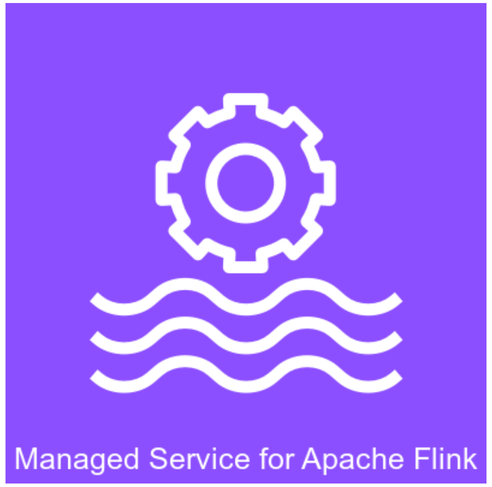
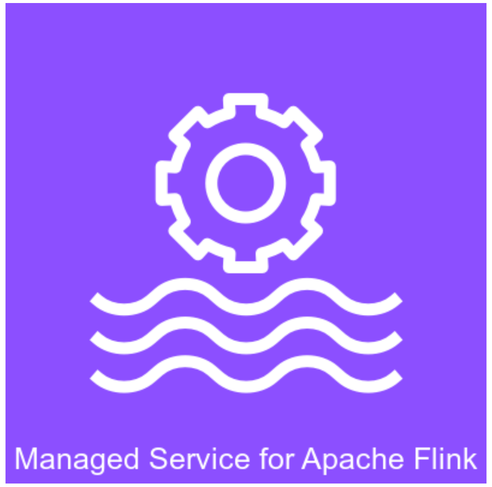


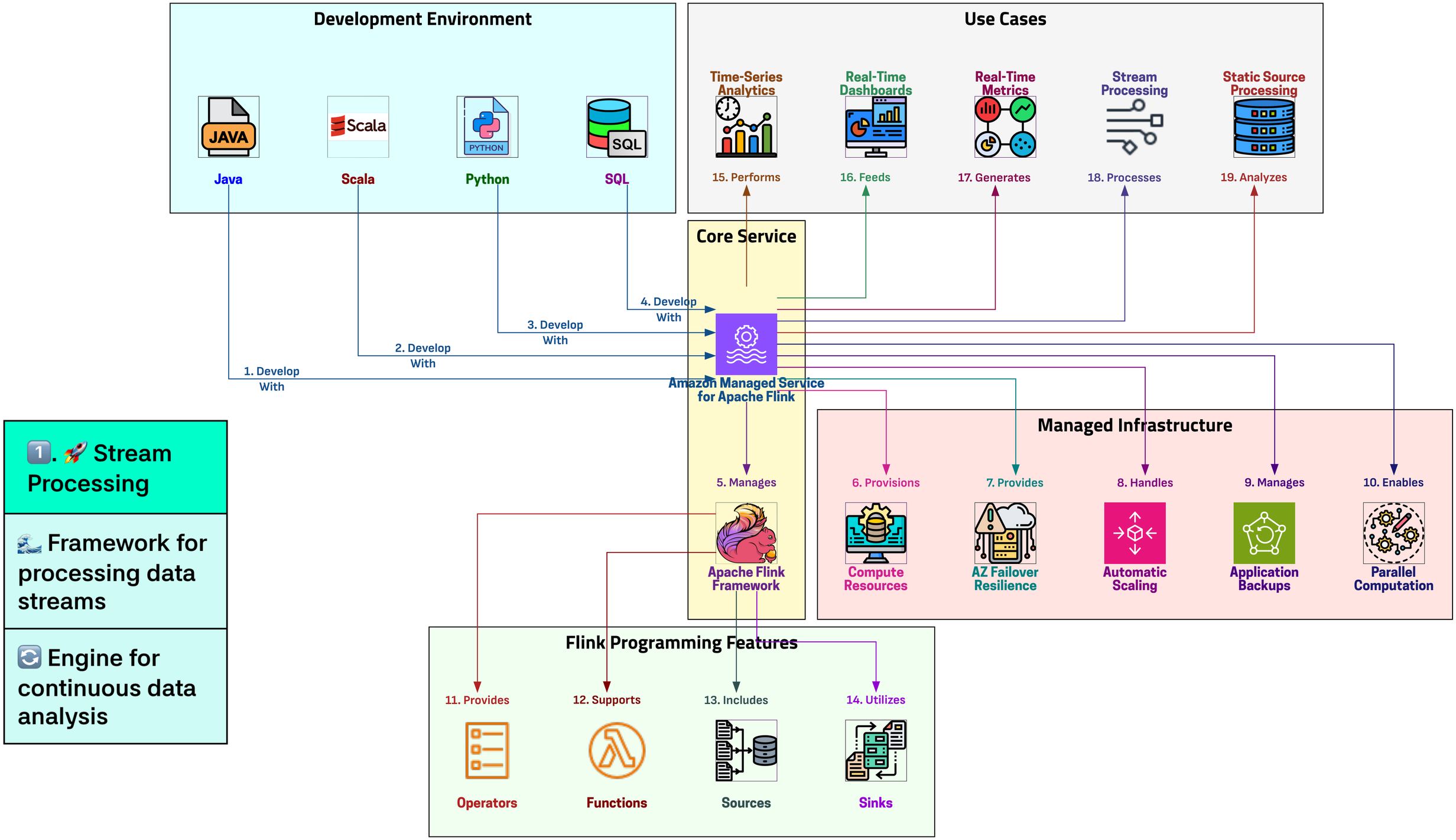
Amazon Managed Service for Apache Flink

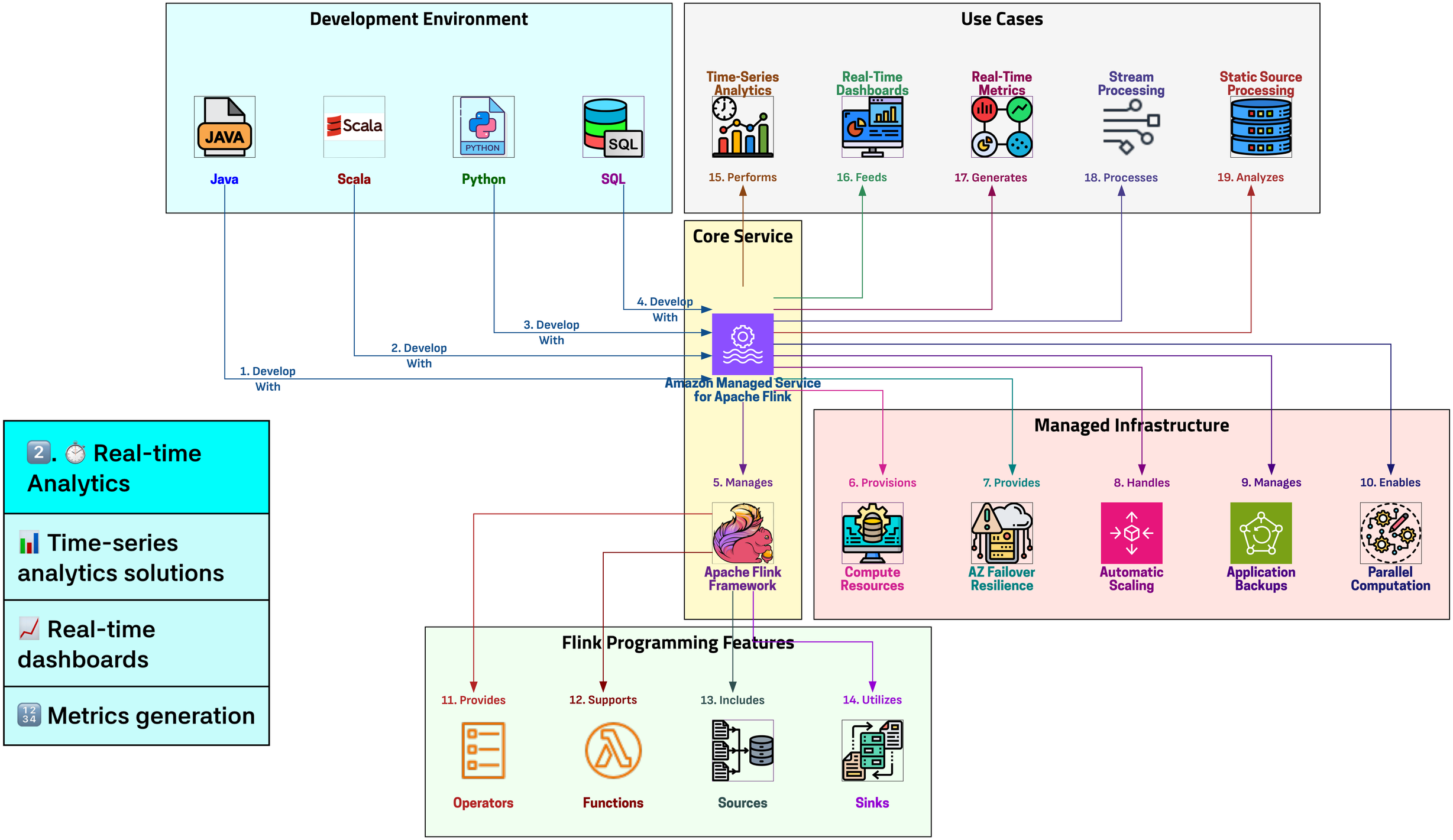


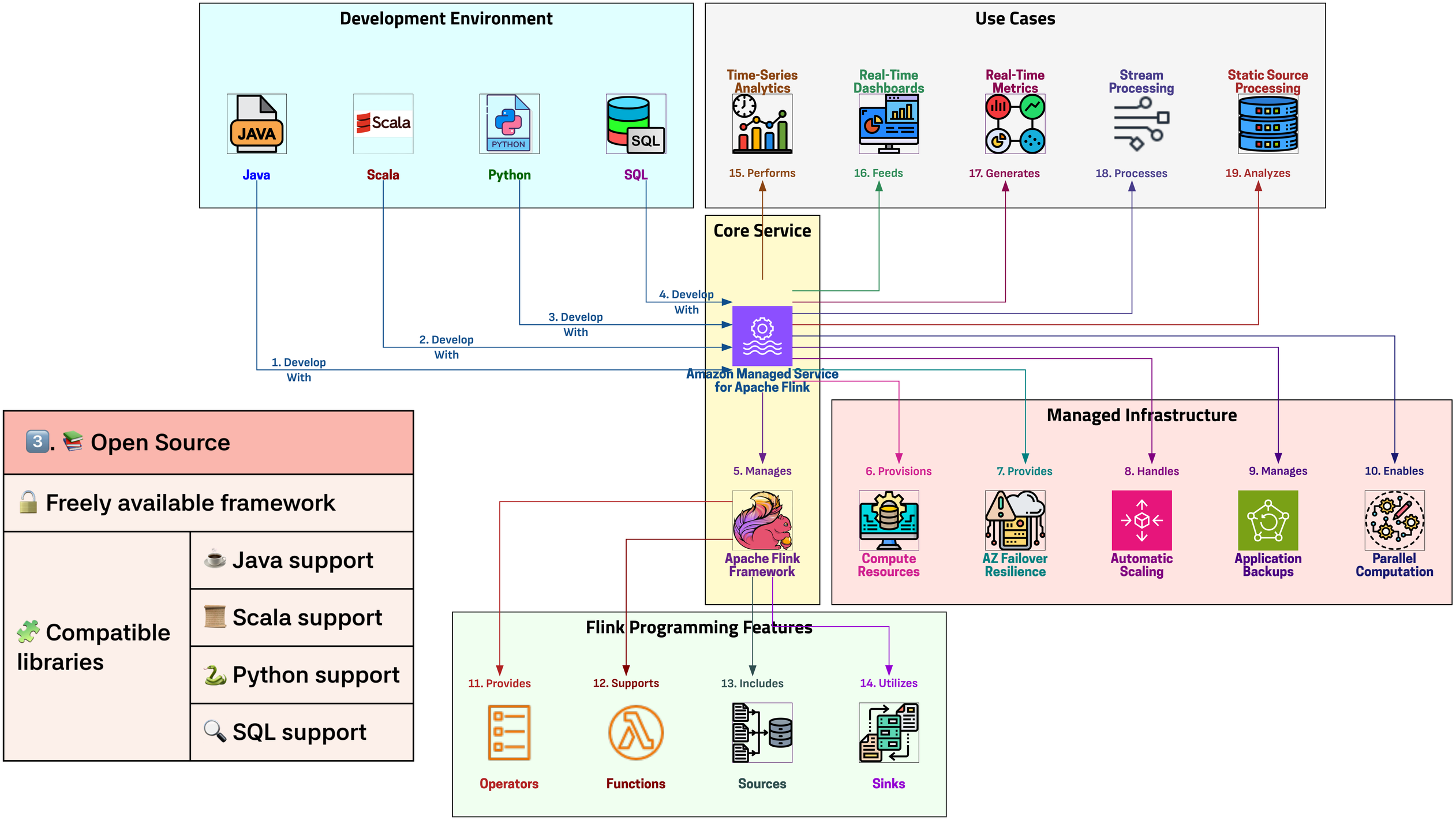
Amazon Kinesis Data Analytics for Apache Flink

