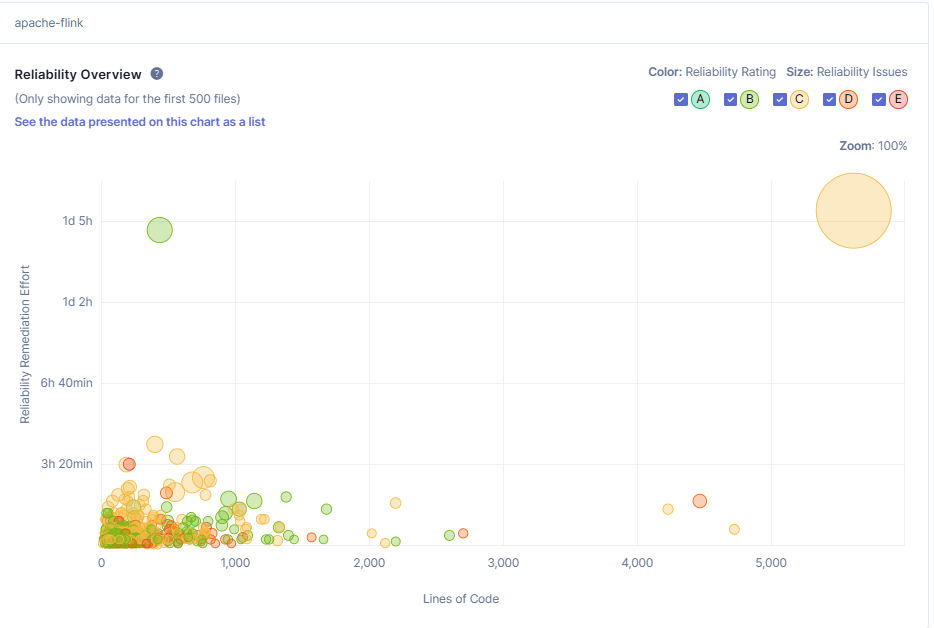
### 5. Fault Tolerance Improvements

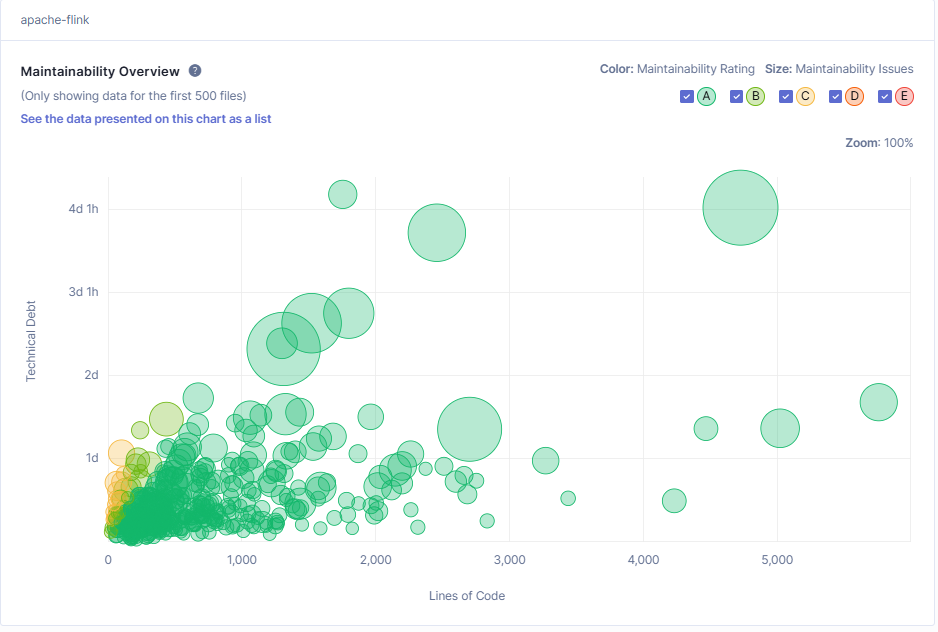
* Prevent data loss by increasing backup options and using replication strategies.

