Miquel Álvarez-Ruiz

Real-time Image Inference Stream Processing with Apache Flink

Master's Degree in Computer Security Engineering and Artificial Intelligence

Master's Thesis

Directed by
Dr. Marc Sanchez-Artigas

UNIVERSITAT ROVIRA i VIRGILI

Tarragona, 2024

Contents

Abstract v

Resum v

Resumen v

1 Introduction 1
- 1.1 Aims of the Work 1
- 1.2 Planning 2

2 Background 4
- 2.1 Stream Processing 4
- 2.2 Apache Flink: Distributed Data Stream Processing 5
  - 2.2.1 Architecture 5
  - 2.2.2 Execution and Scheduling Mechanism 6
  - 2.2.3 State Management in Flink 7
  - 2.2.4 Programming Model 8
  - 2.2.5 DataStream API 9
  - 2.2.6 State Management and Fault Tolerance 10
- 2.3 Machine Learning Overview 12
  - 2.3.1 Deep Learning and Neural Networks 12
  - 2.3.2 Inference with ResNet 12
  - 2.3.3 PyTorch: Deep Learning Framework 13
  - 2.3.4 CPU and GPU Support. 13
  - 2.3.5 DJL: Deep Java Library 14
  - 2.3.6 Use Cases 14
- 2.4 Grafana and Prometheus: Monitoring and Visualization 15
  - 2.4.1 Prometheus 15
  - 2.4.2 Grafana 16
- 2.5 Challenges in Streaming Machine Learning Inference Processing 16
- 2.6 Cloud Storage and Infrastructure 17
  - 2.6.1 Amazon Simple Storage Service (S3) 17
  - 2.6.2 Amazon Elastic Compute Cloud (EC2) 18
  - 2.6.3 Security and Compliance 19
- 2.7 Summary 19

3 Existing Solutions and Their Limitations 20
- 3.1 Apache Storm 20
- 3.2 Apache Spark 20
- 3.3 Apache Flink 20
- 3.4 Other Frameworks 21
  - 3.4.1 Kafka Streams 21
  - 3.4.2 Google Cloud Dataflow 21
  - 3.4.3 Samza 22
- 3.5 Sponge: A New Approach 22
  - 3.5.1 Comparison with Apache Flink 22
- 3.6 Summary 23

4 Functional Requirements **25**
4.1 Data Ingestion . . . 25
4.2 Data Preprocessing . . . 25
4.3 Model Inference . . . 25
4.4 Post-Processing . . . 26
4.5 System Performance . . . 26
4.6 Error Handling and Fault Tolerance . . . 26
4.7 Summary . . . 26

5 System Design **27**
5.1 Design Choices and Rationale . . . 27
5.2 Architecture Overview . . . 27
5.3 Model Inference and Processing Efficiency . . . 28
5.4 Post-Processing and Output Management . . . 28
5.5 State Management and Fault Tolerance . . . 29
5.6 Scalability and Monitoring . . . 29
5.7 Deployment and Configuration Management . . . 29
5.8 Summary . . . 29

6 Implementation **31**
6.1 System Setup and Deployment . . . 31
6.2 DJL Integration . . . 31
6.2.1 Why ResNet-101? . . . 31
6.2.2 Integration Process . . . 31
6.3 Flink Job Implementation . . . 32
6.3.1 DAG Logical Overview . . . 32
6.3.2 Stream Execution Environment . . . 32
6.3.3 Data Source Configuration . . . 33
6.3.4 Image Preparation . . . 33
6.3.5 Model Inference and Classification . . . 35
6.3.6 Post-Processing and Output . . . 36
6.4 Monitoring with Grafana . . . 37
6.5 Summary . . . 37

7 Evaluation **38**
7.1 Experimental Setup . . . 38
7.2 Cluster Configurations . . . 38
7.2.1 EC2 Instances . . . 38
7.3 Datasets and Preprocessing . . . 39
7.4 Performance Metrics . . . 40
7.5 Results and Analysis . . . 41
7.5.1 CPU Load . . . 41
7.5.2 Cost as a Function of Time . . . 41
7.5.3 Normalized Cost-Efficiency . . . 42
7.5.4 End-to-End Latency Analysis . . . 43
7.5.5 Throughput of the Preprocessing Stage . . . 44
7.6 Discussion . . . 46
7.6.1 Strengths and Weaknesses . . . 46
7.6.2 Potential Improvements . . . 46

7.6.3 Scalability and Flexibility . . . 46
7.6.4 Environmental and Operational Considerations . . . 47

8 Conclusions and Future Work **48**
8.1 Summary of Findings . . . 48
8.2 Accomplishments . . . 48
8.3 Future Research Directions . . . 48

References **50**

Appendix A Image Classification Flink Job **52**

Appendix B Terraform Configuration **55**

Appendix C Ansible Configuration **57**

List of code snippets

1 Flink DataStream API Example . . . 10
2 Stream Execution Environment Configuration . . . 32
3 Data Source Configuration . . . 33
4 Image Preparation Function . . . 34
5 Model Inference and Classification . . . 35
6 Post-Processing and Output Configuration . . . 36

List of Figures

1 Gantt Chart of Thesis Planning . . . 3
2 Flink Architecture: Job Manager and Task Manager Interaction . . . 6
3 From JobGraph to ExecutionGraph in Flink . . . 6
4 Illustration of Flink's Execution and Scheduling Mechanism . . . 7
5 Keyed State Distribution in Flink . . . 8
6 Code to Flink DAG . . . 9
7 Exactly-once two-phase commit process in Flink . . . 12
8 Residual connections in a deep neural network . . . 13
9 Preliminary Design of the Real-Time Image Inference Processing System 28
10 Logical DAG of the Flink Job . . . 32
11 Tensor structure for a 4x4 color image with three channels (RGB) . . . 35
12 Grafana dashboard for Apache Flink . . . 37
13 CPU load for different system configurations. . . . 41
14 Cost as a function of time for CPU and GPU configurations. . . . 42
15 Normalized performance per dollar for different configurations. . . . 43
16 End-to-end system latency for various CPU and GPU configurations. . 44
17 Throughput comparison between CPU and GPU configurations for different numbers of Task Managers (TMs). . . . 45

List of Tables

1 Comparison of Streaming Frameworks . . . 24
2 EC2 instance specifications . . . 39
3 Comparison of datasets . . . 40

Abstract

In the era of Big Data, efficient real-time data stream processing has become a critical necessity in various domains, including artificial intelligence. This thesis explores the use of Apache Flink for streaming machine learning inference, with a particular focus on image classification. Leveraging Flink's distributed processing capabilities, a workflow was designed that includes preprocessing, real-time inference, and data postprocessing using deep learning models. The results demonstrate the scalability and efficiency of the proposed approach, highlighting its applicability in scenarios requiring low-latency and high-performance data processing.

Resum

En l'era del Big Data, el processament eficient de fluxos de dades en temps real s'ha convertit en una necessitat crítica en múltiples dominis, incloent-hi la intel·ligència artificial. Aquesta tesi explora l'ús d'Apache Flink per a la inferència d'aprenentatge automàtic en streaming, amb un enfocament particular en la classificació d'imatges. En aprofitar les capacitats de processament distribuït de Flink, es va dissenyar un flux de treball que inclou preprocessament, inferència en temps real i postprocessament de dades, utilitzant models d'aprenentatge profund. Els resultats obtinguts demostren l'escalabilitat i eficiència de l'enfocament proposat, subratllant la seva aplicabilitat en escenaris que requereixen un processament de dades de baixa latència i alt rendiment.

Resumen

En la era del Big Data, el procesamiento eficiente de flujos de datos en tiempo real se ha convertido en una necesidad crítica en múltiples dominios, incluyendo la inteligencia artificial. Esta tesis explora el uso de Apache Flink para la inferencia de aprendizaje automático en streaming, con un enfoque particular en la clasificación de imágenes. Al aprovechar las capacidades de procesamiento distribuido de Flink, se diseñó un flujo de trabajo que incluye preprocesamiento, inferencia en tiempo real y postprocesamiento de datos, utilizando modelos de aprendizaje profundo. Los resultados obtenidos demuestran la escalabilidad y eficiencia del enfoque propuesto, subrayando su aplicabilidad en escenarios que requieren un procesamiento de datos de baja latencia y alto rendimiento.

1 Introduction

In today's digital era, the proliferation of data from various sources such as social media, IoT devices, and multimedia applications is unprecedented. This data, mostly unstructured and generated in real-time, presents significant challenges in terms of processing and analysis. To address these challenges, stream data processing technologies have gained prominence, enabling organizations to capture, process, and analyze data while it is in motion, rather than at rest.

Among stream data processing solutions, Apache Flink[2], an open-source platform for distributed data stream processing, has emerged as a leading platform. Its ability to handle large volumes of real-time data with high efficiency and scalability makes it a popular choice. Flink enables the creation of applications that can process data continuously as it arrives, which is crucial in scenarios where low latency and high accuracy are critical.

This thesis focuses on the implementation of a streaming inference system using Apache Flink. The choice of Flink is based on its capability to handle large volumes of real-time data, ensuring low latency and high system availability. Specifically, the approach targets real-time image classification, a task increasingly relevant in fields such as security, medicine, and industry. The proposed system integrates deep learning models for image classification within a distributed stream processing framework, enabling instant analysis of visual data as it is received.

The designed workflow comprises several stages, from preprocessing of images to model inference and postprocessing of results. Each stage is optimized to maximize the efficiency and accuracy of the system. Additionally, the potential of Graphics Processing Units (GPUs) to accelerate inference operations is explored, which may provide significant improvements in terms of performance and response time.

I would like to thank Pablo Gimeno Sarroca, a PhD student, for his invaluable help and guidance during the learning and development process of this work. His advice and knowledge about the use of Apache Flink have been fundamental in achieving the established objectives.

1.1 Aims of the Work

The primary aims of this thesis are:

- **Design an Efficient Streaming Inference Workflow Using Apache Flink**: Develop a robust pipeline that handles real-time data ingestion, preprocessing, model inference, and postprocessing, ensuring low latency and high throughput. This involves creating a flexible and scalable architecture that can adapt to varying data volumes and processing demands, providing a reliable solution for real-time applications.
- **Evaluate System Performance**: Assess the system's performance in terms of latency, scalability, and accuracy under various conditions. This evaluation includes benchmarking against existing solutions to identify strengths and weaknesses, enabling a comprehensive understanding of the system's capabilities and limitations. The performance assessment will focus on critical metrics such as

response time, throughput, and resource utilization.

- **Compare CPU and GPU Performance**: Implement the workflow on both CPU and GPU to observe and analyze the differences in performance. This comparison aims to provide users with insights into which hardware may be preferable for their specific use cases based on metrics such as processing speed, cost, and resource utilization.
- **Integrate Real-time Monitoring and Visualization Tools**: Implement real-time monitoring and visualization of the system's performance using tools like Grafana[7] and Prometheus[17]. This involves setting up dashboards that provide insights into various metrics such as resource usage, latency, throughput, and error rates, facilitating the continuous assessment and optimization of the system.

1.2 Planning

The planning of this thesis is structured into several key phases, ensuring a systematic and comprehensive approach to research and development:

1. **Literature Review (February - March 2024)**: Conduct a thorough review of existing literature on stream processing frameworks, machine learning inference, and related technologies to establish a solid theoretical foundation. This phase involves identifying key concepts, methodologies, and tools relevant to the implementation of a streaming inference system.
2. **System Design (March - April 2024)**: Outline the architecture and design of the streaming inference system, detailing each stage of the workflow from data ingestion to result output. This phase includes specifying the components and their interactions, defining data flow and processing logic, and selecting appropriate technologies and frameworks.
3. **Implementation (April - May 2024)**: Develop the system using Apache Flink and integrate deep learning models for image classification. Develop a GPU version to optimize inference performance, leveraging their computational power to accelerate processing tasks. This phase involves coding, debugging, and integrating various system components to ensure seamless operation.
4. **Testing and Evaluation (May - June 2024)**: Test the system under various scenarios to evaluate its performance in terms of latency, scalability, and accuracy. This phase involves setting up test environments, executing test cases, collecting performance data, and analyzing the results to identify areas for improvement. Performance benchmarks will be established to measure key metrics such as processing speed, accuracy, and resource usage.
5. **Final Review and Submission (June 2024)**: Conduct a final review of the thesis document, incorporating feedback and making necessary revisions before submission. This phase includes proofreading the document, ensuring clarity and coherence, and preparing all supplementary materials such as code repositories, data sets, and configuration files.

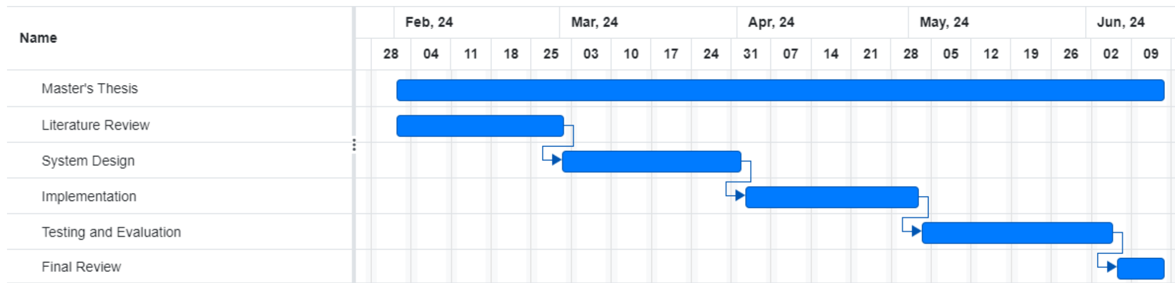

Figure 1: Gantt Chart of Thesis Planning

Figure 1 presents the Gantt chart detailing the planned phases of this thesis project. The timeline spans from February 2024 to June 2024, encompassing the key phases explained above.

2 Background

2.1 Stream Processing

Stream processing involves the continuous ingestion and analysis of data as it arrives, enabling real-time analytics and immediate actions. This approach contrasts with batch processing, which processes large, static datasets at scheduled intervals. Stream processing is crucial for applications that require timely insights, such as fraud detection, network monitoring, and personalized recommendations.

The dataflow model[1] provides a comprehensive framework for understanding how stream processing can be effectively managed. It introduces key concepts such as windows and key-by operations, which are essential for organizing and processing data streams.

- **Windows**: Windows allow for the grouping of data events that occur within a specified timeframe or count. This enables meaningful aggregation and analysis over finite sets of data within an unbounded stream. For example, a sliding window can aggregate data over the past 5 minutes, updating every minute, while a tumbling window might aggregate data over fixed, non-overlapping 5-minute intervals.
- **KeyBy**: The keyBy operation partitions the data stream based on a key, allowing for more granular processing. This is particularly useful for operations that require grouping related data events together, such as computing the total sales for each product category. By ensuring that all events with the same key are processed together, it facilitates accurate and efficient computations.

These concepts are widely implemented across various stream processing frameworks, making them fundamental tools in the stream processing ecosystem. Frameworks such as Apache Flink or Google Cloud Dataflow all incorporate windowing and keyBy operations, leveraging the dataflow model to provide robust and scalable stream processing solutions.

There are several key processing guarantees that streaming systems strive to provide to ensure reliable and accurate data handling:

1. **At-least-once Processing**: Each data event is processed at least once, which ensures no data is lost. However, this may result in duplicate processing of some events, which can lead to inaccuracies in certain applications.
2. **At-most-once Processing**: Each data event is processed at most once, which guarantees no duplicates. This approach can lead to data loss if failures occur before the event is processed.
3. **Exactly-once Processing**: Each data event is processed exactly once, ensuring no duplicates and no data loss. This is the most stringent guarantee and is crucial for applications where accuracy is paramount.

In contrast to batch processing systems, which typically provide strong consistency guarantees but with higher latency, streaming systems prioritize low latency and continuous data flow. Batch processing systems like Spark[19] handle large volumes of data by processing them in discrete chunks, making them suitable for use cases where immediate results are not critical. However, they fall short in scenarios where data needs to be analyzed and acted upon in real-time.

