Apache Flink

Daniel Hinojosa

Unlocking the Power of Real-Time Data Processing


NFJS

In this Presentation

- Flink
- Purpose
- Components
- Types of Processing
- Developing a Flink Application
- Deploying a Flink Application
- Sources and Sinks
- Fault Tolerance
- Partitioning
- Stateful Operations
- Time and Windowing
- Why the Confluent acquisition?

Flink

Flink

- **Apache Flink** is a streaming data processing framework designed for **scalable**, **high-throughput**, and **low-latency** data processing.
- It supports both **real-time stream processing** and **batch processing** in a unified architecture.

Features of Flink

- **Stream-first Architecture:** Flink processes data as continuous streams, with batch processing treated as a special case of streaming.
- **Event Time Processing:** Flink natively handles event-time semantics, enabling precise time-based operations like windowing and joins.
- **Stateful Stream Processing:** Offers robust support for maintaining application state across events, with exactly-once consistency guarantees.
- **Fault Tolerance:** Utilizes distributed snapshots and checkpoints to ensure data integrity and application recovery.
- **Scalability:** Flink scales horizontally to handle massive datasets across clusters of commodity hardware.
- **Flexible APIs:** Provides APIs in Java, Scala, and Python, including DataStream API, Table API, and SQL for diverse use cases.
- **Deployment Flexibility:** Can run on standalone clusters, Kubernetes, YARN, Mesos, and integrates with popular cloud platforms.

Flink Use Cases

Use Cases of Flink

- **Event Driven Applications -** Flink's common source is Kafka, but there are other sources. When an event arrives we can process it, transform it, and load it onto a Kafka topic, or other sinks like databases

- **Analytics and Processing -** Flink's common source is Kafka, but there are other sources. When an event arrives we can process it, transform it, and load it onto a Kafka topic, or other sinks like databases

- **Fraud Detection -** Flink's windowing makes it suitable to capture suspicious activity

- **Machine Learning and Predictive Analysis -** Flink provides a common set of estimators that can be used to analyze data, like KNN, LogisticRegression, and more

Type of Processing

Batching

Running Historic Data

Batching

- **Definition:** Finite, bounded datasets processed in discrete chunks.
- **Use Cases:** Historical data analysis, periodic ETL jobs, data warehousing, offline reporting.
- **Strengths:** Optimized for throughput, efficient resource utilization for static datasets.

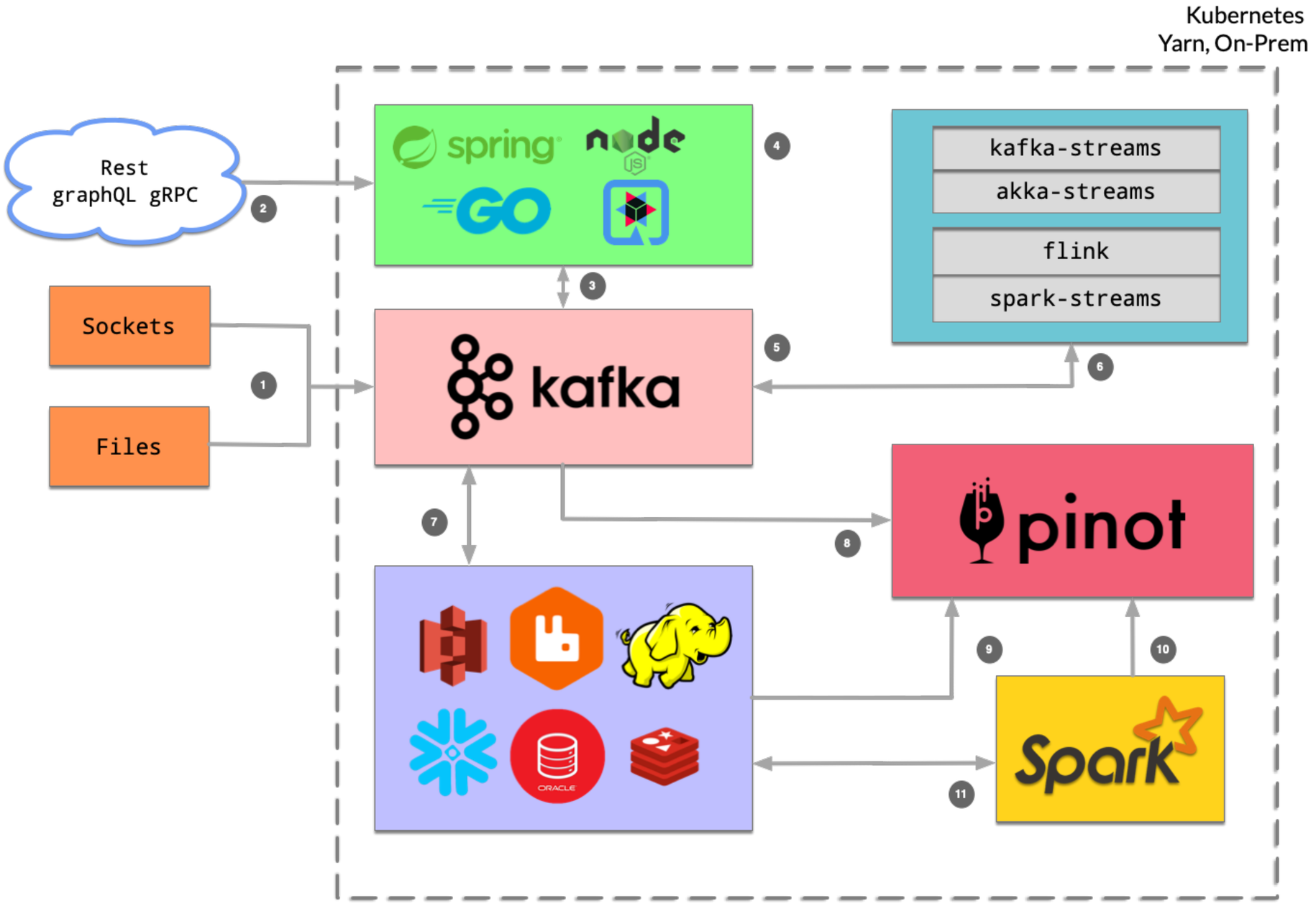
Streaming

Real Time Data Analysis

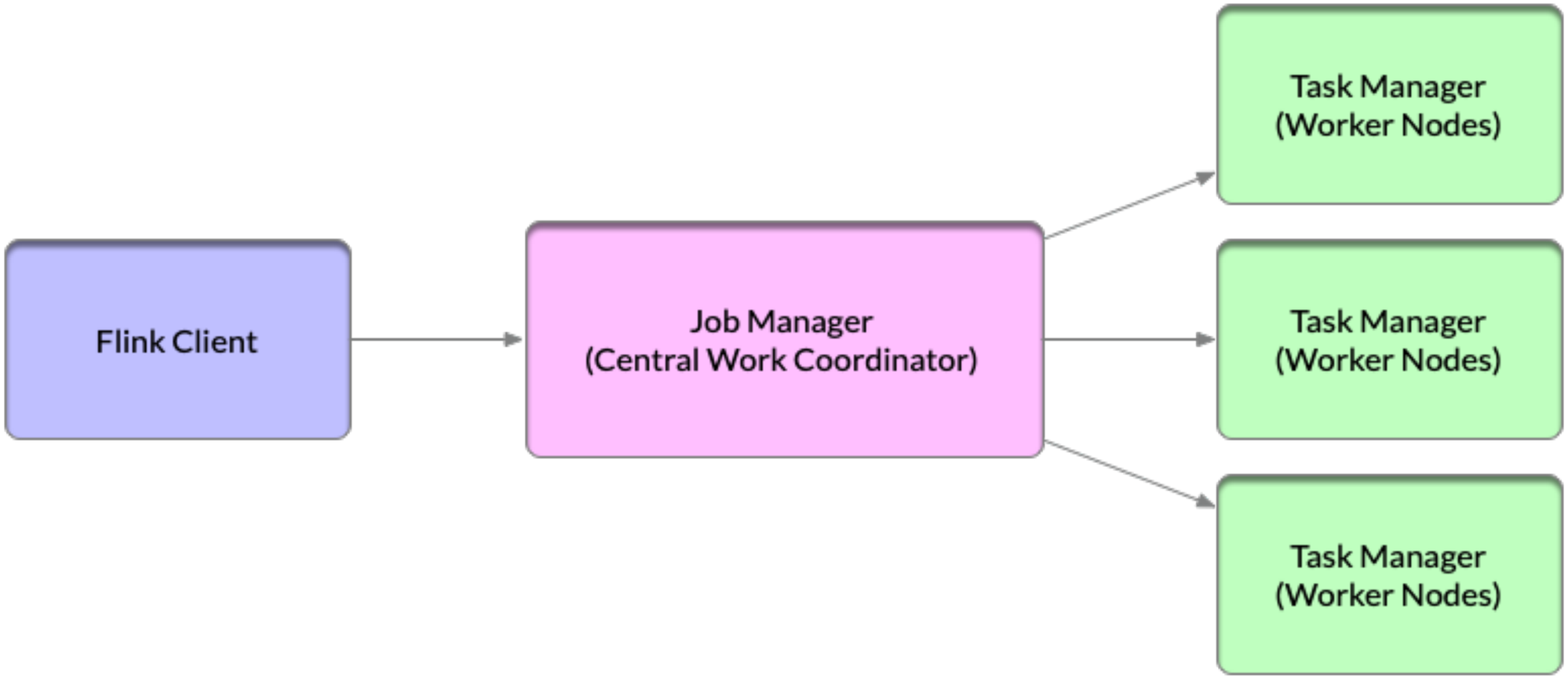
Streaming

- **Definition:** Continuous, unbounded flow of data processed in real-time as it arrives.
- **Use Cases:** Real-time analytics, event-driven applications, fraud detection, monitoring systems.
- **Strengths:** Low-latency processing, real-time insights, dynamic handling of late-arriving data.

Big Data Architecture


Kubernetes
Yarn, On-Prem
Rest
graphQL gRPC
Sockets
Files
spring
node
GO
kafka
kafka-streams
akka-streams
flink
spark-streams
pinot
ORACLE
Spark
1
2
3
4
5
6
7
8
9
10
11

Components of Flink


Flink Client
Job Manager
(Central Work Coordinator)
Task Manager
(Worker Nodes)
Task Manager
(Worker Nodes)
Task Manager
(Worker Nodes)

Job Manager

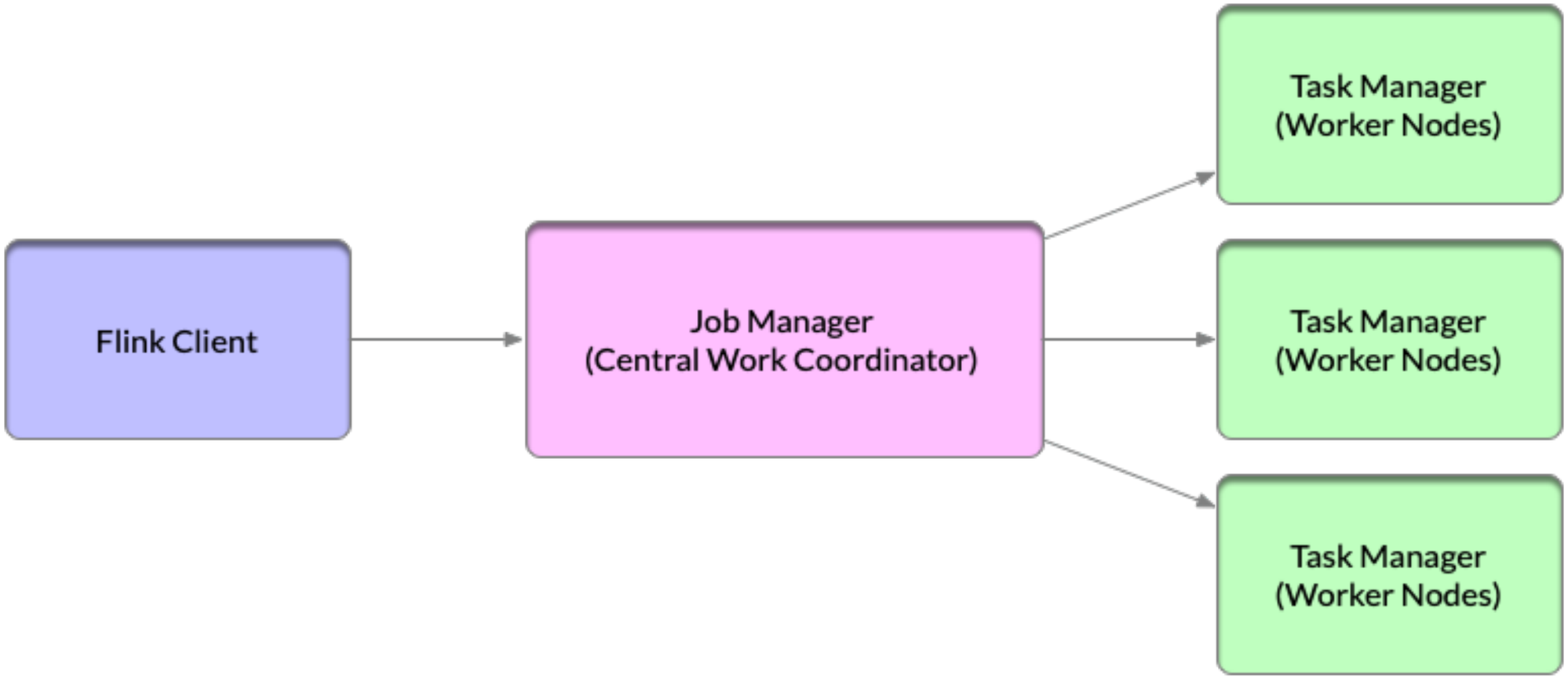
- Acts as the brain of the Flink cluster, responsible for *coordinating and managing jobs*.
- Responsibilities:
  - **Job Scheduling:** Assigns tasks to Task Managers.
  - **Checkpointing & Fault Tolerance:** Ensures consistency in case of failures using checkpoints and savepoints.
  - **Resource Management:** Allocates resources dynamically across the cluster.
  - **Failure Recovery:** Monitors Task Managers and restarts failed tasks.
  - **Communication with Clients:** Accepts job submissions from users and provides status updates.

Task Manager

- Executes the actual data processing tasks assigned by the Job Manager.
- Responsibilities:
  - **Task Execution:** Runs tasks in parallel through slots.
  - **State Management:** Manages local state and interacts with Job Manager for checkpointing.
  - **Data Exchange:** Handles network communication between tasks (e.g., shuffling data).
  - **Resource Allocation:** Each Task Manager has multiple task slots, defining how many parallel tasks it can run.