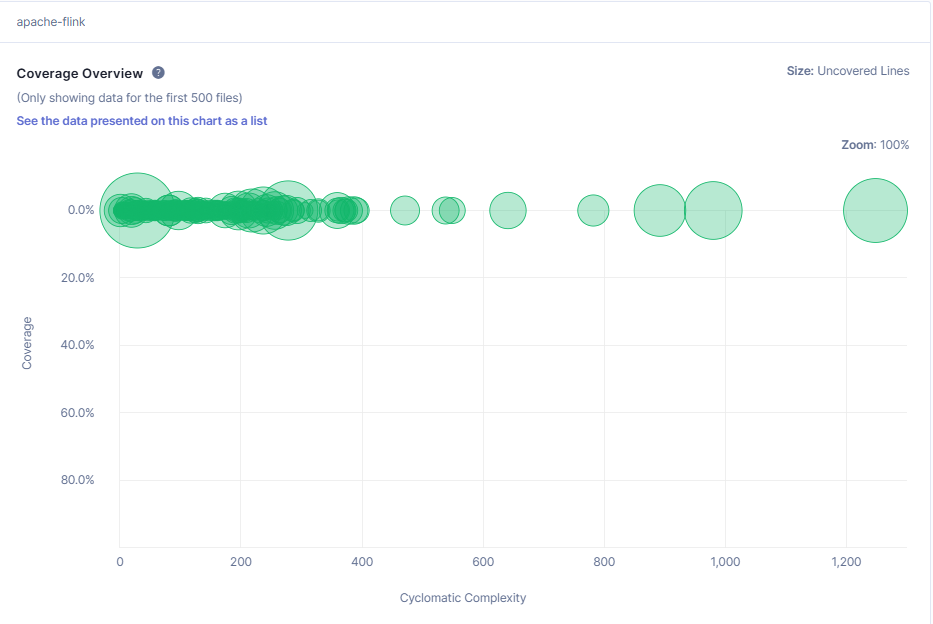
### 6. Scalability Improvements

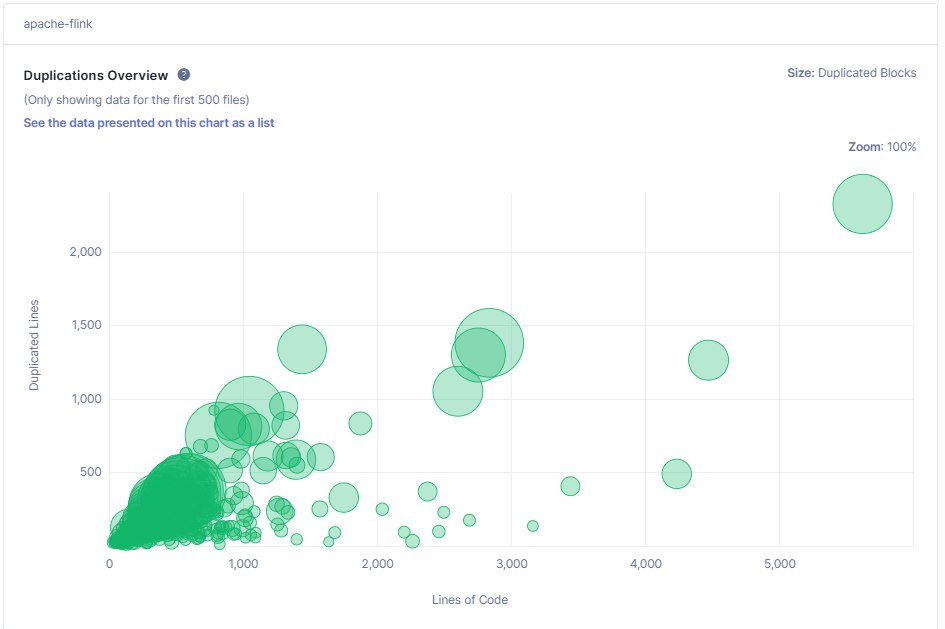
* Use autoscaling to add new nodes to the Flink cluster automatically when needed.
* Adjust Flink jobs dynamically based on their resource needs.
* Apply custom partitioning strategies for better data distribution and balanced workloads.