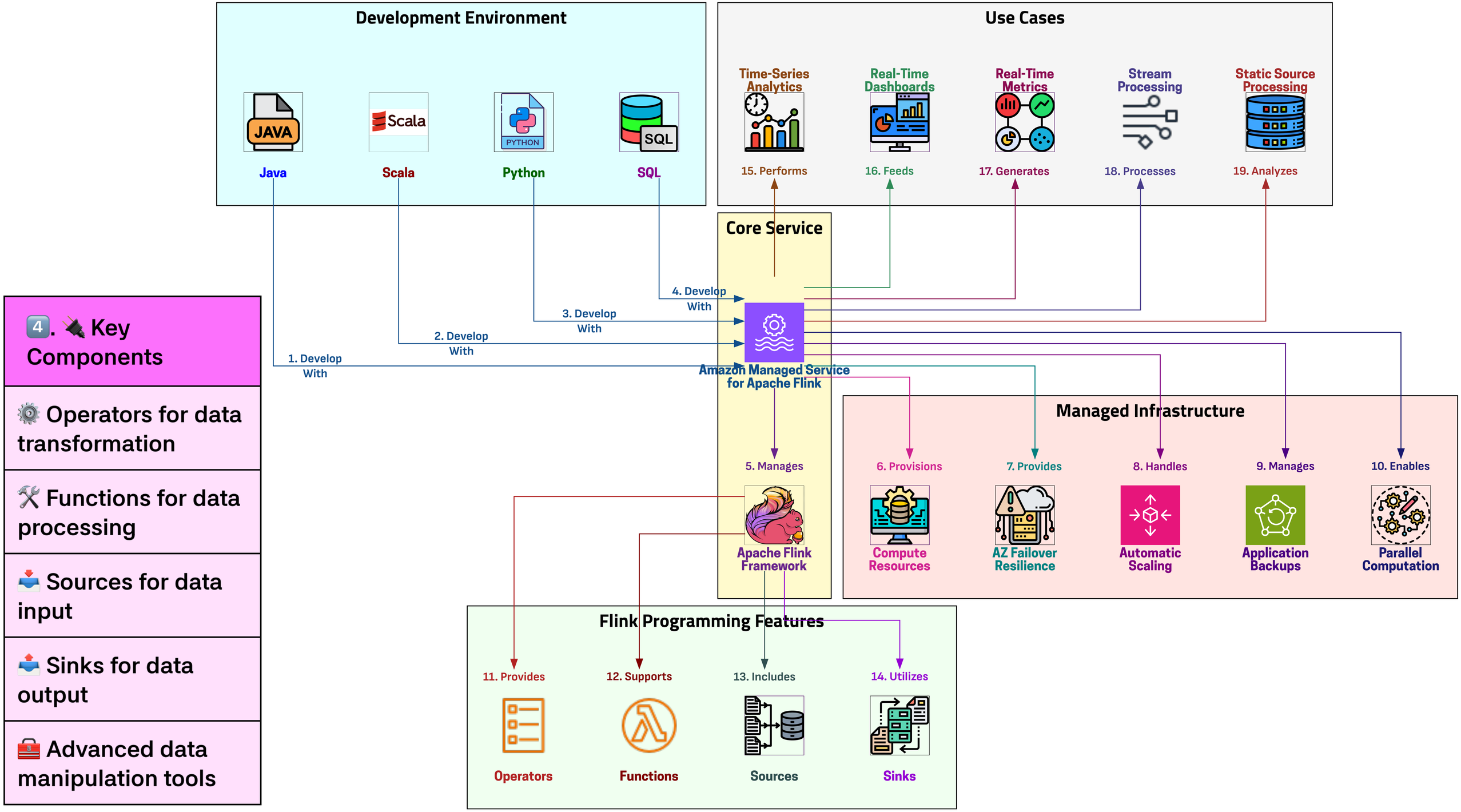
Table of Contents

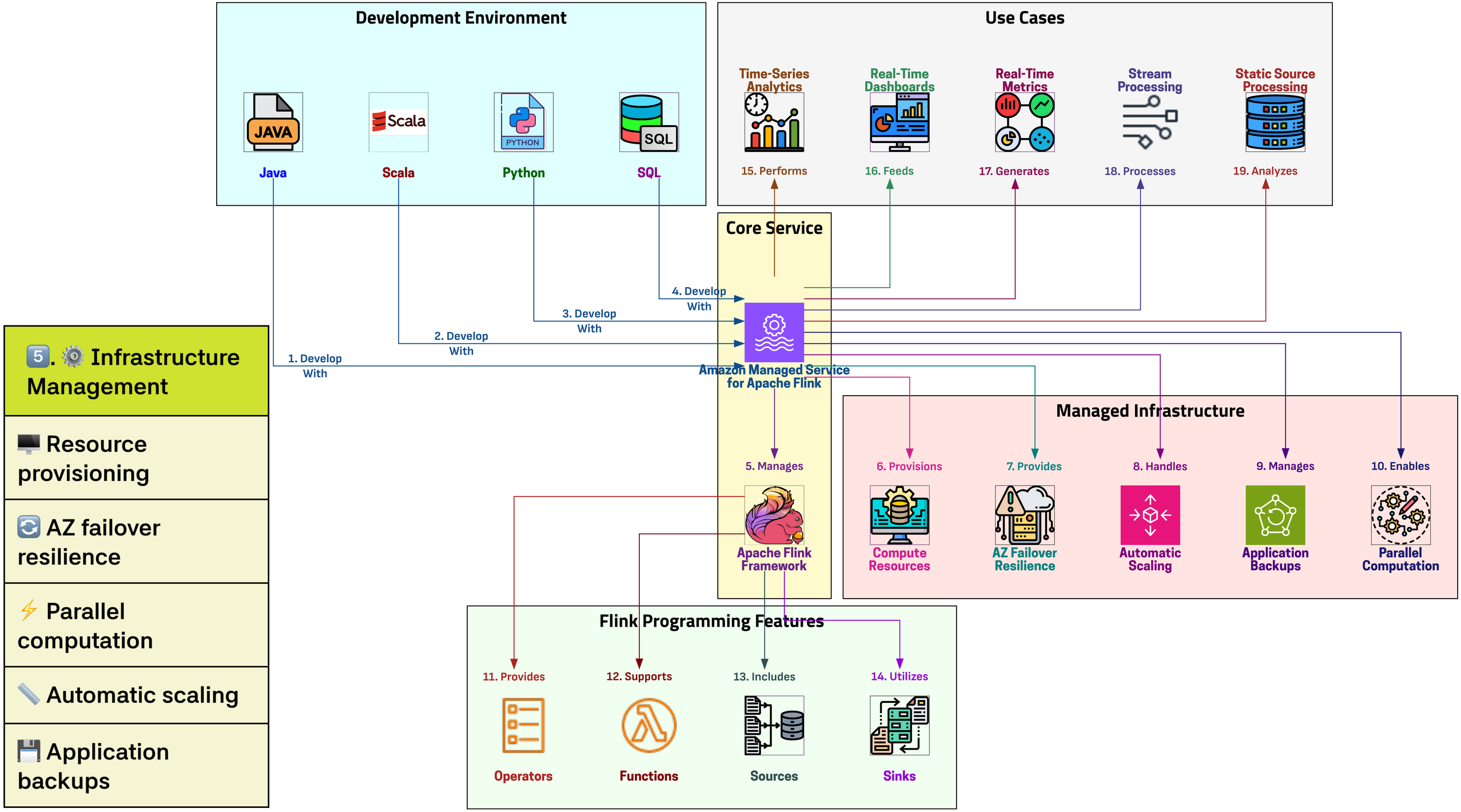
Amazon Managed Service for Apache Flink

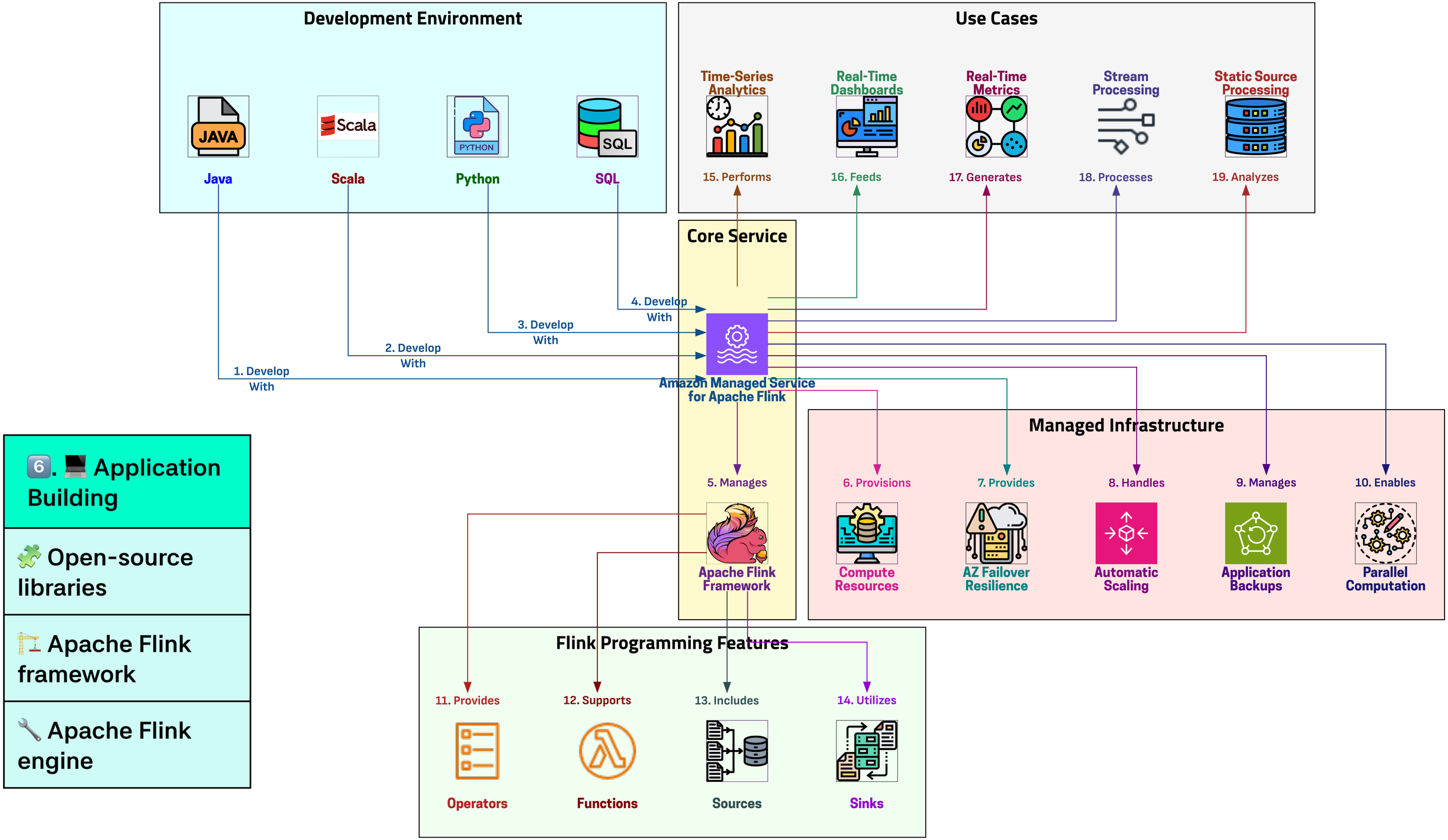

Development Environment
Java
Scala
Python
SQL
Use Cases
Time-Series Analytics
Real-Time Dashboards
Real-Time Metrics
Stream Processing
Static Source Processing
15. Performs
16. Feeds
17. Generates
18. Processes
19. Analyzes
Core Service
4. Develop With
3. Develop With
2. Develop With
1. Develop With
Amazon Managed Service for Apache Flink
5. Manages
Apache Flink Framework
Managed Infrastructure
6. Provisions
7. Provides
8. Handles
9. Manages
10. Enables
Compute Resources
AZ Failover Resilience
Automatic Scaling
Application Backups
Parallel Computation
Flink Programming Features
11. Provides
12. Supports
13. Includes
14. Utilizes
Operators
Functions
Sources
Sinks
1️⃣. 🚀 Stream Processing
🌊 Framework for processing data streams
🔄 Engine for continuous data analysis