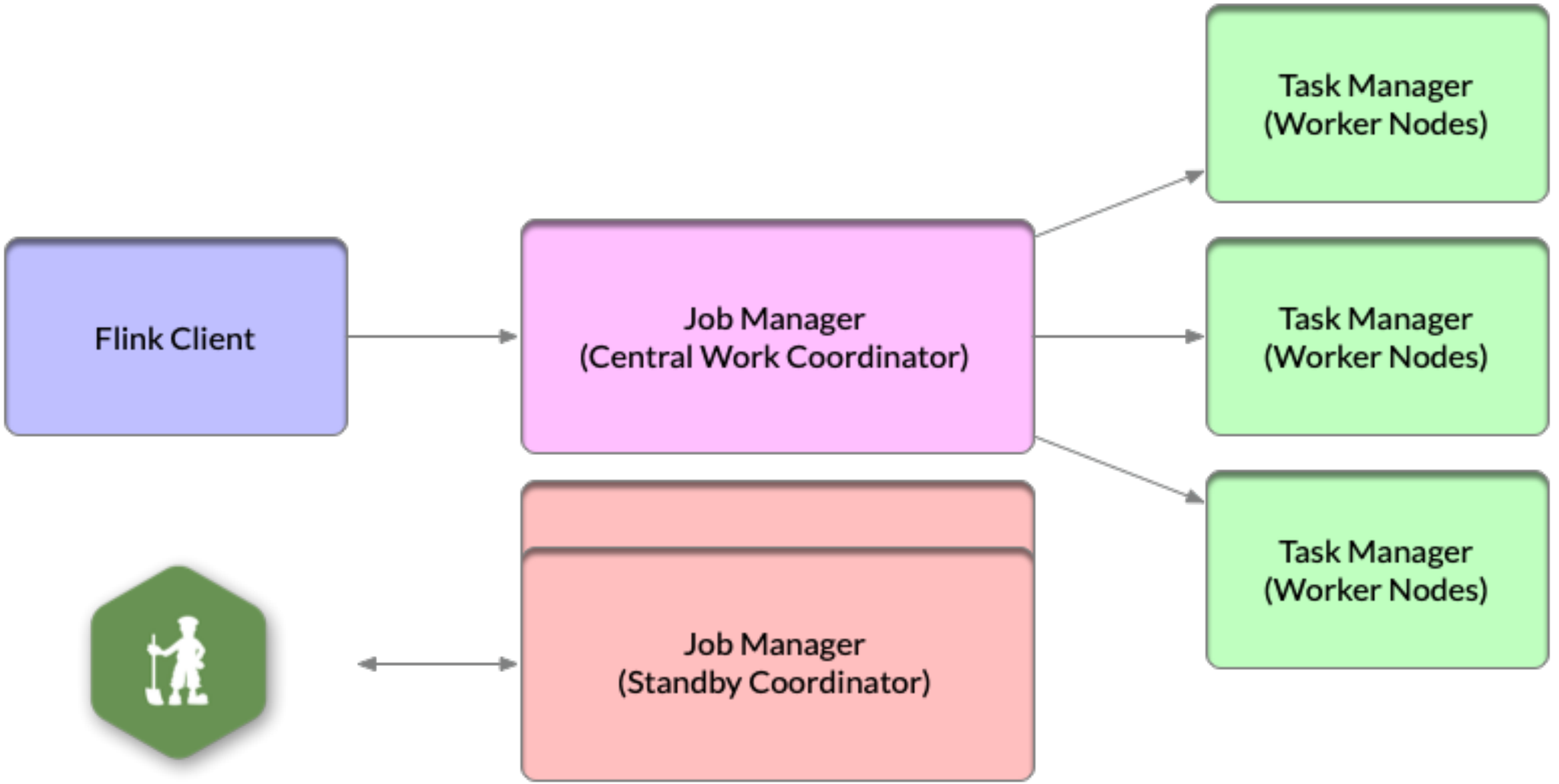
Client

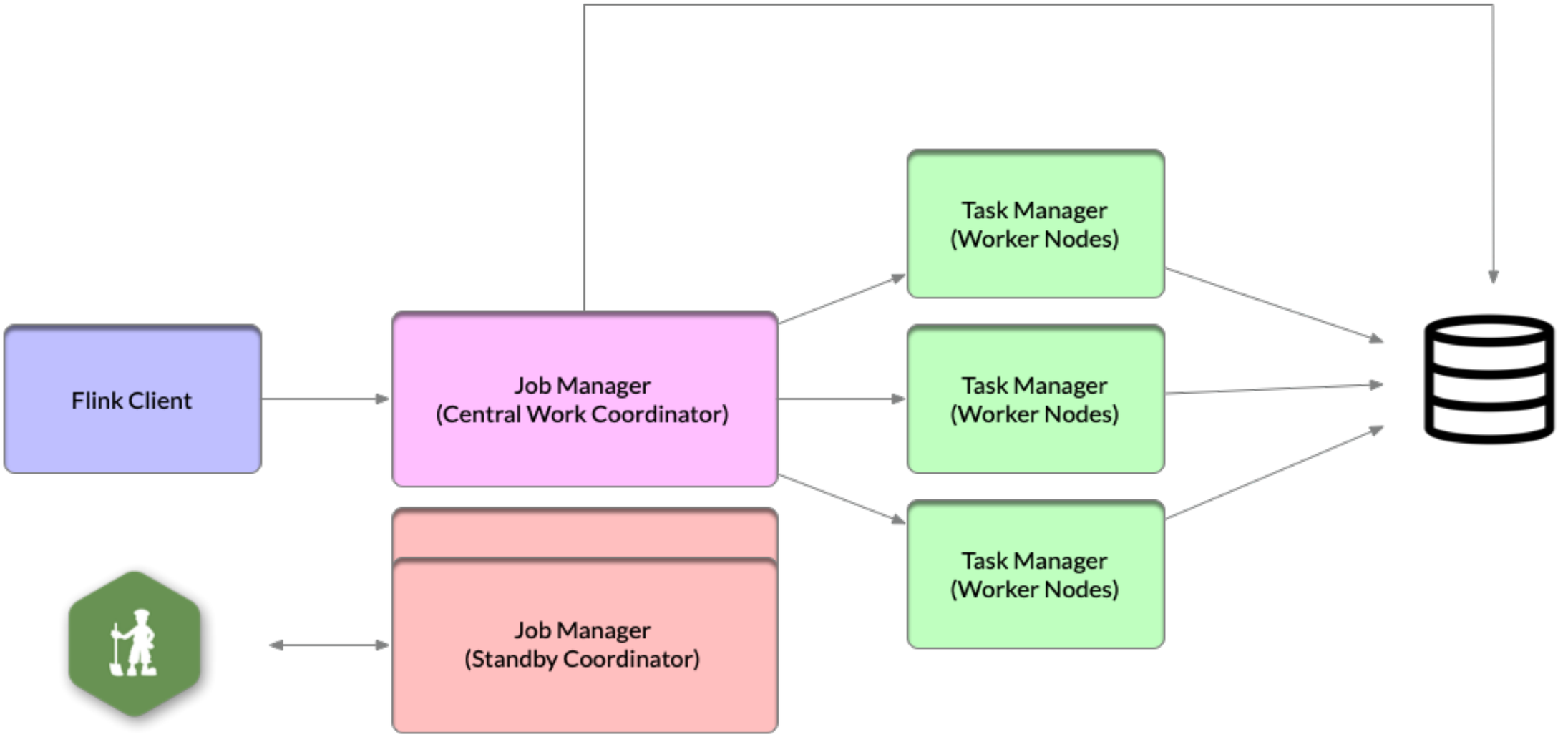
- The entry point for users to submit jobs to the Flink cluster.
- Responsibilities:
  - **Job Submission:** Sends the job's dataflow graph to the Job Manager.
  - **Job Monitoring:** Allows users to track the job status and fetch logs/results.
  - **Deployment Options:**
    - Command-line interface (`flink run job.jar`).
    - REST API for programmatic submissions.
    - Flink UI
    - Integrated into IDEs or other applications.


Flink Client
Job Manager
(Central Work Coordinator)
Task Manager
(Worker Nodes)
Task Manager
(Worker Nodes)
Task Manager
(Worker Nodes)

Job Manager Redundancy

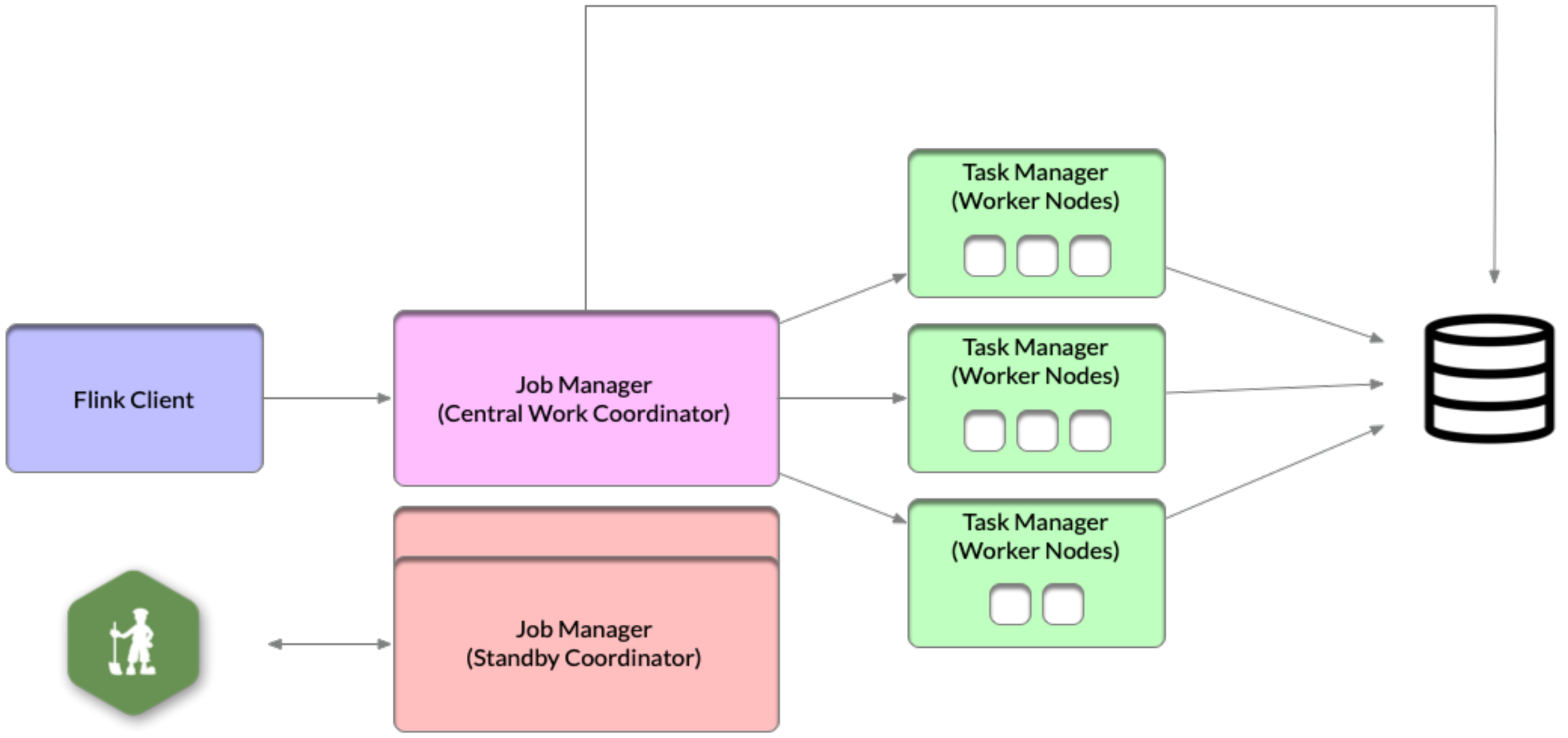
- Job Manager can be made highly available by eliminating any single point of failure
- Flink supports multiple standby Job Managers: if the active one fails, another takes over without disrupting the running jobs
- Flink uses Zookeeper to manage leader election and metadata storage.
- One active JobManager handles the job execution, and other standby will wait for failover.


Flink Client
Job Manager
(Central Work Coordinator)
Task Manager
(Worker Nodes)
Task Manager
(Worker Nodes)
Task Manager
(Worker Nodes)
Job Manager
(Standby Coordinator)


Flink Client
Job Manager
(Central Work Coordinator)
Task Manager
(Worker Nodes)
Task Manager
(Worker Nodes)
Task Manager
(Worker Nodes)
Job Manager
(Standby Coordinator)

How Jobs are Run

- The **Job Manager** receives the submitted job and translates it into a **Job Graph**, which is a high-level representation of the data flow including sources, transformations, and sinks
- Job Graph is converted into an **Execution Graph**. An Execution Graph break the job into tasks and defines parallelism. *Each task is split into subtasks based on the parallelism*
- A **Job Manager** requests **slots** from available **TaskManagers**. Every TaskManager has multiple slots to run the subtasks
- A Job Manager uses the **Slot Manager** to allocate resources based on parallelism settings, data locality, and load balancing.
- Each subtask represents a partition of the data


Flink Client
Job Manager
(Central Work Coordinator)
Job Manager
(Standby Coordinator)
Task Manager
(Worker Nodes)
Task Manager
(Worker Nodes)
Task Manager
(Worker Nodes)

Flink API

DataStream API

Streaming API for Flink

What is the DataStream API?

- The DataStream API is Flink’s core API for processing unbounded and bounded streams of data.
- It supports event-time, processing-time, and ingestion-time semantics.
- Designed for real-time, low-latency data processing with flexible, fine-grained control over transformations.

When to use the DataStream API?

- **Real-Time Processing:** When you need to process data as it arrives (e.g., live analytics, fraud detection).
- **Complex Event Processing:** When simple SQL or Table operations aren't sufficient.
- **Stateful Stream Processing:** To maintain state across events (e.g., session tracking, windowing).
- **Fine-Grained Control:** When you need more control than declarative APIs like SQL provide.