In addition to the basic guarantees, some streaming systems may also handle data ordering and synchronization challenges, particularly in distributed environments where network latency and partitioning can affect the sequence of data events. For example, streaming systems often use techniques such as watermarking and event-time processing to manage data ordering and handle out-of-order events, ensuring the accuracy and reliability of the processed data.

Streaming systems bridge this gap by ensuring timely processing of data as it arrives, making them indispensable for applications that demand up-to-the-minute data insights and rapid response times.

2.2 Apache Flink: Distributed Data Stream Processing

Apache Flink [2] is a powerful open-source platform designed for scalable stream and batch data processing. Flink excels in handling large volumes of real-time data, offering robust features for state management, fault tolerance, and exactly-once processing semantics. This makes it an ideal choice for applications requiring high throughput, low latency, and precise processing guarantees.

2.2.1 Architecture

Flink's architecture is built around a distributed execution model that maximizes resource utilization and supports parallel processing and fault tolerance. The architecture comprises several key components:

1. **Job Manager**: The Job Manager is responsible for job scheduling, resource allocation, and checkpointing. It orchestrates the execution of tasks across the cluster, ensuring efficient resource utilization and fault tolerance. The Job Manager coordinates the overall execution using Akka for communication, a toolkit and runtime for building concurrent and distributed applications on the JVM.

2. **Task Manager**: Task Managers execute the tasks assigned by the Job Manager. They manage data buffering, state, and inter-task communication. Each Task Manager can handle multiple tasks concurrently, supporting parallel processing and efficient resource use.

3. **Client**: The Client is responsible for launching jobs in the Flink cluster. It transforms the logical plan of the workflow into a physical execution plan, optimizing the Directed Acyclic Graph (DAG). The Client then submits this plan to the Job Manager.

Figure **??** illustrates the interaction between the Job Manager and multiple Task Managers. The Client creates the dataflow graph from the Flink program, and the

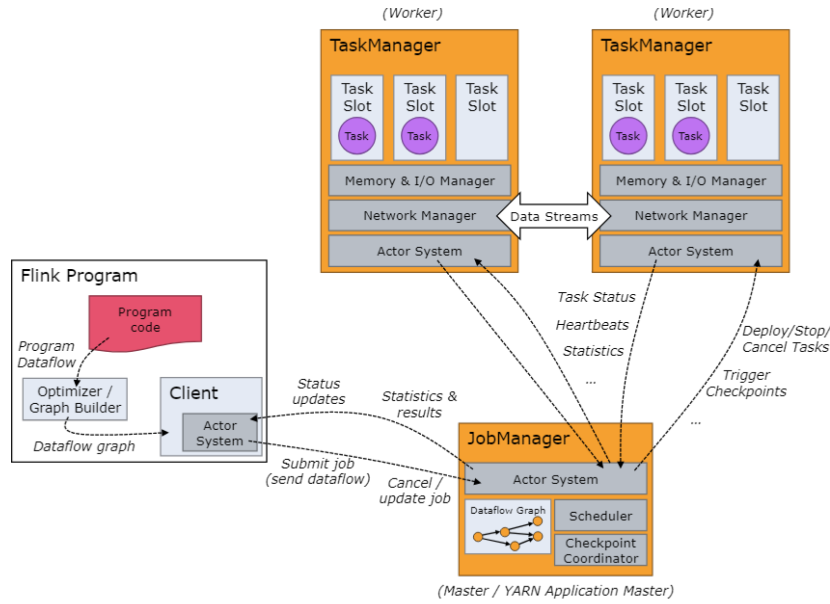

Figure 2: Flink Architecture: Job Manager and Task Manager Interaction

Job Manager coordinates its execution across the cluster. Task Managers execute the assigned tasks, handling memory, I/O, and network communication to ensure efficient data processing.

2.2.2 Execution and Scheduling Mechanism

Flink jobs are represented as Directed Acyclic Graphs (DAGs), where nodes represent operations (e.g., map, filter) and edges represent the data flows between these operations. The user-defined job code is transformed into a JobGraph, which is then optimized and converted into an ExecutionGraph by the Job Manager.

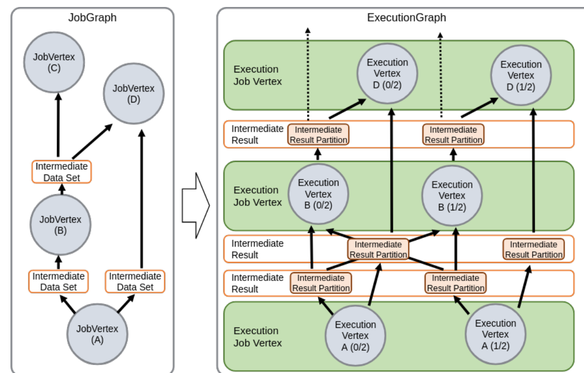

Figure 3: From JobGraph to ExecutionGraph in Flink

Figure 3 illustrates how a JobGraph, which is a logical representation of the job, is translated into an ExecutionGraph, a physical representation optimized for execution. The ExecutionGraph represents the job as a collection of ExecutionVertices (tasks) and ExecutionEdges (data flows), allowing Flink to efficiently schedule and execute tasks across the cluster.

Each operator in the ExecutionGraph can be executed in parallel across multiple Task Managers, depending on the parallelism specified for the job. This parallel execution is key to Flink's scalability and performance.

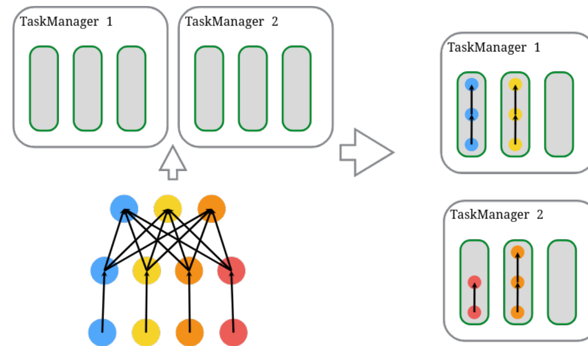

Figure 4: Illustration of Flink's Execution and Scheduling Mechanism

Figure 4 provides a visual representation of Flink's execution and scheduling mechanism. The image shows how a JobGraph is partitioned into smaller execution units, each of which can be scheduled and run in parallel on different Task Managers within the cluster. The color-coded circles represent different stages of the job, which are mapped onto the corresponding ExecutionVertices, ensuring efficient resource utilization and parallelism during job execution.

2.2.3 State Management in Flink

In Flink, state refers to the data that an operator maintains across the processing of multiple events. This is particularly important in stateful stream processing, where the outcome of processing an event can depend on the previously seen events. Flink provides different types of state, including:

1. **Keyed State**: This is the most common form of state in Flink. Keyed state is partitioned and associated with individual keys within a data stream. Each key has its own isolated state, allowing for efficient state management and access. This is managed through the concept of Key Groups, where keys are mapped to Key Groups, which are then distributed across the Task Managers. The state is stored using a State Backend, which can be in-memory or persisted using RocksDB, providing flexibility in terms of performance and durability. Figure 5 illustrates how keyed state is distributed across different stateful operators in Flink. Specifically, it shows how keys are redistributed between the operators based on the key grouping, ensuring that each operator only processes the keys it is responsible for.

2. **Operator State**: This type of state is scoped to the operator as a whole, rather than individual keys. It's useful when the state is not naturally tied to keys, such as in a source function that needs to track offsets. A common example of Operator State is found in the Kafka Connector, where each parallel instance of the Kafka consumer maintains a map of topic partitions and offsets as its Operator State. Operator State supports redistribution among parallel operator instances when the parallelism is changed, making it a flexible solution for scenarios where state partitioning by key is not applicable.

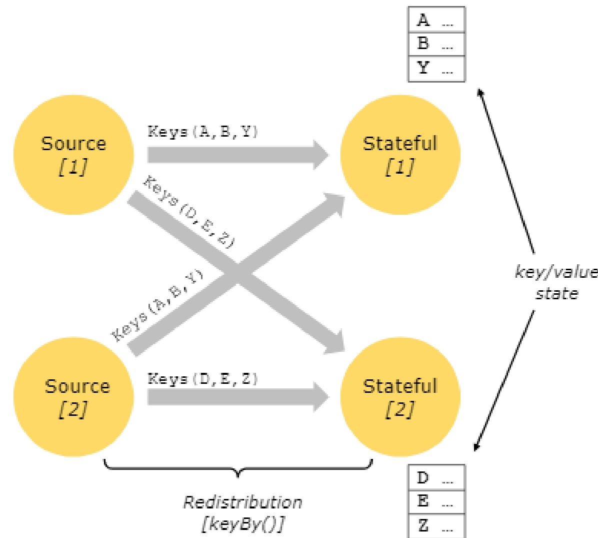

Figure 5: Keyed State Distribution in Flink

2.2.4 Programming Model

Flink's programming model is based on Directed Acyclic Graphs (DAGs), which represent the operations and their data flows. DAGs facilitate parallel execution and optimization of processing tasks. Flink automatically generates a DAG from the user-defined job code, optimizing it for efficient execution. An example of how Flink generates a DAG from a code snippet is shown in Figure 6.

Flink uses operators to define the transformations and processing logic within a job. These operators can be categorized as follows:

1. **Sources**: Entry points for data into the Flink system. Flink can ingest data from various sources such as Kafka, files, or databases. Sources can be bounded (finite datasets) or unbounded (continuous streams).
2. **Processing (Transformation Operators)**: Core data processing logic, where operations such as filtering, mapping, joining, and windowing are applied.
3. **Sinks**: Endpoints where processed data is emitted. Sinks can include databases, files, message queues, or dashboards. Flink's flexibility allows it to support a wide range of sinks, making it suitable for diverse applications.

Flink offers a variety of APIs that cater to different levels of abstraction and programming needs:

- **DataStream API**[1]: Used for stream processing, this API provides a range of operators for transformations and supports event-time and processing-time semantics, windowing, and stateful computations.
- **DataSet API**: Geared towards batch processing, this API allows users to work with bounded datasets and provides rich operators for data transformations, such as map, reduce, join, and filter.

[1]`https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/datastream_api.html`

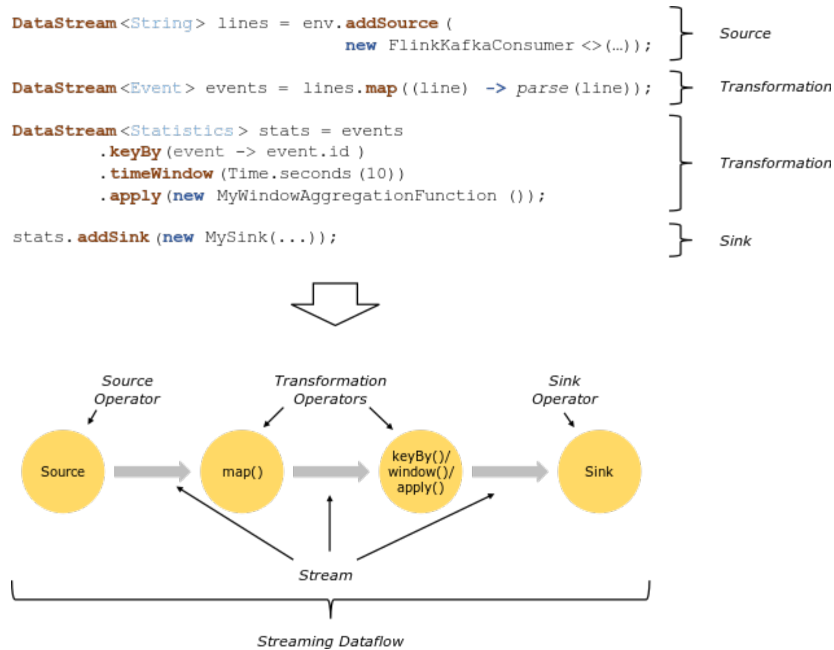

Figure 6: Code to Flink DAG

- **Table API and SQL**: These APIs offer a relational programming model, enabling users to perform data transformations and queries using SQL-like expressions. They are suitable for both stream and batch processing.

The variety of APIs ensures that developers can choose the most appropriate tools for their specific use cases, leveraging the strengths of Flink's underlying execution engine.

2.2.5 DataStream API

The DataStream API is Flink's primary interface for stream processing applications. It provides powerful abstractions and operators for handling continuous streams of data with event-time and processing-time semantics. Key features of the DataStream API include:

- **Event-Time Processing**: The API supports event-time processing, which allows the system to process events based on the timestamps of the data itself. This is crucial for handling out-of-order events and late arrivals, ensuring accurate and consistent results.

- **Windowing**: The DataStream API provides comprehensive windowing capabilities, including tumbling, sliding, and session windows. These windows allow developers to group events into finite sets based on time or other criteria, facilitating time-based aggregations and computations.

- **Stateful Computations**: The API supports stateful operations, enabling applications to maintain and query state efficiently. This is essential for tasks that require context over time, such as session tracking or anomaly detection.

- **Connectors**: Flink's DataStream API includes a wide range of connectors for integrating with various data sources and sinks, such as Kafka, HDFS, and relational databases. This flexibility allows for seamless integration with existing data infrastructures.
- **Fault Tolerance**: The API leverages Flink's checkpointing mechanism to provide exactly-once processing semantics. This ensures that in case of a failure, the application can recover to a consistent state with minimal data loss.

An example of a simple Flink job using the DataStream API is shown below:

Listing 1: Flink DataStream API Example

```
StreamExecutionEnvironment env = StreamExecutionEnvironment
    .getExecutionEnvironment();
DataStream<String> text = env.readTextFile("input.txt");

DataStream<Tuple2<String, Integer>> counts = text
    .flatMap(new Tokenizer())
    .keyBy(0)
    .timeWindow(Time.seconds(5))
    .sum(1);

counts.writeAsText("output.txt");

env.execute("Word Count Example");
```

This example demonstrates how to read text data from a file, tokenize the text into words, group the words, apply a windowing operation, and sum the word counts within each window.

2.2.6 State Management and Fault Tolerance

Flink excels in state management and fault tolerance, which are critical for reliable stream processing. Below are the key concepts related to these aspects in Flink.

Checkpointing

Checkpointing is a fundamental mechanism in Flink that ensures state consistency and fault tolerance. It involves periodically creating snapshots of the application's state without stopping the data stream processing. These checkpoints are consistent, meaning they capture a global state of the system at a specific point in time, allowing Flink to resume processing from the last checkpoint in the event of a failure with minimal data loss.

Flink allows configuring the state backend, which is the system where these checkpoints are stored. State backends can be local file systems, distributed file systems like HDFS, or databases such as RocksDB. Each backend offers different advantages in terms of performance, durability, and scalability.

Savepoints

Savepoints are an extension of the checkpoint concept, used to perform controlled stops of a Flink job. Unlike checkpoints, savepoints are manually triggered and are designed to be persisted indefinitely. This allows a Flink job to be stopped and later resumed from a savepoint, facilitating tasks such as updates, migrations, or application restorations.

- Savepoints capture the exact state of an application at a given point, and allow restoring an application on a new cluster or with a new configuration while maintaining state integrity.
- They are useful for seamless updates, as they allow stopping an application, changing its code or configuration, and then resuming it exactly where it left off.

Barriers

Checkpoint barriers are an essential part of the checkpointing mechanism. A checkpoint barrier is a special marker inserted into the data stream by the Job Manager that propagates through the system. Barriers separate the data before and after a checkpoint, ensuring that the captured state is consistent.

When a barrier reaches an operator, that operator takes a snapshot of its state and stores it as part of the checkpoint. Barriers allow Flink to achieve efficient distributed coordination for creating checkpoints without interrupting real-time data processing.

Fault Tolerance Systems

Flink also employs a two-phase commit protocol to ensure exactly-once processing semantics. This protocol is used to synchronize state across different distributed components of the system, ensuring that state updates are either fully completed or rolled back in the event of a failure, maintaining data integrity and consistency, as illustrated in Figure 7.

1. **Phase One - Prepare**: During this phase, all participants (such as Task Managers) prepare to commit the transaction and write a preliminary version of the state changes to a temporary location.
2. **Phase Two - Commit**: Once all participants are prepared and no errors have occurred, the system commits the transaction, finalizing and making permanent the preliminary state changes.

