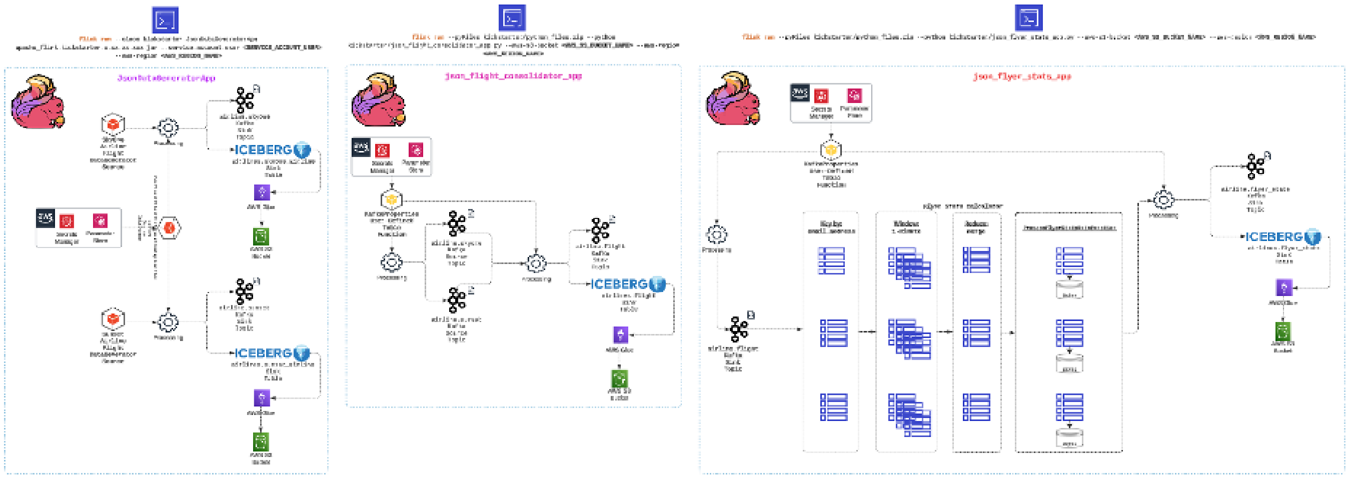
Apache Flink v2.1 Kickstarter

This project demonstrates the **real-world power of Apache Flink**—a distributed stream processing engine built for **low-latency, high-throughput, and exactly-once stateful computation**. As a core component of the **signalRoom technology stack**, it highlights how Flink's advanced features—such as event-time semantics, checkpointing, watermarks, and dynamic scaling—can be leveraged to build resilient, production-grade streaming pipelines. Staying true to our mission of **empowering developers through practical, open knowledge**, this project extends the concepts from our *"Building Apache Flink Applications in Java"* series into runnable, end-to-end examples that showcase how to design, deploy, and optimize Flink jobs for real business impact.

Embarking on this journey, you'll explore how Flink seamlessly integrates with modern data infrastructures, including **Confluent Cloud for Apache Flink**, **AWS Secrets Manager**, **AWS Systems Manager Parameter Store**, and **Apache Iceberg tables in Snowflake**. This project not only illustrates Flink's capabilities in handling complex event processing and state management but also demonstrates best practices for secure configuration management and scalable data storage.



Flink App	Description

Flink App	Description
`Data Generator App`	This Apache Flink application simulates realistic flight activity for the fictional airlines **Sunset Air** and **Sky One Airlines**, showcasing seamless integration across modern data streaming technologies. Using Flink's high-throughput processing capabilities, flight events are continuously published to dedicated **Kafka topics** (`airline.sunset` and `airline.skyone`), enabling real-time monitoring, transformation, and analytics. In parallel, the same synthetic data is written to **Apache Iceberg tables** (`apache_kickstarter.airlines.sunset_airline` and `apache_kickstarter.airlines.skyone_airline`) stored in **AWS S3**, creating a unified, scalable foundation for both streaming and historical analysis. Built entirely in **Java**, this implementation overcomes current **PyFlink** limitations—namely, the absence of a Python-native data generator source—while demonstrating best practices for stateful stream processing and unified batch/stream architecture. The result is a powerful, extensible data pipeline that bridges **real-time event processing** and **analytical persistence**, illustrating how Flink can serve as the backbone of next-generation streaming systems.
`Flight Consolidator App`	This Apache Flink application ingests flight event data from the **`airline.sunset`** and **`airline.skyone`** Kafka topics, transforms and standardizes the records into a unified **`airline.flight`** Kafka topic, and persists them to the **`apache_kickstarter.airlines.flight`** Apache Iceberg table. Implemented in both **Java** and **Python**, it demonstrates cross-language stream processing, schema unification, and seamless integration between Kafka, Flink, and Iceberg for end-to-end real-time and analytical data workflows.
`Flyer Stats App`	This Apache Flink application consumes flight event data from the **`airline.flight`** Kafka topic, performs real-time aggregations to compute **flyer-level statistics**, and publishes the results to the **`airline.flyer_stats`** Kafka topic as well as the **`apache_kickstarter.airlines.flyer_stats`** Apache Iceberg table. Implemented in both **Java** and **Python**, it demonstrates how to build unified, stateful streaming and analytical pipelines for continuous aggregation and historical insight.