Recipe for Building a Stream API

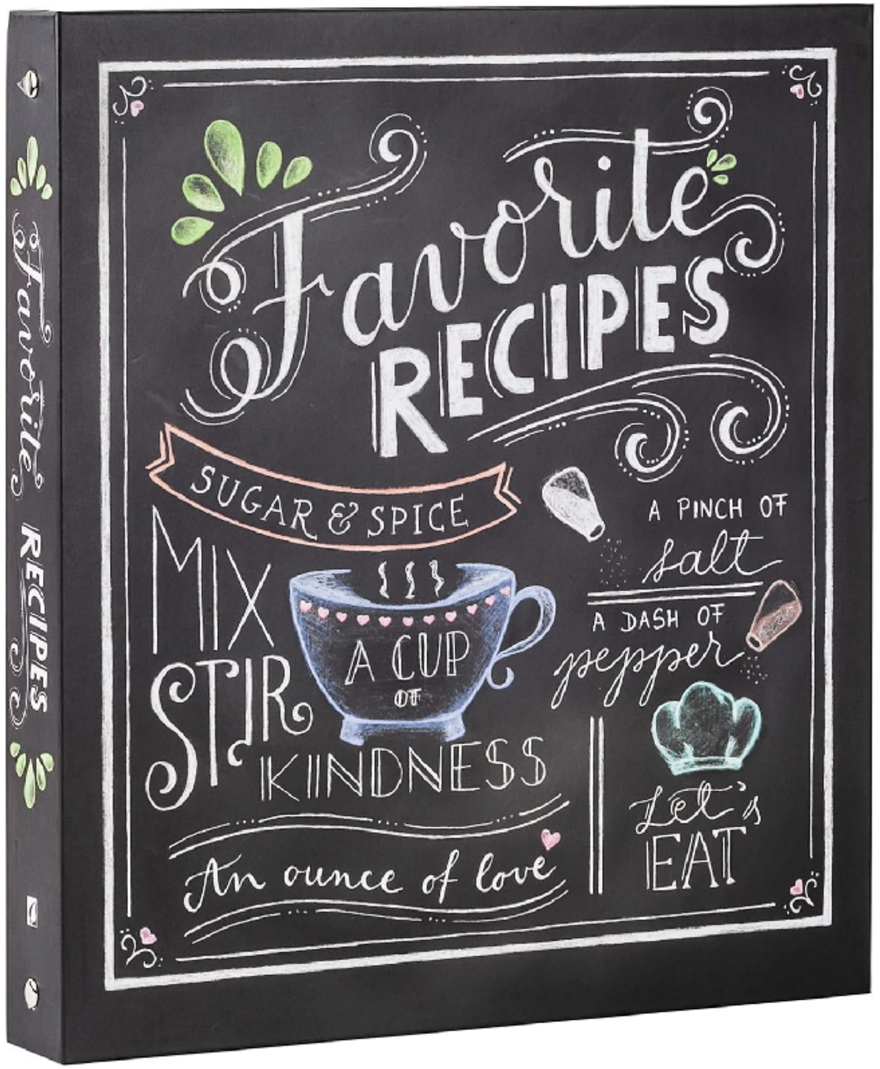
1. Create a `StreamExecutionEnvironment` as an entry point

2. Using the environment decide where you are getting the data from, there are many choices. This is a *source*

3. Apply transformations: `map`, `filter`, `keyBy`, etc.

4. Create *a sink*, where the data will end up. This can be the console, Kafka, a database, or more.

5. Execute the stream with a name



Lab: Running and Submitting a Basic Stream

- Let’s see the anatomy of a basic stream, hooked up to Kafka and sending to Kafka
- We will use the Stream API

Table API & SQL

Using Table Structure and SQL

Table API and SQL

- Table API and SQL in Flink provide a high-level, declarative way to process both batch and streaming data.
- Table API can simplify complex transformations with SQL-like syntax
- Uses Flink’s Catalyst Optimizer to optimize complex queries in the most efficient way possible
- Integrates with Connectors like Kafka, JDBC, Hive, HDFS, etc.
- Support Functional-Style Programming using Java & Scala APIs or you can use Flink SQL

Streaming SQL Example in Flink

```
INSERT INTO results
SELECT color, COUNT(*)
FROM events
WHERE color <> orange
GROUP BY color;
```

Lab: Table API and Flink SQL

- Let’s view what the table API and Flink SQL looks like

Building Your Application

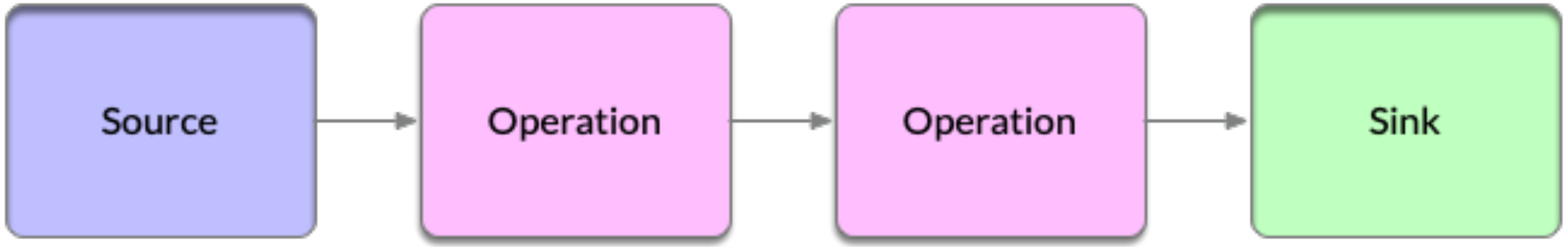
Building an Application

- A **Flink Job** is a standard Java job, with a `main` method.
- Depending on your use case, you may need to package your Flink application in different ways before it gets deployed to a Flink environment.
- If you want to create a JAR for a Flink Job and use only Flink dependencies without any third-party dependencies (i.e. using the *filesystem connector* with JSON format), you do not need to create an uber/fat JAR or shade any dependencies.
- If you want to create a JAR for a Flink Job and use external dependencies not built into the Flink distribution, you can either add them to the classpath of the distribution or shade them into your uber/fat application JAR.

Sources and Sinks

Sources and Sinks

- **Sources** - Where data enters the Flink pipeline, how flink connects to external systems to ingest data.
  - Sources can either be *bounded* (finite datasets) or *unbounded* (infinite datasets)
- **Sinks** - Where data is written or output after computation.
  - Sinks define how Flink delivers results to external systems like databases, message queues, or files.



Common Sources & Sinks

What are some sources and sinks available?

Kafka

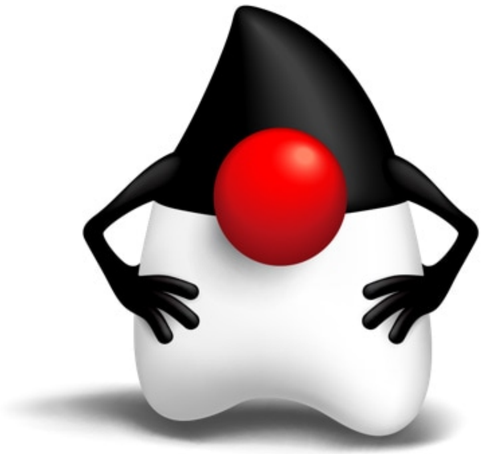
- As a source ingest data from a topic
- Perform real time analysis using flink
- If your destination requires results posted to a topic then use a Kafka sink
- This is typically one of the most common sources used in streaming since Kafka represents the data backplane

Sockets and Files

- Flink can ingest any information from a socket or as a file source.
- The file source can either be distributed (HDFS, S3, or GCS) or local.
- With files, can either be done as batch, or streaming where it monitors for new files

JDBC

- Reads data from relational databases using JDBC connectors.
- As long as there is a JDBC connector you can read that into Flink
- From Flink you can write to the database sink.
- May have idempotent upserts and exactly-once semantics



Elastic Search and Open Search

- Flink integrates with both Elasticsearch and OpenSearch as sinks.
- Commonly used for searchable storage, dashboarding, and analytics.
- Supported via Flink’s Elasticsearch connector (also works with OpenSearch).

Architectural Design with Flink

CQRS Pattern

Command Query Responsibility Segregation

Command Query Responsibility Segregation

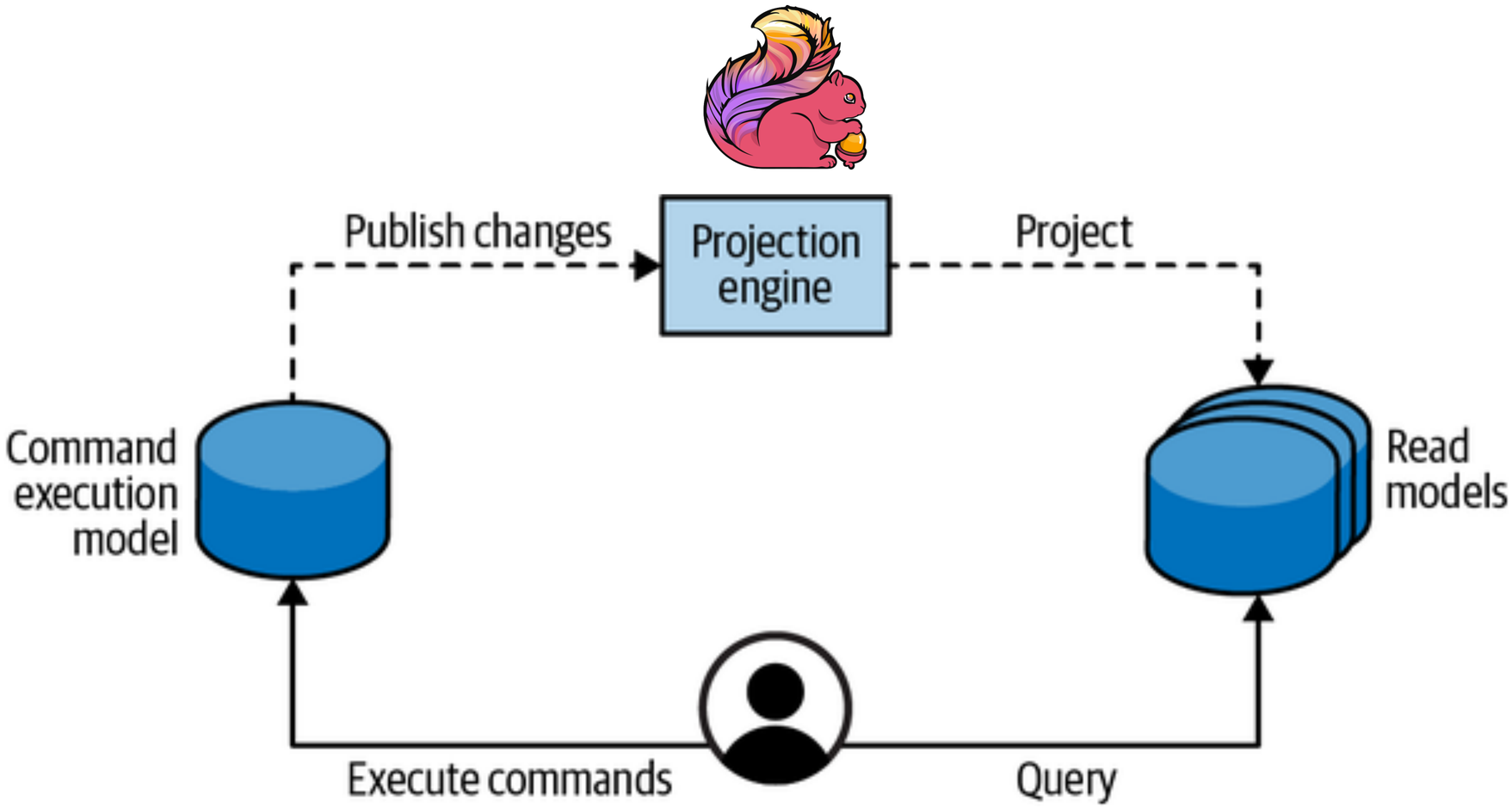
- Provides the possibility of materializing projected models into physical databases that can be used for flexible querying options
  - Pattern segregates the responsibilities of the system's models.
  - There are two types of models:
    - **The command execution model**
    - **The read models**

Command Execution Model

- CQRS devotes a single model to executing operations that modify the system's state (system commands).
- This model is used to implement the business logic, validate rules, and enforce invariants.
- The command execution model is also the only model representing strongly consistent data—the system's source of truth.
- It should be possible to read the strongly consistent state of a business entity and have optimistic concurrency support when updating it.

Read Models

- The system can define as many models as needed to present data to users or supply information to other systems.
- A read model is a pre-cached projection.
- It can reside in a durable database, flat file, or in-memory cache. Proper implementation of CQRS allows for wiping out all data of a projection and regenerating it from scratch.
- This also enables extending the system with additional projections in the future—models that couldn't have been foreseen originally.
- Read models are read-only.
- None of the system's operations can directly modify the read models' data.



Learning Domain Driven Design Vlad Khononov Published by O'Reilly Media, Inc.

Lab: CQRS

- Let’s implement an outbox with Flink SQL, Kafka

Outbox Pattern

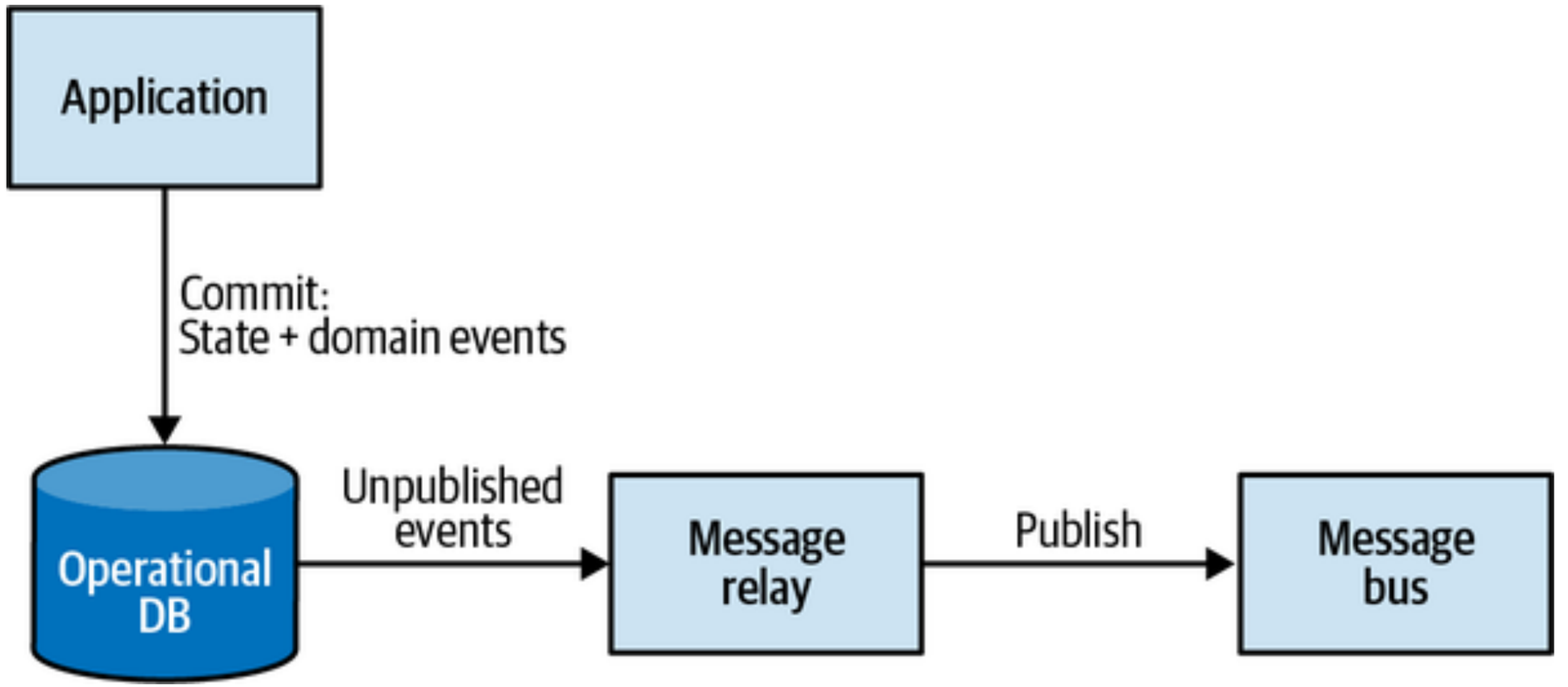
Dealing with Transaction issues across different systems

Sending to Database and Messaging

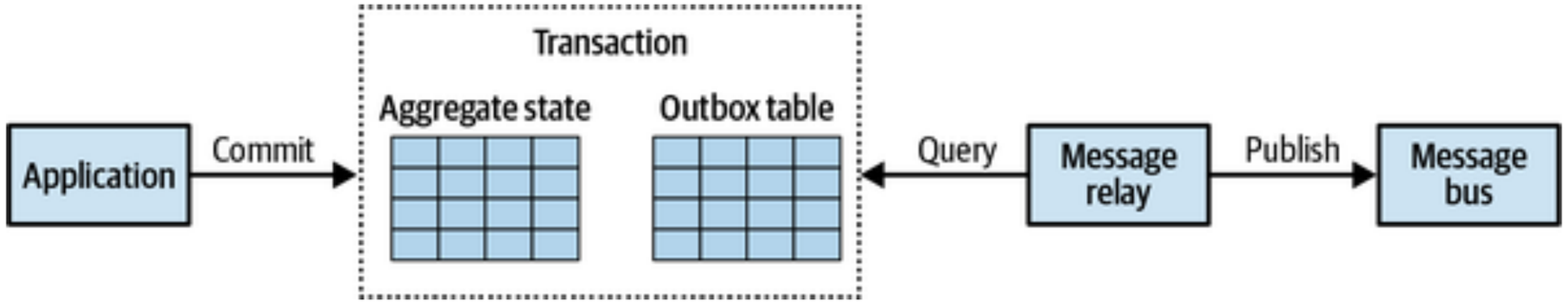
```
public class Campaign {
    private EventRepository repository; // In the aggregate?
    private MessageBus messageBus; // In the aggregate?
    private List<Location> locations;

    public void deactivate(string reason) {
        for (Location loc : locations) {
            loc.deactivate();
        }
        isActive = false;
        var newEvent = new CampaignDeactivated(id, reason);
        events.append(newEvent);   //Where will this fail? Here?
        messageBus.publish(newEvent); //Or Here?
    }
}
```

Issues with the Code

- First, the event will be dispatched before the aggregate's new state is committed to the database.
- A subscriber may receive the notification that the campaign was deactivated, but it would contradict the campaign's state.
- Second, what if the database transaction fails to commit because of a race condition, subsequent aggregate logic rendering the operation invalid, or simply a technical issue in the database?
- Even though the database transaction is rolled back, the event is already published and pushed to subscribers, and there is no way to retract it.