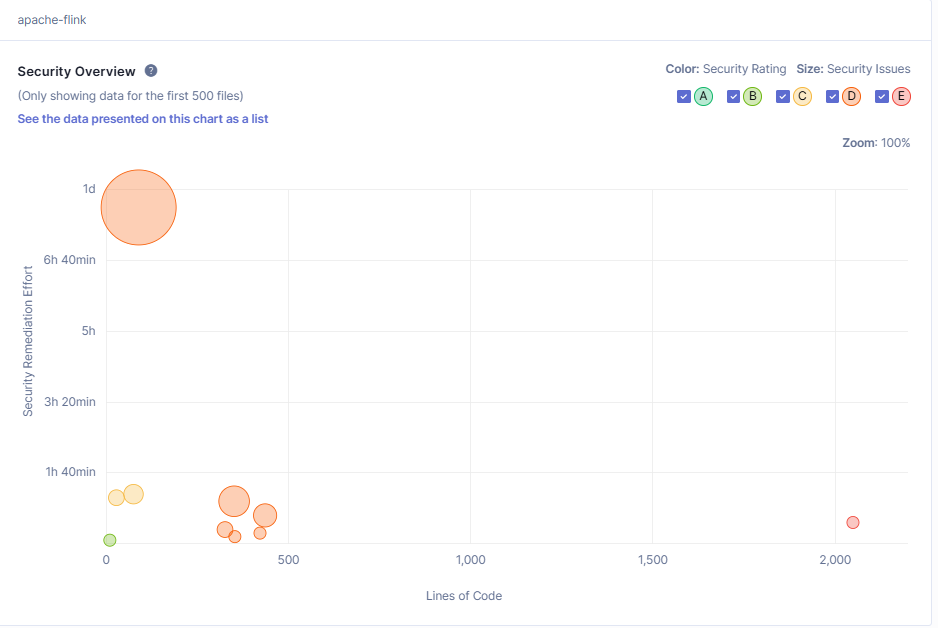
## Tool Output

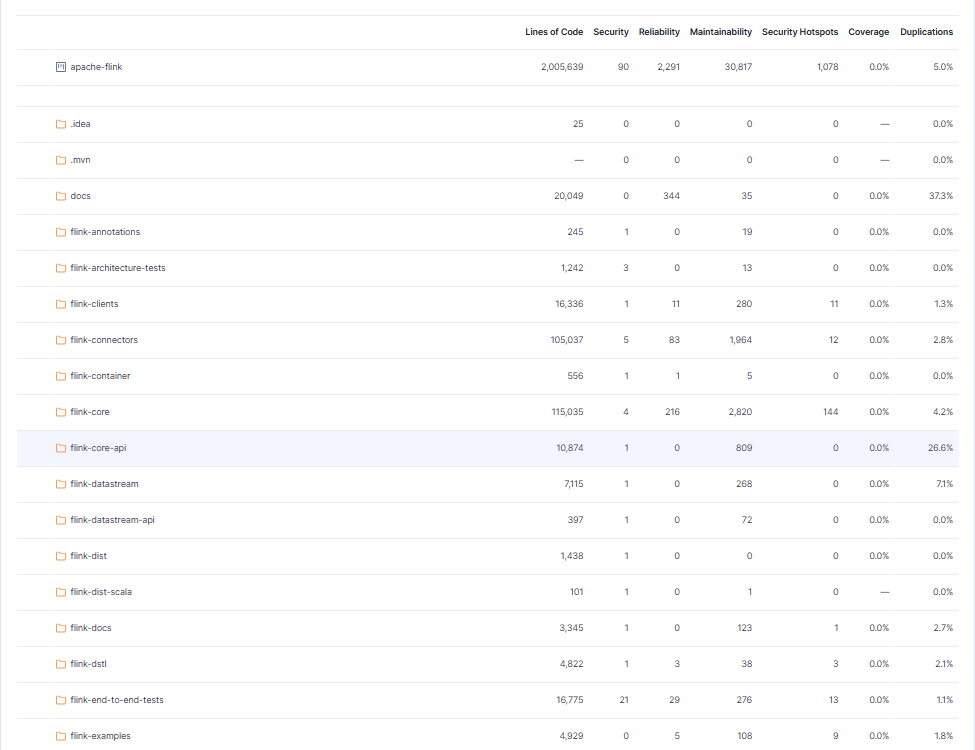


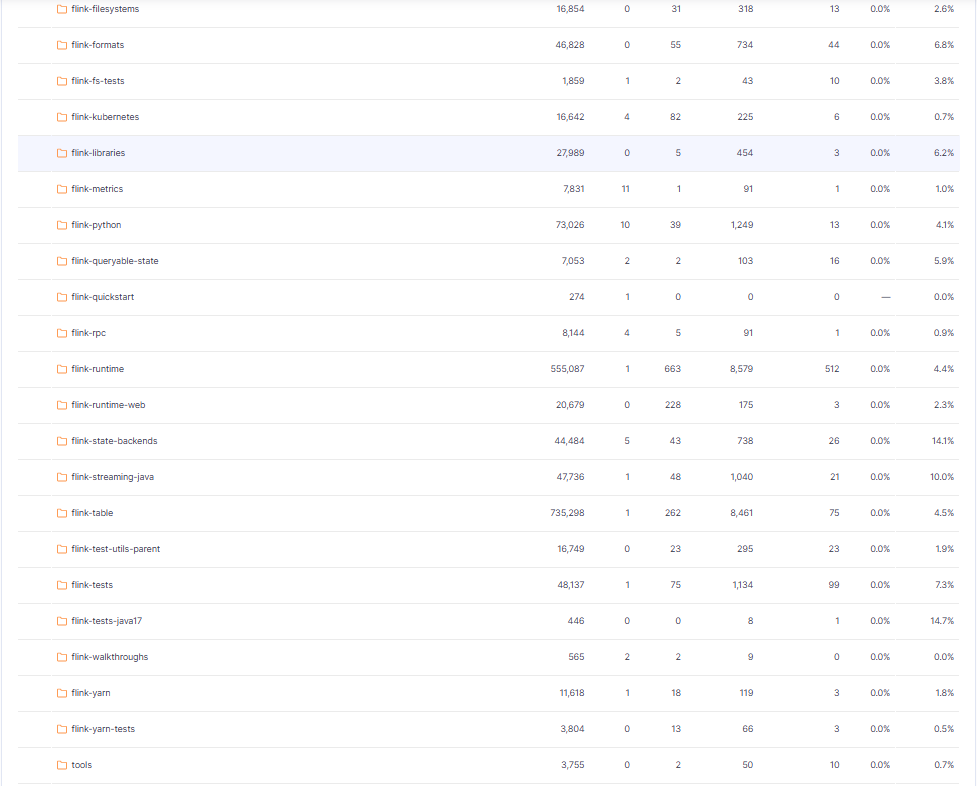


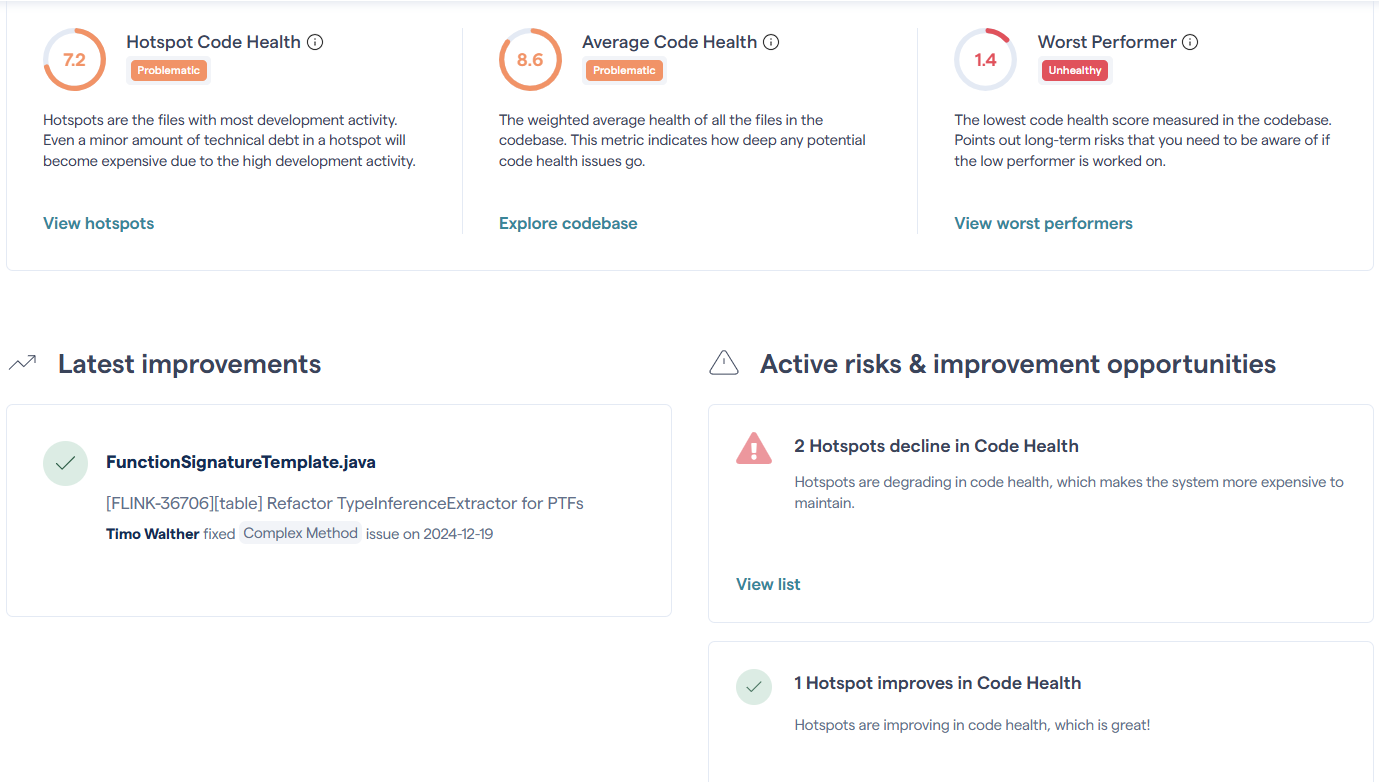




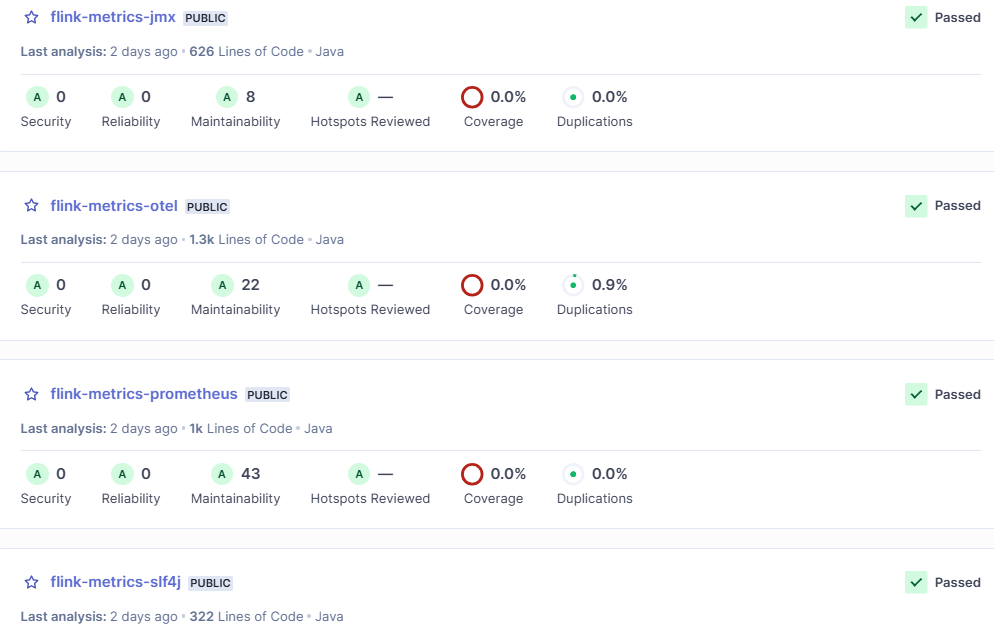
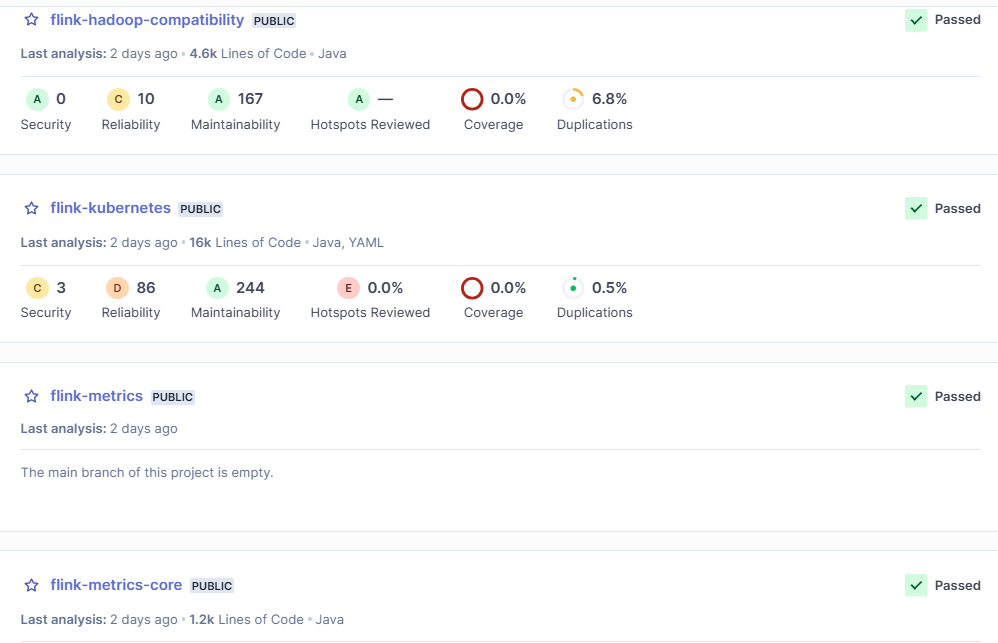
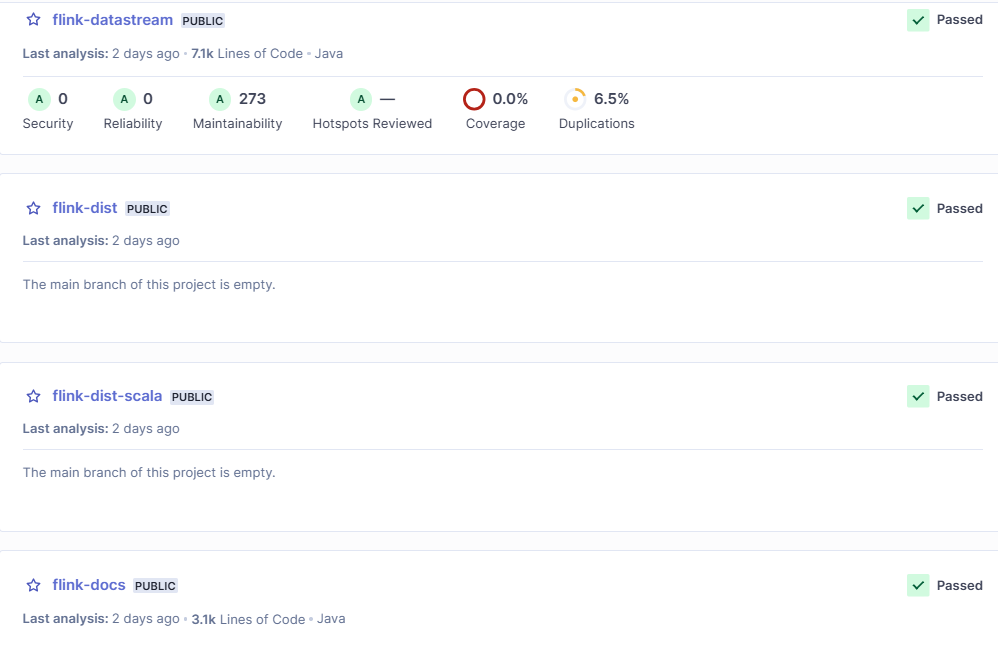