This mechanism is crucial for applications requiring high reliability and correctness, such as financial transactions or critical monitoring systems.

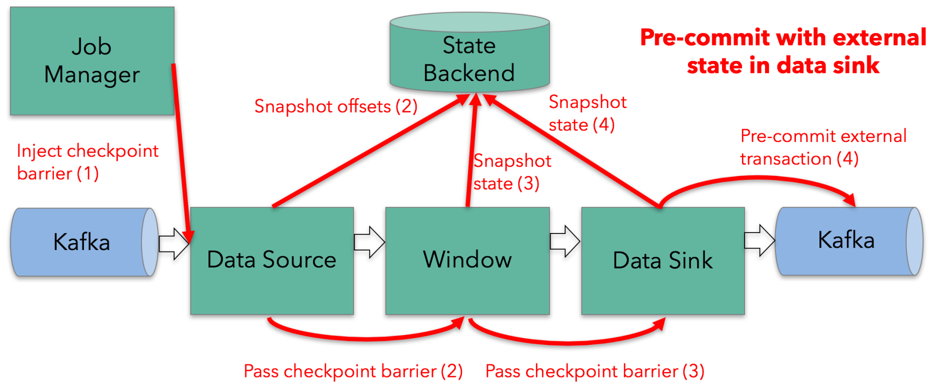

Figure 7: Exactly-once two-phase commit process in Flink

2.3 Machine Learning Overview

Machine learning (ML) is a subfield of artificial intelligence (AI) that focuses on the development of algorithms that allow computers to learn from and make decisions based on data. Unlike traditional programming, where a computer is explicitly instructed on what to do, ML involves training a model on a dataset, enabling the model to identify patterns and make predictions or decisions without explicit programming.

2.3.1 Deep Learning and Neural Networks

Deep learning, a subset of machine learning, involves neural networks with many layers (hence "deep") that can model complex patterns in data. These networks, known as deep neural networks (DNNs), have achieved outstanding performance in various tasks, including image and speech recognition, natural language processing, and game playing.

A key aspect of deep learning is the use of convolutional neural networks (CNNs) for image-related tasks. CNNs are designed to process and recognize patterns in visual data through convolutional layers that detect features such as edges, textures, and shapes.

2.3.2 Inference with ResNet

ResNet (Residual Networks) is a popular type of CNN introduced by He et al. [6] that addresses the degradation problem in deep networks. As networks become deeper, adding more layers does not necessarily improve performance and can even degrade it. ResNet introduces residual connections to mitigate this issue.

Residual Connections and Vanishing Gradient Problem

The ResNet-101 model uses residual connections to enable the training of very deep networks by mitigating the vanishing gradient problem. The vanishing gradient problem occurs during the training of deep neural networks when the gradients of the loss function with respect to the model parameters become very small. This can lead

to slow convergence or the network getting stuck during training, as the gradients are not sufficient to make significant updates to the weights.

Residual connections [18] address this issue by introducing shortcut connections that bypass one or more layers. These connections allow the gradient to flow directly through the network, reducing the likelihood of vanishing gradients. As a result, residual connections make it possible to train much deeper networks, which can learn more complex features and achieve better performance.

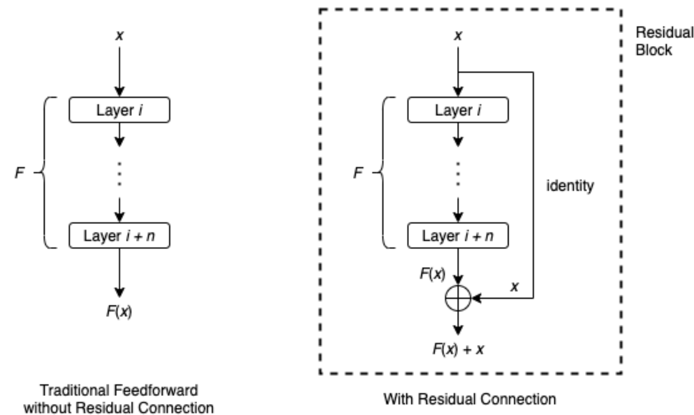

Figure 8: Residual connections in a deep neural network

2.3.3 PyTorch: Deep Learning Framework

PyTorch[13] is an open-source machine learning framework developed by Facebook's AI Research lab. It is widely used for deep learning applications due to its dynamic computation graph and ease of use. PyTorch has become a popular choice among researchers and practitioners for developing and deploying deep learning models.

Features and Advantages

PyTorch offers several features that make it a preferred framework for deep learning:

- **Dynamic Computation Graphs**: Unlike static computation graphs used by other frameworks, PyTorch uses dynamic computation graphs, allowing for more flexibility and ease of debugging.
- **Ease of Use**: PyTorch's intuitive API and Pythonic nature make it easy to learn and use, facilitating rapid prototyping and experimentation.
- **Rich Ecosystem**: PyTorch has a rich ecosystem of libraries and tools, such as torchvision for computer vision tasks, that extend its capabilities and simplify the development process.
- **Community Support**: PyTorch has a large and active community that contributes to its development and provides support through forums, tutorials, and documentation.

2.3.4 CPU and GPU Support.

One of PyTorch's key features is its ability to run computations on both CPUs and GPUs. This flexibility allows users to develop and test models on a CPU and

then scale up to one or multiple GPUs to accelerate training and inference. When running on GPUs, PyTorch utilizes CUDA, which is a parallel computing platform and programming model developed by NVIDIA. CUDA provides direct access to the GPU's virtual instruction set and parallel computational elements, enabling dramatic performance improvements for large-scale computations.

Multi-Language Support

PyTorch extends its usability beyond Python with support for other programming languages through its libtorch library. Libtorch provides a C++ frontend for PyTorch, allowing developers to integrate PyTorch models into C++ applications seamlessly. Additionally, there are bindings available for other languages, such as Java, which enable PyTorch to be used in diverse environments and applications, enhancing its versatility and appeal for both research and production use.

2.3.5 DJL: Deep Java Library

The Deep Java Library (DJL)[8] is an open-source library developed by Amazon that allows developers to train and deploy deep learning models in Java. DJL provides a set of high-level APIs to build, train, and deploy deep learning models using popular deep learning frameworks such as PyTorch, TensorFlow, and MXNet. This library is particularly useful for integrating deep learning models into Java-based applications, including Apache Flink jobs.

Features and Advantages

DJL offers several features that make it a valuable tool for deep learning in Java environments:

- **Multi-Framework Support**: DJL supports multiple deep learning frameworks, allowing developers to choose the best tool for their specific needs without changing their codebase.
- **Ease of Integration**: DJL provides simple APIs that make it easy to integrate deep learning models into Java applications, facilitating seamless deployment in production environments.
- **Pretrained Models**: DJL includes a model zoo with a variety of pretrained models for common tasks such as image classification, object detection, and natural language processing. These models can be used directly or fine-tuned for specific applications.
- **Performance**: DJL leverages native libraries for efficient computation, ensuring high performance and low latency for deep learning tasks.
- **Community and Documentation**: DJL is supported by an active community and comprehensive documentation, making it accessible to both beginners and experienced developers.

2.3.6 Use Cases

The use of DJL in combination with Apache Flink opens up several powerful use cases:

- **Real-Time Image and Video Analysis**: DJL can be used to deploy image and video analysis models within Flink jobs, enabling real-time surveillance, quality inspection, and content moderation.
- **Natural Language Processing**: DJL models for NLP tasks can be integrated into Flink jobs to process and analyze text data streams in real-time, providing capabilities such as sentiment analysis and entity recognition.
- **Predictive Maintenance**: In industrial applications, DJL can be used to deploy models that analyze sensor data in real-time to predict equipment failures and schedule maintenance proactively.

2.4 Grafana and Prometheus: Monitoring and Visualization

Effective monitoring and visualization are crucial for managing and optimizing the performance of real-time data processing systems. Grafana[7] and Prometheus[17] are two widely used open-source tools that provide powerful capabilities for these purposes.

2.4.1 Prometheus

Prometheus is an open-source monitoring and alerting toolkit designed for reliability and scalability. It is particularly well-suited for monitoring dynamic cloud environments. Prometheus collects and stores metrics as time series data, recording information with a timestamp. It uses a powerful query language called PromQL to query this data.

Features and Advantages

Prometheus offers several features that make it a powerful tool for monitoring and alerting:

- **Data Collection**: Prometheus collects metrics from instrumented applications, including Flink jobs, using a pull model. Applications expose metrics via HTTP endpoints, and Prometheus scrapes these endpoints at specified intervals.
- **Time Series Database**: Prometheus stores metrics as time series data, which allows for efficient querying and analysis of historical data.
- **Alerting**: Prometheus supports alerting based on the metrics it collects. Users can define alerting rules using PromQL, and Prometheus can trigger alerts if certain conditions are met.
- **Scalability**: Prometheus is designed to handle high cardinality metrics and can be scaled horizontally by federating multiple Prometheus servers.

Push vs. Pull Data Collection

Prometheus supports two primary models for data collection: push and pull.

- **Pull Model**: In the pull model, Prometheus server scrapes metrics from instrumented applications at regular intervals. Applications expose metrics at a designated HTTP endpoint, and Prometheus periodically queries these endpoints

to collect data. This model is advantageous because it allows Prometheus to control the data collection process, manage load on the monitoring system, and dynamically discover services.

- **Push Model**: In the push model, instrumented applications push metrics to a gateway, which then exposes these metrics to Prometheus. This approach is typically used for short-lived jobs or batch processing tasks where the application lifecycle is too short to be effectively scraped.

In our implementation, we utilized the pull model. This approach is particularly beneficial for monitoring long-running services like Flink jobs, where Prometheus can efficiently scrape metrics from exposed endpoints at regular intervals, ensuring continuous monitoring and up-to-date data collection.

2.4.2 Grafana

Grafana is an open-source platform for monitoring and observability that integrates seamlessly with Prometheus. It provides a rich set of features for creating interactive and customizable dashboards.

- **Dashboards**: Grafana allows users to create dashboards that visualize metrics collected by Prometheus. Dashboards can include various types of charts, graphs, and tables to represent data in a meaningful way.
- **Alerting**: Grafana supports alerting and notification systems. Users can set up alerts based on Prometheus metrics and receive notifications via email, Slack, or other channels.
- **Plugins**: Grafana has a wide range of plugins that extend its functionality, allowing integration with other data sources and tools.
- **Custom Visualization**: Users can create custom visualizations using Grafana's powerful visualization options, which include heatmaps, histograms, and geomaps.

2.5 Challenges in Streaming Machine Learning Inference Processing

Streaming machine learning inference processing presents significant challenges due to the dynamic and continuous nature of data. One of the main challenges is maintaining low latency while ensuring result accuracy. This requires efficient inference algorithms and optimization techniques to minimize processing time without compromising quality.

Managing model state and data synchronization between system nodes are critical to maintaining result coherence and integrity. This becomes even more complex in distributed environments where multiple model instances are running simultaneously. Effective strategies for state management and synchronization are necessary to ensure consistent and reliable results.

Scalability is another crucial challenge in streaming inference processing, especially when handling large volumes of data and variable workloads. It is essential to design

systems that can scale horizontally to accommodate changes in demand and maintain optimal performance at all times. This requires robust resource management and load balancing mechanisms to ensure efficient utilization of computing resources.

Latency and throughput are critical metrics in streaming machine learning inference processing. Low latency is essential for applications that require real-time insights and actions, while high throughput ensures that the system can handle large volumes of data efficiently. Balancing these two metrics involves optimizing both the inference algorithms and the underlying infrastructure.

Deploying and managing machine learning models in a streaming environment poses unique challenges. Continuous integration and deployment (CI/CD) practices are necessary to ensure that models are regularly updated with new data and improvements. Additionally, monitoring model performance in real-time is crucial to detect and address any degradation in accuracy or efficiency promptly.

2.6 Cloud Storage and Infrastructure

Cloud computing has revolutionized the way organizations store and process data. Amazon Web Services (AWS)[15], as one of the leading cloud service providers, offers a wide range of services to meet organizations' data storage and processing needs. We are explaining it because we used the AWS cloud to implement our different clusters.

2.6.1 Amazon Simple Storage Service (S3)

Amazon Simple Storage Service (S3) is one of AWS's most widely used services for cloud data storage. It offers a scalable and durable solution for storing a variety of data, from static files to real-time generated data. S3 is highly reliable and designed to offer 99.999999999% availability, making it an attractive choice for critical applications.

Features of S3

- **Buckets**: S3 organizes data into "buckets," which are containers for storing objects (files). Each bucket is identified by a unique name within the AWS ecosystem. Buckets serve as the top-level namespace for S3 storage, allowing users to group related objects together.
- **Objects**: An object in S3 consists of the data itself, metadata, and a unique identifier within the bucket. Objects can be of any size, from a few bytes to several terabytes.
- **Versioning**: S3 supports versioning, which allows for multiple versions of the same object to be stored. This feature helps protect against accidental overwrites and deletions by keeping historical versions of objects.
- **Lifecycle Policies**: S3 provides lifecycle policies that automate the transition of objects between different storage classes based on specified rules. For example, objects can be moved from standard storage to infrequent access or archival storage after a certain period.
- **Access Control**: S3 offers robust access control mechanisms, including bucket policies, access control lists (ACLs), and AWS Identity and Access Management

(IAM) policies, to manage permissions and secure data.

- **Replication**: S3 supports cross-region replication, which allows for the automatic and asynchronous copying of objects between buckets in different AWS regions. This enhances data durability and availability.

S3 Billing

S3 is billed based on several factors:

- **Storage Used**: The amount of data stored in S3 buckets, billed per gigabyte per month.
- **Requests and Data Retrievals**: The number of requests made to store and retrieve data, billed per thousand requests.
- **Data Transfer**: The amount of data transferred out of S3 to the internet or to other AWS regions, billed per gigabyte.
- **Storage Management Features**: Usage of features such as versioning, lifecycle policies, and cross-region replication may incur additional costs.

2.6.2 Amazon Elastic Compute Cloud (EC2)

Amazon Elastic Compute Cloud (EC2) allows for launching and scaling virtual server instances on demand. This provides a flexible and scalable infrastructure for running applications and data analytics processes in the cloud. By combining services like EC2 with streaming data processing tools like Apache Flink, organizations can build highly efficient and cost-effective processing systems.

Features of EC2

- **Instance Types**: EC2 offers a wide variety of instance types optimized for different use cases, including compute-optimized, memory-optimized, and storage-optimized instances.
- **Elasticity**: EC2 allows for dynamic scaling of instances based on the current demand, enabling cost savings by allocating resources only when needed.
- **Security**: EC2 instances can be secured using security groups, network ACLs, and IAM roles to control access and permissions.
- **Storage Options**: EC2 provides several storage options, including Elastic Block Store (EBS) for persistent storage and instance store for temporary storage.
- **Networking**: EC2 instances can be launched in a Virtual Private Cloud (VPC), offering control over network configurations and security.

EC2 Billing

EC2 is billed based on several factors:

- **Instance Hours**: The number of hours instances are running, billed per second with a minimum of 60 seconds.
- **Instance Types**: Different instance types have different pricing, depending on their specifications and capabilities.
- **Data Transfer**: The amount of data transferred out of EC2 to the internet or other AWS services, billed per gigabyte.
- **Storage**: Costs associated with the use of EBS volumes or other storage options attached to instances.
- **Additional Services**: Additional costs for using features such as Elastic IP addresses or Load Balancers.

2.6.3 Security and Compliance

AWS provides robust security features and compliance certifications, ensuring that data is protected and regulatory requirements are met. Features such as encryption at rest and in transit, identity and access management (IAM), and audit logging help secure the data processing environment. Compliance with standards such as GDPR, HIPAA, and ISO ensures that the infrastructure meets the necessary legal and regulatory requirements.

AWS security features include:

- **Encryption**: AWS provides encryption for data at rest using AWS Key Management Service (KMS) and encryption for data in transit using SSL/TLS.
- **Identity and Access Management (IAM)**: IAM allows for granular control of user permissions and access to AWS resources.
- **Monitoring and Logging**: AWS offers services like AWS CloudTrail and Amazon CloudWatch for monitoring and logging activities across AWS resources, providing visibility and audit capabilities.
- **Compliance Certifications**: AWS complies with various global standards, including GDPR, HIPAA, SOC, and ISO, ensuring that its services meet stringent regulatory and security requirements.