Development Environment
Java
Scala
Python
SQL
Use Cases
Time-Series Analytics
Real-Time Dashboards
Real-Time Metrics
Stream Processing
Static Source Processing
15. Performs
16. Feeds
17. Generates
18. Processes
19. Analyzes
Core Service
4. Develop With
3. Develop With
2. Develop With
1. Develop With
Amazon Managed Service for Apache Flink
5. Manages
Apache Flink Framework
Managed Infrastructure
6. Provisions
7. Provides
8. Handles
9. Manages
10. Enables
Compute Resources
AZ Failover Resilience
Automatic Scaling
Application Backups
Parallel Computation
Flink Programming Features
11. Provides
12. Supports
13. Includes
14. Utilizes
Operators
Functions
Sources
Sinks
2. ⏱️ Real-time Analytics
📊 Time-series analytics solutions
📈 Real-time dashboards
🔢 Metrics generation


Development Environment
Java
Scala
Python
SQL
Use Cases
Time-Series Analytics
Real-Time Dashboards
Real-Time Metrics
Stream Processing
Static Source Processing
15. Performs
16. Feeds
17. Generates
18. Processes
19. Analyzes
Core Service
4. Develop With
3. Develop With
2. Develop With
1. Develop With
Amazon Managed Service for Apache Flink
5. Manages
Apache Flink Framework
Managed Infrastructure
6. Provisions
7. Provides
8. Handles
9. Manages
10. Enables
Compute Resources
AZ Failover Resilience
Automatic Scaling
Application Backups
Parallel Computation
Flink Programming Features
11. Provides
12. Supports
13. Includes
14. Utilizes
Operators
Functions
Sources
Sinks
3. Open Source
Freely available framework
Compatible libraries
Java support
Scala support
Python support
SQL support


Development Environment
Java
Scala
Python
SQL
Use Cases
Time-Series Analytics
Real-Time Dashboards
Real-Time Metrics
Stream Processing
Static Source Processing
15. Performs
16. Feeds
17. Generates
18. Processes
19. Analyzes
Core Service
4. Develop With
3. Develop With
2. Develop With
1. Develop With
Amazon Managed Service for Apache Flink
5. Manages
Apache Flink Framework
Managed Infrastructure
6. Provisions
7. Provides
8. Handles
9. Manages
10. Enables
Compute Resources
AZ Failover Resilience
Automatic Scaling
Application Backups
Parallel Computation
Flink Programming Features
11. Provides
12. Supports
13. Includes
14. Utilizes
Operators
Functions
Sources
Sinks
4️⃣. 🔌 Key Components
⚙️ Operators for data transformation
🛠️ Functions for data processing
📥 Sources for data input
📤 Sinks for data output
🧰 Advanced data manipulation tools


Development Environment
Java
Scala
Python
SQL
Use Cases
Time-Series Analytics
Real-Time Dashboards
Real-Time Metrics
Stream Processing
Static Source Processing
15. Performs
16. Feeds
17. Generates
18. Processes
19. Analyzes
Core Service
4. Develop With
3. Develop With
2. Develop With
1. Develop With
Amazon Managed Service for Apache Flink
5. Manages
Apache Flink Framework
Managed Infrastructure
6. Provisions
7. Provides
8. Handles
9. Manages
10. Enables
Compute Resources
AZ Failover Resilience
Automatic Scaling
Application Backups
Parallel Computation
Flink Programming Features
11. Provides
12. Supports
13. Includes
14. Utilizes
Operators
Functions
Sources
Sinks
5. Infrastructure Management
Resource provisioning
AZ failover resilience
Parallel computation
Automatic scaling
Application backups


Development Environment
Java
Scala
Python
SQL
Use Cases
Time-Series Analytics
Real-Time Dashboards
Real-Time Metrics
Stream Processing
Static Source Processing
15. Performs
16. Feeds
17. Generates
18. Processes
19. Analyzes
Core Service
4. Develop With
3. Develop With
2. Develop With
1. Develop With
Amazon Managed Service for Apache Flink
5. Manages
Apache Flink Framework
Managed Infrastructure
6. Provisions
7. Provides
8. Handles
9. Manages
10. Enables
Compute Resources
AZ Failover Resilience
Automatic Scaling
Application Backups
Parallel Computation
Flink Programming Features
11. Provides
12. Supports
13. Includes
14. Utilizes
Operators
Functions
Sources
Sinks
6. Application Building
Open-source libraries
Apache Flink framework
Apache Flink engine

Apache Flink vs Apache Flink Studio

1️⃣ . 🔄 **Two service options**: 🛠️ Managed Service for Apache Flink, 🖥️ Managed Service for Apache Flink Studio

2️⃣ . 🧰 **Managed Service for Apache Flink capabilities**: 👷 Build applications using IDE of choice, 🔤 Programming language support: ☕ Java, 📜 Scala, 🐍 Python, 📊 API options: 🌊 Datastream API, 📋 Table API

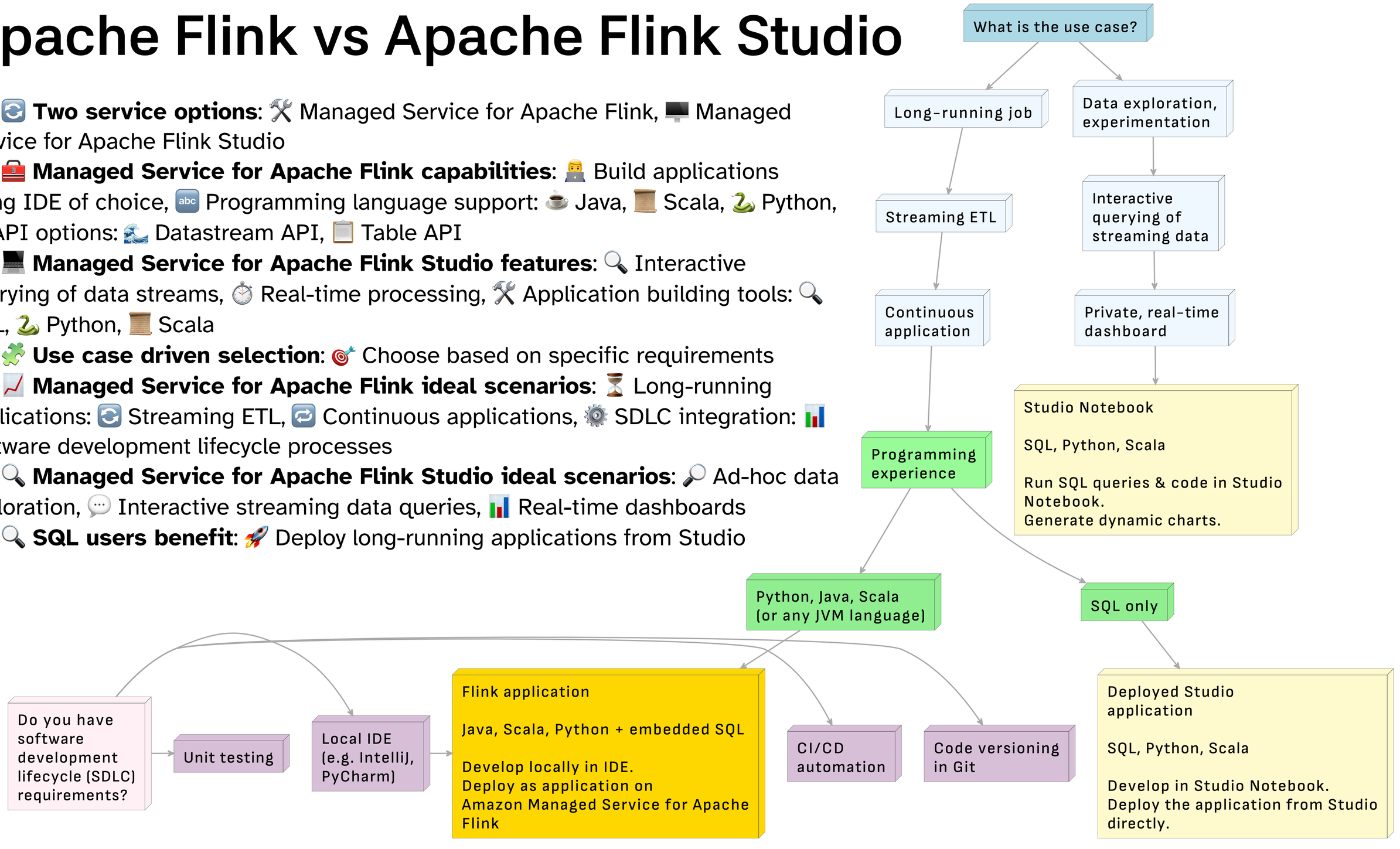
3️⃣ . 💻 **Managed Service for Apache Flink Studio features**: 🔍 Interactive querying of data streams, ⏱️ Real-time processing, 🛠️ Application building tools: 🔍 SQL, 🐍 Python, 📜 Scala

4️⃣ . 🧩 **Use case driven selection**: 🎯 Choose based on specific requirements

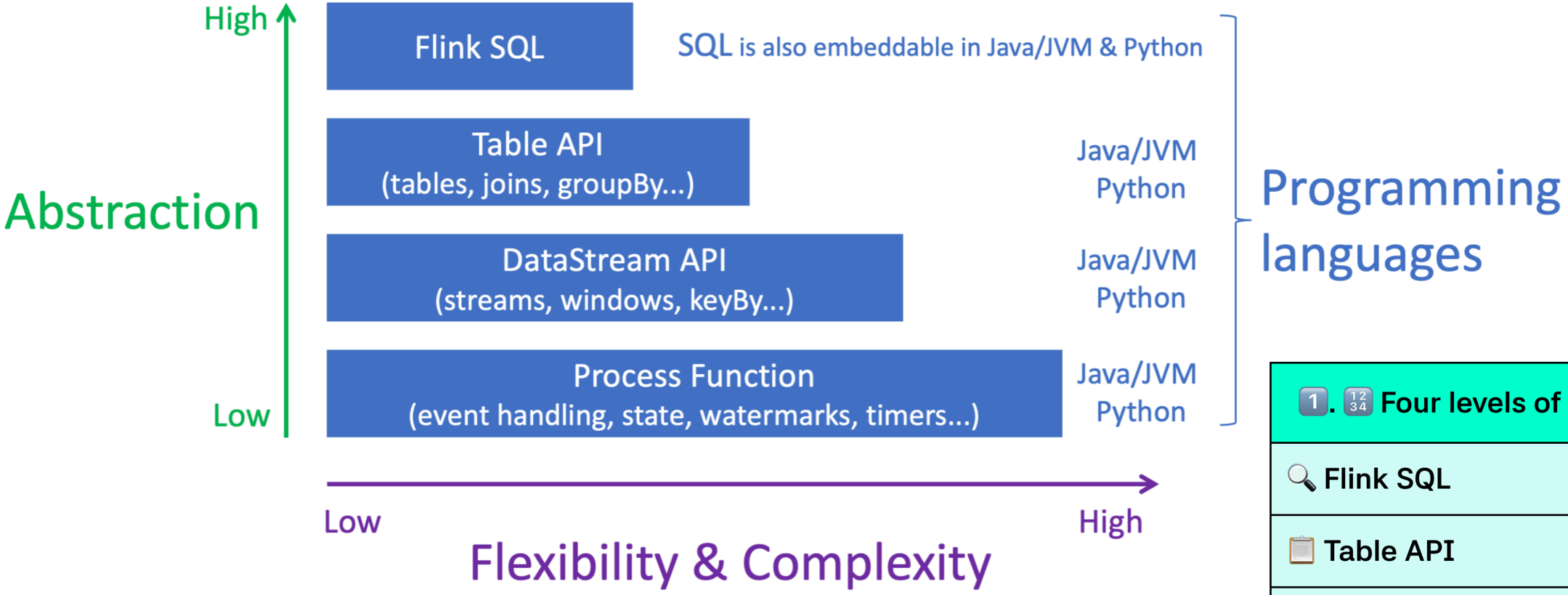
5️⃣ . 📈 **Managed Service for Apache Flink ideal scenarios**: ⏳ Long-running applications: 🔄 Streaming ETL, 🔁 Continuous applications, ⚙️ SDLC integration: 📊 Software development lifecycle processes

6️⃣ . 🔍 **Managed Service for Apache Flink Studio ideal scenarios**: 🔎 Ad-hoc data exploration, 💬 Interactive streaming data queries, 📊 Real-time dashboards