Learning Domain Driven Design Vlad Khononov Published by O'Reilly Media, Inc.



Learning Domain Driven Design Vlad Khononov Published by O'Reilly Media, Inc.

Lab: Transactional Outbox

- Let’s implement an outbox with Flink SQL, Kafka

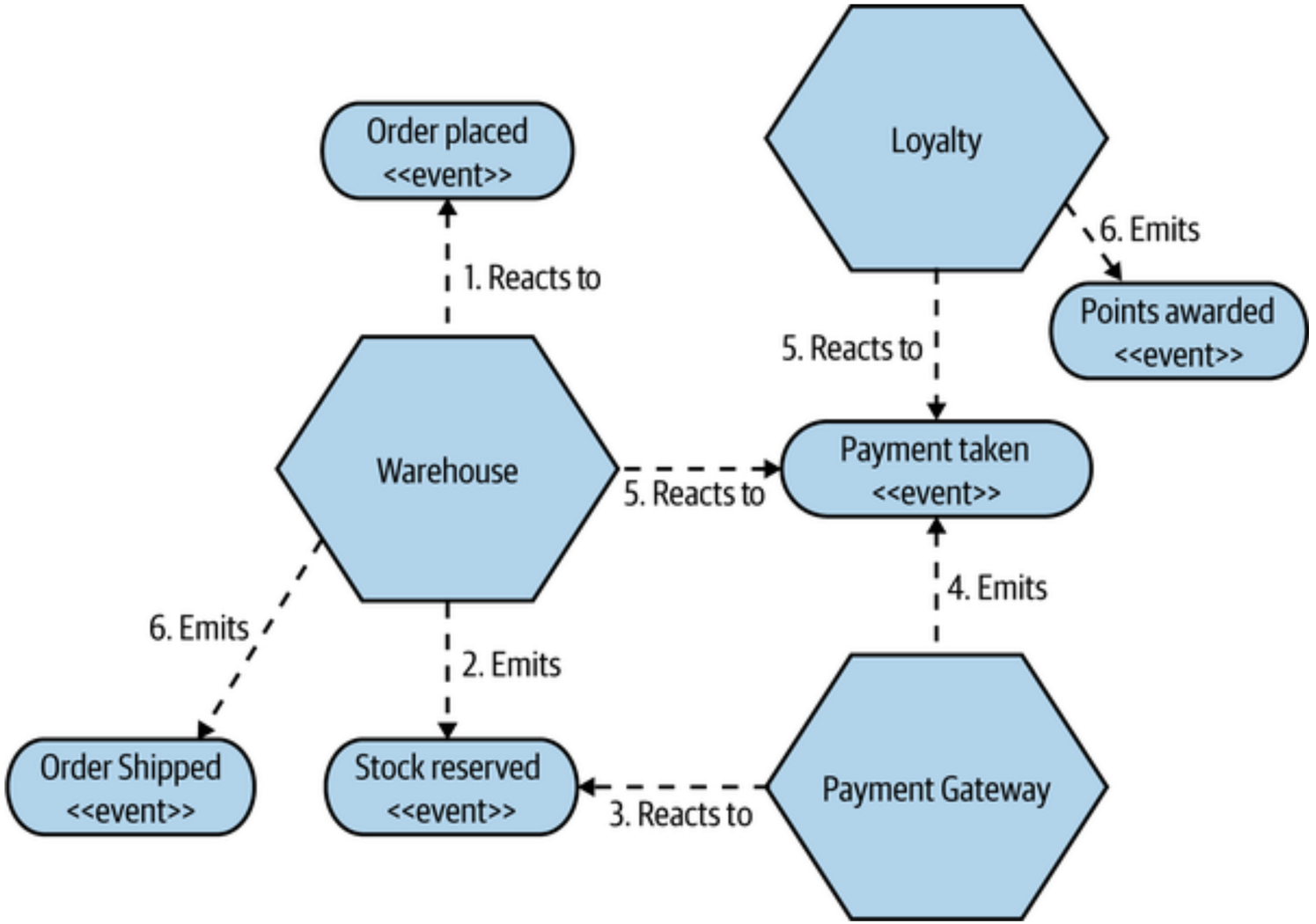
Saga Pattern

Reacting to Events as they happen

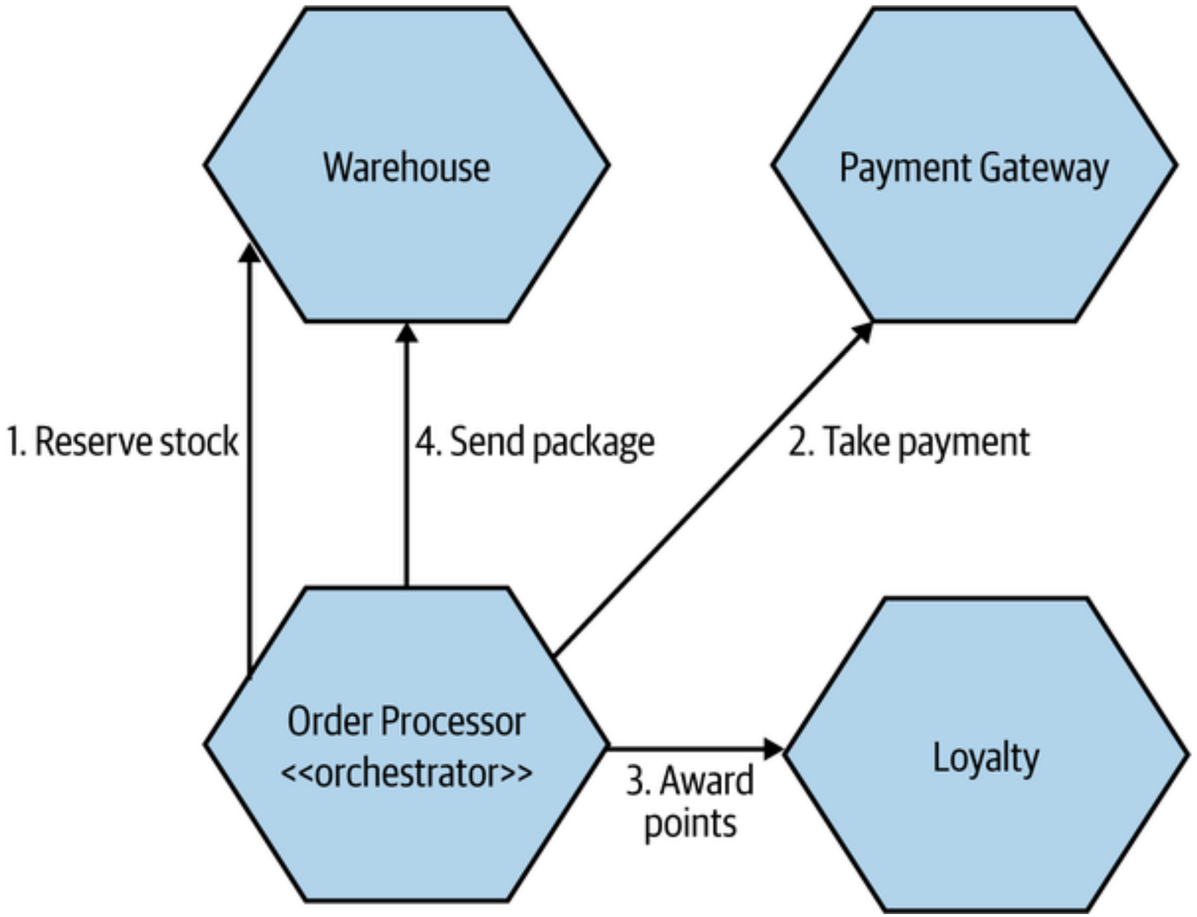
Saga Pattern

- A saga is a long-running business process.
- It’s long-running not necessarily in terms of time, as sagas can run from seconds to years, but rather in terms of transactions: a business process that spans multiple transactions
- Transactions can be handled not only by aggregates but by any component emitting domain events and responding to commands
- Saga listens to the events emitted by the relevant components and issues subsequent commands to the other components. If one of the execution steps fails, the saga is in charge of issuing relevant compensating actions to ensure the system state remains consistent

Learning Domain Driven Design Vlad Khononov Published by O'Reilly Media, Inc.



Building Microservices, 2nd Edition Sam Newman Published by O'Reilly Media, Inc.



Building Microservices, 2nd Edition Sam Newman Published by O'Reilly Media, Inc.

Using the Flink UI

Flink UI

- The **Flink UI** (also known as the Web Dashboard) is a web-based interface that provides real-time monitoring, management, and debugging capabilities for Flink jobs.
- It runs as part of the JobManager and is typically accessible at `http://<jobmanager-host>:8081`.
- The UI is essential for observing job execution, tracking performance, and debugging failures without needing to dive into logs or metrics manually.

Features of Flink UI

- **Dashboard Overview -** Displays an overview of all running, completed, and failed jobs. Quick insights into job IDs, statuses, uptime, and task counts.
- **Job Graph Visualization -** Visualizes the execution plan and data flow of Flink jobs. Shows how data moves through operators like `map`, `filter`, `keyBy`, and `window`.
- **Task & Operator Monitoring -** Track parallelism and subtask statuses for each operator. View task-level metrics like records processed, failures, and restarts.
- **Metrics Dashboard -** Throughput (records per second). Latency and watermarks for event-time processing. Backpressure information indicating bottlenecks.

Features of Flink UI (Continued)

- **Checkpoint & Savepoint Management -** Monitor the progress and status of checkpoints and savepoints. Trigger manual savepoints directly from the UI for safe job updates.
- **TaskManager & Resource Monitoring -** View TaskManager health, including memory usage, CPU load, and slot allocation.
- **Logs and Exception Handling -** Identify underutilized or overloaded TaskManagers for resource optimization.
- **Backpressure Detection -** Identify operators causing backpressure in the pipeline. Helps optimize resource allocation and parallelism settings.

Features of Flink UI (Continued)

- **Job Cancellation and Restart -** Cancel, pause, or restart jobs directly from the UI. Resume from the latest checkpoint to minimize data loss.
- **Configuration Management -** Review the current configuration settings for jobs and clusters. Useful for debugging issues related to timeouts, parallelism, and state backends.
- **Submit Jobs Directly** - Submit jobs directly by uploading a JAR file containing your Flink application.