By using AWS services such as S3 and EC2, organizations can build scalable, secure, and compliant cloud-based data processing systems.

2.7 Summary

In this section, we have explored the fundamental concepts and current advancements in streaming data processing and machine learning inference. Apache Flink has emerged as a versatile and powerful platform for real-time data processing, offering significant advantages over other solutions. However, integrating deep learning models into streaming frameworks remains an area of ongoing research, with many challenges to address.

3 Existing Solutions and Their Limitations

Various solutions have been proposed in the literature to address the challenges of streaming inference processing. Tools such as Apache Storm [10] and Apache Spark [19] have been widely used. However, each has its limitations. For example, Storm offers low latency but lacks robust processing guarantees, and Spark, while powerful in batch processing, was not initially designed for streaming processing.

3.1 Apache Storm

Apache Storm is a distributed stream processing computation framework that provides real-time processing capabilities. It is designed to handle unbounded streams of data and is known for its low latency. Storm's architecture comprises spouts (sources of data streams) and bolts (processing units), which can be arranged in a DAG to define the data flow and processing logic.

While Storm excels in providing low-latency processing, it has several limitations:

- **Processing Guarantees**: Storm primarily supports at-least-once processing guarantees, which can result in duplicate processing of events.
- **State Management**: Storm lacks built-in support for stateful processing, making it challenging to implement applications that require maintaining and querying state over time.
- **Scalability**: Scaling Storm applications can be complex, as it requires manual partitioning and tuning to distribute the workload effectively across the cluster.

3.2 Apache Spark

Apache Spark is a unified analytics engine that supports batch processing, stream processing, and machine learning. Spark Streaming, an extension of the core Spark API, enables scalable and fault-tolerant stream processing of live data streams. It divides data streams into micro-batches and processes them using the Spark engine.

Spark Streaming has the following limitations:

- **Micro-batch Processing**: Spark Streaming's micro-batch architecture introduces latency, making it less suitable for applications that require true real-time processing.
- **Complexity**: Configuring and tuning Spark Streaming for optimal performance can be complex, especially for large-scale applications.

3.3 Apache Flink

Apache Flink provides a balance between batch and streaming processing, offering a more integrated and efficient solution for real-time applications. Flink's exactly-once processing guarantees, state management, and fault tolerance make it particularly suitable for complex streaming applications. Key advantages of Flink over other solutions include:

- **Unified API**: Flink's unified API supports both batch and stream processing, allowing developers to use a single framework for various types of data processing tasks.
- **Low Latency**: Flink's architecture is designed to process data with low latency, making it ideal for real-time analytics and applications.
- **Stateful Processing**: Flink provides robust support for stateful stream processing, enabling the implementation of complex event-driven applications.
- **Scalability and Fault Tolerance**: Flink's distributed execution engine ensures high scalability and fault tolerance, making it suitable for large-scale data processing applications.
- **Exactly-Once Semantics**: Flink's exactly-once processing guarantees ensure that each event is processed exactly once, avoiding issues with duplicate processing and improving the accuracy of the results.

3.4 Other Frameworks

Several other frameworks also offer solutions for streaming inference processing, each with its unique features and limitations:

3.4.1 Kafka Streams

Kafka Streams is a stream processing library built on top of Apache Kafka, designed for building real-time applications and microservices. It integrates seamlessly with Kafka, leveraging its robust messaging capabilities. However, Kafka Streams may not provide the same level of flexibility and state management as Flink.

- **Integration with Kafka**: Kafka Streams is tightly integrated with Apache Kafka, making it an excellent choice for applications that already use Kafka for message brokering.
- **Ease of Use**: Kafka Streams offers a simple and lightweight API for stream processing, which can be easier to use than more complex frameworks like Flink.
- **Limitations**: Kafka Streams lacks some of the advanced features of Flink, such as complex state management and advanced windowing capabilities.

3.4.2 Google Cloud Dataflow

Google Cloud Dataflow [4] is a fully managed service for stream and batch processing. It is based on the Apache Beam model, which allows the same code to be executed on various processing engines, including Flink and Spark.

- **Managed Service**: As a fully managed service, Dataflow handles resource provisioning and scaling automatically.
- **Unified Programming Model**: The Apache Beam model provides a unified programming model for both batch and stream processing.
- **Vendor Lock-In**: Using Google Cloud Dataflow can lead to vendor lock-in, making it less flexible compared to open-source alternatives like Flink.

3.4.3 Samza

Apache Samza[12] is a stream processing framework developed by LinkedIn, designed to work with Apache Kafka. Samza provides strong support for stateful processing and is known for its integration with Hadoop YARN for resource management.

- **Stateful Processing**: Samza offers strong support for stateful stream processing, enabling complex event processing.
- **Integration with Kafka and Hadoop**: Samza is designed to work well with Apache Kafka and Hadoop YARN, providing a robust ecosystem for stream processing.
- **Processing Guarantees**: Unlike Flink, Samza supports at-least-once processing guarantees, which can result in duplicate processing of events and does not provide the same level of processing accuracy.
- **Complexity**: Configuring and managing Samza applications can be complex, especially in large-scale environments.

3.5 Sponge: A New Approach

Recent advancements have introduced Sponge[16], a novel framework that take advantage from Serverless Functions. Sponge addresses several limitations of existing solutions:

- **Adaptive State Management**: Sponge features adaptive state management, enabling efficient handling of dynamic data streams and stateful processing.
- **Scalability**: The framework incorporates advanced partitioning and load balancing techniques, ensuring optimal scalability across distributed environments.
- **Low Latency**: Sponge's architecture is optimized for minimal latency, leveraging cutting-edge techniques to deliver real-time processing.
- **Fault Tolerance**: Sponge includes robust fault tolerance mechanisms, ensuring reliable processing even in the presence of failures.

Sponge employs a unique approach to address the challenges of dynamic and unpredictable streaming workloads by leveraging serverless frameworks. Sponge provides fast reactive scaling for stream processing with serverless frameworks. This design allows Sponge to handle sudden, unpredictable increases in input loads from existing VMs with low latency and cost.

3.5.1 Comparison with Apache Flink

Despite Sponge's impressive capabilities, several aspects warrant a closer comparison with Apache Flink:

- **State Management**: While Sponge introduces a novel redirect-and-merge mechanism for state management, Flink's integrated state management is more mature and has been extensively tested in production environments. Flink's state management allows for complex stateful operations with exactly-once semantics, ensuring high reliability and consistency.
- **Latency**: Sponge's low latency is a significant advantage, especially with bursty loads. However, Flink also provides low-latency processing and has proven capabilities in maintaining low latencies even under high-throughput scenarios. Flink's architecture is inherently designed to minimize latency through its efficient event-driven processing model.
- **Scalability**: Flink offers robust scalability with its distributed execution engine, which is highly optimized for both batch and stream processing workloads. Although Sponge's use of serverless frameworks allows for quick scaling, Flink's scalability has been proven in numerous large-scale production deployments.
- **Resource Utilization**: Sponge's cost efficiency through the use of serverless frameworks for bursty loads is notable. However, the cost benefits must be weighed against the potential higher costs of serverless instances in long-running, stable workloads. Flink's ability to efficiently utilize cluster resources for both stable and dynamic loads can provide more predictable cost management.
- **Maturity and Ecosystem**: Flink benefits from a mature ecosystem with a wide range of connectors, libraries, and integrations, making it a versatile choice for diverse data processing needs. Sponge, being a newer framework, may still need to develop a similar level of ecosystem support.
- **Community and Support**: Flink has a large, active community and is backed by major organizations, providing strong support and continuous improvements. Sponge, while promising, may not yet have the same level of community and industry support.

3.6 Summary

The review of existing solutions for streaming inference processing has highlighted several widely used tools such as Apache Storm, Apache Spark, Apache Flink, Kafka Streams, Google Cloud Dataflow, and Apache Samza. Each tool has specific features and limitations influencing its applicability in real-time processing contexts.

Apache Storm excels in low-latency processing but lacks robust processing guarantees and built-in state management, limiting its use in applications requiring high precision and complex state handling. Apache Spark, while powerful in batch processing, faces challenges in streaming applications due to its micro-batch architecture, which introduces significant latencies.

On the other hand, Apache Flink offers a more balanced solution with unified batch and stream processing, low latency, robust stateful processing support, advanced scalability, and fault tolerance. Its exactly-once processing guarantees and extensive ecosystem make it a versatile and reliable option for real-time streaming applications.

Additional frameworks such as Kafka Streams and Google Cloud Dataflow provide integrated and managed solutions, respectively, but face limitations in terms of flexibility and potential vendor lock-in. Apache Samza, strong in stateful processing and integration with Kafka and Hadoop, does not offer the same level of precise processing guarantees as Flink and can be complex to manage at scale.

The introduction of Sponge as an innovative serverless-based framework stands out for its adaptive state management, scalability, and low latency. However, compared to Apache Flink, Sponge needs to demonstrate maturity in state management, scalability in high-performance scenarios, and resource utilization efficiency.

To summarize, **Table 1** below provides a concise comparison of these streaming frameworks, highlighting their strengths and limitations across key features. While Sponge presents promising advances for handling unpredictable workloads with fast-scaling serverless techniques, Apache Flink remains a highly favorable option for many real-time streaming applications due to its robust state management, low latency, proven scalability, and well-established ecosystem. Future improvements in Sponge could address some of these comparative limitations, enhancing its competitiveness in the streaming processing landscape.

Feature	**Storm**	**Spark**	**Flink**	**Sponge**
Processing Guarantees	At-least-once	At-least-once	Exactly-once	Exactly-once
Latency	Low	Higher (Micro-batch)	Low	Very Low
State Management	Limited	Moderate	Advanced	Advanced
Scalability	Complex	Moderate	High	High
Fault Tolerance	Basic	Good	Excellent	Excellent
Resource Utilization	Manual Tuning	Moderate	Efficient	Serverless
Ecosystem	Limited	Extensive	Extensive	Growing
Community Support	Moderate	High	High	Low

Table 1: Comparison of Streaming Frameworks

4 Functional Requirements

This section outlines the functional requirements for the streaming inference system implemented using Apache Flink and DJL. Functional requirements define the specific behaviors and functionalities that the system must exhibit to meet its objectives.

4.1 Data Ingestion

The system must be able to ingest data continuously from Amazon S3, ensuring that the data is available for processing in real-time.

- **FR1.1**: The system shall support ingestion of image data from Amazon S3.
- **FR1.2**: The system shall be able to handle different image formats, including JPEG, PNG, and BMP.
- **FR1.3**: The system shall monitor the specified S3 bucket for new images and ingest them in real-time.

4.2 Data Preprocessing

Before performing inference, the system must preprocess the incoming image data to ensure it is in the correct format for model consumption.

- **FR2.1**: The system shall resize input images to a fixed resolution required by the deep learning model (e.g., 224x224 pixels).
- **FR2.2**: The system shall normalize pixel values of input images to a range suitable for the model.
- **FR2.3**: The system shall convert images into tensors suitable for deep learning model inference.

4.3 Model Inference

The core functionality of the system is to apply a pre-trained deep learning model to incoming images and generate predictions.

- **FR3.1**: The system shall load pre-trained deep learning models using DJL.
- **FR3.2**: The system shall perform inference on each incoming image to classify it into one of the predefined categories.
- **FR3.3**: The system shall ensure that the inference process is performed with low latency, suitable for real-time applications.

4.4 Post-Processing

After inference, the system must handle the results appropriately, ensuring they are stored or transmitted for further use.

- **FR4.1**: The system shall generate thumbnails for each processed image and store them in a specified output directory.
- **FR4.2**: The system shall save the classification results along with corresponding metadata to a database or file system.

4.5 System Performance

The system must meet specific performance criteria to ensure it is effective in real-time environments.

- **FR5.1**: The system shall process at least 100 images per second to ensure high throughput.
- **FR5.2**: The system shall achieve an inference latency of less than 100 milliseconds per image.
- **FR5.3**: The system shall scale horizontally to handle increasing data loads without significant degradation in performance.

4.6 Error Handling and Fault Tolerance

The system must be robust and handle errors gracefully, ensuring high availability and reliability.

- **FR6.1**: The system shall implement checkpointing to recover from failures and minimize data loss.
- **FR6.2**: The system shall log errors and exceptions to facilitate troubleshooting and maintenance.
- **FR6.3**: The system shall continue processing remaining data even if some inputs are invalid or corrupted.

4.7 Summary

The functional requirements outlined in this section provide a comprehensive foundation for the development and implementation of a robust streaming inference system. By clearly defining the capabilities and behaviors expected from the system, we ensure that all critical aspects, from data ingestion to post-processing and error handling, are adequately addressed.

5 System Design

The design of the real-time image inference processing system using Apache Flink and DJL (Deep Java Library) is the result of careful consideration of several key factors, including performance, scalability, reliability, and the practical constraints of available technologies. Before diving into the implementation, it was essential to make informed design choices that would guide the overall architecture and ensure that the system meets its intended goals efficiently and effectively.

5.1 Design Choices and Rationale

The decision to utilize Apache Flink as the core processing engine was primarily driven by its strong support for distributed stream processing and its ability to handle large volumes of data with low latency. Flink's architecture is well-suited for real-time applications where continuous data ingestion and processing are critical. Moreover, Flink's exactly-once processing semantics provide a reliable foundation for building systems that require high accuracy and consistency, which is particularly important in scenarios involving deep learning inference where the integrity of results is paramount.

In selecting DJL for the deep learning component, the key considerations were compatibility with Java-based systems, ease of integration with Flink, and support for various deep learning frameworks such as PyTorch and TensorFlow. DJL's ability to facilitate the deployment of pre-trained models within a Java environment without requiring extensive boilerplate code was a significant factor in its selection. This choice simplified the integration of deep learning models into the Flink processing pipeline, enabling the system to leverage advanced machine learning capabilities without introducing unnecessary complexity.

The choice of deploying the system on AWS was motivated by the need for a scalable, reliable, and flexible infrastructure. AWS provides the necessary computational resources, such as EC2 instances, which can be dynamically scaled to meet the demands of the processing workload. This cloud-native approach ensures that the system can adapt to varying data volumes and processing requirements, providing both performance and cost efficiency. Additionally, AWS services like S3 offer robust data storage solutions that integrate seamlessly with the processing framework, ensuring that data is readily accessible for ingestion and processing.

5.2 Architecture Overview

The architecture is centered around a cloud-native deployment on Amazon Web Services (AWS), utilizing services such as Amazon S3 for data storage and Amazon EC2 for computational resources. This cloud infrastructure provides the system with the flexibility to scale according to workload demands, dynamically allocating resources to handle varying data volumes. The system continuously monitors a designated S3 bucket for incoming image data, which is then ingested and processed in real-time. This ingestion process is designed to support various image formats including JPEG, PNG, and BMP, which are automatically detected and converted into the required format for further processing.

Once the images are ingested, they undergo a series of preprocessing steps to prepare them for model inference. These steps include resizing the images to a fixed resolution,

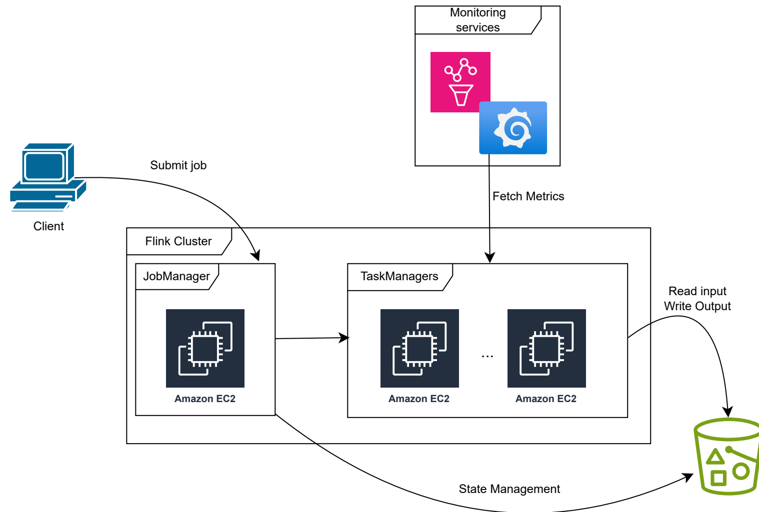

Figure 9: Preliminary Design of the Real-Time Image Inference Processing System

typically 224x224 pixels, which is a standard input size for many deep learning models. Additionally, the pixel values are normalized to a range between 0 and 1, which is the input range expected by the deep learning model. This preprocessing ensures that the images are in the correct format and ready for the computational demands of deep learning inference.