7️⃣ . 🔍 **SQL users benefit**: 🚀 Deploy long-running applications from Studio

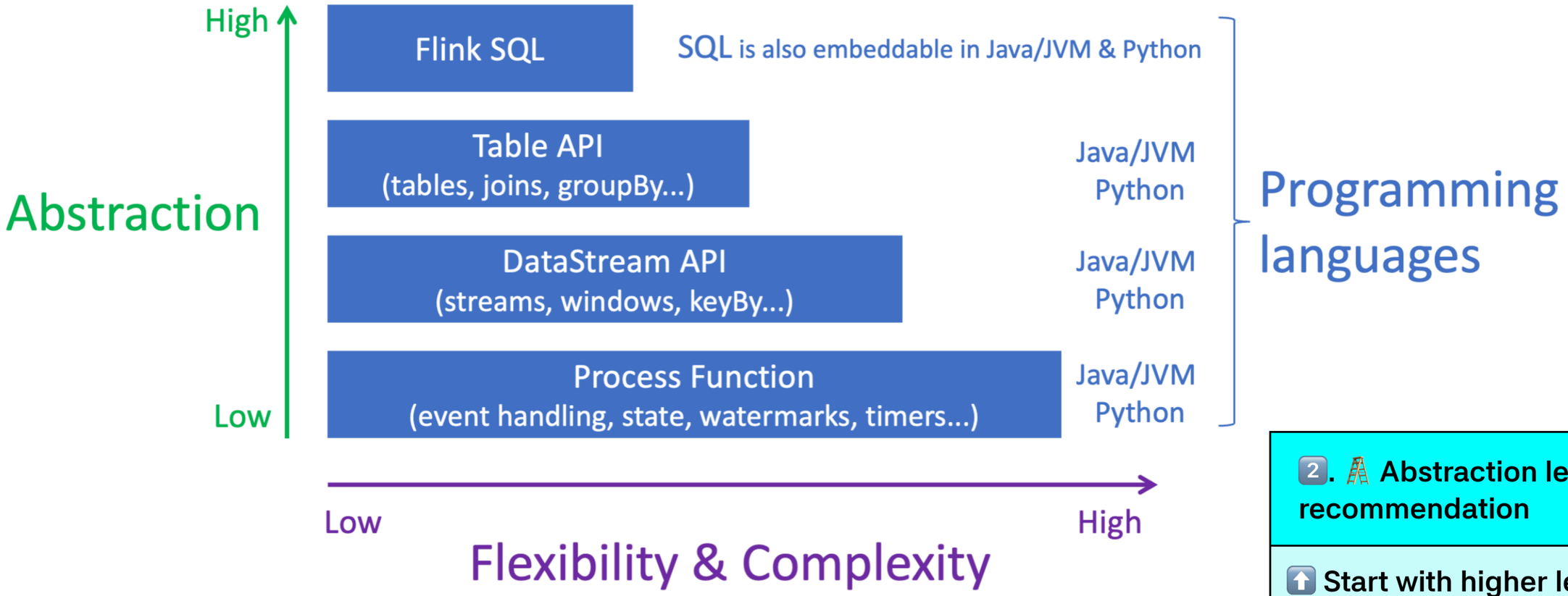


Apache Flink APIs



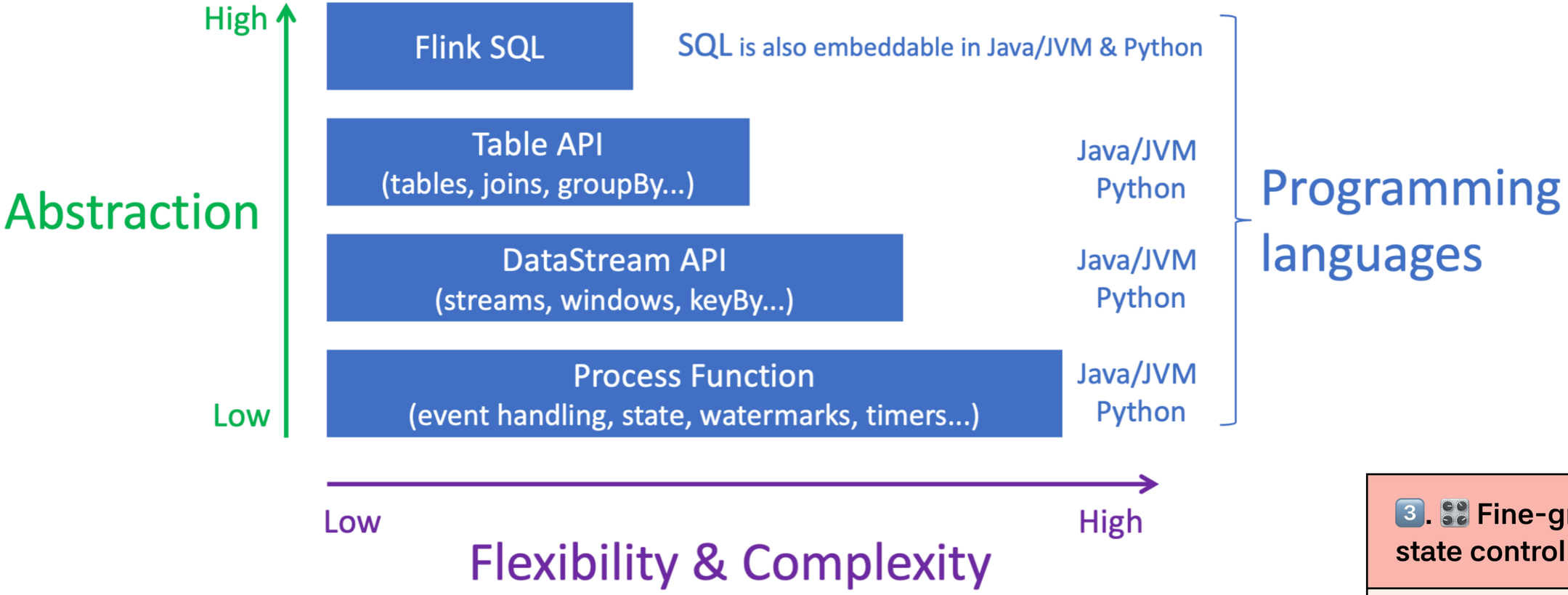
1️⃣. 🔢 Four levels of API abstraction	
🔍 Flink SQL	
📋 Table API	
🌊 DataStream API	
⚙️ Process Function	🔄 Used with DataStream API
☁️ All supported in Amazon Managed Service	

Apache Flink APIs



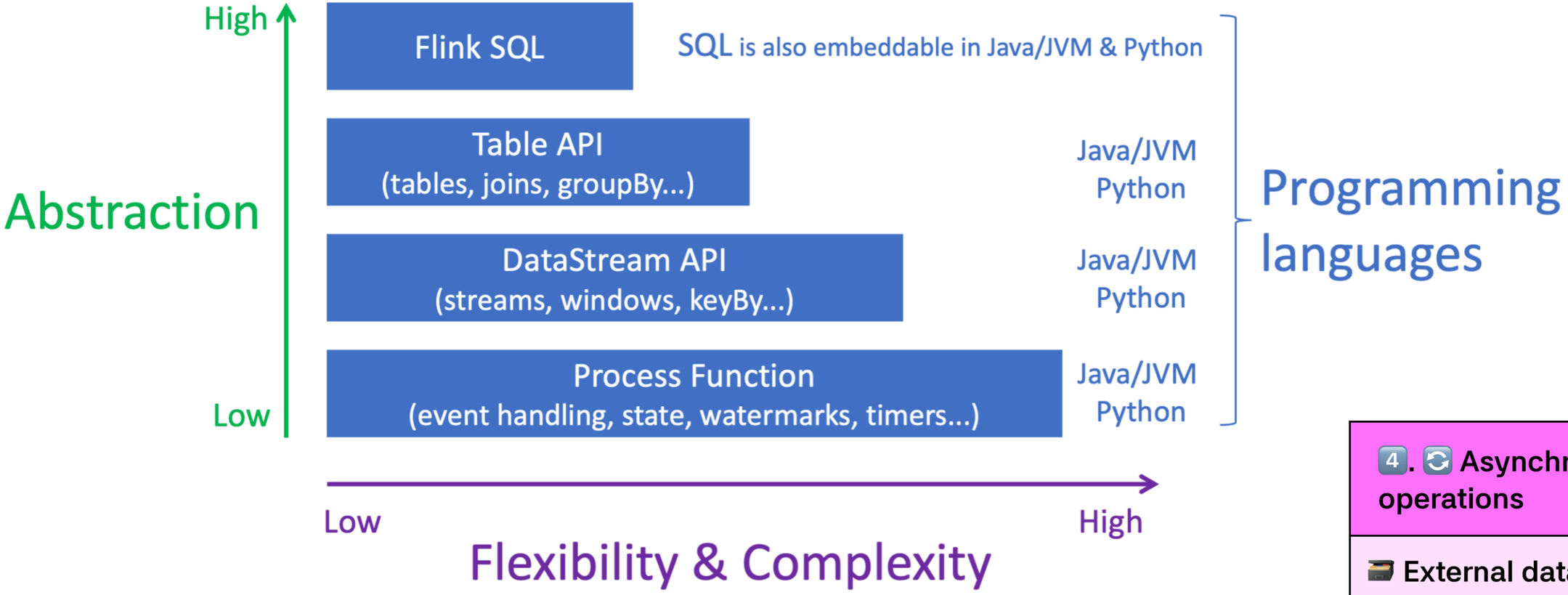
2️⃣. 🪜 Abstraction level recommendation	
⬆️ Start with higher levels when possible	
🌟 Some features exclusive to DataStream API	☕ Java support
	🐍 Python support
	📜 Scala support

Apache Flink APIs



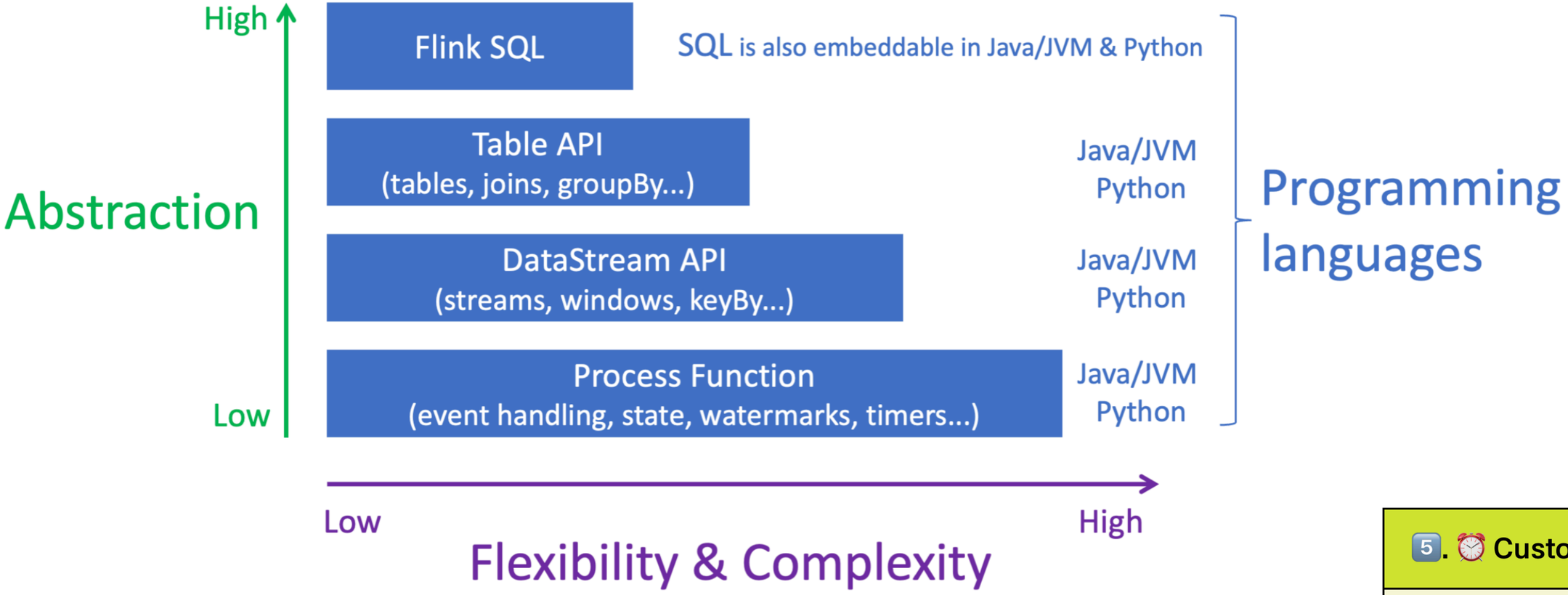
3️⃣. 🎛️ Fine-grained state control
💾 State management capabilities
🔧 Detailed control options