Originally created by **Wade Waldron**, *Staff Software Practice Lead at Confluent Inc.*, these applications are built to showcase **Apache Flink's enterprise-grade capabilities** through realistic, production-scale scenarios. The workflow begins by **securely retrieving Kafka Cluster and Schema Registry API keys** from **AWS Secrets Manager**, followed by loading essential **Kafka client configuration parameters** from **AWS Systems Manager Parameter Store**. This design highlights best practices for **secure, automated configuration management** and **real-world Flink integration across AWS and Confluent Cloud environments**.

We go beyond simple transformation and enrichment — this project **streams data into Kafka topics and *Apache Iceberg Tables***, purpose-built for **high-performance, large-scale analytics**. Iceberg brings **ACID compliance**, schema evolution, and **highly scalable, real-time data processing with durable storage**, forming the analytical backbone of this architecture.

Our exploration of **Apache Flink** extends beyond Java into **Python-based applications**, showcasing the power of **Flink SQL**, **Table API**, and **DataStream API**. Each serves a distinct purpose:

- **Flink SQL** for declarative, real-time analytics
- **Table API** for a blend of SQL expressiveness and programmatic control
- **DataStream API** for efficient, event-driven stream processing

Together, they deliver a **unified, flexible, and scalable approach** to solving complex streaming challenges—bridging real-time and batch processing with elegance and performance.

Run the Docker containers (fully compatible with both **Apple Silicon (M-series)** and **x86 architectures**) and experience firsthand how Apache Flink can power **robust, enterprise-grade streaming applications** that are as practical as they are powerful.

Table of Contents

1.0 The Apache Flink Kick-Starter Just Got Smarter: From Stream Processing to Real-Time Intelligence

The `apache_flink-kickstarter` project has been completely refreshed to keep pace with the fast-moving evolution of **Apache Flink 2.1** and **Apache Iceberg 1.10**. This isn't just a routine maintenance update— it's a strategic move that advances your real-time data platform into the next era of streaming and AI-driven decision-making.

The upgrade updates the project with the latest APIs, performance improvements, and intelligent runtime hooks introduced since Apache Flink v1.20 and Apache Iceberg v1.7. The result: faster pipelines, improved observability, and new features that merge data engineering and data science.

1.1 Apache Flink v2.1: Real-Time AI and Smarter Streaming

If you've been holding steady on `v1.20`, the `v2.1` release is the inflection point where **stream processing meets machine learning**:

- **AI/ML inside the stream:** Native model DDLs and inference functions allow you to deploy and run trained models directly in your Flink jobs—no sidecars, no hacks.
- **Richer SQL & Table API:** Handle complex and semi-structured data with new Process Table Functions (PTFs), faster joins, and more expressive queries.
- **Operational superpowers:** Enhanced runtime hooks enable detailed performance tuning, state inspection, and adaptive write strategies.
- **Evolution of purpose:** Flink's story is shifting from *real-time data processing* to *real-time intelligence*.

1.2 Apache Iceberg v1.10: A Lakehouse Built for Modern Workloads

For those on Apache Iceberg v1.7.x, version v1.10 marks the point when the open-table format truly matures:

- **Broader ecosystem support:** Native compatibility with the latest Flink and Spark versions for a seamless hybrid stack.
- **Smarter table specification:** Enables support for geospatial types, variant columns, and deletion vectors—enhancing advanced analytics and governance capabilities.
- **Operational finesse:** Smarter compaction, more precise stats, and streaming-friendly enhancements that simplify managing long-running jobs.
- **Future-proofing your platform:** Keeping it up-to-date guarantees access to the newest spec features and ongoing support from engine developers and vendors.