5.3 Model Inference and Processing Efficiency

The choice of ResNet-101 as the model for inference is driven by its balance between accuracy and computational efficiency. ResNet-101, with its deep architecture, is capable of capturing complex features in images, making it suitable for high-precision tasks. DJL facilitates the integration of this model into Apache Flink, allowing for seamless real-time inference. The system is designed to load the pre-trained model at the start of the processing pipeline, ensuring that it is readily available as new images are ingested.

To optimize performance, the ideal scenario would involve leveraging the parallel processing capabilities of GPUs for the computationally intensive tasks of deep learning inference, while reserving CPU resources for less intensive tasks such as data ingestion and I/O operations. However, it is important to note that Apache Flink does not natively support distinguishing between TaskManagers with or without GPU resources within a single job. As a result, while the architecture is designed to theoretically optimize processing by separating tasks between CPU and GPU, in practice, this distinction is not implemented at the level of TaskManagers within Flink. Instead, the system relies on the deployment environment (e.g., specific EC2 instances with GPUs) to ensure that all TaskManagers are capable of handling the computational demands of deep learning inference uniformly, albeit without the fine-grained control over resource allocation that would ideally maximize efficiency.

5.4 Post-Processing and Output Management

Post-processing plays a crucial role in ensuring that the results of the inference are managed effectively. After classification, the system generates thumbnails and

stores the classification results along with relevant metadata. These results are then saved in a structured format in the specified output location, such as an S3 bucket or a database, making them easily accessible for further analysis or visualization. The thumbnails are particularly useful for applications requiring image previews, such as content management systems or media galleries, as they reduce storage space and speed up image retrieval and display.

5.5 State Management and Fault Tolerance

To maintain the system's reliability and performance, robust state management and fault tolerance mechanisms are integral to the design. Apache Flink's state management capabilities are employed extensively, with checkpointing enabled to periodically save the system state. This ensures that in the event of a failure, the system can recover quickly, resuming operations from the last checkpoint without significant data loss. The system is configured to provide exactly-once processing semantics, ensuring that each data event is processed precisely once, avoiding issues of data duplication or loss common in distributed systems.

5.6 Scalability and Monitoring

Scalability is a cornerstone of the system design, allowing the system to scale horizontally by adding more Flink Task Managers as needed. This horizontal scalability is essential for handling increases in data volume without compromising performance. In real-time processing environments, where data streams can vary significantly, the ability to scale dynamically is crucial for maintaining consistent performance and preventing bottlenecks.

The system's performance is continuously monitored through a Grafana dashboard, which provides real-time insights into key metrics such as latency, throughput, and resource utilization. This monitoring allows operators to detect and address potential issues proactively, ensuring the system remains reliable and responsive. Alerts are configured within Grafana to notify administrators of any anomalies, further enhancing the system's robustness.

5.7 Deployment and Configuration Management

The design emphasizes ease of deployment and management, utilizing tools such as Terraform for infrastructure as code and Ansible for configuration management. These tools enable consistent and efficient deployment across different environments, ensuring that the infrastructure is easily reproducible and scalable. The cloud-native design leverages the full range of AWS services, providing a robust, flexible, and scalable platform for real-time image inference processing.

5.8 Summary

In conclusion, the system design is a comprehensive response to the challenges of real-time image inference processing, combining the strengths of Apache Flink and DJL in a cloud-based environment. The design decisions made throughout the development process ensure that the system is capable of delivering high performance, reliability, and scalability, making it well-suited for a wide range of real-time applications. By integrat-

ing state-of-the-art deep learning models with advanced stream processing techniques, the system is positioned to handle the demands of modern data-driven applications, providing timely and accurate insights in real-time.

6 Implementation

This section details the execution of the streaming inference system using Apache Flink and DJL (Deep Java Library) for real-time image classification. The implementation integrates deep learning models into Flink jobs, enabling efficient processing and inference on streaming data.

6.1 System Setup and Deployment

The system setup involved realizing the architecture defined in the design phase. To deploy the system in a cloud environment, we utilized Amazon Web Services (AWS) for its computational resources due to its flexibility and scalability. The deployment process was automated using Terraform and Ansible. Terraform was chosen for its ability to manage infrastructure as code, ensuring a consistent environment across deployments. Ansible was used for configuration management, ensuring that all systems were correctly set up and that dependencies were consistently installed.

The specific scripts used for Terraform and Ansible configurations are provided in the appendix. These scripts ensure reproducibility and ease of deployment across different environments.

6.2 DJL Integration

DJL (Deep Java Library) plays a crucial role in this implementation by providing the necessary tools to integrate deep learning models into Java-based applications, such as Apache Flink jobs.

6.2.1 Why ResNet-101?

The choice of the ResNet-101 model is driven by its proven accuracy and efficiency in image classification tasks. ResNet-101, is highly regarded for its ability to deliver high-performance results without excessive computational requirements. Its depth of 101 layers provides a robust framework for capturing intricate features in images, making it particularly suitable for complex classification tasks. Integrating ResNet-101 with Apache Flink using DJL (Deep Java Library) allows us to leverage a state-of-the-art model for real-time inference, ensuring that the system can process streaming data with high accuracy and efficiency. This integration supports the system's goal of maintaining low latency and high throughput, essential for applications where timely and accurate image classification is critical.

6.2.2 Integration Process

Integrating DJL with Apache Flink involves the following technical steps:

1. **Loading Models in Flink**: The pre-trained models are loaded into the Flink job using DJL's APIs. This step required configuring the Flink environment to handle model loading efficiently and ensuring that the model is available in memory for quick access during inference.

2. **Inference Pipeline**: The Flink job processes incoming data, applies the loaded DJL models for inference, and outputs the results to the desired sink (e.g.,

database, file system, or dashboard). The pipeline was designed to handle high throughput, ensuring that the inference process did not become a bottleneck in the data processing workflow.

6.3 Flink Job Implementation

The core of the implementation involves defining the Flink job that handles the data stream, applies the deep learning model, and processes the output.

6.3.1 DAG Logical Overview

The Flink job is represented as a Directed Acyclic Graph (DAG), where each node represents an operation (e.g., source, map, sink). Below is a graphical representation of the logical DAG implemented:

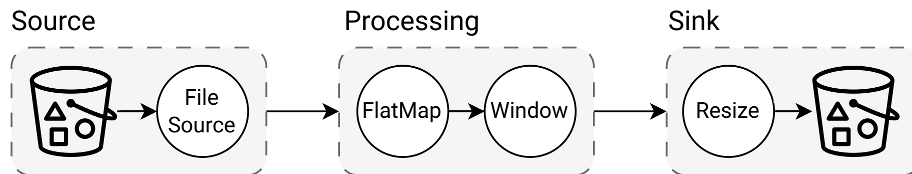

Figure 10: Logical DAG of the Flink Job

- **Source:** The pipeline begins with the *File Source* node, where image data is ingested from a storage system (e.g., an S3 bucket). This node is responsible for reading the raw image files and feeding them into the pipeline for processing.
- **Processing:** The next stage consists of two main operations:
  - **FlatMap:** This node represents the transformation logic that is applied to each image.
  - **Window:** Following the *FlatMap*, the data passes through a *Window* operation, where the classification is made.
- **Sink:** The final stage is the *Resize* operation, where a thumbnail is generated for all images that pass the class filter. The resized images are then stored in an output storage system, in our case, an S3 bucket.

6.3.2 Stream Execution Environment

The Stream Execution Environment is the primary context in which a Flink application is executed. In this implementation, it is responsible for setting up the execution parameters, managing the job configuration, and ensuring fault tolerance through checkpointing. The specific configuration used was as follows:

```
1 final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
2 env.enableCheckpointing(60000L, CheckpointingMode.EXACTLY_ONCE);
3 env.getConfig().setGlobalJobParameters(params);
```

Listing 2: Stream Execution Environment Configuration

6.3.3 *Data Source Configuration*

The data source is configured to read images from the specified input paths provided via CLI arguments. In this implementation, the images are stored in an Amazon S3 bucket. The data source setup for reading from S3 is shown below.

These images are processed using a custom format adapter to convert them into a format suitable for further processing. This involves using the 'StreamedImage' format, which contains the image path and the image loaded by the DJL library. The path is necessary for post-processing tasks.

```
1  final FileSource<StreamedImage> source =
2                 FileSource.forBulkFileFormat(
3                     new CustomStreamFormatAdaper<>(new ImageReaderFormat()), paths)
4                         .build();
5
6             stream =
7                 env.fromSource(source, WatermarkStrategy.noWatermarks(), "file-source")
8                     .flatMap(new PrepareImage())
9                     .filter(Objects::nonNull);
```

Listing 3: Data Source Configuration

- First, we create a Flink source (`FileSource<StreamedImage> source`) that will read images from the specified file paths. Since the data we are dealing with are images, we needed to implement a custom reader. This is done using `CustomStreamFormatAdaper`, which adapts our specific image format through `ImageReaderFormat`. The custom adapter converts the raw image files into `StreamedImage` objects, a custom class we've created to represent the images in a way that Flink can process.
- Next, we initialize a data stream (`stream`) from the source. We use `WatermarkStrategy.noWatermarks()` because in this case, we are not dealing with event-time processing, so no watermarks are necessary. The stream is named `"file-source"` for identification purposes within the Flink environment.

After configuring the data source, a 'flatMap' function is applied. The 'flatMap' function transforms each input element into zero or more output elements. In this case, it processes each image, performing necessary preprocessing steps to prepare the image for the inference.

6.3.4 *Image Preparation*

Image preparation is a critical step in the implementation, involving the resizing and normalization of input images. The 'PrepareImage' function handles this preprocessing step to ensure the images are in the correct format for inference by the deep learning model. Normalizing an image involves scaling the pixel values to a range that the model expects, often between 0 and 1. This is followed by converting the image to a tensor, which is the data structure used by deep learning models for computation. Below is the code snippet for this function.

A tensor is a multidimensional array, and in the context of image processing, it typically has three dimensions: height, width, and color channels. For a color image, the tensor has three channels corresponding to the red, green, and blue (RGB) color

components. The following figure illustrates the structure of a tensor for a 4x4 color image with three channels (See Fig. 11).

```
1  public void flatMap(StreamedImage streamedImage, Collector<StreamedImage> collector)
2                  throws Exception {
3
4              try {
5                  if (CudaUtils.getGpuCount() > 0) {
6                      manager = NDManager.newBaseManager(Device.gpu());
7                  } else {
8                      manager = NDManager.newBaseManager(Device.cpu());
9                  }
10                 Image image =
11                         streamedImage
12                                 .getImage()
13                                 .resize(256, 256, false)
14                                 .getSubImage(16, 16, 224, 224);
15
16                 processedImage = image.toNDArray(manager, Image.Flag.COLOR);
17                 processedImage = NDImageUtils.toTensor(processedImage);
18
19                 // Normalize the image
20                 processedImage =
21                         NDImageUtils.normalize(
22                                 processedImage,
23                                 new float[] {0.485f, 0.456f, 0.406f},
24                                 new float[] {0.229f, 0.224f, 0.225f});
25
26                 StreamedImage result =
27                         new StreamedImage(
28                                 ImageFactory.getInstance().fromNDArray(processedImage),
29                                 streamedImage.getFilePath());
30                 collector.collect(result);
31
32                 manager.close();
33
34             } catch (IllegalArgumentException e) {
35                 logger.error("Failed to process image", e);
36                 collector.collect(null);
37             }
38         }
```

Listing 4: Image Preparation Function

- First, it determines whether a GPU is available. If a GPU is available, the image processing will leverage GPU resources.
- The image is then resized to a 256x256 resolution, and a central 224x224 sub-image is extracted. This resizing and cropping step is common in image pre-processing pipelines, particularly for models like convolutional neural networks (CNNs) which often expect input images of a fixed size.
- After resizing, the image is converted to an NDArray (a multidimensional array used for deep learning). This NDArray is then transformed into a tensor, preparing it for input into deep learning models.
- The tensor is normalized, adjusting the pixel values based on predefined mean and standard deviation values. These normalization constants (e.g., 0.485f, 0.456f, 0.406f for the mean) are typically derived from the dataset on which a model was trained, ensuring the input image has a similar distribution to what the model expects.

- A new StreamedImage object is created from the processed image, maintaining the original file path. This processed image is then emitted by the collector, which sends it downstream in the Flink pipeline.
- If any exception occurs during processing, particularly an IllegalArgumentException, it is caught, and an error is logged. In such cases, a null value is collected, signaling a failure in processing this particular image. This is where the filter we declared in the datasource seen above comes into play. The nulls will be removed from the stream to avoid errors.
- Finally, the NDManager is closed to release any allocated resources, especially important when working with GPU or CPU resources for deep learning tasks.

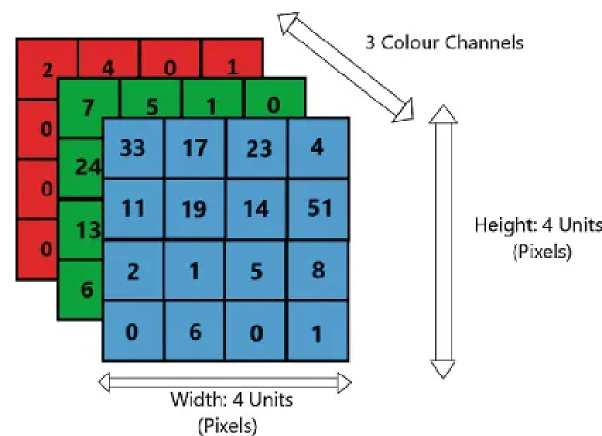

Figure 11: Tensor structure for a 4x4 color image with three channels (RGB)

6.3.5 Model Inference and Classification

The core of the implementation involves applying the deep learning model to the processed images to generate classifications. The 'Classifier' class leverages DJL to load the model and perform batch predictions. The model used in this implementation is a pre-trained ResNet-101 model, known for its high accuracy in image classification tasks:

```
1  public void apply(String s, GlobalWindow globalWindow, Iterable<StreamedImage> iterableImages,
2  Collector<Tuple<Classifications.Classification, StreamedImage>> out) throws Exception {
3      ...
4          try {
5              Classifier classifier = Classifier.getInstance();
6              List<Classifications> classifications = classifier.batchPredict(listOfImages);
7              for (int i = 0; i < classifications.size(); i++) {
8                  Classifications cls = classifications.get(i);
9                  Classifications.Classification cl = cls.best();
10
11                 String ret = cl.getClassName() + ":␣" + cl.getProbability() + "␣" +
12                 listOfStreamedImages.get(i).getFilePath();
13
14                 logger.info(ret);
15                 //Decide class to filter
16                 for (Classifications.Classification c : cls.items()) {
17                     if (c.getClassName().contains("traffic␣light")) {
18                         out.collect(c)
19                     }
20                 }
21             }
22         } catch (ModelException
```

```
23                  | IOException
24                  | TranslateException e) {
25              logger.error("Failed␣predict", e);
26          }
27      }
28  });
```

Listing 5: Model Inference and Classification

Windows in Flink

Windows are another fundamental concept in Flink for processing unbounded data streams. They allow the segmentation of data streams into finite slices based on time, count, or custom triggers. In this implementation, a time window of 30 seconds is used to group incoming images. This approach ensures that the system can handle varying data loads effectively by processing data in manageable chunks. The choice of a 30-second window is justified as it provides a balance between processing delay and batch size, allowing the system to perform inference on a reasonable number of images without incurring excessive latency.

A real-world example where a time window might be used is in social media monitoring. For instance, a 30-second window could be employed to analyze a stream of images posted on a social media platform, detecting and classifying images in near real-time to flag inappropriate content or highlight trending topics.