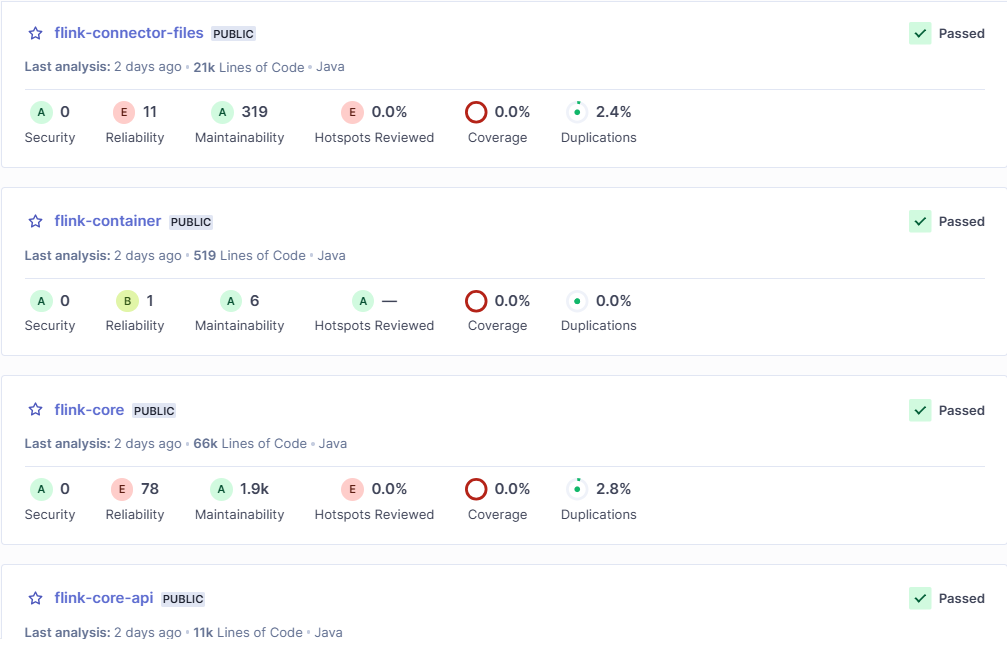
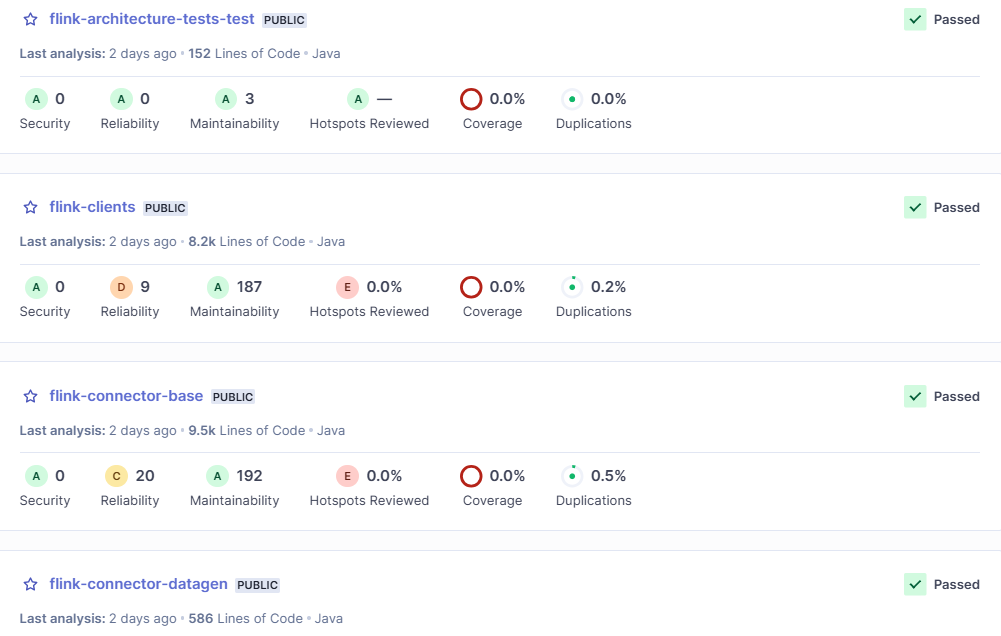
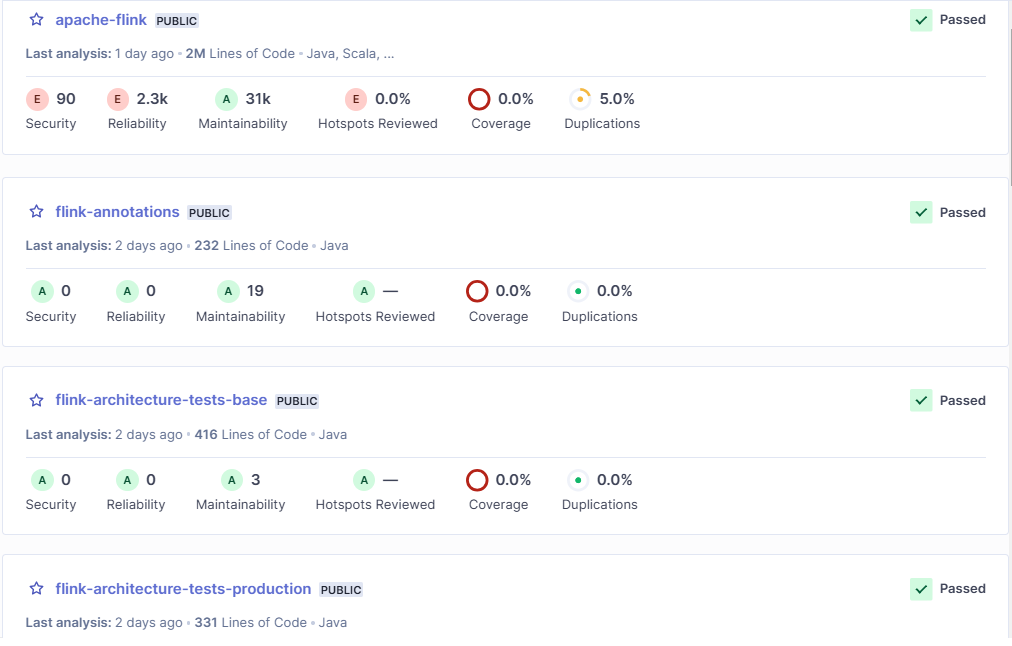
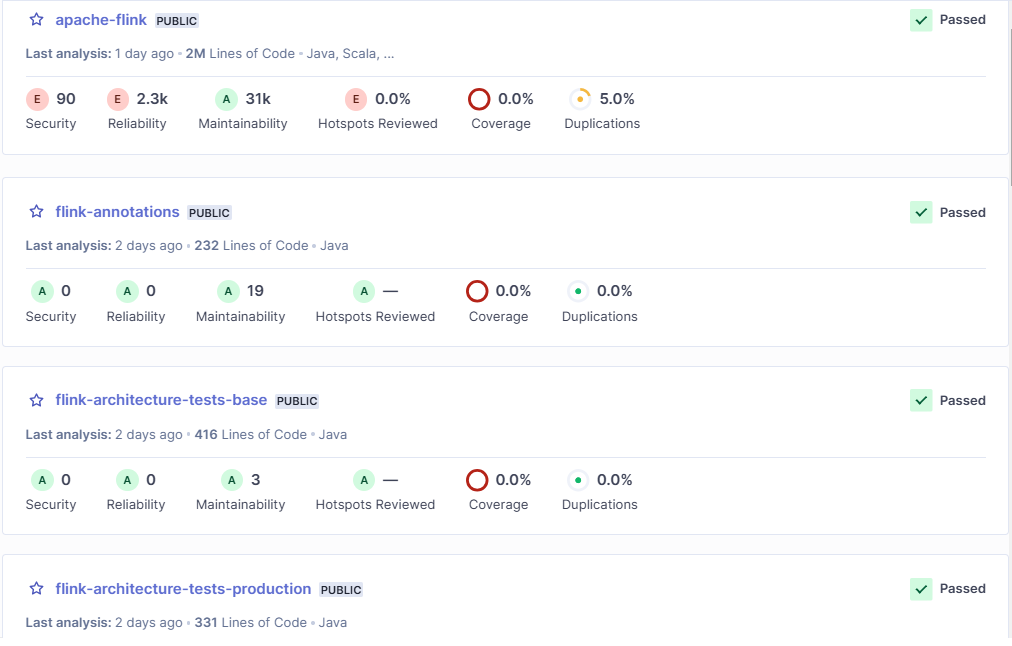


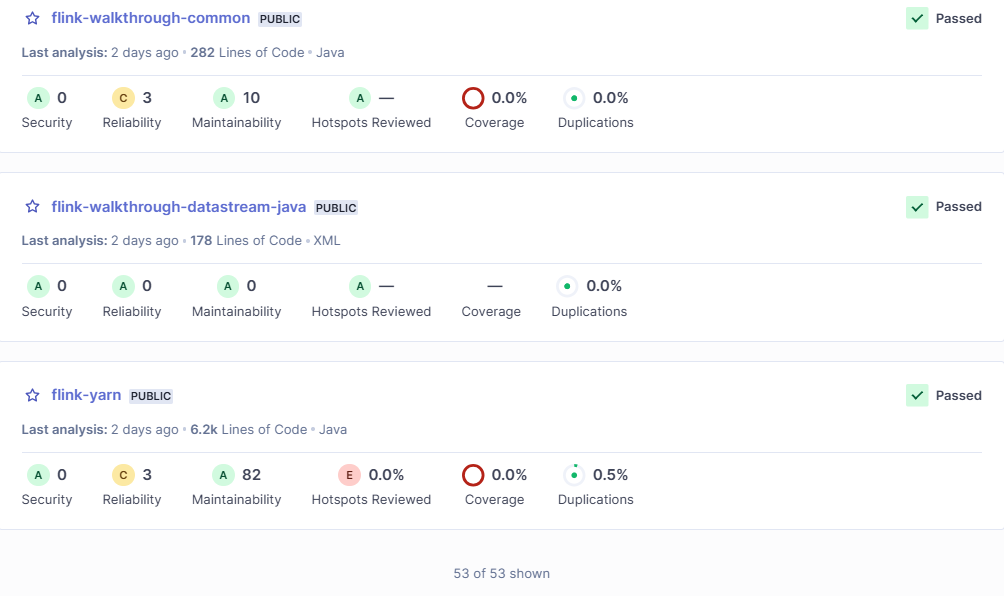
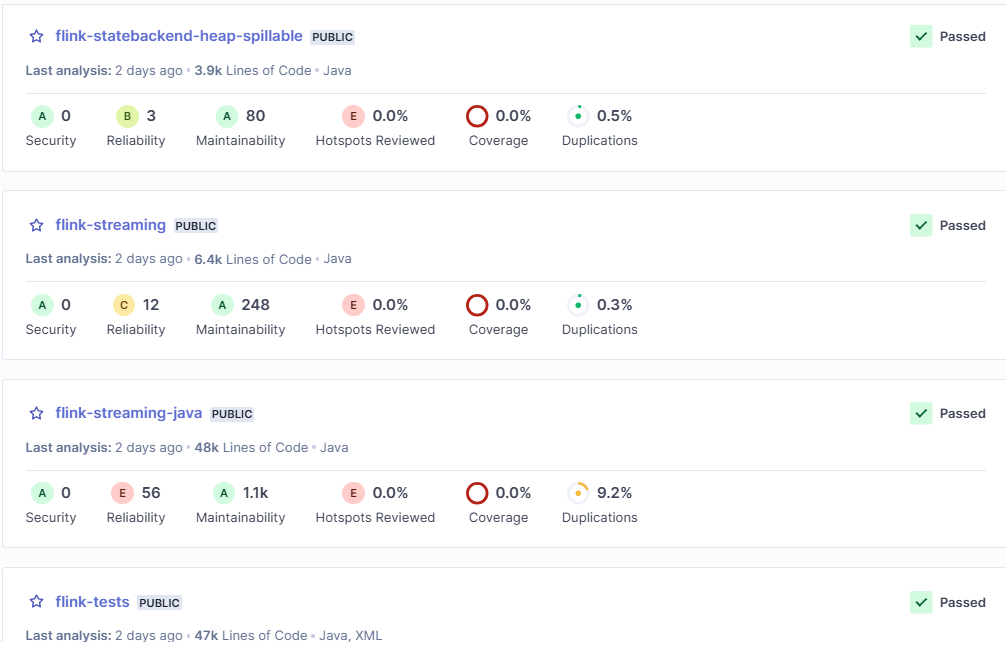






Below are the analysis results of both Flink's overall