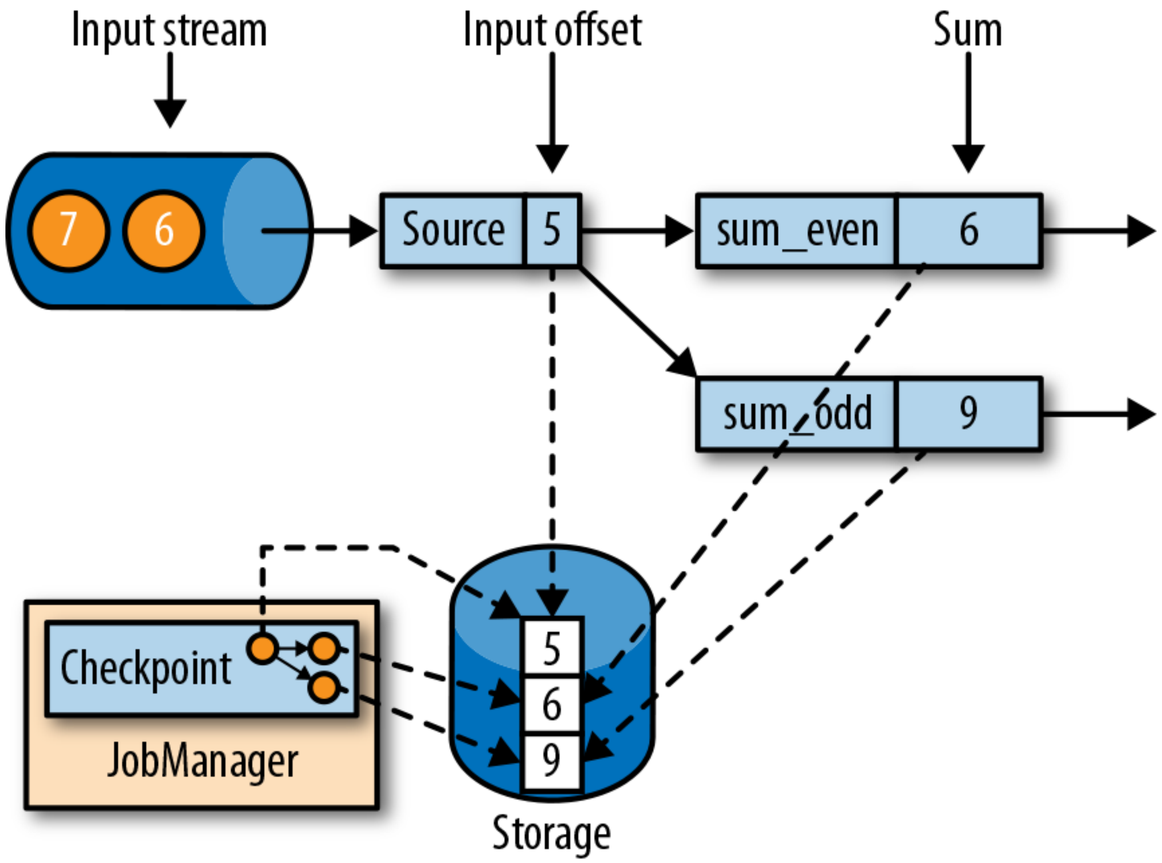
Fault Tolerance

Checkpoints

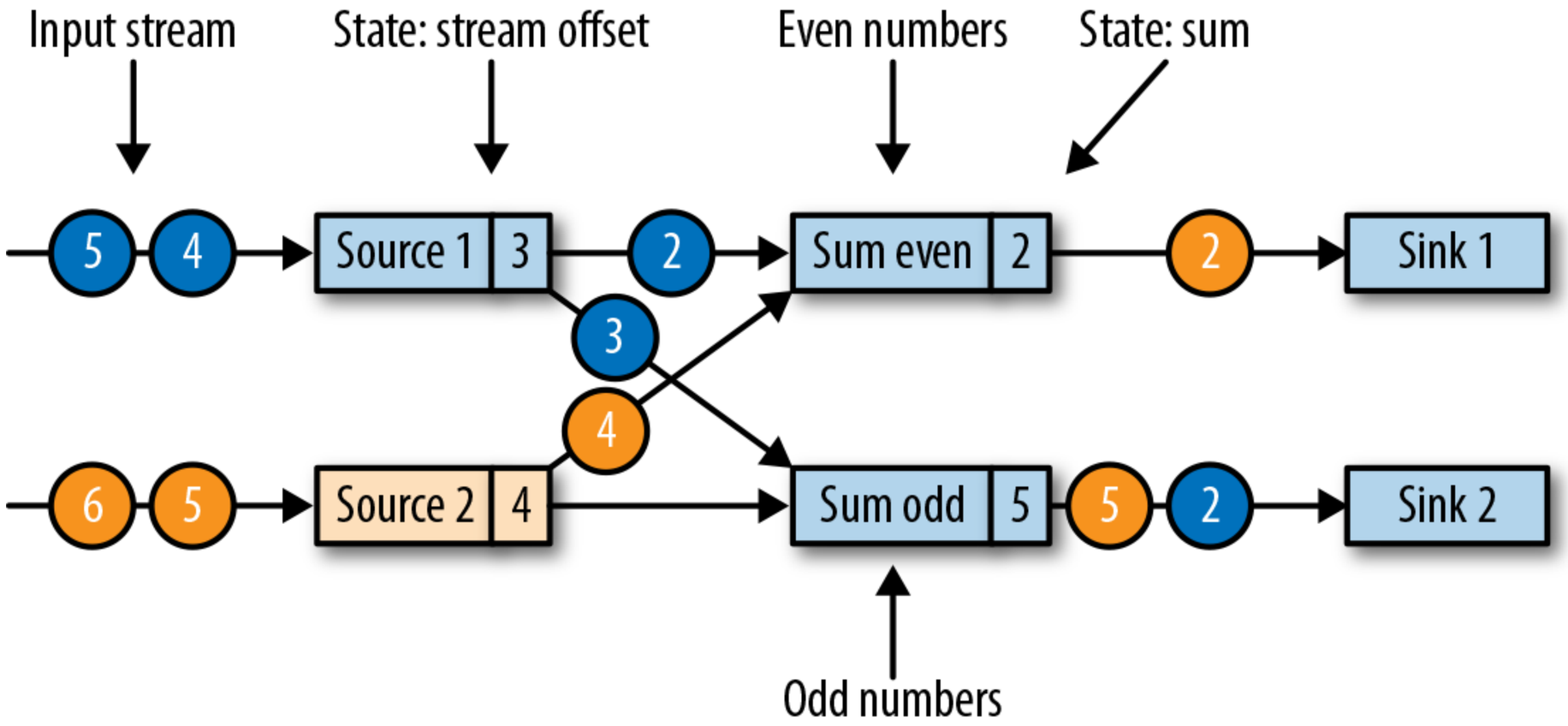
- A checkpoint in Flink is a snapshot of the entire state of a streaming application at a specific point in time.
- It’s a mechanism for fault tolerance, ensuring that if a failure occurs, the application can recover from the last successful checkpoint without data loss.

How do they work?

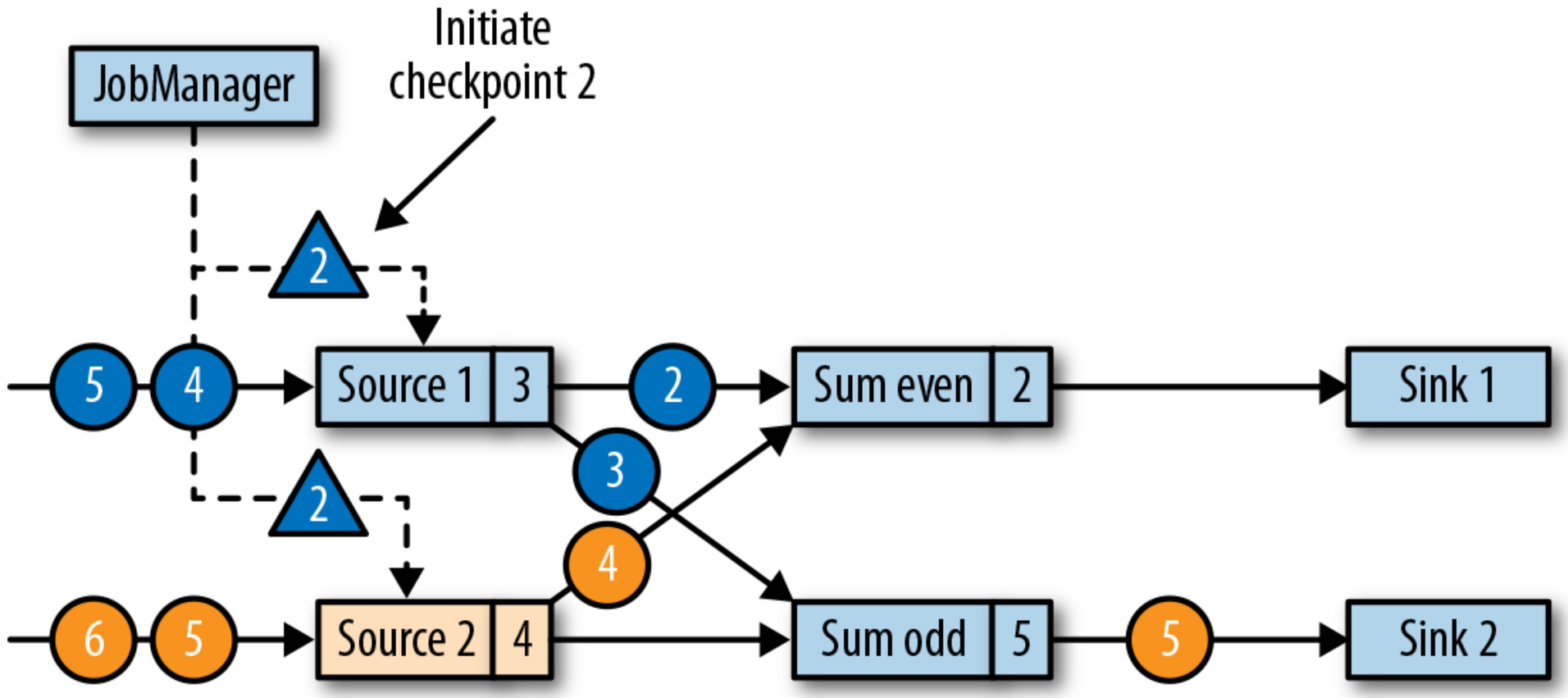
- The **Job Manager** periodically triggers checkpoints based on the configured interval. Example: Every 5 minutes, Flink creates a checkpoint of the current state.
- **Checkpoint barriers** are injected into the timeline. When an operator receives a barrier, it *takes a snapshot of its state*
- The snapshot includes:
  - **Keyed state:** State partitioned by keys (e.g., aggregations per user).
  - **Operator state:** State maintained by the operator itself.
- State is then backed up into memory, filesystem, or RocksDB
- Once all operators have successfully stored their state, the checkpoint is marked as completed.
- If a failure occurs, Flink will restart the job from this checkpoint.



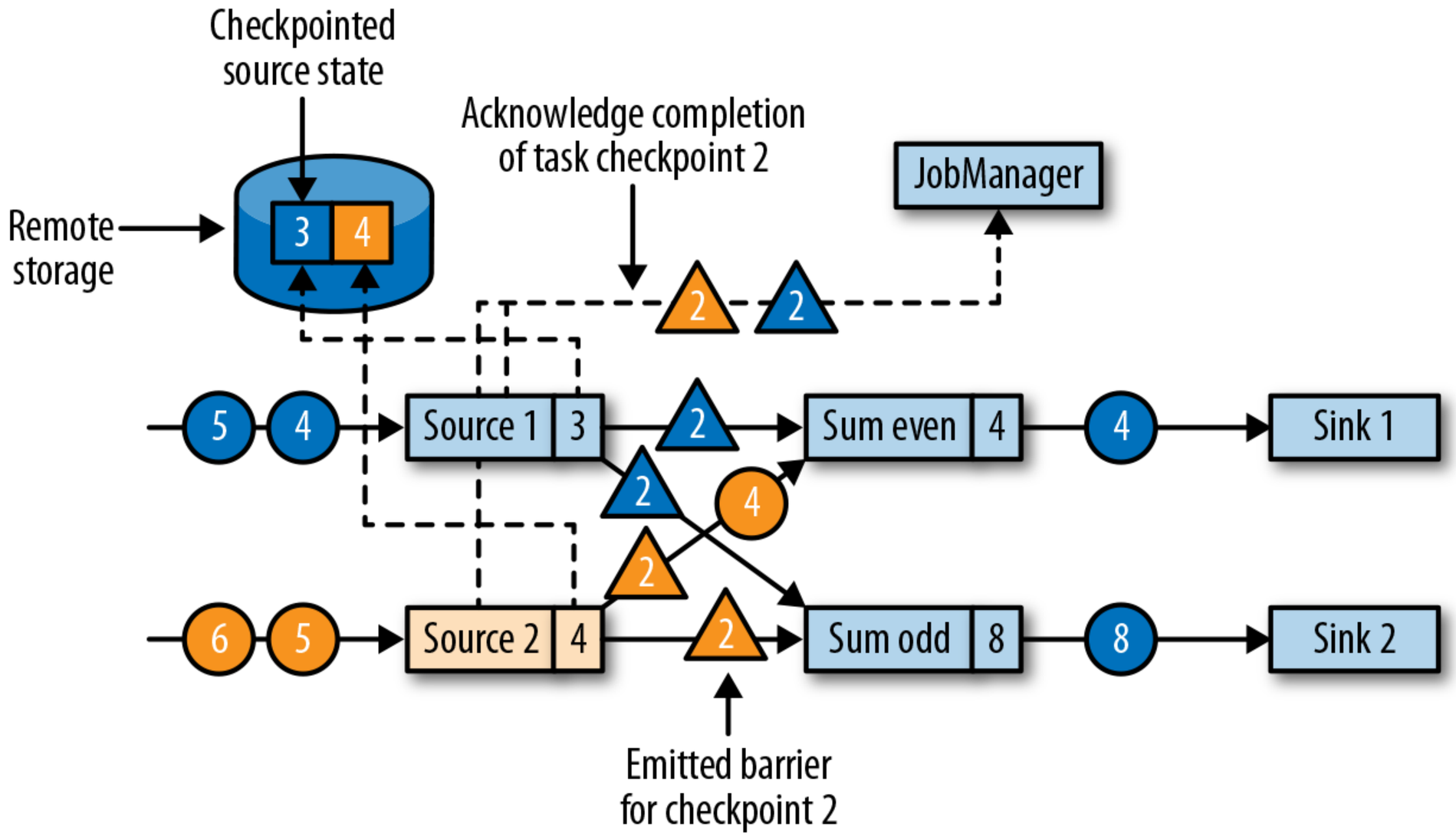
Stream Processing with Apache Flink Fabian Hueske, Vasiliki Kalavri



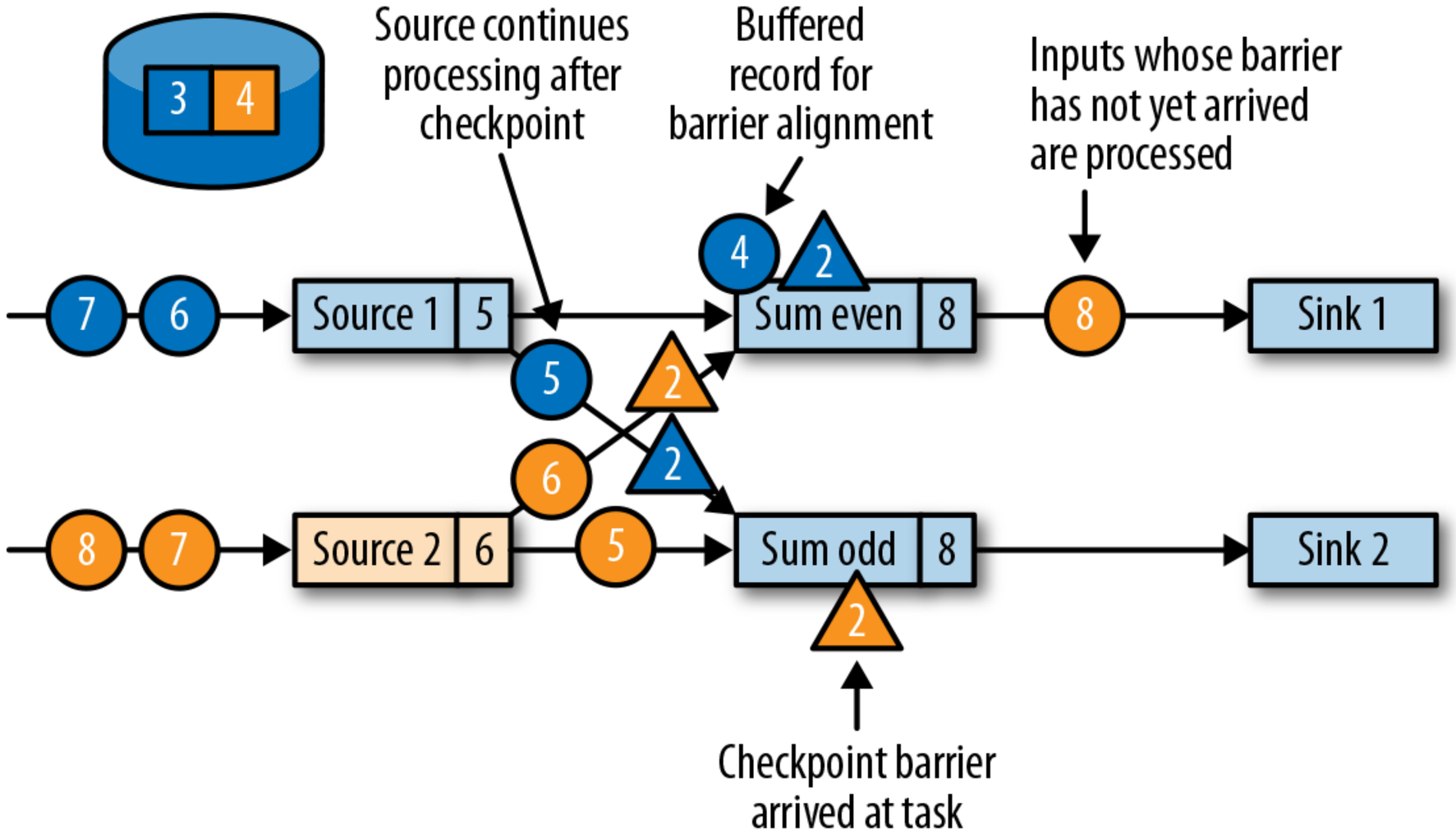
Stream Processing with Apache Flink Fabian Hueske, Vasiliki Kalavri