Apache Flink APIs



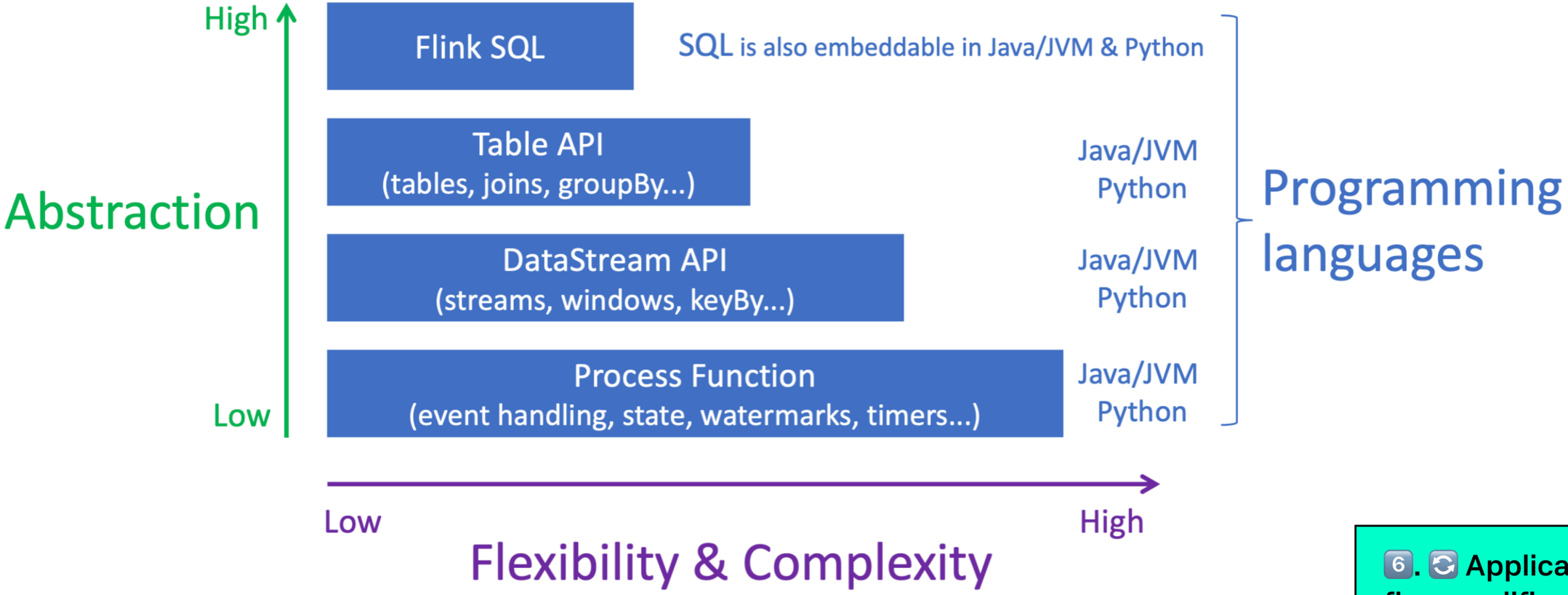
4️⃣. 🔄 Asynchronous operations
🗃️ External database calls
🔌 Endpoint connections
🧠 Inference operations

Apache Flink APIs



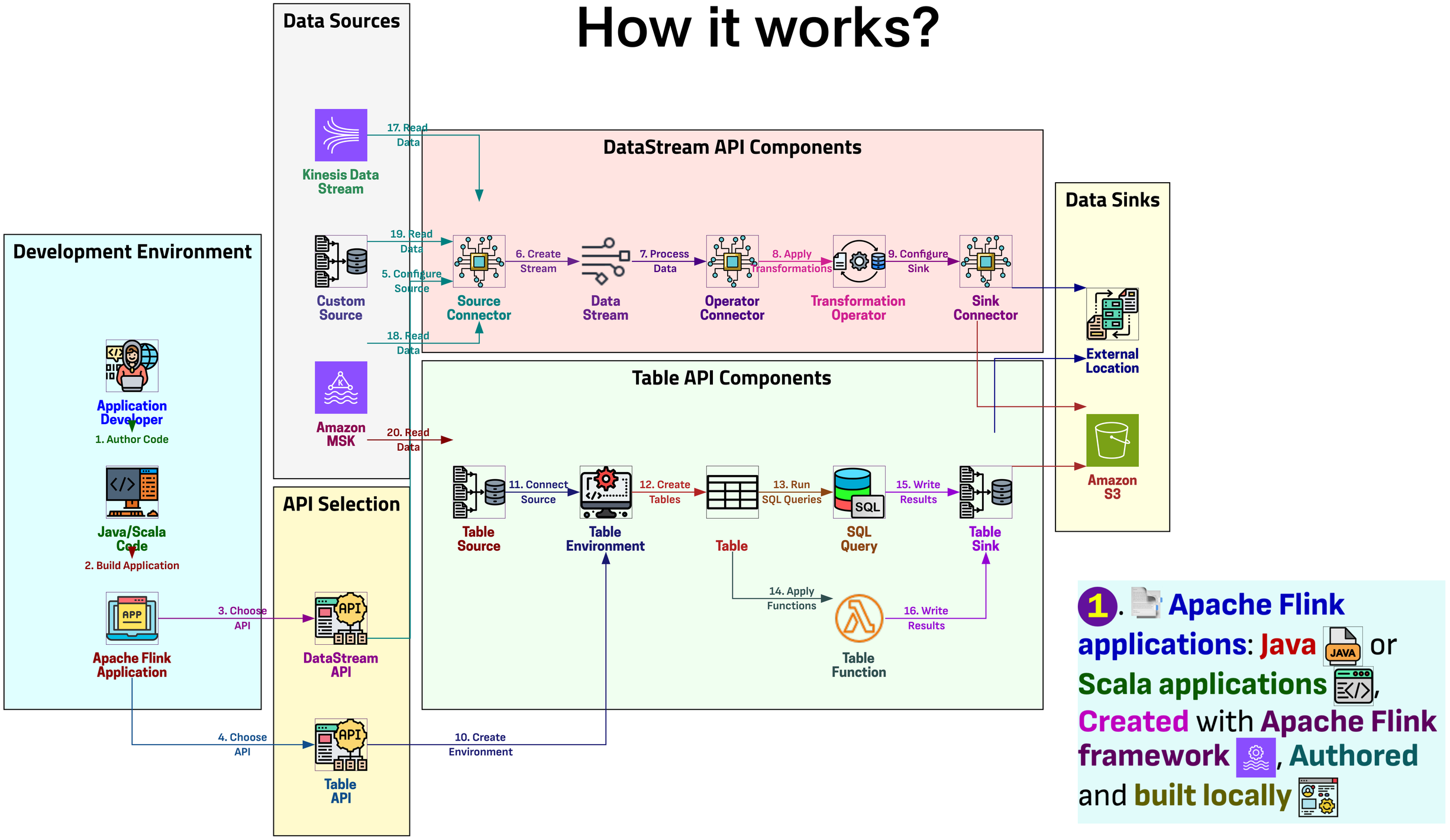
5️⃣. ⏰ Custom timers
🪟 Specialized windowing
⏱️ Late event handling

Apache Flink APIs



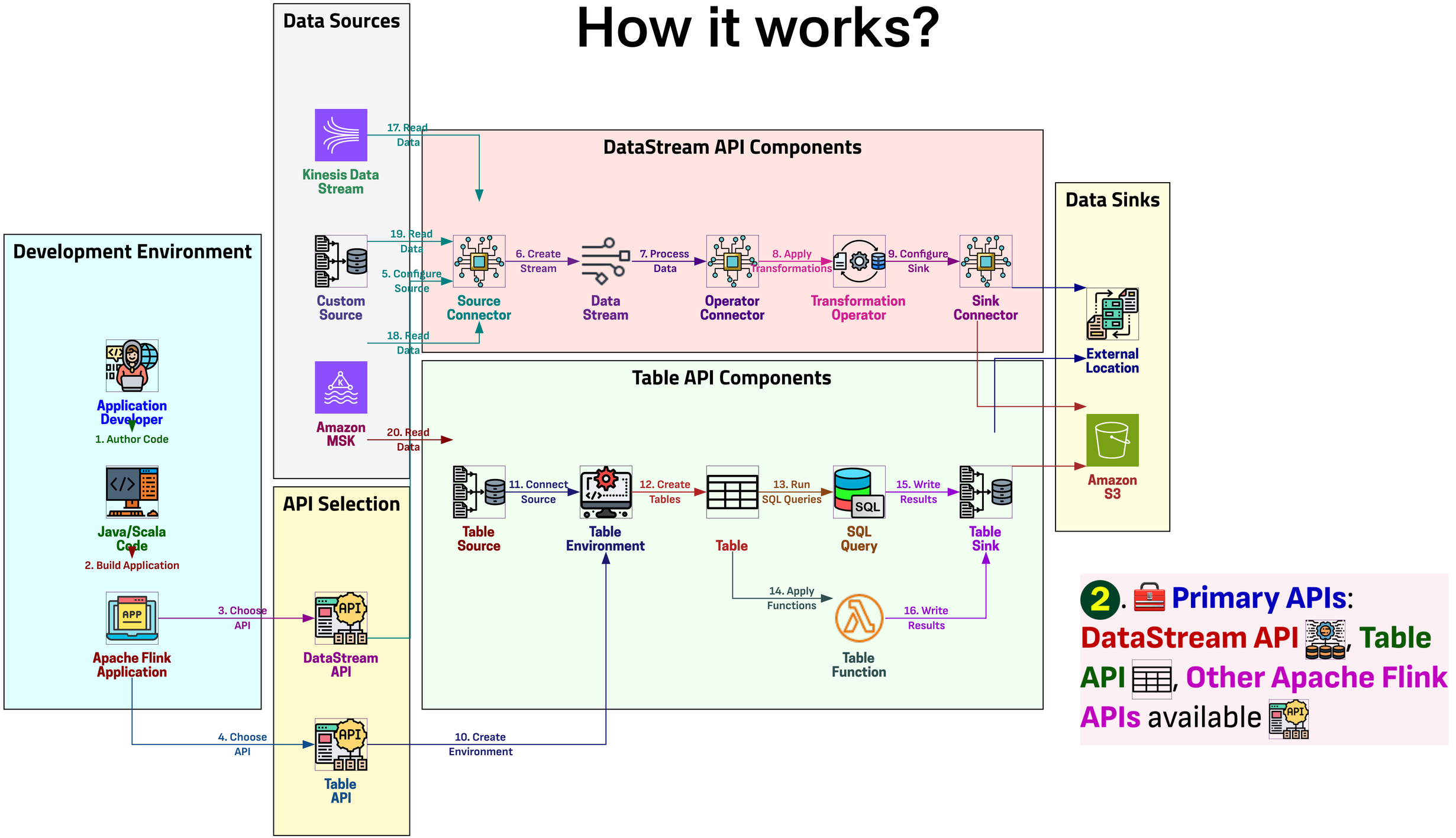
6️⃣. 🔄 Application flow modification
💾 Without resetting state
🔀 Greater flexibility

How it works?