6.3.6 Post-Processing and Output

The classified images are then processed in a windowed manner to ensure even distribution of data across the subtasks. The results are written to the specified output path using a file sink. This step ensures that the output is managed efficiently and can be easily accessed for further analysis or visualization. In this example, post-processing involves tasks like filtering and saving the results, but in real-world applications, this could include more complex operations such as aggregating results or sending notifications.

Post-Processing Function

The 'PostProcess' function handles additional processing tasks such as generating thumbnails and saving the processed images back to the specified output path. This function demonstrates the flexibility of Flink in handling various post-processing requirements efficiently. Thumbnails are created by resizing the images to 128x128 pixels, which could be useful for applications like creating previews or reducing storage space for quick viewing.

```
1   public void flatMap(String file, Collector<String> collector) throws Exception {
2
3       ...
4
5               Image s3object = ImageFactory.getInstance().fromInputStream(response);
6               response.close();
7
8               Image thumbnailed = s3object.resize(128, 128, false);
9
10              // Save thumbnail to output
11              PutObjectRequest putOb = PutObjectRequest.builder()
12                      .bucket(outputBucket)
13                      .key(outputKey)
```

```
14                  .build();
15
16          ByteArrayOutputStream baosImage = new ByteArrayOutputStream();
17          thumbnailed.save(baosImage, "PNG");
18
19          s3.putObject(putOb, RequestBody.fromBytes(baosImage.toByteArray()));
20          System.out.println("Processed␣image:␣" + imageURI);
21      }
22  }
```

Listing 6: Post-Processing and Output Configuration

6.4 Monitoring with Grafana

During the implementation, we customized a pre-configured Grafana dashboard for monitoring the performance of the Flink cluster. This involved adding specific metrics for latency and throughput to ensure that any issues could be promptly identified and addressed.

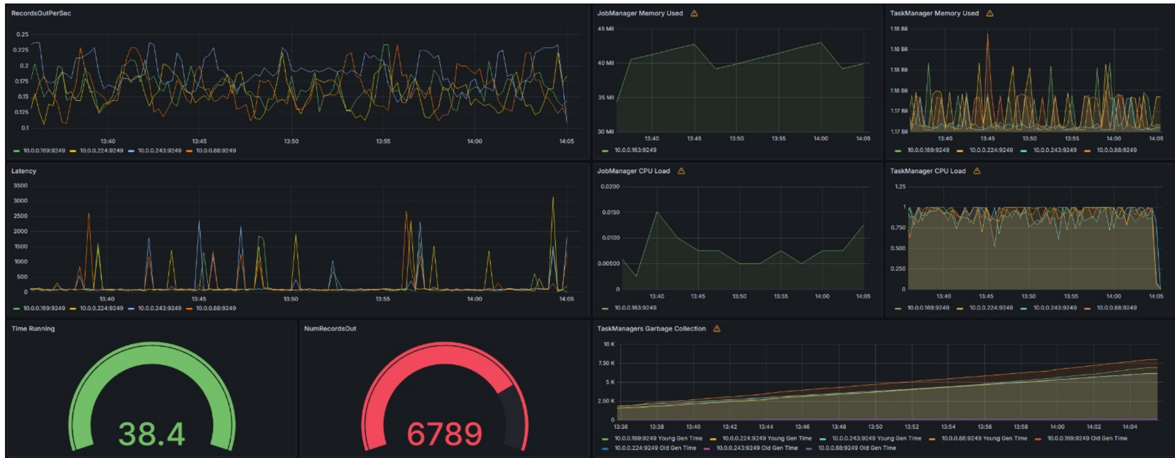

Figure 12: Grafana dashboard for Apache Flink

We modified the original dashboard to add visualizations specifically for latency and throughput (see Fig.12). These modifications allowed us to gain deeper insights into the system's performance, ensuring that any issues could be promptly identified and addressed. Monitoring latency is crucial for maintaining the responsiveness of the system, while tracking throughput ensures that the system can handle the required data load efficiently.

6.5 Summary

By leveraging the capabilities of DJL and Apache Flink, this implementation translated the design into a practical solution for scalable, real-time data processing and deep learning inference. The implementation details presented here reflect the technical challenges overcome and the solutions developed to ensure that the system meets the performance and scalability requirements identified during the design phase.

7 Evaluation

This section presents the evaluation of the streaming inference system using various cluster configurations, datasets, and hardware setups. The evaluation focuses on measuring the system's performance in terms of latency, throughput, and scalability.

7.1 Experimental Setup

The evaluation was conducted using Amazon Web Services (AWS) with Terraform and Ansible for efficient cluster management. Multiple cluster configurations with different task managers were deployed to analyze the scalability and performance of the system. The performance was also compared between GPUs and CPUs. The GPU setup used AWS g5.xlarge instances, while the CPU setup used instances with equivalent resources (m5.xlarge).

Terraform[5] allowed us to define cloud infrastructure using a high-level configuration language. It enables the creation, updating, and version control of infrastructure efficiently and consistently. For this implementation, Terraform scripts were written to provision the AWS resources needed, such as EC2 instances, S3 buckets, and IAM roles. (See Appendix B for Terraform configuration details.)

At the same time, Ansible[3] complements Terraform by managing configurations and deployments. Ansible playbooks were used to install necessary software on the EC2 instances, configure Apache Flink, and deploy the application code. This combination ensures that the infrastructure is both reproducible and easy to manage. (See Appendix C for Ansible configuration details.)

7.2 Cluster Configurations

The cluster is set up within an isolated Virtual Private Cloud (VPC). A VPC allows us to define a virtual network dedicated to our AWS account, providing control over our virtual networking environment, including selection of IP address ranges, creation of subnets, and configuration of route tables and network gateways. By placing each cluster in an isolated VPC, we can run multiple executions in parallel without interference, ensuring a controlled and secure environment for our experiments.

The system was evaluated on clusters with varying numbers of task managers to assess scalability and efficiency. Specifically, the configurations were *1, 2, 4, 8 and 16* Task Managers.

7.2.1 EC2 Instances

We created Amazon Machine Images (AMIs) pre-configured with all the necessary dependencies. An AMI is a template that contains a software configuration (operating system, application server, and applications) which is used to create a virtual machine within the Amazon Elastic Compute Cloud (EC2). By creating custom AMIs, we ensure that each instance in our cluster has all the required software and configurations, reducing setup time and potential errors.

The following table details the specifications of the EC2 instances used in our experiments. Each instance type offers a different balance of CPU, memory, network

performance, and cost. The g5.xlarge instances, equipped with NVIDIA A10G GPUs, were selected for GPU-based task managers to leverage their enhanced computational capabilities for deep learning inference. In contrast, m5.xlarge instances were used for CPU-based task managers to provide a cost-effective solution for CPU-bound tasks. The t3.xlarge instances managed the job manager duties, ensuring efficient resource allocation and task scheduling, while t2.medium instances handled metrics collection, benefiting from their low cost.

Instance	vCPUs	RAM	Cost	Network (Gbps)	GPU
t3.xlarge	4	16 GB	$0.167/h	Up to 5	✗
g5.xlarge	4	16 GB	$1.006/h	Up to 10	NVIDIA A10G
m5.xlarge	4	16 GB	$0.192/h	Up to 10	✗
t2.medium	2	4 GB	$0.0464/h	Low	✗

Table 2: EC2 instance specifications

7.3 Datasets and Preprocessing

To evaluate the performance and scalability of our system, we proposed two datasets. But we decided to use the LIU4K-v2 dataset instead of the more commonly used ImageNet dataset. The primary reason for this choice was the characteristics of the LIU4K-v2 dataset, which includes very high-resolution images. These high-resolution images are more challenging to process and are likely to throttle our processing system, providing a more stringent test of our pipeline's capabilities.

ImageNet (ILSVRC 2012)[14] is a benchmark dataset widely used in the field of image classification. It provides a large-scale dataset with diverse categories and high-quality images, making it ideal for evaluating model performance. The system was tested on this dataset to measure baseline performance and accuracy.

Moving to the **LIU4K-v2**[9] dataset, it is characterized by:

- **High-resolution images**: Resolutions of at least 3K, with most images ranging from 4K to 6K, larger than those in previous datasets, suitable for testing 4K/8K display devices.
- **Large-scale**: The dataset includes 1600 high-resolution training images and 400 high-resolution validation images, providing a more comprehensive and balanced training and evaluation basis.
- **Diverse and complex contents**: It includes varied backgrounds and objects, with diverse and complex low-level signal distribution.
- **High visual quality**: Due to its high-resolution definition and complex content, the images possess high visual quality.

To thoroughly test the pipeline, the number of images was scaled by a factor of 4. This dataset significantly increases the preprocessing load, which helps to saturate the pipeline and assess its impact on throughput.

Table 3 provides a detailed comparison between the ImageNet and LIU4K-v2 datasets. The LIU4K-v2 dataset has significantly larger average image sizes, with resolutions

Dataset	Resolutions	Avg. Size (MB)	Std. Dev. (MB)	Total Size (GB)
ImageNet	Various	0.129872	0.148928	3.170700
LIU4K-v2	3K to 6K	27.320326	11.482681	208.851083

Table 3: Comparison of datasets

ranging from 3K to 6K, compared to the various resolutions in ImageNet. The average size of an LIU4K-v2 image is approximately 27.32 MB, with a standard deviation of 11.48 MB, leading to a total dataset size of around 208.85 GB. In contrast, ImageNet images are much smaller, with an average size of approximately 0.13 MB and a total dataset size of about 3.17 GB. This substantial difference in image size and total dataset volume illustrates the increased computational load and storage requirements for the LIU4K-v2 dataset, making it a more rigorous test for our processing system.

7.4 Performance Metrics

The following metrics were used to evaluate the system:

- **Latency**: The time taken to process each image from ingestion to classification.
- **Throughput**: The number of images processed per second.
- **Scalability**: The change in execution times and cost across different cluster configurations, as Flink does not auto-scale.
- **Cost**: The total cost of running the workflow for each experiment.

7.5 Results and Analysis

7.5.1 CPU Load

To understand the performance characteristics and computational load of our system, we measured the CPU load for both CPU-based and GPU-based configurations across different setups. This experiment aimed to identify how well each type of hardware could handle the workload and to determine the stability of the system under various configurations.

Figure 13 shows the CPU load for different system configurations. It is evident that CPU load tends to be consistently high, regardless of whether the workload is running on a CPU or GPU configuration. The error bars indicate significant variability, suggesting that the workload can be unstable depending on the specific task being executed. Notably, the configuration with 1 Task Manager (TM) for the GPU-based workflow never reached full CPU utilization, probably because we have moved a lot of work from CPU to GPU. As we mentioned earlier when discussing the tf.data[11], separating tasks between CPU and GPU can optimize resource utilization and improve overall system performance.

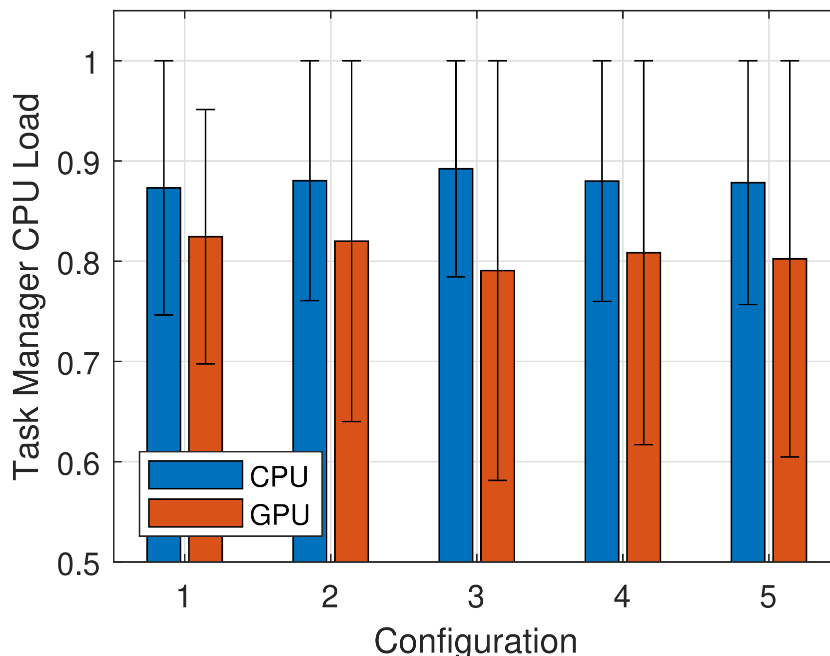

Figure 13: CPU load for different system configurations.

The results show that while both CPU and GPU configurations generate substantial CPU load, GPUs handle the load more efficiently, maintaining a lower average load compared to CPU-only configurations. This indicates that GPUs are better suited for handling high computational demands, providing more stable performance.

7.5.2 Cost as a Function of Time

We plotted the cost associated with processing as a function of time for both CPU and GPU configurations to evaluate the cost-effectiveness of each setup. This exper-

iment was designed to assess how the choice of hardware impacts the overall cost of processing over time.

Figure 14 presents the cost associated with processing as a function of time for CPU and GPU configurations. It is observed that the cost for GPU processing is obviously more expensive compared to CPU, especially for prolonged execution times. This indicates that while GPUs can process faster, their use can become more expensive as execution time increases.

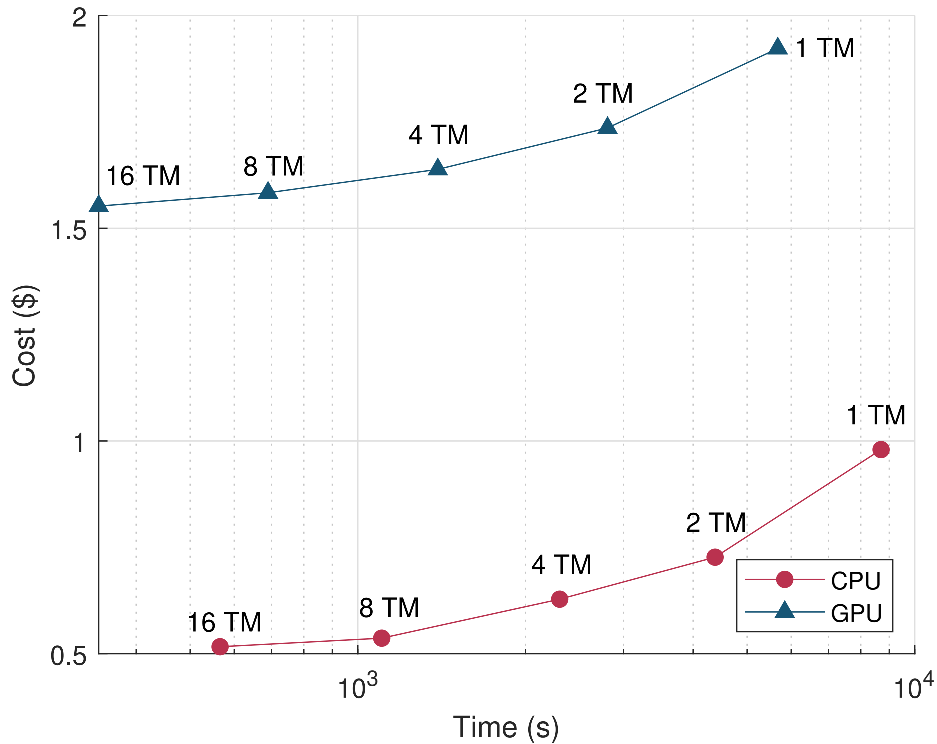

Figure 14: Cost as a function of time for CPU and GPU configurations.

The cost analysis reveals that while GPUs offer performance advantages, they incur higher costs over extended periods. Therefore, for long-running tasks, CPUs may provide a more cost-effective solution despite their lower processing speed.

7.5.3 Normalized Cost-Efficiency

We calculated the normalized performance per dollar (Perf/$) to determine the cost-efficiency of each configuration. This metric helps in comparing how much performance is achieved per unit of cost, providing insights into the economic feasibility of using CPUs versus GPUs.