1.3 Behind the Scenes: Code, Performance, and Best Practices

The refreshed `apache_flink-kickstarter` codebase not only adopts new APIs but also incorporates modern engineering principles:

- Deprecated calls replaced with **new unified APIs**.
- Best practices for **state management** and **checkpointing** to ensure reliability.
- Optimized **serialization**, **deserialization**, and **connector performance** for Kafka, Iceberg, and S3.
- **Cleaner configuration** and **modularized build logic** for easier deployment across environments.

- Added **thread-safety** for random number generation in Flink applications.

These changes turn the project into both a quick-start example and a ***living reference architecture*** for designing production-grade Apache Flink + Apache Iceberg applications in 2025.

1.4 Why It Matters

The shift from batch analytics to continuous intelligence is occurring now. Apache Flink v2.1 and Apache Iceberg v1.10 are not just incremental upgrades—they symbolize the merging of ***streaming***, ***AI***, and ***lakehouse architecture***.

By upgrading today, you're not just boosting performance—you're securing your data platform for the next era of real-time, model-driven decision systems.

2.0 Are these examples better described as Flink Jobs or Flink Applications-—and why does the distinction matter?

> *"What's in a name? That which we call a rose by any other name would smell just as sweet."*
>
> *-- William Shakespeare*

Did you know? In the Flink ecosystem, what many call "jobs" are actually better described as **Flink applications**—and that distinction matters.

A Flink application isn't just a single computation or stream transformation—it's a **self-contained, event-driven system** that coordinates state management, fault tolerance, and complex dataflows across distributed clusters. In essence, each one is a **mini software ecosystem**, designed to solve real-world data challenges at scale.

By calling them *applications*, we highlight their **depth, sophistication, and architectural completeness**. They don't just process data—they **model business logic, orchestrate stateful workflows, and power streaming intelligence** much like any robust production-grade software system.

(Curious why the terminology matters? Check out the rationale here!)

3.0 Let's get started!

As of **October 4, 2024**, **Apache Flink on Confluent Cloud** unlocks incredible potential for real-time stream processing—along with a few key nuances developers should know. Currently, **Flink on Confluent Cloud** does **not** support the **DataStream API**, and its **Table API** has **limited functionality** (see supported features here), supporting Java and Python-based Flink applications. But don't worry—**this repository bridges those gaps** with practical, production-ready examples.

3.1 What You'll Learn

In this repo, you'll find **Java-based examples using the Flink DataStream API** to go beyond Flink's standard capabilities—like:

- Securely retrieving **Kafka Cluster API keys** from **AWS Secrets Manager**
- Dynamically loading **Kafka client configuration** from **AWS Systems Manager Parameter Store** —all directly integrated within a running Flink job.

Prefer Python? You're covered there too. Using the **Flink Table API** and **User-Defined Table Functions (UDTFs)**, you can achieve the same secure, dynamic configuration loading while staying fully in Python.

3.2 Beyond Kafka: Iceberg Integration

We don't just stream data into Kafka topics—this project also demonstrates how to **sink data into Apache Iceberg tables**, combining **real-time event streaming** with **durable, analytical storage** for hybrid batch/stream processing.

3.3 Run It Locally — Fast

Spin up an **Apache Flink 2.1.0 cluster locally** in minutes using Docker—fully compatible with both **Apple Silicon (M-series)** and **x86 machines**. The Flink cluster connects seamlessly to **Confluent Cloud** for Kafka and Schema Registry resources, giving you a realistic, end-to-end development environment.

3.4 Dive In

Explore the examples, run the containers, and see how to build **secure, scalable, enterprise-grade Flink applications** that push the limits of what's possible with modern data streaming. The future of real-time data pipelines starts here.

3.5 Getting Started

Follow these steps to set up your environment and launch the complete **Apache Flink Kickstarter** stack—locally or in the cloud.