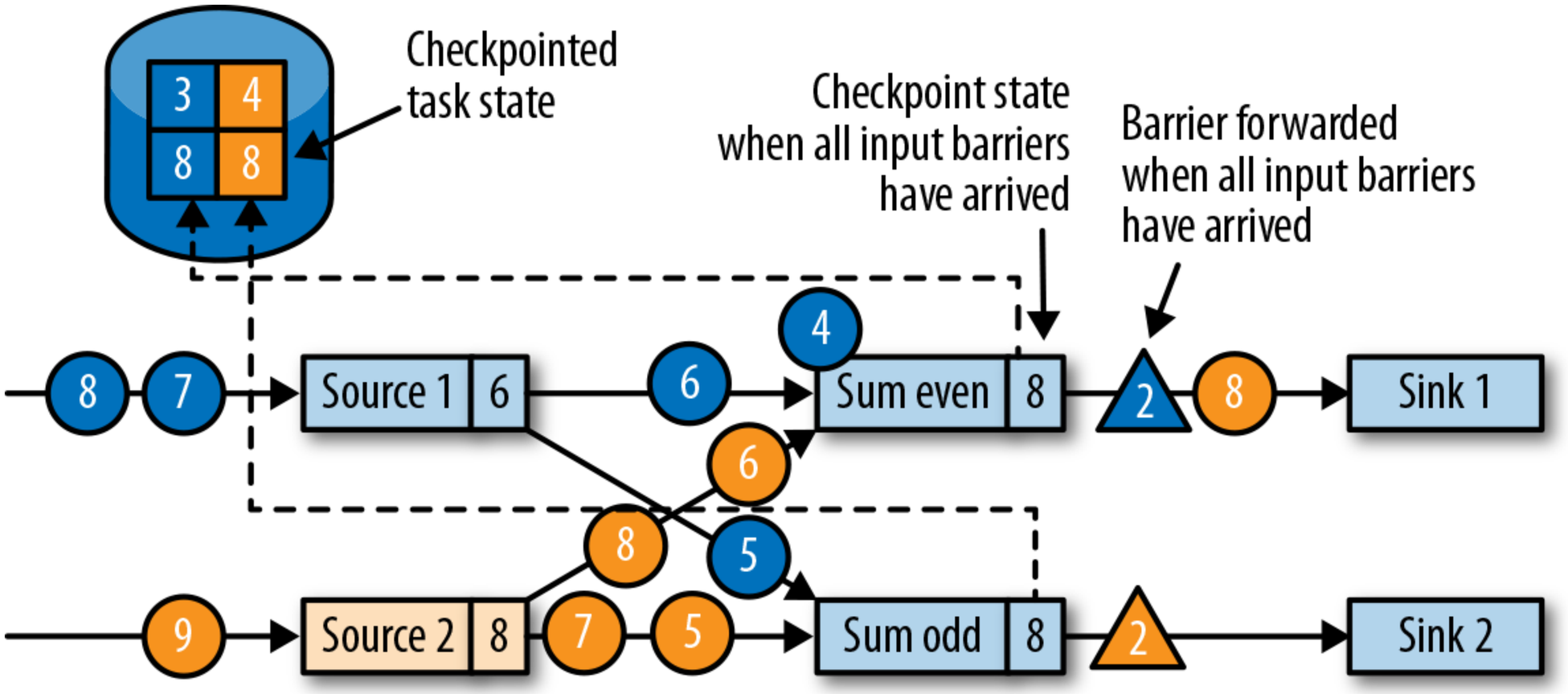
Stream Processing with Apache Flink Fabian Hueske, Vasiliki Kalavri



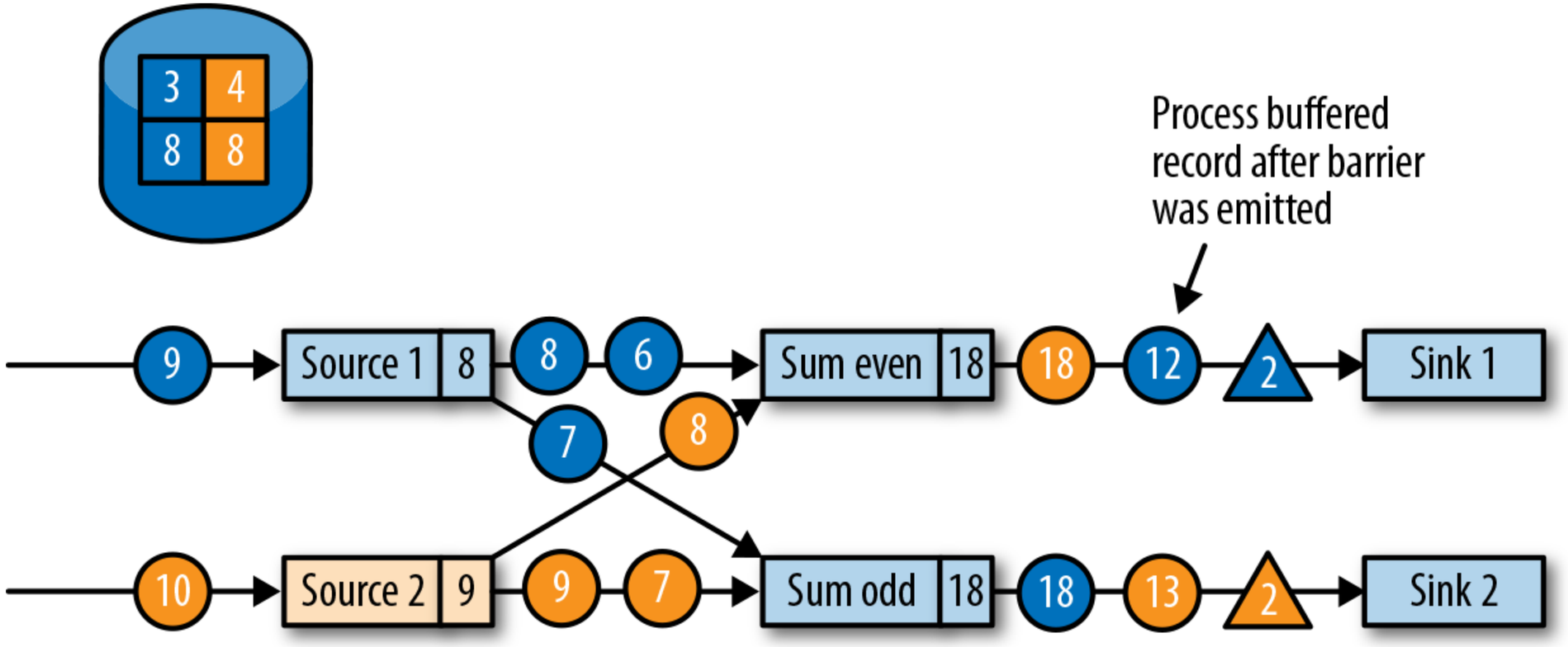
Stream Processing with Apache Flink Fabian Hueske, Vasiliki Kalavri



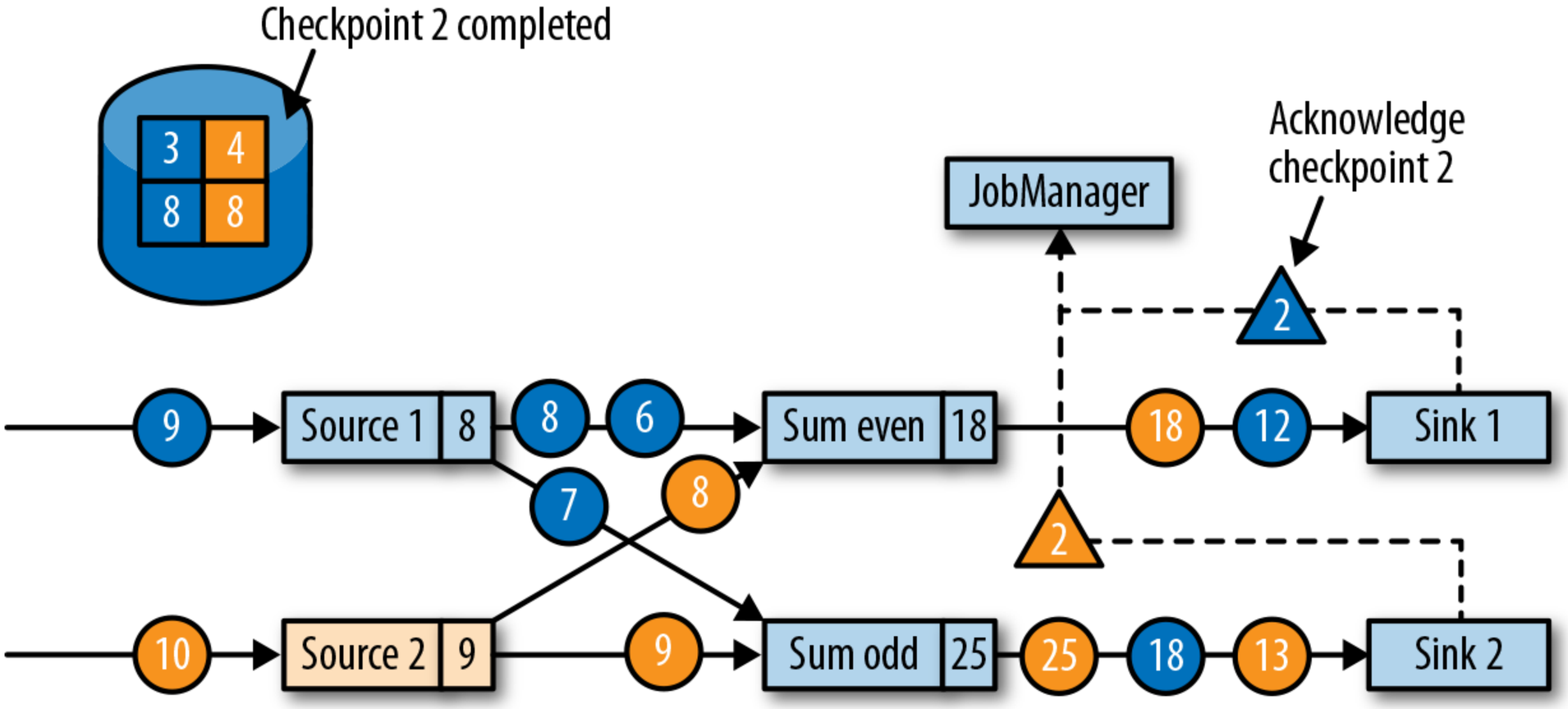
Stream Processing with Apache Flink Fabian Hueske, Vasiliki Kalavri



Stream Processing with Apache Flink Fabian Hueske, Vasiliki Kalavri



Stream Processing with Apache Flink Fabian Hueske, Vasiliki Kalavri



Stream Processing with Apache Flink Fabian Hueske, Vasiliki Kalavri

Savepoints

- A **savepoint** in Flink is a manually triggered snapshot of a streaming application's state.
- Unlike automatic checkpoints, savepoints are designed for operational tasks like *upgrades*, *maintenance*, and *migration*.
- Provides a consistent recovery point for stopping, modifying, and restarting jobs.
- Essential for version upgrades, code changes, or state migrations without data loss.
- Offers operator-level flexibility: You can rescale or modify the job graph while retaining state.

Partitioning

Partitioning

- Partitioning in Flink refers to how data is distributed across different parallel tasks (operators) during processing.
- It determines which parallel instance of an operator processes which subset of data.
- Partitioning is essential for scalability, parallelism, and efficient processing in distributed environments.

How is Partitioning Used?

- Parallel Processing: Enables the system to distribute the workload across multiple parallel task slots.
- State Management: In keyed state operations (e.g., aggregations), partitioning ensures that all events with the same key are processed by the same operator.
- Data Locality & Optimization: Ensures efficient data locality, reducing shuffling and network overhead.
- Consistency: Guarantees that stateful operations, such as reduce or window, operate on consistent partitions of data.

Parallelism

Parallelism

- Parallelism in Flink refers to the number of concurrent tasks that execute parts of a data processing job.
- Flink jobs are broken into operators (e.g., `map`, `filter`, `reduce`), and each operator can run in multiple parallel instances called subtasks.
- Parallelism ensures that Flink can process large datasets efficiently by distributing the workload across multiple CPU cores or nodes in a cluster.

How is Parallelism Achieved in Stream API?

- Global Processing

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.setParallelism(4);  // Sets parallelism to 4 for the entire job
```

- Operator Level Processing

```
DataStream<String> stream = env.fromElements("A", "B", "C").setParallelism(2);
```

- Source Based Processing

```
KafkaSource<String> source = KafkaSource.<String>builder()
    .setBootstrapServers("localhost:9092")
    .setTopics("my_topic")  // Parallelism depends on Kafka partition count
    .build();
```

How is Parallelism Achieved in Stream API?

- Processing in Sinks