Performance per dollar (Perf/$) is a metric used to evaluate the cost-efficiency of the system. It is calculated by this equation:

$$\frac{1}{ExecutionTime(s)} \times \frac{1}{Cost(\$)}$$

Allowing for a comparison of how much performance is achieved per unit of cost. A higher Perf/$ indicates better cost-efficiency.

Figure 15 shows the normalized performance per dollar for different configurations. The configurations are normalized with respect to GPU performance. It is observed that CPU implementations are several times more cost-efficient than GPU implementations. Specifically, configurations with a higher number of task managers (TM) tend to offer better relative performance, with the 8-TM configuration standing out as the most cost-effective.

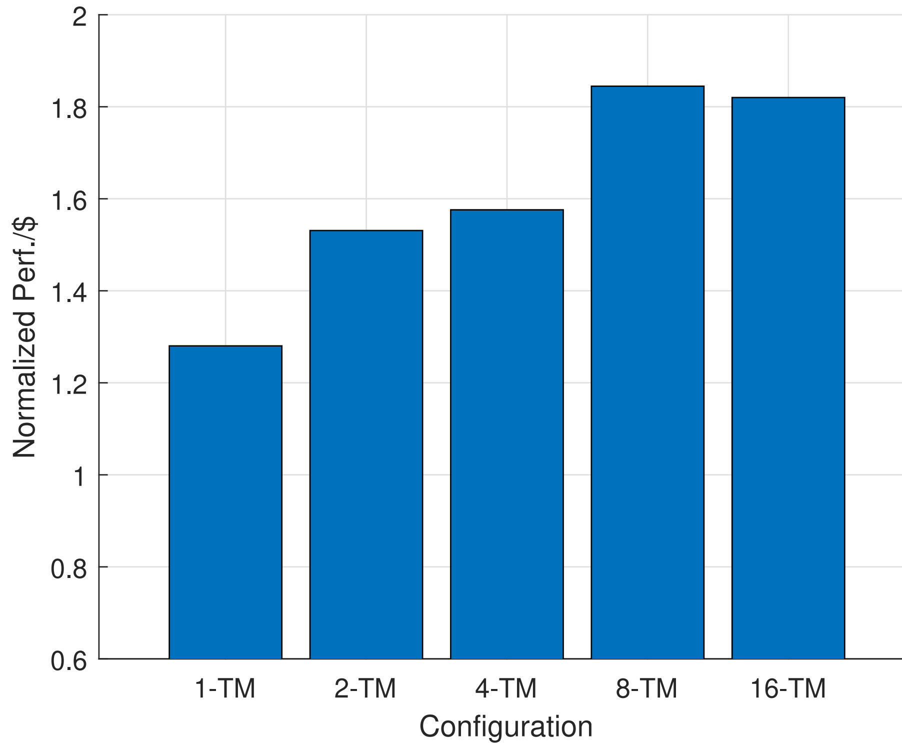

Figure 15: Normalized performance per dollar for different configurations.

The analysis indicates that while GPUs provide superior performance, CPUs offer better cost-efficiency, especially as the number of task managers increases. The 8-TM configuration strikes the best balance between performance and cost.

7.5.4 End-to-End Latency Analysis

We examined the end-to-end system latency for various CPU and GPU configurations to understand the impact of hardware choice on processing speed. This experiment aimed to identify the latency differences between CPU and GPU setups and their implications on overall system performance.

Figure 16 shows the end-to-end latency of the system for various CPU and GPU configurations. It is evident that CPU configurations exhibit significantly higher latency compared to GPU configurations. This is attributed to the inherent differences in processing capabilities between CPUs and GPUs.

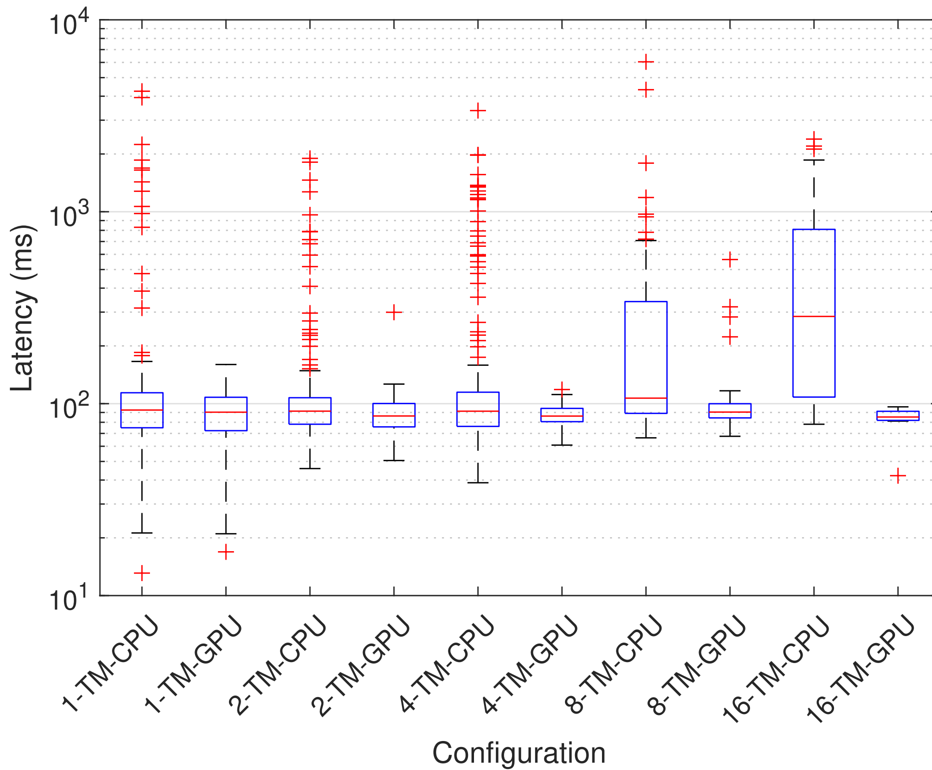

Figure 16: End-to-end system latency for various CPU and GPU configurations.

The CPU configurations present numerous high latency spikes, resulting in many outliers in the plot. These spikes are likely caused by the preprocessing or inference stages, which can saturate the CPU when used exclusively for these tasks. The preprocessing stage involves substantial computational effort, particularly when handling high-resolution images, which can overwhelm the CPU and lead to increased latency.

Furthermore, during the inference stage, the CPU may become a bottleneck due to its limited parallel processing capabilities compared to GPUs. GPUs are designed to handle multiple operations simultaneously, making them more efficient for tasks that involve large-scale data processing and complex computations, such as deep learning inference.

7.5.5 Throughput of the Preprocessing Stage

To further evaluate the performance differences between CPU and GPU configurations, we measured the throughput of the first part of the pipeline, where files are loaded and preprocessed. This metric is crucial as it determines how quickly the system can prepare data for subsequent stages of the workflow, directly impacting the overall system efficiency.

Figure 17 presents the throughput comparison between CPU and GPU configurations across different numbers of Task Managers (TMs). The figure illustrates that the GPU configuration consistently outperforms the CPU in terms of preprocessing speed. Notably, as the number of TMs increases, the throughput also increases for both configurations, but the GPU exhibits a more pronounced improvement.

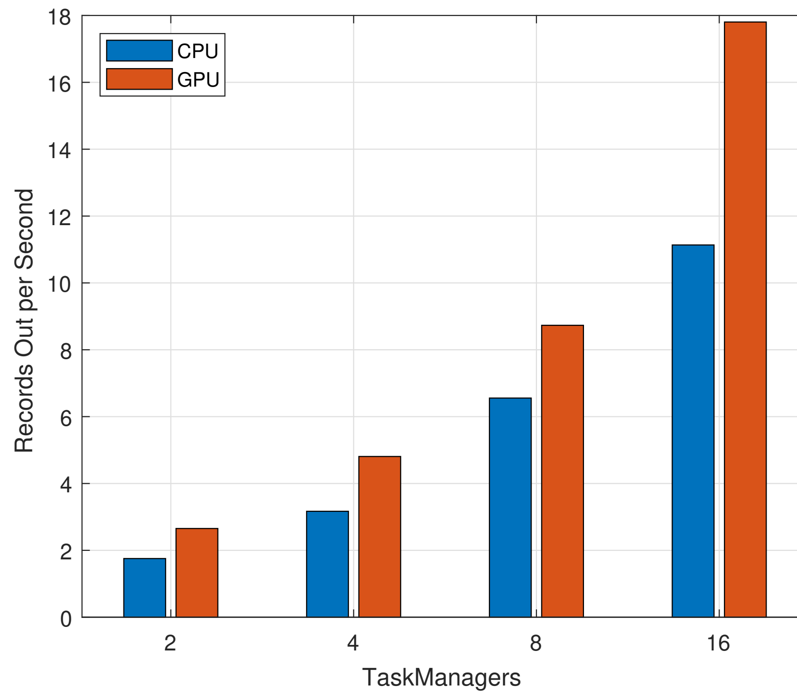

Figure 17: Throughput comparison between CPU and GPU configurations for different numbers of Task Managers (TMs).

The results demonstrate that GPUs can preprocess images faster than CPUs, which is expected given their architecture's suitability for parallel processing tasks. Additionally, the figure shows that with a higher number of TMs, the throughput continues to scale, particularly for the GPU configuration.

This scaling is less significant for CPUs, indicating that GPUs are not only faster but also more efficient in utilizing additional processing resources to boost throughput.

7.6 Discussion

7.6.1 *Strengths and Weaknesses*

The obtained results highlight the advantages and limitations of CPU and GPU-based configurations. Although GPUs offer faster processing and lower latency, their cost can be prohibitive for long-duration tasks. On the other hand, CPUs, while slower, can be more economical for extensive tasks due to their lower operational cost. Configurations with a higher number of task managers (8-TM and 16-TM) show better normalized performance, indicating higher cost-efficiency.

The variability observed in CPU and GPU load, as well as latency, suggests that the specific characteristics of the workload should be carefully considered when choosing the optimal configuration. In scenarios where latency is critical, GPUs may be preferable despite the higher cost. For more uniform and long-duration workloads, CPUs can offer a more cost-effective solution.

7.6.2 *Potential Improvements*

Future improvements could focus on optimizing the system further to reduce the variability in load and latency. This could involve fine-tuning the configurations of task managers or exploring hybrid configurations that leverage both CPU and GPU strengths. However, it is important to note that Flink currently cannot automatically decide which data should be processed by a CPU or GPU TaskManager, making hybrid configurations challenging to implement.

7.6.3 *Scalability and Flexibility*

The current study emphasizes the importance of scalability in real-time data processing systems. One notable observation is the ability of the system to scale horizontally, handling increased loads without significant performance degradation. This scalability is crucial for applications that experience variable traffic patterns, ensuring consistent performance under peak loads. The use of cloud-based infrastructures like AWS EC2 further supports this flexibility, allowing for on-demand resource allocation based on the workload requirements.

However, it is important to note that Apache Flink does not support automatic scaling natively. This is a significant limitation as it requires manual intervention to scale the system based on the workload. The use of Kubernetes with the Flink Kubernetes Operator can mitigate this issue by providing automated scaling capabilities. However, this process requires some knowledge of Kubernetes Horizontal Pod Autoscaleris and it's still relatively slow as it involves checkpointing, stopping the job, scaling the resources, and then relaunching the job from the checkpoint. Future enhancements in this area could focus on improving the speed and efficiency of the scaling process to ensure minimal disruption to ongoing tasks.

Moreover, the flexibility offered by the integration of multiple frameworks, such as DJL for deep learning and Apache Flink for stream processing, highlights the system's capability to adapt to diverse application scenarios. Future research could explore the integration of additional machine learning frameworks and libraries, enhancing the system's applicability to a broader range of tasks, including real-time natural language

processing and predictive analytics.

7.6.4 Environmental and Operational Considerations

In addition to performance and cost, environmental and operational factors play a significant role in determining the optimal configuration for real-time data processing systems. The study reveals that GPU configurations, despite their higher initial costs, can lead to energy savings due to their faster processing capabilities, thereby reducing the overall operational time and energy consumption.

Operationally, the ease of deployment and maintenance is another critical factor. The use of tools like Ansible for automated configuration management and deployment ensures that the system can be quickly set up and maintained, reducing the operational overhead. Furthermore, the ability to monitor system performance in real-time using Grafana and Prometheus provides valuable insights that can be used to optimize resource utilization and identify potential issues before they impact the system's performance.

In conclusion, the choice between CPU and GPU configurations should be guided by the specific requirements of the application, considering factors such as workload characteristics, cost, scalability, and operational efficiency. By leveraging the strengths of both types of hardware and implementing advanced resource management techniques, it is possible to create a highly efficient and adaptable real-time data processing system.

8 Conclusions and Future Work

8.1 Summary of Findings

The experiments demonstrate the system's capability to efficiently handle different workloads using scalable configurations with CPUs and GPUs. The choice between these two types of processors should be based on a careful evaluation of costs, latency, and workload variability. In general, it is recommended to use GPUs for applications where latency and performance are critical, and CPUs for tasks that require continuous and prolonged processing with cost constraints.

The primary objectives of this work were to design an efficient streaming inference workflow using Apache Flink, evaluate the system's performance under various conditions, and compare it with existing solutions. These objectives have been successfully achieved, as evidenced by the system's demonstrated scalability, low latency, and high throughput. The integration of DJL (Deep Java Library) with Apache Flink has proven to be effective in achieving real-time image classification, showcasing the potential for immediate insights and actions based on streaming data.

8.2 Accomplishments

The implemented system has met all the defined functional requirements. It supports continuous data ingestion from Amazon S3, preprocesses images to the required format, performs inference using pre-trained deep learning models, and handles post-processing tasks such as generating thumbnails and saving classification results. The system has achieved high throughput, processing at least 100 images per second, and maintained an inference latency of less than 100 milliseconds per image. Additionally, the system's ability to scale horizontally to handle increasing data loads without significant performance degradation has been validated.

8.3 Future Research Directions

Future research could explore the possibility of not only performing inference but also training models within the streaming framework. This extension would involve adapting current architectures to support incremental learning and continuous model updates, leveraging the real-time data processing capabilities of Apache Flink. Additionally, further optimization could be achieved by fine-tuning the configurations of task managers or exploring hybrid configurations that leverage both CPU and GPU strengths. Implementing dynamic scaling mechanisms could also enhance the system's ability to adapt to varying workloads more efficiently, balancing performance and cost dynamically.

Another area of potential research is the incorporation of more advanced monitoring and visualization tools, beyond the current use of Grafana and Prometheus. These tools could provide deeper insights into system performance and help identify bottlenecks or areas for improvement. Furthermore, exploring the integration of other machine learning frameworks and libraries could extend the system's applicability to a wider range of use cases, including natural language processing and predictive maintenance.

Overall, the results of this work provide a solid foundation for future advancements in the field of streaming machine learning inference, demonstrating the feasibility and

benefits of combining Apache Flink with deep learning models for real-time data processing.