3.5.1 Cloud Prerequisites

Before you begin, ensure you have access to the following cloud accounts:

- **AWS Account** — with **SSO configured**
- **Confluent Cloud Account** — for Kafka and Schema Registry resources
- **Docker Account** — to pull and run containers
- **GitHub Account** — with **OIDC configured** for AWS authentication
- **Snowflake Account** — for data warehousing and analytics
- **Terraform Cloud Account** — for automated infrastructure provisioning

3.5.2 Local Prerequisites

Make sure the following tools are installed on your local machine:

- **AWS CLI version 2**
- **Confluent CLI version 4 or higher**
- **Docker Desktop**
- **Java JDK 21**
- **Python 3.12**
- **Terraform CLI version 1.13.4**

3.5.3 Clone the Repository

```
git clone https://github.com/j3-signalroom/apache_flink-kickstarter.git
cd apache_flink-kickstarter
```

3.5.4 Set Up Your Environment

You can set up the complete environment in one of two ways:

- **Option A:** Configure and run **Terraform Cloud** locally.
- **Option B:** Use **GitHub Actions** to automatically create all resources.

Both methods provision a fully functional **end-to-end data streaming architecture**, including:

- A **Confluent Cloud environment** with a Kafka cluster and pre-configured example topics
- **AWS Secrets Manager** for secure Kafka API key storage
- **AWS Systems Manager Parameter Store** for Kafka client configuration (consumer/producer properties)
- An **AWS S3 bucket** with a dedicated `warehouse/` folder serving as the **landing zone for Apache Iceberg Tables**—populated by two Python-based Flink applications

3.5.5 Run Flink Locally

You can run **Apache Flink** locally on your Mac or use the **provided Docker containers** to launch Flink and Iceberg seamlessly—compatible with both **Apple Silicon (M-series)** and **x86 architectures**.

3.5.6 You're Ready!

Once deployed, you'll have a **complete enterprise-grade data streaming environment**—secure, scalable, and ready to power both **real-time** and **analytical** use cases.

3.6 DevOps in Action: Deploying Terraform

Install the Terraform CLI on your local machine, and make sure you have an HCP Terraform account to run the Terraform configuration. Learn how to set up Terraform Cloud for local use by clicking here.

3.6.1 Deploy Terraform

```
./deploy-terraform.sh <create | delete> --profile=<SSO_PROFILE_NAME>
                                        --confluent-api-key=
<CONFLUENT_API_KEY>
                                        --confluent-api-secret=
<CONFLUENT_API_SECRET>
                                        --snowflake-warehouse=
<SNOWFLAKE_WAREHOUSE>
                                        --service-account-user=
<SERVICE_ACCOUNT_USER>
                                        --day-count=<DAY_COUNT>
                                        --auto-offset-reset=<earliest |
latest>
                                       --number-of-api-keys-to-retain=
<NUMBER_OF_API_KEYS_TO_RETAIN>
                                       --admin-service-user-secrets-root-
path=<ADMIN_SERVICE_USER_SECRETS_ROOT_PATH>
```

Argument placeholder	Replace with
`<SSO_PROFILE_NAME>`	your AWS SSO profile name for your AWS infrastructue that host your AWS Secrets Manager.
`<CONFLUENT_API_KEY>`	your organization's Confluent Cloud API Key (also referred as Cloud API ID).
`<CONFLUENT_API_SECRET>`	your organization's Confluent Cloud API Secret.
`<SNOWFLAKE_WAREHOUSE>`	the Snowflake warehouse (or "virtual warehouse") you choose to run the resources in Snowflake.
`<SERVICE_ACCOUNT_USER>`	the Snowflake service account user who is to be assigned the RSA key pairs for its authentication.
`<DAY_COUNT>`	how many day(s) should the API Key be rotated for.
`<AUTO_OFFSET_RESET>`	Use `earliest`, when you want to read the first event in a Kafka topic. Otherwise, specify `latest`.
`<NUMBER_OF_API_KEYS_TO_RETAIN>`	Specifies the number of API keys to create and retain.
`<ADMIN_SERVICE_USER_SECRETS_ROOT_PATH>`	the root path in AWS Secrets Manager for admin service user secrets.

To learn more about this script, click here.

3.7 DevOps in Action: Deploying Terraform in the cloud

In order to run the Terraform configuration from GitHub, the Terraform Cloud API token and Confluent Cloud API Key are required as GitHub Secret variables. Learn how to do to get the Terraform Cloud API token and Confluent Cloud API key here.