```
dataStream.sinkTo(kafkaSink).setParallelism(3);  // Sink runs with 3 parallel instances
```

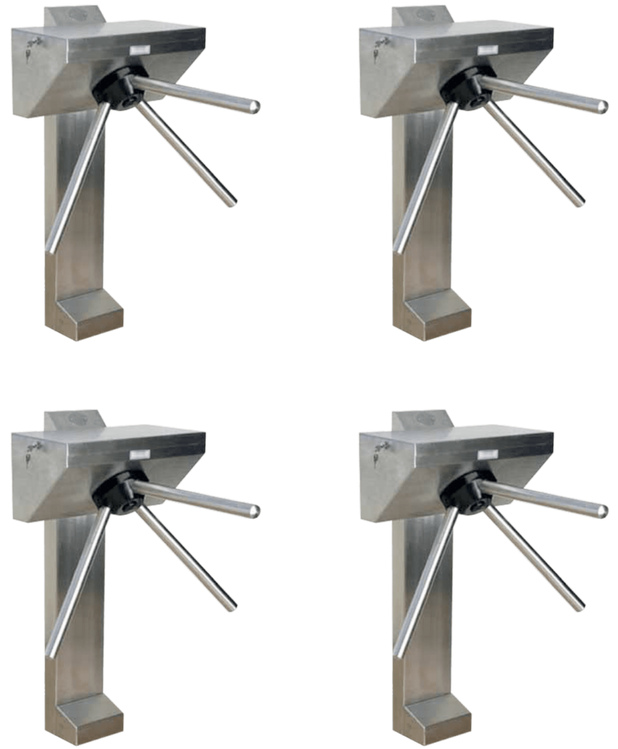
Stateful Operations

State

- State in Flink refers to data that is stored and managed across different events in a stream.
- It allows Flink applications to remember information between events, enabling complex, stateful computations.
- We need state processing, for aggregations (`sum`, `average`, and `count`), for user sessions and activities, detect sequences, remembering processed events

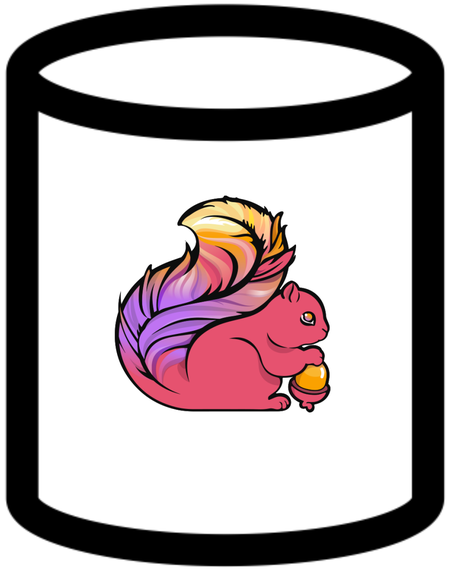
State is Hard

- **Consider the following issues:**
  - **Fault Tolerance:** Ensuring state is consistent and recoverable after failures.
  - **Scalability:** Managing state efficiently as data volume and parallelism increase.
  - **Consistency:** Maintaining exactly-once semantics while handling distributed streams.
  - **Latency:** Keeping processing fast while managing large state.



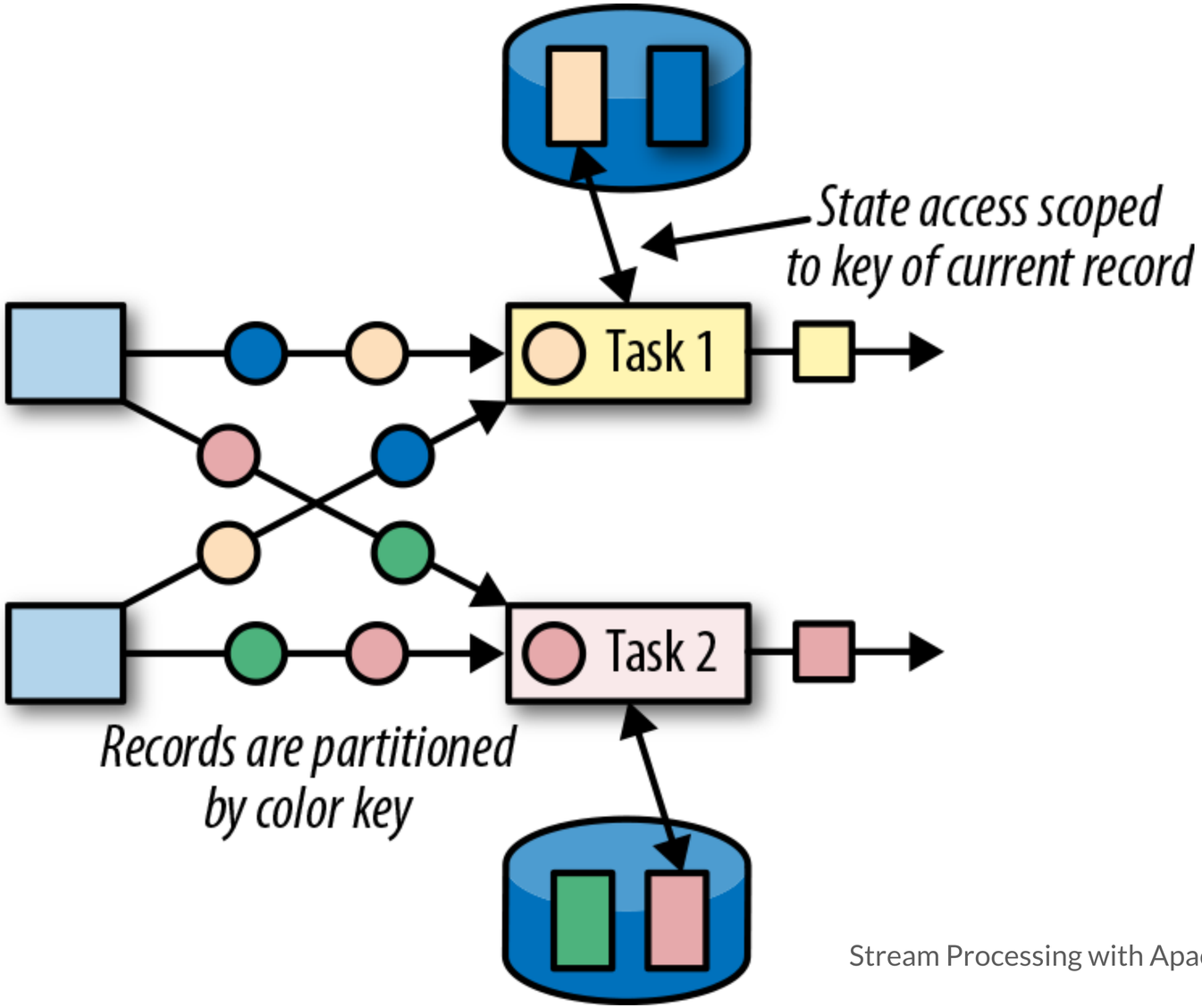
State in Flink

- Flink uses state backends to manage how and where state is stored.
- Common backends:
  - **MemoryStateBackend:** Stores state in-memory (for testing or small state).
  - **RocksDBStateBackend:** Stores state on disk using RocksDB (for large-scale, production systems).
  - **FsStateBackend:** Stores state in filesystem (e.g., HDFS).



Created by Herbert Spencer
from Noun Project

State in Flink



Stream Processing with Apache Flink Fabian Hueske, Vasiliki Kalavri

Demo: State Operations

- Let’s see what a state operation looks like

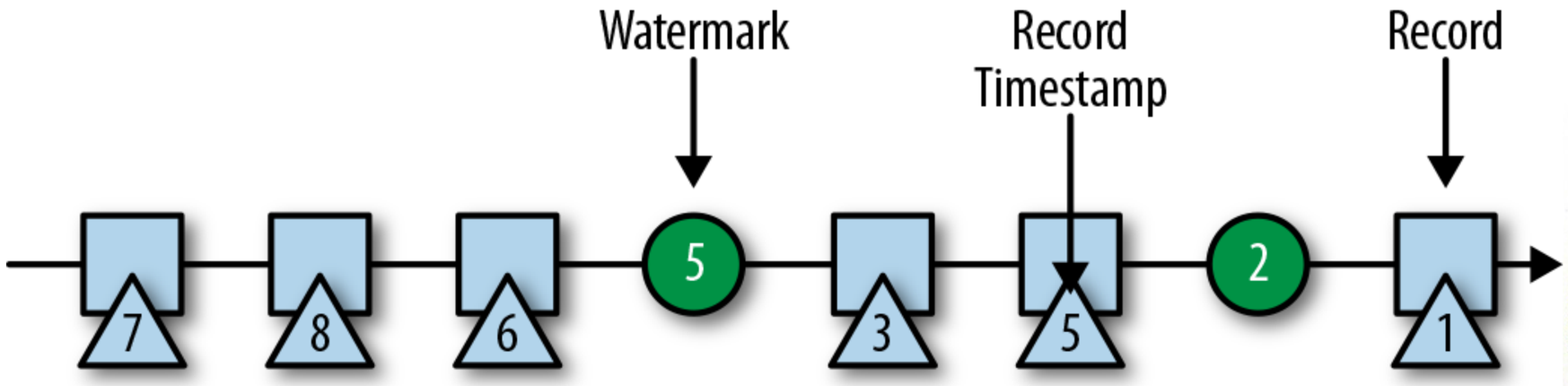
Window Operations

Event Time vs Processing Time

- **Event Time:** The time an event *actually* occurred and extracted from the data itself
- **Processing Time:** The system clock time when Flink processes an event.

Watermarks

- A *watermark* is a special kind of timestamp that Flink uses to track the progress of event time in a data stream.
- It tells Flink: “I’ve likely seen all events up to this point in time, and it’s safe to process them.”

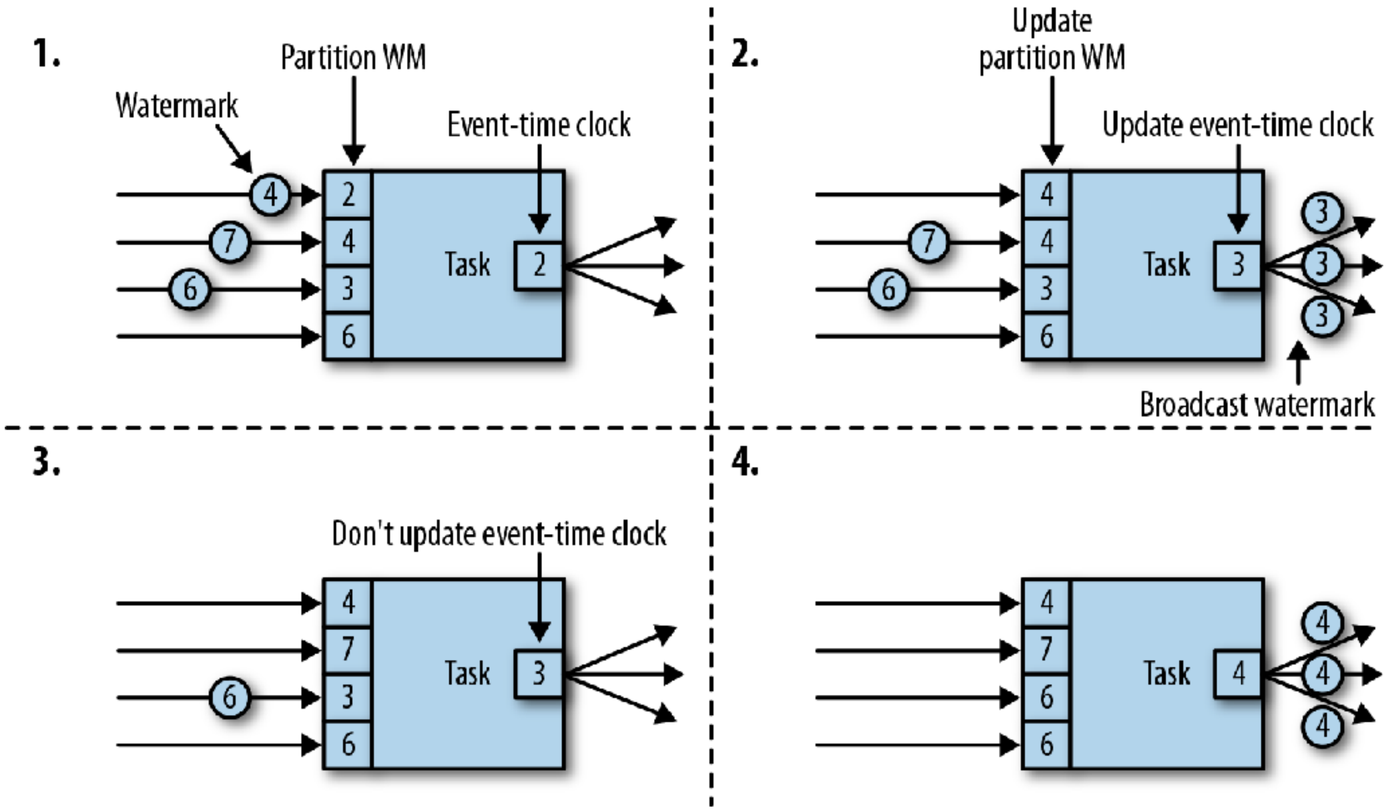


Stream Processing with Apache Flink Fabian Hueske, Vasiliki Kalavri

Parallel Processing Watermarks

- Watermarks are generated at, or directly after, source functions. Each parallel subtask of a source function usually generates its watermarks independently. These watermarks define the event time at that particular parallel source.

Processing of Watermarks



- The example shows how a task with four input partitions and three output partitions receives watermarks, updates its partition watermarks and event-time clock, and emits watermarks.
- Note that the lowest number will be propagated forward

Stream Processing with Apache Flink Fabian Hueske, Vasiliki Kalavri

Tumbling Window

Hop Window

Session Window

Session Window

the last element has a occurred; now we wait

Session Window

waited 5 minutes = make it a window

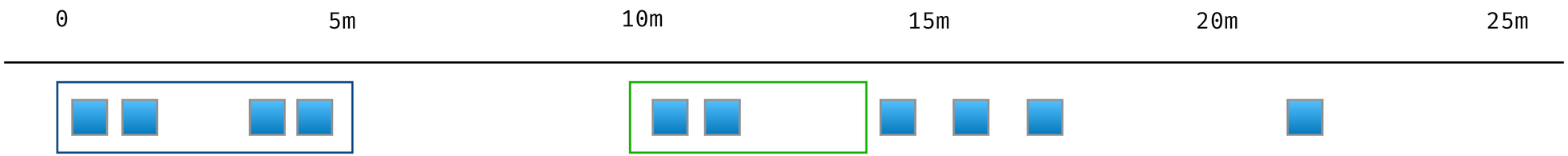
Session Window

we have a new element; we start a new window

Session Window

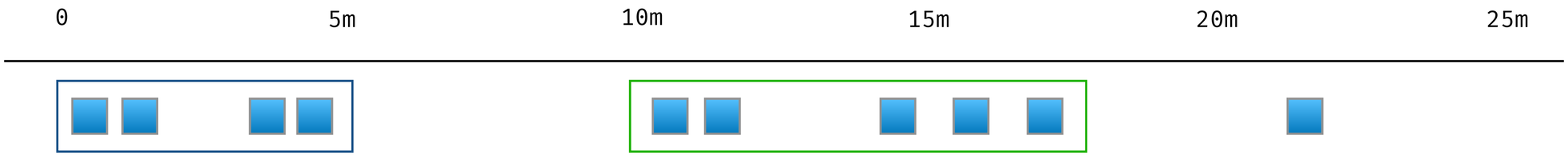
we wait for 5 minutes

Session Window



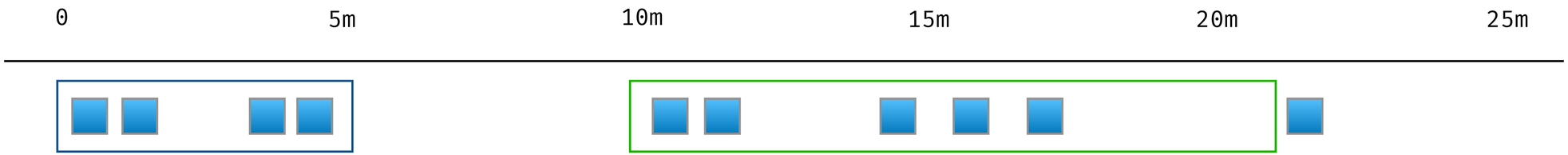
2 minutes elapsed; no windowing yet

Session Window



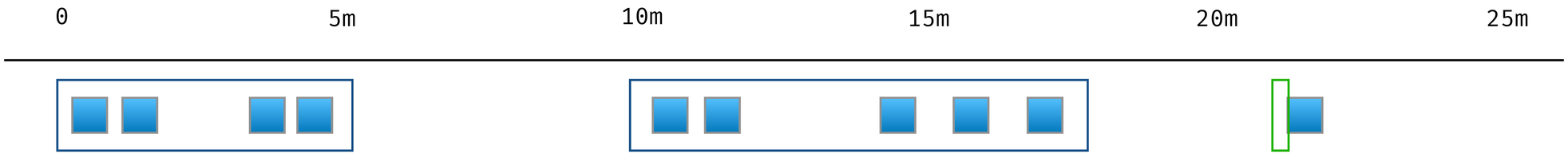
we wait five minutes

Session Window



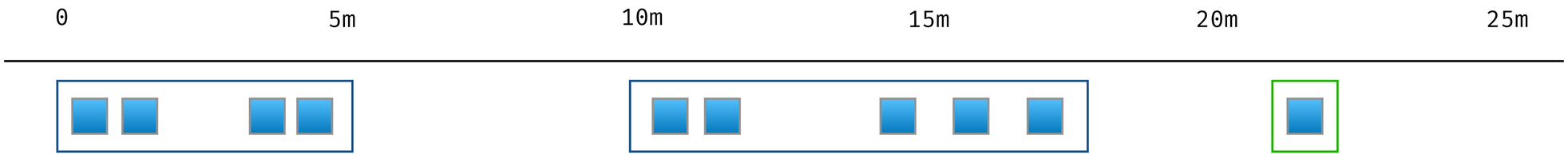
no new elements; we make it a window

Session Window



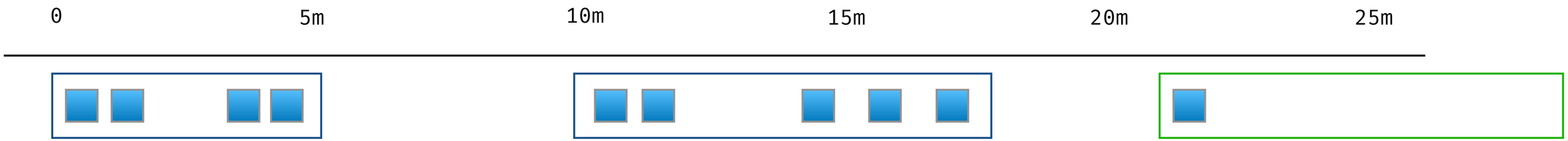
we see something new

Session Window



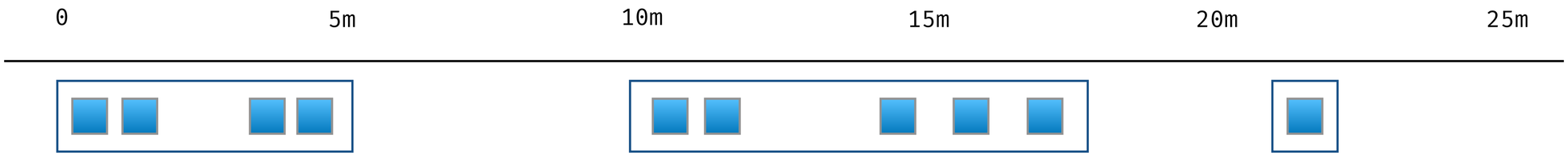
last element; we wait five minutes

Session Window



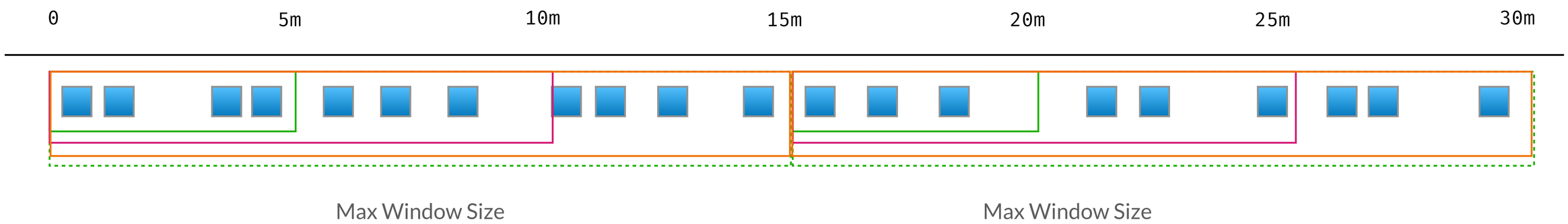
nothing else; we wait 5 minutes

Session Window



that's a new window

Cumulate Window



Demo: Windowing

- Let’s view what windowing looks like in Flink.

Testing

Test Harness

- A **Test Harness** in Flink is a utility that allows you to unit test individual operators (like map, flatMap, processFunction) in isolation.
- It simulates the Flink runtime environment (including state, time, and timers) without needing to run the full job or deploy a cluster.
- **Benefits include:**
  - **Operator Level Testing** - Focus on testing custom logic in a single operator instead of the entire pipeline.
  - **Control Over Time and State** - Simulate event time and processing time. Test how operators behave with timers and stateful processing.
  - Validate how operators behave during state snapshots and recovery scenarios.

Demo: Test Harness

- Let’s see what a test harness looks like to test Flink code without submitted to the server.

User Defined Functions

User Defined Functions

- In the Table/SQL API, UDFs extend SQL-like expressions to perform custom operations that aren’t natively supported. You define them when you need specific transformations or computations within SQL queries or Table API expressions.

- UDF can be a promising solution to call external services, return cached elements, and call ML models

Creating a UDF Scalar Function

Scalar Functions: Operate on a single input row and produce a single output value.