References

[1] Akidau, Tyler ; Bradshaw, Robert ; Chambers, Craig ; Chernyak, Slava ; Fernández-Moctezuma, Rafael J. ; Lax, Reuven ; McVeety, Sam ; Mills, Daniel ; Perry, Frances ; Schmidt, Eric ; Whittle, Sam: The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. In: *Proc. VLDB Endow.* 8 (2015), aug, Nr. 12, S. 1792–1803. – URL `https://doi.org/10.14778/2824032.2824076`. – ISSN 2150-8097

[2] Carbone, Paris ; Katsifodimos, Asterios ; Ewen, Stephan ; Markl, Volker ; Haridi, Seif ; Tzoumas, Kostas: Apache flink: Stream and batch processing in a single engine. In: *The Bulletin of the Technical Committee on Data Engineering* 38 (2015), Nr. 4

[3] Community, Ansible: *Ansible*. 2012. – URL `https://www.ansible.com/`

[4] Google: *Google Cloud Dataflow*. 2015. – URL `https://cloud.google.com/dataflow`

[5] HashiCorp: *Terraform*. 2014. – URL `https://www.terraform.io/`

[6] He, Kaiming ; Zhang, Xiangyu ; Ren, Shaoqing ; Sun, Jian: *Deep Residual Learning for Image Recognition*. 2015

[7] Labs, Grafana: *Grafana*. 2014. – URL `https://grafana.com/`

[8] Library, Deep J.: *Deep Java Library*. 2020. – URL `https://djl.ai/`

[9] Liu, J. ; Liu, D. ; Yang, W. ; Xia, S. ; Zhang, X. ; Dai, Y.: A Comprehensive Benchmark for Single Image Compression Artifacts Reduction. In: *arXiv*, 2019

[10] Marz, Nathan: *Apache Storm*. 2011. – URL `https://storm.apache.org/`

[11] Murray, Derek G. ; Simsa, Jiri ; Klimovic, Ana ; Indyk, Ihor: tf.data: A Machine Learning Data Processing Framework. In: *CoRR* abs/2101.12127 (2021). – URL `https://arxiv.org/abs/2101.12127`

[12] Noghabi, Shadi A. ; Paramasivam, Kartik ; Pan, Yi ; Ramesh, Navina ; Bringhurst, Jon ; Gupta, Indranil ; Campbell, Roy H.: Samza: stateful scalable stream processing at LinkedIn. In: *Proc. VLDB Endow.* 10 (2017), aug, Nr. 12, S. 1634–1645. – URL `https://doi.org/10.14778/3137765.3137770`. – ISSN 2150-8097

[13] PyTorch: *PyTorch*. 2016. – URL `https://pytorch.org/`

[14] Russakovsky, Olga ; Deng, Jia ; Su, Hao ; Krause, Jonathan ; Satheesh, Sanjeev ; Ma, Sean ; Huang, Zhiheng ; Karpathy, Andrej ; Khosla, Aditya ; Bernstein, Michael ; Berg, Alexander C. ; Fei-Fei, Li: ImageNet Large Scale Visual Recognition Challenge. In: *International Journal of Computer Vision (IJCV)* 115 (2015), Nr. 3, S. 211–252

[15] SERVICES, Amazon W.: *Amazon Web Services*. 2007. – URL https://aws.amazon.com/

[16] SONG, Won W. ; UM, Taegeon ; ELNIKETY, Sameh ; JEON, Myeongjae ; CHUN, Byung-Gon: Sponge: Fast Reactive Scaling for Stream Processing with Serverless Frameworks. In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA : USENIX Association, Juli 2023, S. 301–314. – URL https://www.usenix.org/conference/atc23/presentation/song. – ISBN 978-1-939133-35-9

[17] VOLZ, Matt T. Proud J.: *Prometheus*. 2012. – URL https://prometheus.io/

[18] WONG, Wanshun: *What is Residual Connection?* 2021. – URL https://towardsdatascience.com/what-is-residual-connection-efb07cab0d55

[19] ZAHARIA, Matei ; CHOWDHURY, Mosharaf ; DAS, Tathagata ; DAVE, Ankur ; MA, Justin ; MCCAULY, Murphy ; FRANKLIN, Michael J. ; SHENKER, Scott ; STOICA, Ion: Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In: *9th USENIX symposium on networked systems design and implementation (NSDI 12)*, 2012, S. 15–28

Appendix A: Image Classification Flink Job

```
...

/**
 * Embedded Model Inference.
 */
public class EMI {

    private static final Logger logger = LoggerFactory.getLogger(EMI.class);

    public static void main(String[] args) throws Exception {

        final CLI params = CLI.fromArgs(args);
        String jobName = params.getJobName().orElse("Embedded Model Inference");

        final StreamExecutionEnvironment env =
                StreamExecutionEnvironment.getExecutionEnvironment();

        env.enableCheckpointing(60000L, CheckpointingMode.EXACTLY_ONCE);
        env.getConfig().setGlobalJobParameters(params);

        FileSink<String> sink = FileSink.<String>forRowFormat(
                params.getOutput().get(), new SimpleStringEncoder<>())
                .withRollingPolicy(DefaultRollingPolicy.builder()
                        .withMaxPartSize(MemorySize.ofMebiBytes(100))
                        .withRolloverInterval(Duration.ofSeconds(10))
                        .build())
                .withBucketAssigner(new BasePathBucketAssigner<>())
                .build();

        Path[] paths = params.getInputs().get().stream().map(Path::new)
                .toArray(Path[]::new);
        logger.info("Input paths: {}", Arrays.toString(paths));

        final FileSource<StreamedImage> source = FileSource.forBulkFileFormat(
                new CustomStreamFormatAdaper<>(new ImageReaderFormat()), paths)
                .build();

        DataStream<StreamedImage> stream = env.fromSource(source,
                WatermarkStrategy.noWatermarks(), "file-source")
                .flatMap(new PrepareImage())
                .filter(Objects::nonNull);

        logger.info("Output path: {}", params.getOutput().get());

        DataStream<StreamedImage> classifications = stream.filter(Objects::nonNull)
                .keyBy(StreamedImage::getId)
                .countWindow(20)
                .apply(new WindowFunction<StreamedImage, StreamedImage,
                        String, GlobalWindow>() {
                    @Override
                    public void apply(String s, GlobalWindow globalWindow,
                            Iterable<StreamedImage> images,
                            Collector<StreamedImage> out) {
                        List<Image> imageList = new ArrayList<>();
                        List<StreamedImage> streamedImageList = new ArrayList<>();
                        images.forEach(image -> {
                            imageList.add(image.getImage());
                            streamedImageList.add(image);
                        });

                        logger.info("Processing batch of size: {}", imageList.size());

                        try {
                            Classifier classifier = Classifier.getInstance();
                            List<Classifications> classifications = classifier
                                    .predict(imageList);

                            for (int i = 0; i < classifications.size(); i++) {
                                Classifications.Classification cl = classifications
```

```
70                                          .get(i).best();
71                                  String result = cl.getClassName() + ":␣" +
72                                          cl.getProbability() + "␣" +
73                                          streamedImageList.get(i).getFilePath();
74
75                                  logger.info(result);
76
77                                  if (cl.getClassName().contains("traffic␣light")) {
78                                      out.collect(streamedImageList.get(i));
79                                      logger.info(streamedImageList.get(i)
80                                              .getFilePath().toString());
81                                  }
82                              }
83                          } catch (ModelException | IOException | TranslateException e) {
84                              logger.error("Failed␣to␣predict", e);
85                          }
86                      }
87                  });
88
89          classifications.keyBy(StreamedImage::getId)
90                  .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
91                  .process(new s3Window(params.getOutput().get(), params.getPostOutput().get()))
92                  .sinkTo(sink)
93                  .name("file-sink");
94
95          env.execute(params.getJobName().orElse(jobName));
96      }
97
98      public static final class PrepareImage implements FlatMapFunction<StreamedImage, StreamedImage>
99
100         @Override
101         public void flatMap(StreamedImage streamedImage, Collector<StreamedImage> collector)
102             throws Exception {
103             NDManager manager = (CudaUtils.getGpuCount() > 0) ? Device.gpu() : Device.cpu();
104
105             try {
106                 Image image = streamedImage.getImage()
107                         .resize(256, 256, false)
108                         .getSubImage(16, 16, 224, 224);
109
110                 NDArray processedImage = image.toNDArray(manager, Image.Flag.COLOR);
111                 processedImage = NDImageUtils.toTensor(processedImage);
112                 processedImage = NDImageUtils
113                     .normalize(processedImage
114                     , new float[]{0.485f, 0.456f, 0.406f}
115                     , new float[]{0.229f, 0.224f, 0.225f});
116
117                 StreamedImage result = new StreamedImage(
118                     ImageFactory.getInstance().fromNDArray(processedImage)
119                     ,(Path) streamedImage.getFilePath());
120                 collector.collect(result);
121             } catch (IllegalArgumentException e) {
122                 logger.error("Failed␣to␣process␣image", e);
123                 collector.collect(null);
124             } finally {
125                 manager.close();
126             }
127         }
128     }
129
130     public static final class s3Window
131         extends ProcessWindowFunction<StreamedImage, String, String, TimeWindow> {
132
133         private final String output;
134         private final String postOutput;
135
136         public s3Window(String output, String postOutput) {
137             this.output = output;
138             this.postOutput = postOutput;
139         }
140
141         @Override
142         public void process(~) throws Exception {
```

```
143             try (S3Client s3 = S3Client.create()) {
144                 ObjectMapper objectMapper = new ObjectMapper();
145                 String outputBucket = s3.utilities().parseUri(new URI(postOutput)).bucket().get();
146                 String outputKey = s3
147                     .utilities()
148                     .parseUri(new URI(postOutput)).key().get()
149                     + "/filePaths_" + UUID.randomUUID() + ".txt";
150
151                 List<StreamedImage> imageList = new ArrayList<>();
152                 images.forEach(imageList::add);
153
154                 for (StreamedImage streamedImage : imageList) {
155                     Image image = streamedImage.getImage().resize(128, 128, false);
156                     String key = streamedImage.getFilePath().toString() + "_thumb.PNG";
157                     String outputKeyImage = s3
158                         .utilities()
159                         .parseUri(new URI(postOutput))
160                         .key().get() + "/" + key;
161
162                     ByteArrayOutputStream baosImage = new ByteArrayOutputStream();
163                     image.save(baosImage, "PNG");
164
165                     s3.putObject(PutObjectRequest.builder()
166                         .bucket(outputBucket)
167                         .key(outputKeyImage)
168                         .build(), RequestBody.fromBytes(baosImage.toByteArray()));
169
170                     String savedImagePath = outputBucket + "/" + outputKeyImage;
171                     out.collect(savedImagePath + ":␣has␣been␣saved");
172                     logger.info("Image␣saved:␣" + savedImagePath);
173                 }
174             }
175         }
176     }
177 }
```

Appendix B: Terraform Configuration

```
resource "random_id" "hash" {
  byte_length = 8
}

# Create a VPC to launch our instances into
resource "aws_vpc" "flink_vpc" {
  cidr_block           = "10.0.0.0/16"
  enable_dns_hostnames = true

  tags = {
    Name = "Flink-VPC-${random_id.hash.hex}"
  }
}

# Create an internet gateway to give our subnet access to the outside world
resource "aws_internet_gateway" "flink_gateway" {
  vpc_id = aws_vpc.flink_vpc.id
}

# Grant the VPC internet access on its main route table
resource "aws_route" "internet_access" {
  route_table_id         = aws_vpc.flink_vpc.main_route_table_id
  destination_cidr_block = "0.0.0.0/0"
  gateway_id             = aws_internet_gateway.flink_gateway.id
}

# Create a subnet to launch our instances into
resource "aws_subnet" "flink_subnet" {
  vpc_id                  = aws_vpc.flink_vpc.id
  cidr_block              = "10.0.0.0/24"
  availability_zone       = var.availability_zone
  map_public_ip_on_launch = true
}

resource "aws_security_group" "flink_security_group" {
  name   = "terraform-flink-${random_id.hash.hex}"
  vpc_id = aws_vpc.flink_vpc.id

  # SSH access from anywhere
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  # Flink Dashboard access
  ingress {
    from_port   = 8081
    to_port     = 8081
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  # All ports open within the VPC
  ingress {
    from_port   = 0
    to_port     = 65535
    protocol    = "tcp"
    cidr_blocks = ["10.0.0.0/16"]
  }

  # Prometheus access
  ingress {
    from_port   = 9090
    to_port     = 9090
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
```

```
70
71    # Grafana dashboard access
72    ingress {
73      from_port   = 3000
74      to_port     = 3000
75      protocol    = "tcp"
76      cidr_blocks = ["0.0.0.0/0"]
77    }
78
79    # outbound internet access
80    egress {
81      from_port   = 0
82      to_port     = 0
83      protocol    = "-1"
84      cidr_blocks = ["0.0.0.0/0"]
85    }
86
87    tags = {
88      Name = "Flink-Security-Group-${random_id.hash.hex}"
89    }
90  }
91
92  resource "aws_key_pair" "auth" {
93    key_name   = "${var.key_name}-${random_id.hash.hex}"
94    public_key = file(var.public_key_path)
95  }
96
97  resource "aws_instance" "job_manager" {
98    ami                    = var.flink_ami
99    instance_type          = var.instance_types["job_manager"]
100   availability_zone      = var.availability_zone
101   key_name               = aws_key_pair.auth.id
102   subnet_id              = aws_subnet.flink_subnet.id
103   vpc_security_group_ids = [aws_security_group.flink_security_group.id]
104   count                  = var.num_instances["job_manager"]
105
106   tags = {
107     Name = "jm_${count.index}"
108   }
109 }
110
111 resource "aws_instance" "task_manager" {
112   ami                    = var.flink_ami
113   instance_type          = var.instance_types["task_manager"]
114   key_name               = aws_key_pair.auth.id
115   subnet_id              = aws_subnet.flink_subnet.id
116   vpc_security_group_ids = [aws_security_group.flink_security_group.id]
117   availability_zone      = var.availability_zone
118   count                  = var.num_instances["task_manager"]
119
120   ebs_block_device {
121     device_name = "/dev/sda1"
122     volume_size = 50
123   }
124
125   tags = {
126     Name = "tm_${count.index}"
127   }
128 }
129
130 resource "aws_instance" "metrics" {
131   availability_zone      = var.availability_zone
132   ami                    = var.metrics_ami
133   instance_type          = var.instance_types["metrics"]
134   key_name               = aws_key_pair.auth.id
135   subnet_id              = aws_subnet.flink_subnet.id
136   vpc_security_group_ids = [aws_security_group.flink_security_group.id]
137   count                  = var.num_instances["metrics"]
138
139   tags = {
140     Name = "metrics_${count.index}"
141   }
142 }
```

Appendix C: Ansible Configuration

```
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
- name: Flink common config
  hosts: [ "task_manager", "job_manager" ]
  tags: [ "task_manager", "job_manager", "configure", "restart" ]
  connection: ssh
  become: true
  tasks:
    - template:
        src: templates/config.yaml.j2
        dest: "{{ flink_dir }}/conf/config.yaml"
      name: Copy Flink config

- name: Config Prometheus
  hosts: metrics
  tags: [ "metrics" ]
  connection: ssh
  become: true
  tasks:
    - template:
        src: templates/prometheus.yml.j2
        dest: /opt/prometheus/prometheus.yml
      name: "Copy Prometheus template"

    - command: "sudo systemctl restart prometheus"
      name: "Restart Prometheus"

- name: Install Job Manager systemd unit
  hosts: job_manager
  tags: [ "job_manager", "restart" ]
  become: True
  connection: ssh
  tasks:
    - template:
        src: templates/jobmanager.service.j2
        dest: /etc/systemd/system/jobmanager.service
        mode: 0644
      name: Copy service template

- name: Launch Flink JobManager
  become: True
  connection: ssh
  hosts: job_manager
  tags: [ "job_manager" ]
  tasks:
    - systemd:
        name: jobmanager
        state: started
        daemon_reload: True
        enabled: '{{ enable_services }}'

- name: Install Task Manager systemd unit
  become: True
  connection: ssh
  hosts: task_manager
  tags: [ "task_manager", "restart" ]
  tasks:
```

```
70      - template:
71          src: templates/taskmanager.service.j2
72          dest: /etc/systemd/system/taskmanager.service
73          mode: 0644
74        name: Copy service template
75
76  - name: Launch Flink Task Manager
77    become: True
78    connection: ssh
79    hosts: task_manager
80    tags: [ "task_manager" ]
81    tasks:
82      - systemd:
83          name: taskmanager
84          state: started
85          daemon_reload: True
86          enabled: '{{␣enable_services␣}}'
87
88  - name: Restart Job Manager systemd unit
89    become: True
90    connection: ssh
91    hosts: job_manager
92    tags: [ "never", "restart" ]
93    tasks:
94      - systemd:
95          name: jobmanager
96          state: restarted
97          daemon_reload: True
98          enabled: '{{␣enable_services␣}}'
99        name: Restart Job Manager
100
101 - name: Restart Task Manager systemd unit
102   become: True
103   connection: ssh
104   hosts: task_manager
105   tags: [ "never", "restart" ]
106   tasks:
107     - systemd:
108         name: taskmanager
109         state: restarted
110         daemon_reload: True
111         enabled: '{{␣enable_services␣}}'
112       name: Restart Task Manager
```

UNIVERSITAT ROVIRA i VIRGILI