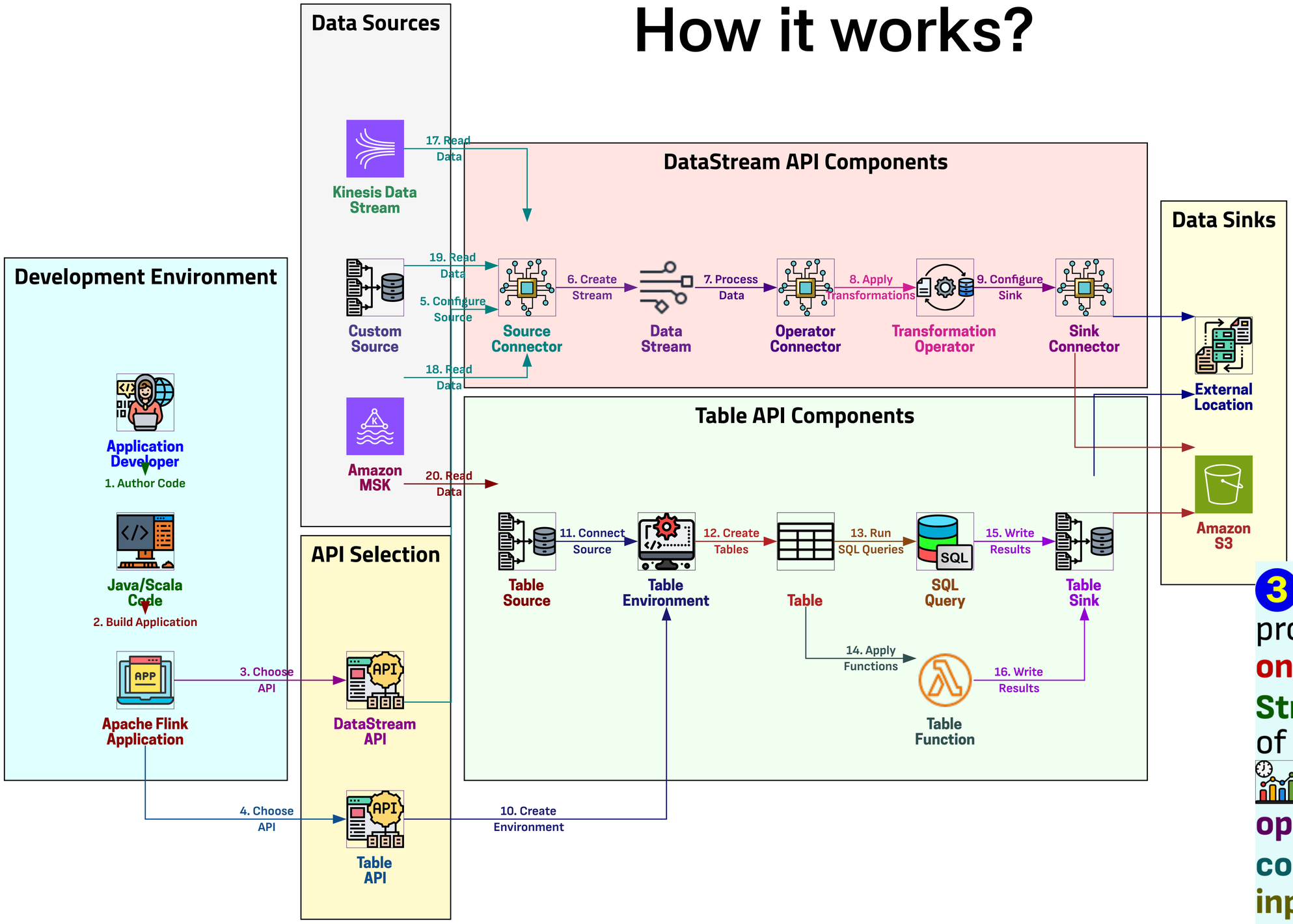
1. Apache Flink applications: Java or Scala applications, Created with Apache Flink framework, Authored and built locally

How it works?



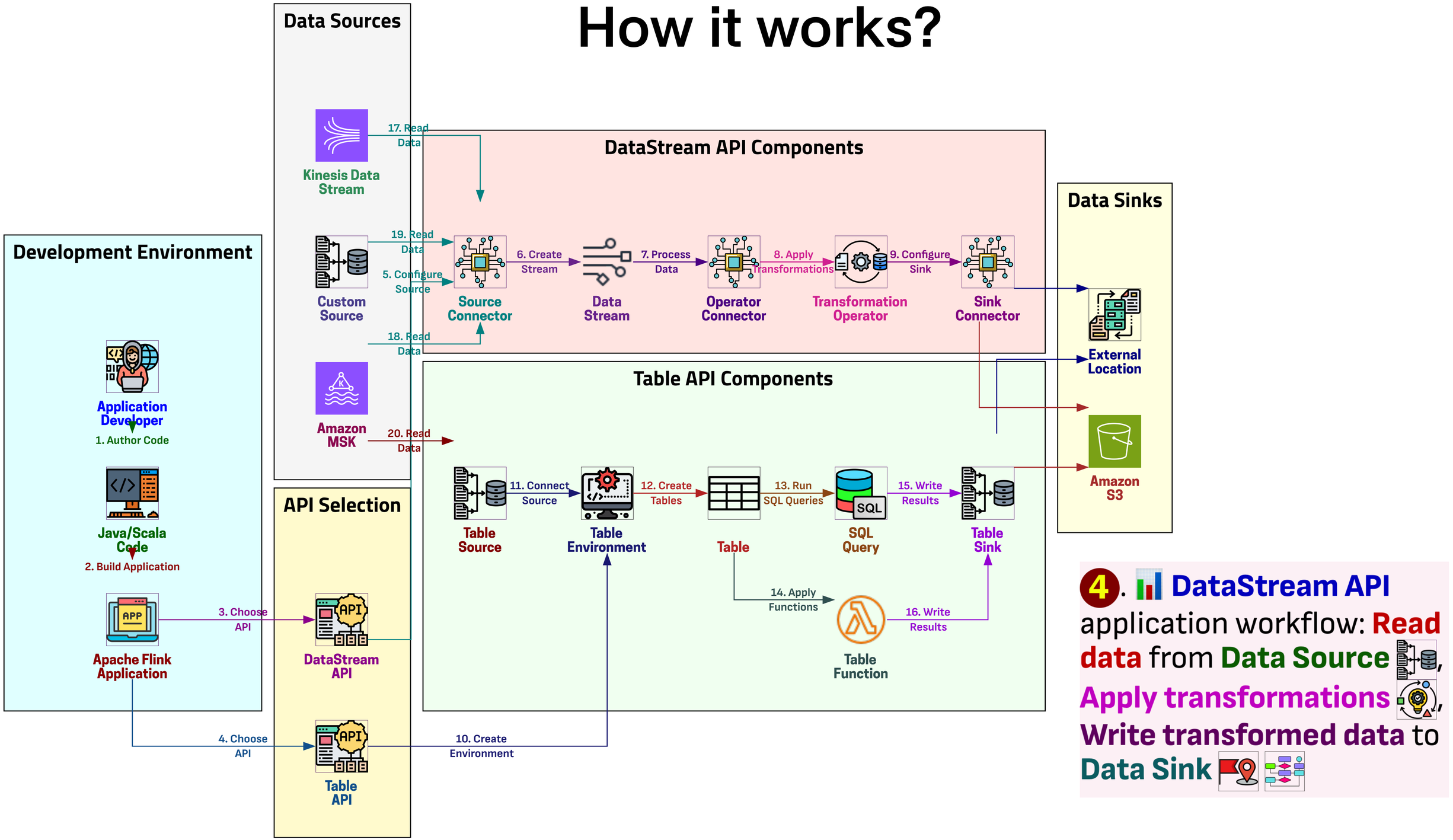
2. 🧰 Primary APIs: DataStream API, Table API, Other Apache Flink APIs available

How it works?



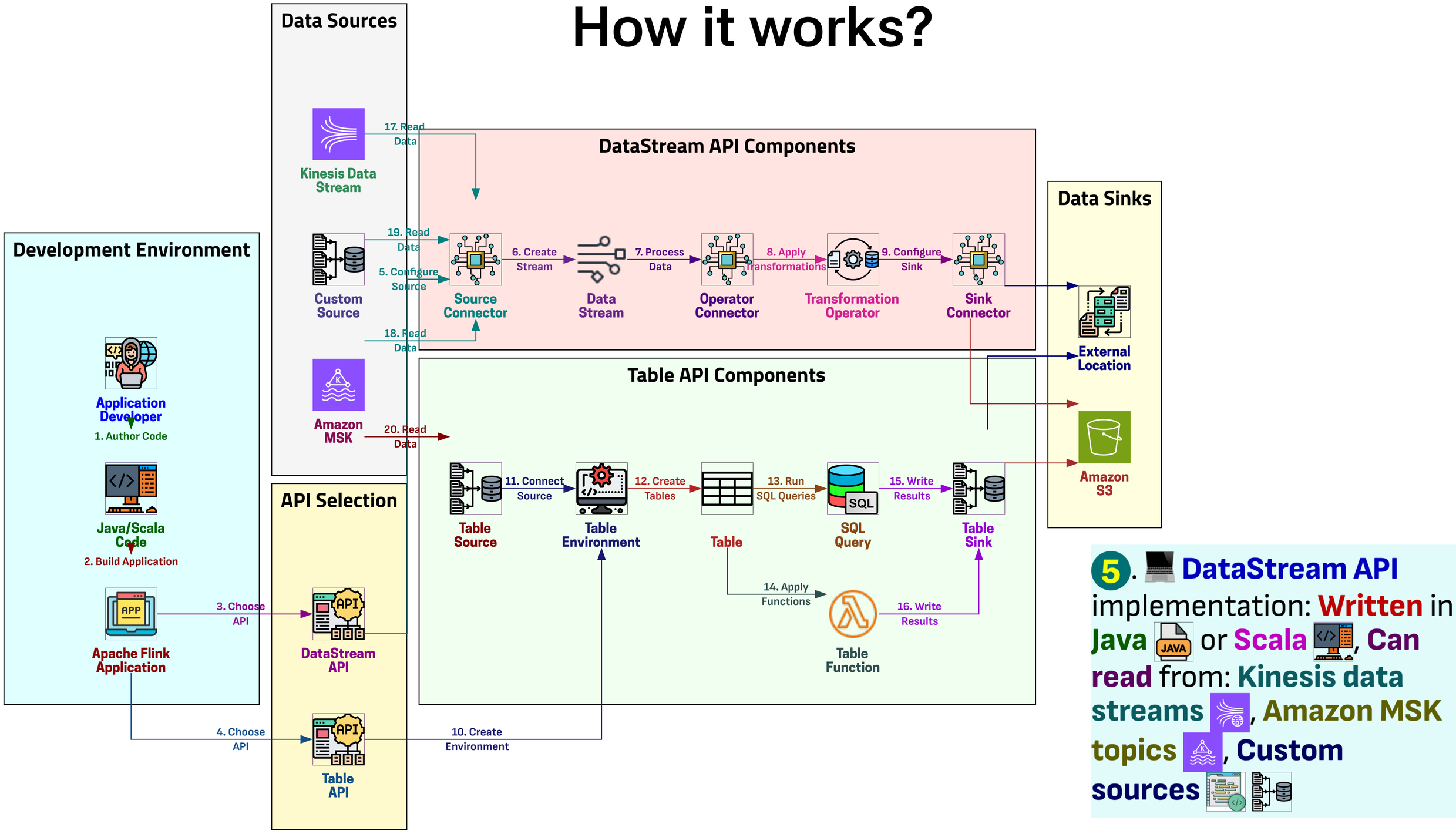
3. 🌊 **DataStream API** programming model: **Based on data streams** 🗄️, **Structured representation** of **continuous data flow** 📊, **Transformation operators** ⚙️🗄️, **Processing components** transforming **input streams** 🔀

How it works?



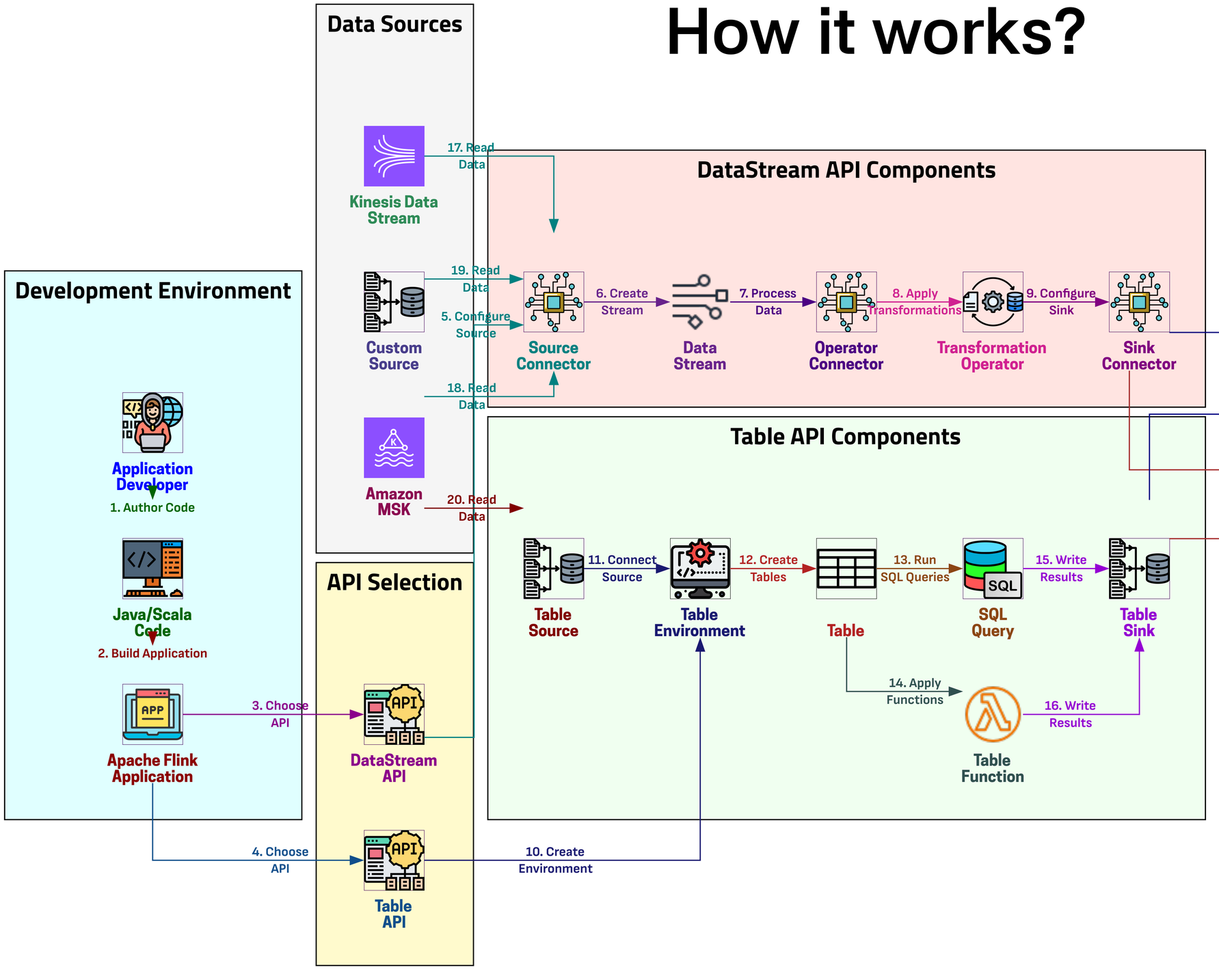
4. 📊 DataStream API application workflow: **Read data** from **Data Source** , **Apply transformations** , **Write transformed data** to **Data Sink**