```
public class UpperCaseFunction extends ScalarFunction {
    public String eval(String input) {
        return input.toUpperCase();
    }
}
```

Creating a UDF Table Function

Table Functions: Return a table (multiple rows) from a single input row.

```
public class SplitFunction extends TableFunction<String> {
    public void eval(String str) {
        for (String s : str.split(",")) {
            collect(s);
        }
    }
}
```

Creating a UDF Aggregate Function

Aggregate Functions: Perform aggregations over multiple input rows.

```
public class SumAggregateFunction extends AggregateFunction<Integer, SumAccumulator> {
    public Integer getValue(SumAccumulator acc) { return acc.sum; }
    public void accumulate(SumAccumulator acc, Integer value) { acc.sum += value; }
}
```

Creating a UDF Table Aggregate Function

Table Aggregate Functions: Perform multiple rows for each group of input rows. This is useful for operations like Top-N queries.

```
// Define the Table Aggregate Function
public static class Top2Scores extends TableAggregateFunction<Integer, Top2Accumulator> {

    public void accumulate(Top2Accumulator acc, Integer score) {
        acc.topScores.add(score);
        if (acc.topScores.size() > 2) {
            acc.topScores.pollLast();  // Keep only top 2 scores
        }
    }

    public void emitValue(Top2Accumulator acc, Collector<Integer> out) {
        for (Integer score : acc.topScores) {
            out.collect(score);  // Emit top scores
        }
    }
}
```

Registering a UDF

Once you design your UDF you can register it by performing the following:

```
tableEnv.createTemporarySystemFunction("UpperCase", UpperCaseFunction.class);
Table result = tableEnv.sqlQuery("SELECT UpperCase(name) FROM users");
```

Flink Extensions

Flink CEP

Complex Event Processing

Flink CEP

- Flink CEP is a library for detecting complex patterns in event streams. It allows you to define and detect sequences of events that meet specific conditions.
- Use Flink CEP when you need to monitor and detect patterns over time, such as **fraud detection**, **anomaly detection**, **e-commerce behavior patterns, network monitoring,** or **alerting systems**.
- It's ideal for scenarios where individual events are not meaningful on their own but gain significance when part of a sequence.
- Use it when your application requires identifying complex, time-sensitive event patterns. If your use case only involves *simple filters or aggregations*, standard `DataStream` operations might suffice.

Demo: Flink CEP

- Let’s what we mean by patterns with CEP

Flink ML

Machine Learning Processing in Flink

Flink ML

- Flink ML is a library for building and deploying machine learning models within Flink’s streaming and batch processing framework.
- It provides scalable algorithms and tools for *preprocessing*, *model training*, and *prediction*.
- Use Flink ML when you need to integrate **real-time machine learning** directly into your data processing pipelines. It is particularly useful for applications that require **online learning**, **real-time predictions**, or **incremental model updates**

Flink Gelly

Graph Processing in Flink

Flink Gelly

- **Flink Gelly** is a graph processing library within **Apache Flink** designed for creating, transforming, and analyzing graph-structured data.
- It provides APIs and algorithms for working with large-scale graphs in both batch and streaming modes.
- Use **Flink Gelly** when you need to perform graph analytics like *PageRank*, community detection, shortest path computation, or graph traversal on large datasets.
- It's particularly useful for applications in social network analysis, fraud detection, and recommendation systems.

Stream Framework Comparisons

Streaming

Streaming

Strengths

Comparing the Strengths of each Stream Application

Flink Strengths and Use Cases

- Real-time streaming with sub-second latency.Advanced stateful processing and event-time handling.
- Supports complex event processing (CEP) for pattern detection.
- Flexible deployment: runs on Kubernetes
- Exactly Once Semantics
- Java, Scala, Python
- Includes ML Library

Kafka Streams Strengths and Use Cases

- Lightweight library that runs within any Java application. Easy setup.
- Tightly integrated with Kafka, offering exactly-once semantics out of the box.
- No separate cluster needed—just your Kafka brokers.
- Simplified deployment and scaling based on Kafka partitioning.
- Java and Scala
- Exactly Once Semantics Achievable

Spark Streaming Strengths and Use Cases

- Unified batch and streaming API using DataFrames.
- Rich ecosystem with libraries for machine learning (MLlib), graph processing (GraphX), and SQL.
- Strong support for big data workflows and ETL pipelines.
- Can run on Hadoop, Kubernetes, and cloud platforms like AWS, Azure, and GCP.
- Java, Scala, Python (PySpark), and R

Akka Streaming Strengths and Use Cases

- Built on the Akka actor model for reactive, event-driven applications.
- Provides fine-grained control over backpressure and asynchronous processing.
- Lightweight, with low-latency and in-memory stream processing.
- Ideal for distributed, concurrent systems.
- Java, Scala

Weaknesses

Comparing the Weaknesses of each Stream Application

Flink Weaknesses and Tradeoffs

- **Complex setup:** Requires managing TaskManagers, JobManagers, and cluster resources.
- **Steeper learning curve:** Advanced features like stateful processing and event-time semantics can be challenging for beginners.
- **Resource-intensive:** Can consume significant memory and CPU, especially for large-scale, stateful jobs.

Kafka Streams Weaknesses and Tradeoffs

- Limited to Kafka: Tight coupling to Kafka means it doesn't work well with other data sources/sinks. *All sources and sinks are tied to Kafka topics*
- KSQLDB, Kafka's SQL platform is a separate service
- No Machine Learning, Complex Event Processing, or Graphing libraries included.

Spark Streaming Weaknesses and Tradeoffs

- Higher latency (Structured Streaming) : Uses micro-batching instead of true event-by-event streaming, which introduces delays.
- Cluster required: Needs a Spark cluster or managed service, making it less suitable for lightweight deployments.
- Not ideal for sub-second latency: While great for large-scale processing, it’s not optimized for ultra-low latency use cases.

Akka Streaming Weaknesses and Tradeoffs

- **Not designed for big data:** Lacks native support for large-scale data processing and distributed analytics.
- **No ML and Graph Analysis:** Doesn’t have the same extensive connectors or libraries as Flink or Spark
- **Steep Learning Curve:** Requires familiarity with the Akka actor model and reactive programming.

Confluent Acquistion

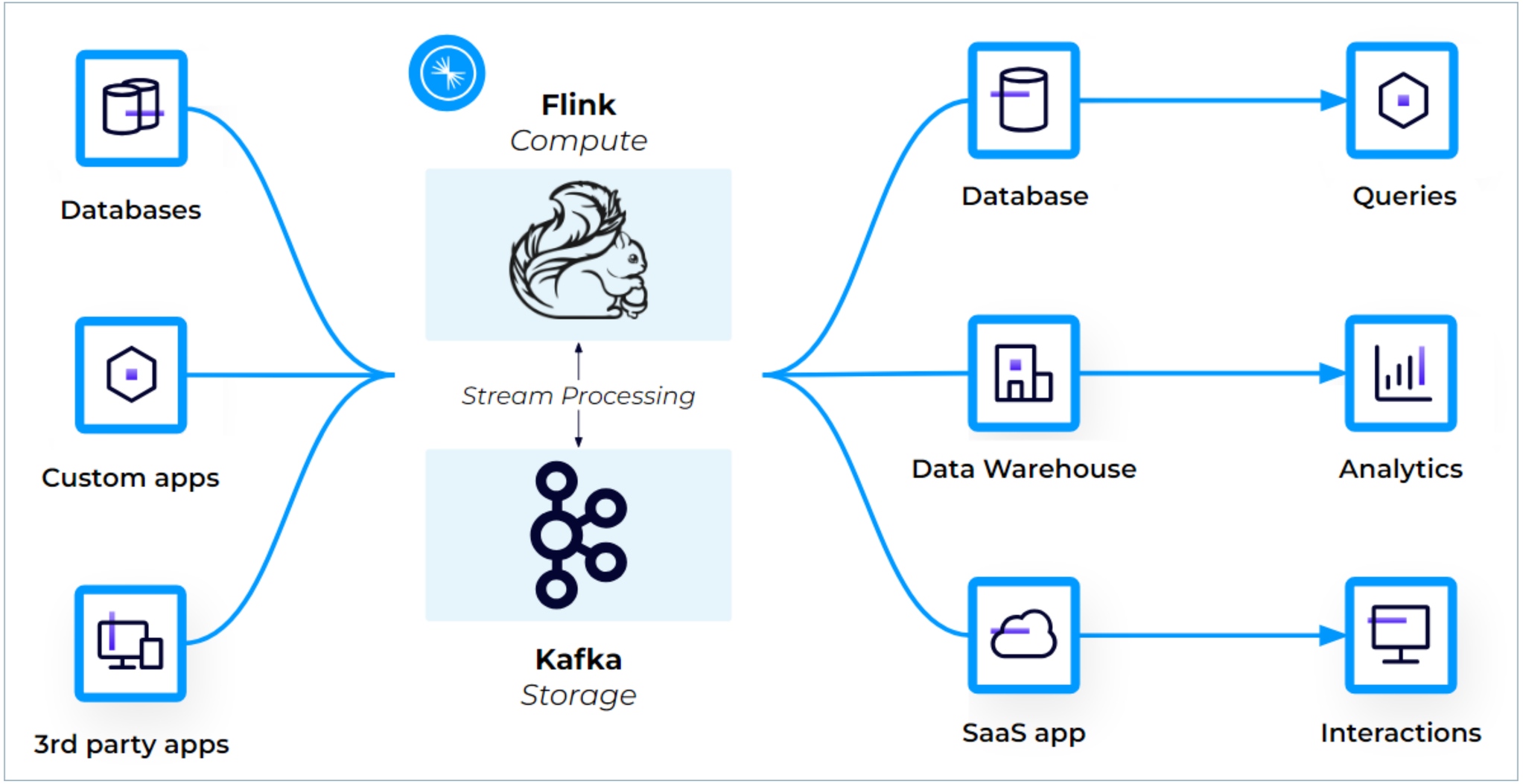
Confluent Acquisition

- Confluent announced its intent to acquire Immerok, a leading contributor to Apache Flink, to accelerate the development of a cloud-native Flink service integrated with Confluent Cloud.
- This acquisition will combine Flink's powerful stream processing capabilities with Confluent's managed Kafka services, offering customers seamless real-time data processing.
- Immerok's team, including key Flink contributors, will enhance Confluent's expertise in stream processing.
- Confluent Cloud was released in 2023.

CONFLUENT

IMMEROK
IMMEROK
IMMEROK
IMMEROK

https://www.confluent.io/press-release/confluent-plans-immerok-acquisition-to-accelerate-cloud-native-apache-flink/


Databases
Custom apps
3rd party apps
Flink
Compute
Stream Processing
Kafka
Storage
Database
Queries
Data Warehouse
Analytics
SaaS app
Interactions

Confluent Cloud and Flink Unified

- Confluent Cloud for Apache Flink is Flink re-imagined as a truly cloud-native service. Confluent's fully managed Flink service enables you to:
  - Easily filter, join, and enrich your data streams with Flink
  - Enable high-performance and efficient stream processing at any scale, without the complexities of managing infrastructure
  - Experience Kafka and Flink as a unified platform, with fully integrated monitoring, security, and governance

https://docs.confluent.io/cloud/current/flink/overview.html

Thank You

- Email: dhinojosa@evolutionnext.com
- Github: https://www.github.com/dhinojosa
- Mastodon: https://mastodon.social/@dhinojosa
- Linked In: http://www.linkedin.com/in/dhevolutionnext