and individual modules, such as Maintenance, security, reliability, coverage and replication.





# References:

1. [Apache Flink Official Architecture](https://flink.apache.org/what-is-flink/flink-architecture/)
2. [Flink Architecture Mindmap](https://blog.devgenius.io/apache-flink-architecture-mindmap-5389ea708524)
3. [Understanding Apache Flink Architecture](https://harshit-sharma.medium.com/understanding-apache-flink-architecture-and-its-components-ee7c67b1ab7d)
4. [Flink Architecture Overview](https://medium.com/big-data-processing/apache-flink-architecture-overview-abbe19199fd0)
5. <https://www.waitingforcode.com/apache-flink/apache-flink-cluster-components-deep-dive/read>
6. <https://alibaba-cloud.medium.com/data-warehouse-in-depth-interpretation-of-flink-resource-management-mechanism-5c13b531abfa>
7. <https://nightlies.apache.org/flink/flink-docs-release-1.11/concepts/flink-architecture.html>
8. <https://alibaba-cloud.medium.com/advanced-apache-flink-tutorial-1-analysis-of-runtime-core-mechanism-1cf13e521f3c>
9. <https://aws.amazon.com/what-is/apache-flink/>
10. <https://nexocode.com/blog/posts/what-is-apache-flink/>
11. <https://codescene.io>
12. <https://www.sonarsource.com/products/sonarqube/>
13. [Next generation code analysis | CodeScene](https://codescene.com/)

# 