How it works?



5. 💻 DataStream API implementation: Written in Java or Scala, Can read from: Kinesis data streams, Amazon MSK topics, Custom sources

How it works?

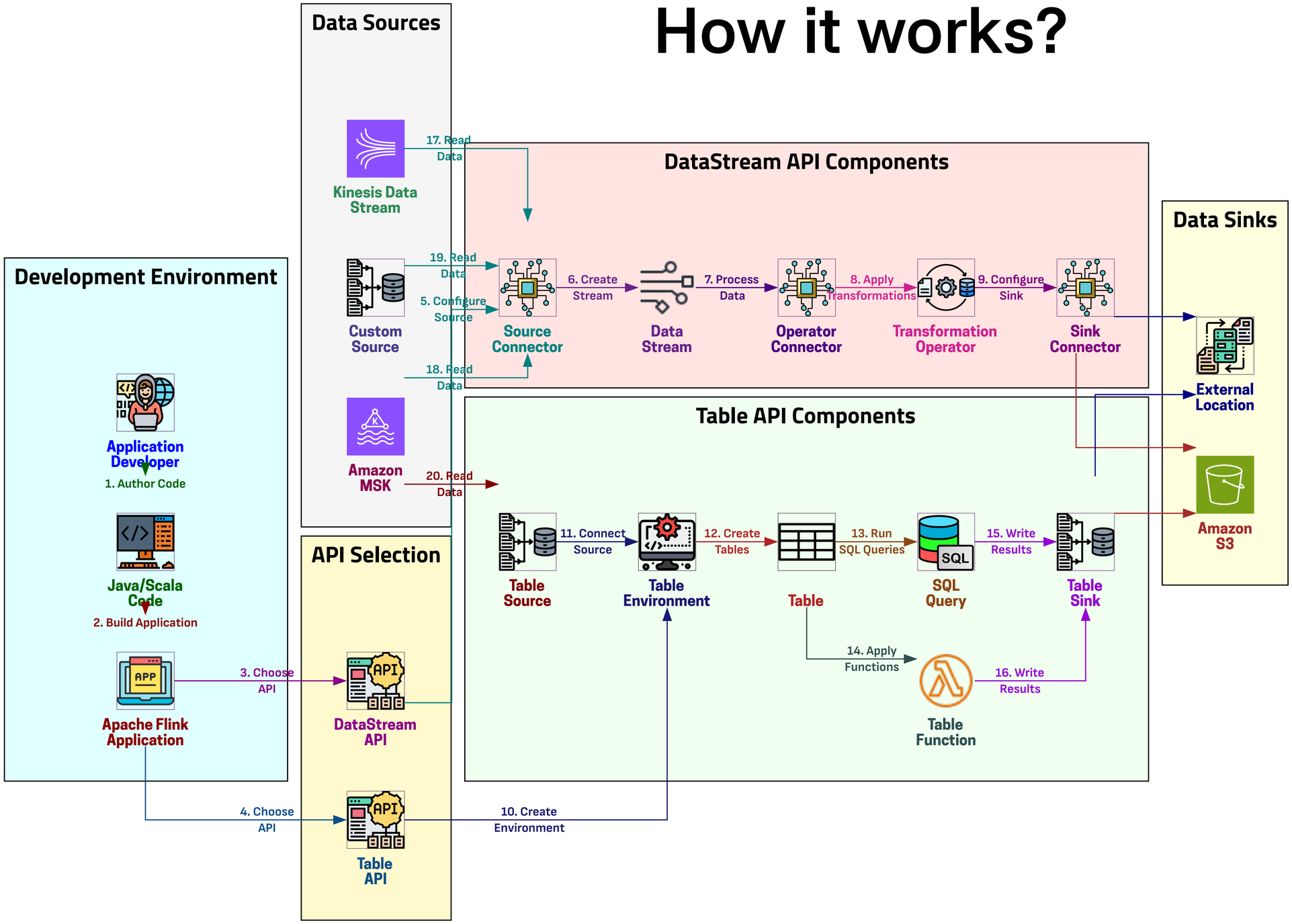


6. 🔌 **Connector types**:

Source connectors : **Reading external data** ,

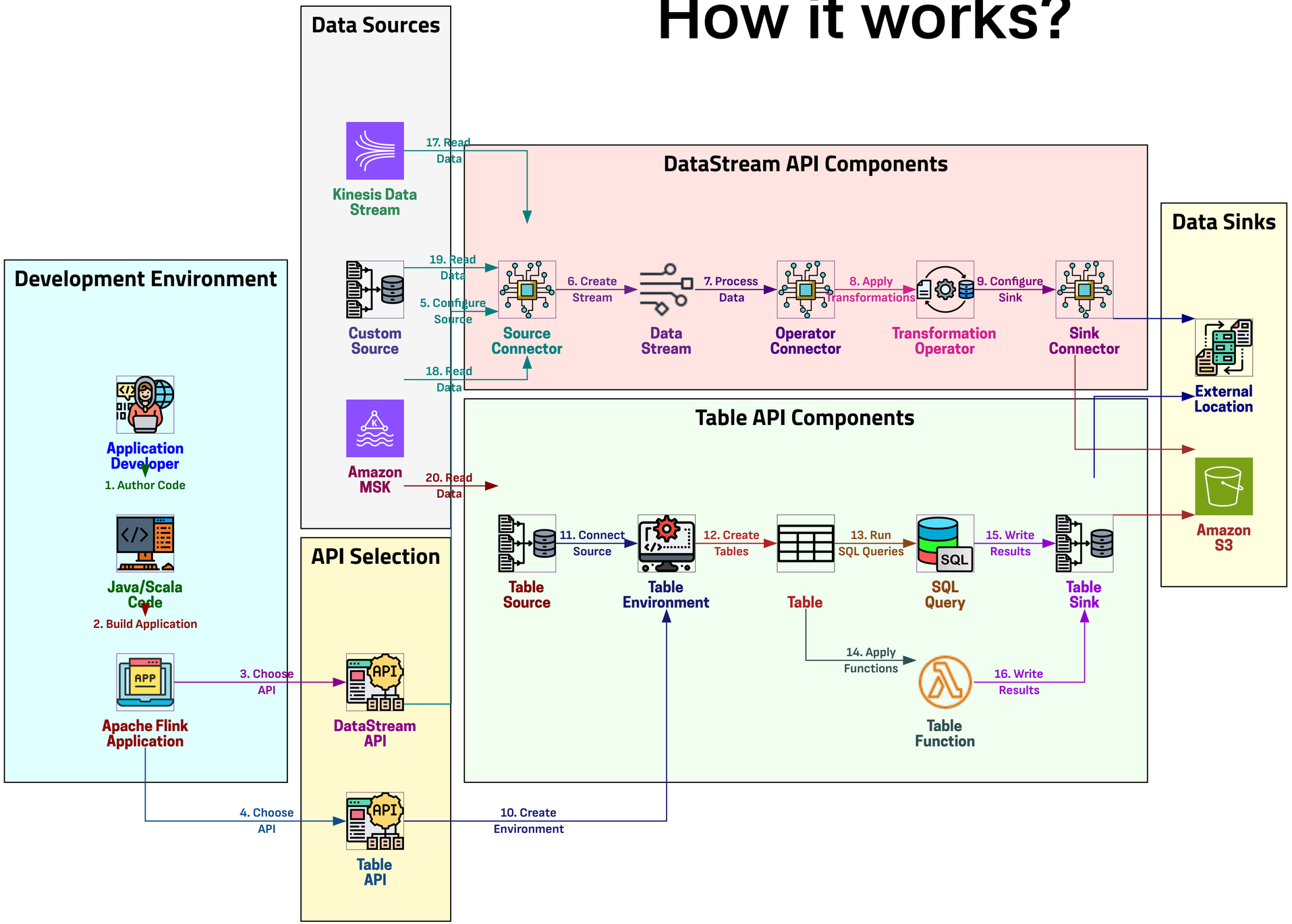
Sink connectors : **Writing** to **external locations** , **Operator connectors** : **Processing data** within **application**

How it works?



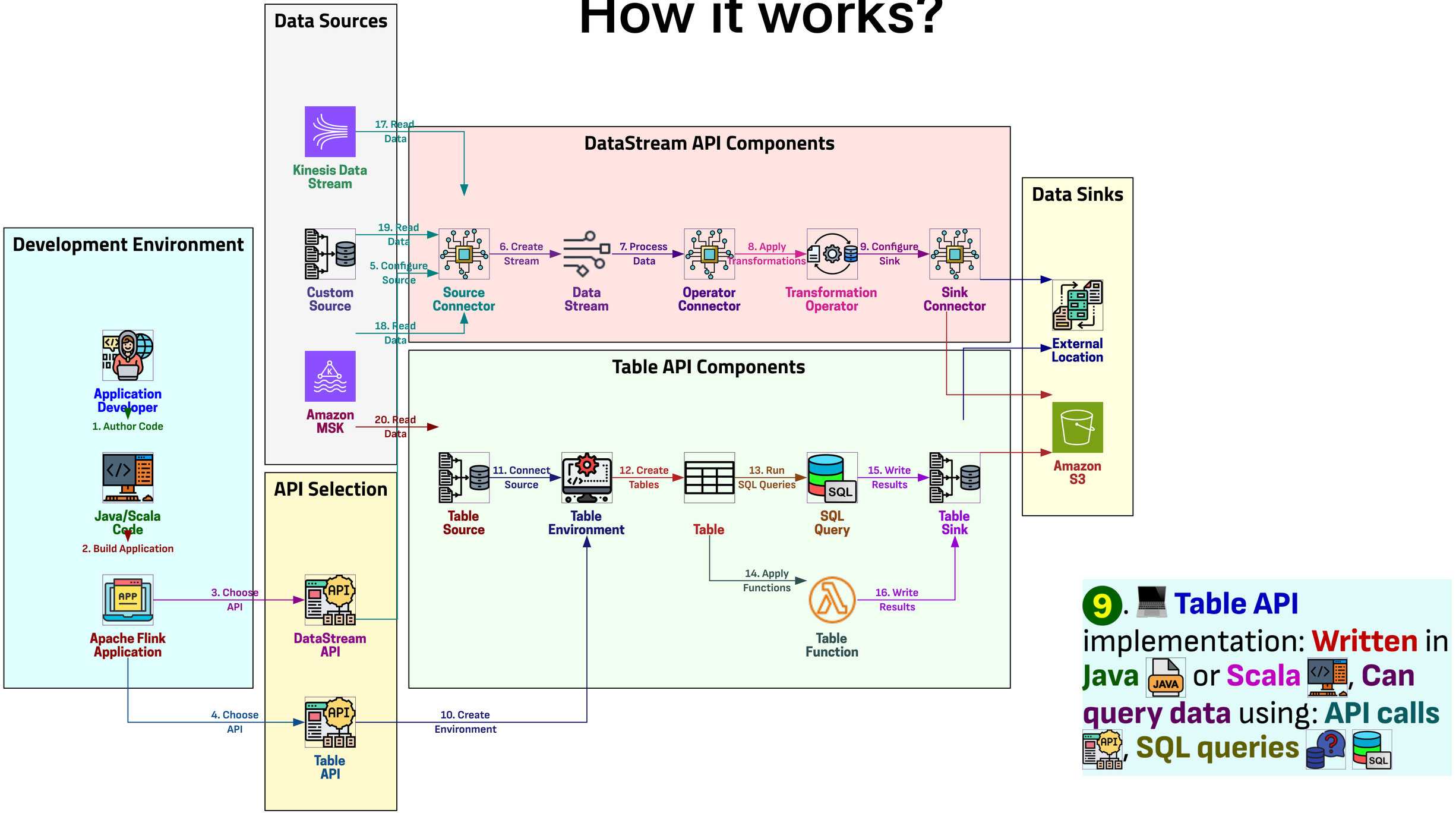
7. 📋 **Table API** programming model components: **Table Environment** : **Interface** to **underlying data** , **Tables** : **Objects** providing **access** to **SQL tables/views** , **Table Sources** : **Reading** from **external sources** , **Table Functions** : **Transforming data** , **Table Sinks** : **Writing** to **external locations**

How it works?