3.7.1 Run from the cloud

Follow these steps:

a. **Deploy the Repository**: Ensure that you have cloned or forked the repository to your GitHub account.

b. **Set Required Secrets and Variables**: Before running any of the GitHub workflows provided in the repository, you must define at least the `AWS_DEV_ACCOUNT_ID` variable (which should contain your AWS Account ID for your development environment). To do this:

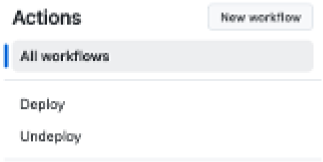
- Go to the **Settings** of your cloned or forked repository in GitHub.
- Navigate to **Secrets and Variables** > **Actions**.
- Add the `AWS_DEV_ACCOUNT_ID` and any other required variables or secrets.

c. **Navigate to the Actions Page**:

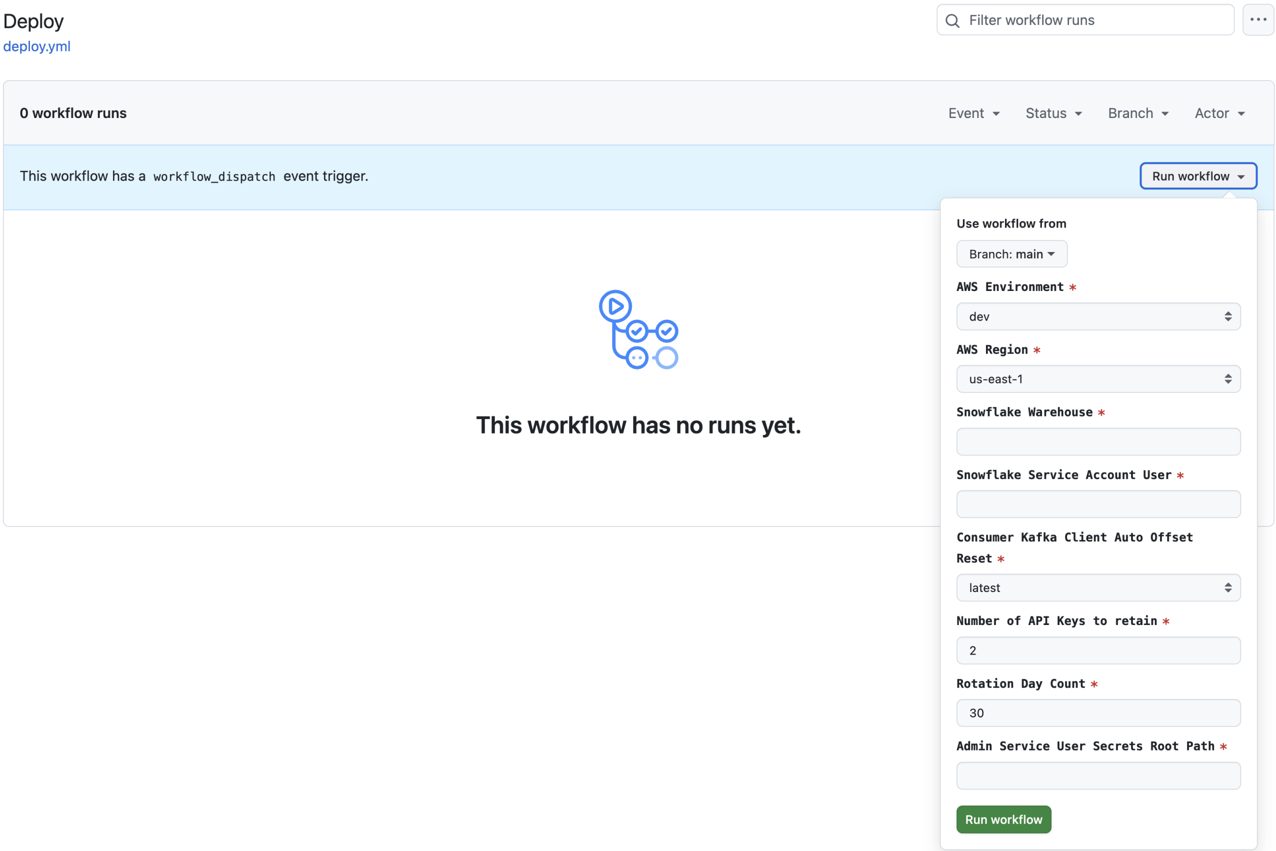
- From the cloned or forked repository on GitHub, click on the **Actions** tab.

d. **Select and Run the Deploy Workflow**:

- Find the **Deploy workflow** link on the left side of the Actions page and click on it.



- On the **Deploy workflow** page, click the **Run workflow** button.
- A workflow dialog box will appear. Fill in the necessary details and click **Run workflow** to initiate the Terraform deployment process.



This workflow has a `workflow_dispatch` event trigger.