8. 📋 **Table API** application workflow: **Create TableEnvironment**, **Create tables**: **Using SQL queries**, **Using API functions**, **Run queries** on **tables**, **Apply transformations** on **results**, **Write results** to **Table Sink**

How it works?



9. 💻 Table API implementation: Written in Java or Scala, Can query data using: API calls, SQL queries

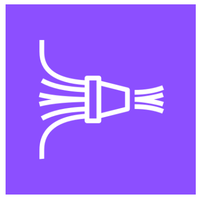
Connectors (Sources & Sinks)

1. Apache Kafka (source/sink)

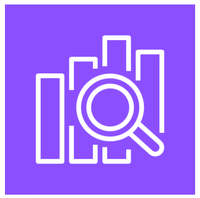
2. Apache Cassandra (sink)

3. Amazon DynamoDB (sink)

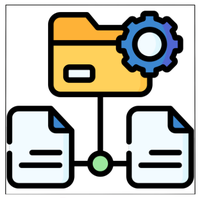
4. Amazon Kinesis Data Streams (source/sink)



5. Amazon Kinesis Data Firehose (sink)



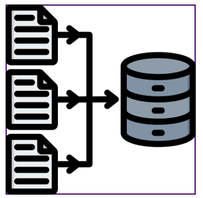
6. Elasticsearch (sink)



7. FileSystem (sink)

8. RabbitMQ (source/sink)

9. Google PubSub (source/sink)



10. Hybrid Source (source)

11. Apache NiFi (source/sink)

12. Apache Pulsar (source)

13. JDBC (sink)

Use Kinesis data streams

```java
// Configure the KinesisStreamsSource
Configuration sourceConfig = new Configuration();
sourceConfig.set(KinesisSourceConfigOptions.STREAM_INITIAL_POSITION, KinesisSourceConfigOptions.InitialPosition.TRIM_HORIZON);    1

// Create a new KinesisStreamsSource to read from specified Kinesis Stream.
KinesisStreamsSource<String> kdsSource =
        KinesisStreamsSource.<String>builder()    2
                .setStreamArn("arn:aws:kinesis:us-east-1:123456789012:stream/test-stream")    3
                .setSourceConfig(sourceConfig)    4
                .setDeserializationSchema(new SimpleStringSchema())    5
                .setKinesisShardAssigner(ShardAssignerFactory.uniformShardAssigner())    6
                .build();    7
```

1. Set the initial position for consuming the Kinesis stream to `TRIM_HORIZON`, which means that the consumer will start reading from the oldest record in the stream. This is optional, and if not specified, the default is `LATEST`, which means reading only new records that arrive after the consumer starts.
2. Initialize the builder pattern for creating a `KinesisStreamsSource` with a generic type parameter of `String`, indicating that the records will be deserialized into String objects.
3. Specify the ARN (Amazon Resource Name) of the Kinesis stream to read from. This uniquely identifies the stream within AWS and includes the AWS account ID, region, and stream name.
4. Apply the configuration settings defined earlier to the Kinesis stream source, which includes the initial position setting.
5. Set the deserialization schema to `SimpleStringSchema`, which will convert the raw Kinesis records into String objects. Different schemas can be used depending on the data format.
6. Configure the shard assigner strategy that determines how Kinesis shards are distributed among parallel subtasks. The `uniformShardAssigner` distributes shards evenly. This setting is optional and defaults to uniform assignment if not specified.
7. Finalize the builder pattern and create the `KinesisStreamsSource` instance with all the configured settings.

Create a KafkaSource

```java
KafkaSource<String> source = KafkaSource.<String>builder()
    .setBootstrapServers(brokers)    1
    .setTopics("input-topic")    2
    .setGroupId("my-group")    3
    .setStartingOffsets(OffsetsInitializer.earliest())    4
    .setValueOnlyDeserializer(new SimpleStringSchema())    5
    .build();

env.fromSource(source, WatermarkStrategy.noWatermarks(), "Kafka Source");    6
```

1. Specifies the Kafka broker addresses to connect to, using the `brokers` variable which should contain a comma-separated list of host:port pairs
2. Defines the Kafka topic to consume messages from, in this case "input-topic"
3. Sets the consumer group ID to "my-group", which is used by Kafka to track the consumption progress
4. Configures the consumer to start reading from the earliest available offset in the topic, ensuring no messages are missed
5. Specifies a deserializer that processes only the value portion of Kafka records and converts them to String objects
6. Adds the configured Kafka source to the Flink execution environment, with no watermark strategy for event time processing, and assigns the name "Kafka Source" for monitoring and visualization

Transform data using Operators

1️⃣. ⚙️ Apache Flink operators

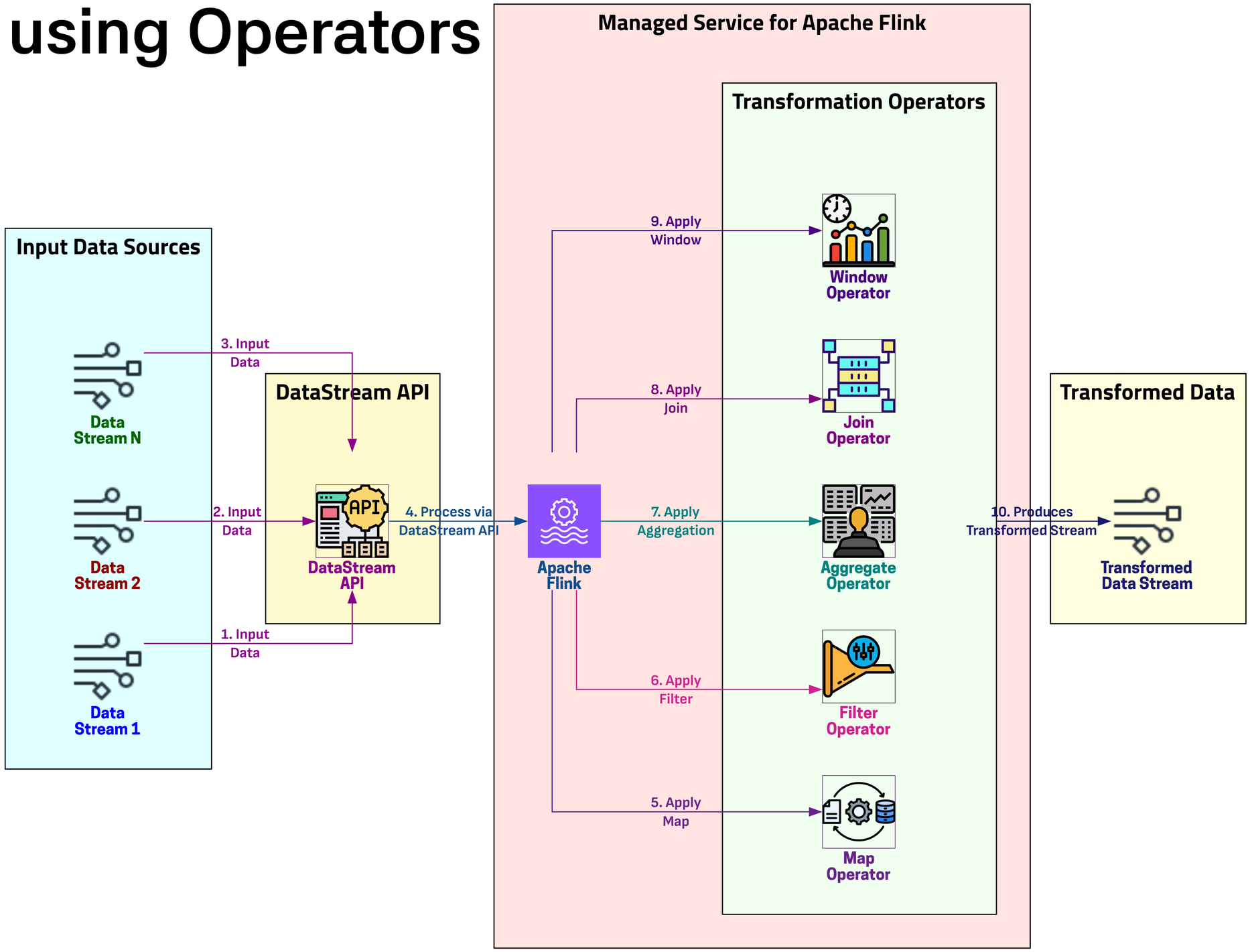
🔄 Transforms data streams

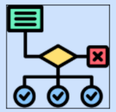
🔀 Converts one or more streams into new stream

✏️ Produces modified data

2️⃣. 🧰 Available operators

📦 25+ pre-built stream processing operators



#	Operator Name	Description
1	 Map	Takes one element  and produces one element  .
2	 FlatMap	Takes one element  and produces zero, one, or more elements  .
3	 Filter	Evaluates a boolean function  for each element and retains those for which the function returns true  .

4	KeyBy	Logically partitions a stream into disjoint partitions. All records with the same key are assigned to the same partition.
5	Reduce	A "rolling" reduce on a keyed data stream . Combines the current element with the last reduced value and emits the new value .

6	Window	Groups data in each key according to some characteristic (e.g., data that arrived within the last 5 seconds) on already partitioned KeyedStreams.
7	WindowAll	Groups all stream events according to some characteristic (e.g., data that arrived within the last 5 seconds) on regular DataStreams. Often a non-parallel transformation.

8	Window Apply	Applies a general function to the window as a whole .
9	WindowReduce	Applies a functional reduce function to the window and returns the reduced value .
10	Union	Merges two or more data streams , creating a new stream containing all elements from all input streams .

Use transform operators

```
1  DataStream<ObjectNode> output = input.map(     1
2      new MapFunction<ObjectNode, ObjectNode>() {     2
3          @Override
4          public ObjectNode map(ObjectNode value) throws Exception {     3
5              return value.put("TICKER", value.get("TICKER").asText() + " Company");     4
6          }
7      }
8  );
```

1. Creates a new transformed `DataStream` by applying a map operation to the `input` stream, which contains `ObjectNode` elements
2. Instantiates an anonymous class implementing the `MapFunction` interface, specifying both input and output types as `ObjectNode`
3. Implements the required `map` method that will be applied to each element in the input stream
4. Modifies the incoming `ObjectNode` by appending " Company" to the value of the "TICKER" field and returns the modified node

Use aggregation operators

```
DataStream<ObjectNode> output = input.keyBy(node -> node.get("TICKER").asText())     1
    .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))     2
    .reduce((node1, node2) -> {     3
        double priceTotal = node1.get("PRICE").asDouble() + node2.get("PRICE").asDouble();     4
        node1.replace("PRICE", JsonNodeFactory.instance.numberNode(priceTotal));     5
    return node1;     6
});
```

1. Partitions the input `DataStream` by the value of the "TICKER" field, ensuring that all records with the same ticker symbol are processed by the same parallel task
2. Applies a tumbling window of 5 seconds in processing time, grouping records that arrive within the same 5-second interval
3. Defines a reduce function that will be applied to all records within each window for each key
4. Calculates the sum of the "PRICE" values from two `ObjectNode` records by extracting the double values
5. Updates the first node by replacing its "PRICE" field with the newly calculated total using `JsonNodeFactory` to create a new number node
6. Returns the modified first node as the result of the reduction, accumulating values as the reducer processes more elements in the window

Thanks for Watching