Use workflow from

Branch: main

AWS Environment * : dev

AWS Region * : us-east-1

Snowflake Warehouse * :

Snowflake Service Account User * :

Consumer Kafka Client Auto Offset Reset * : latest

Number of API Keys to retain * : 2

Rotation Day Count * : 30

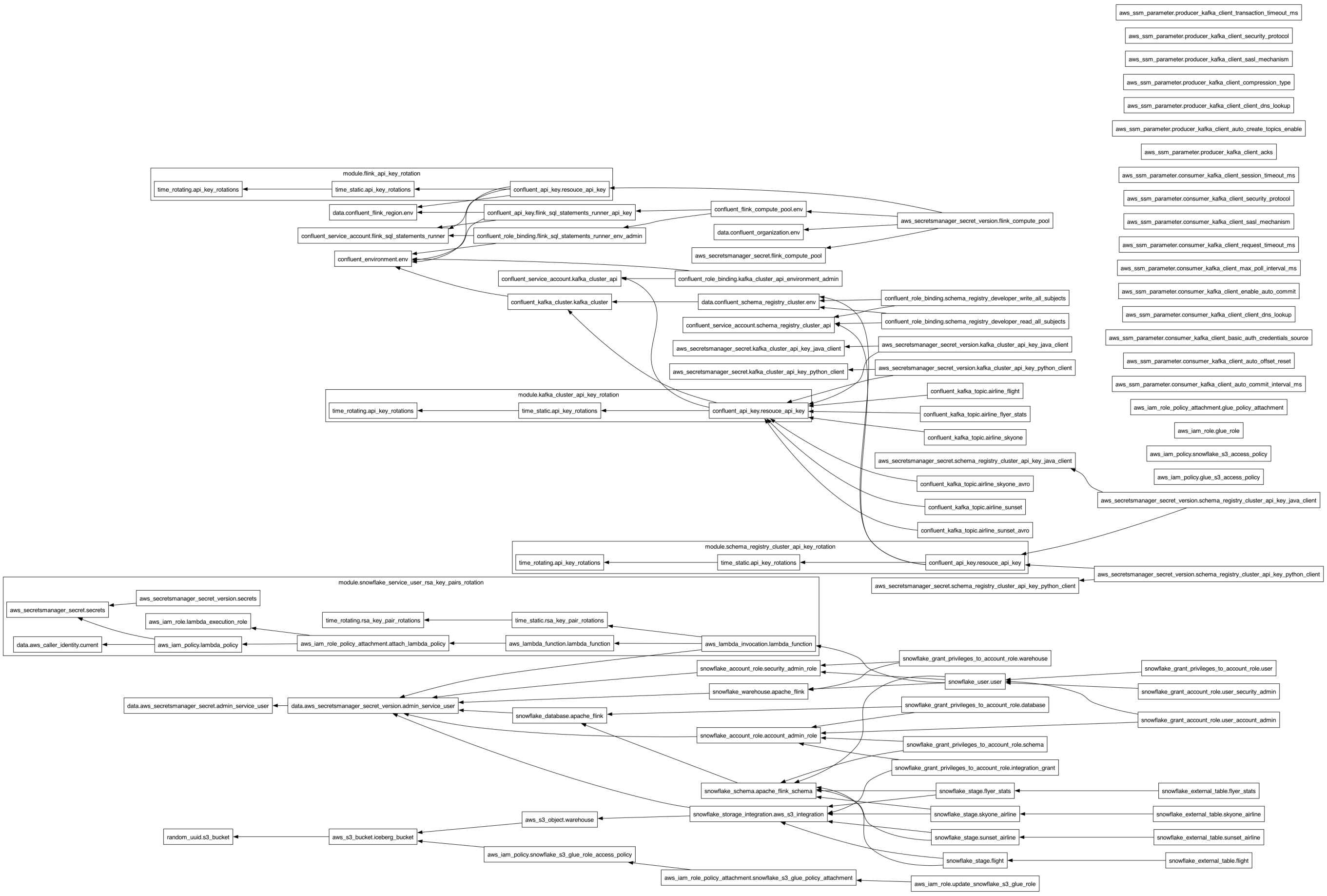
Admin Service User Secrets Root Path * :

Run workflow

By following these steps, you will run the Terraform configuration directly from GitHub, leveraging GitHub Actions for automation and deployment.

3.8 Visualizing the Terraform Configuration

Below is the Terraform visualization of the Terraform configuration. It shows the resources and their dependencies, making the infrastructure setup easier to understand.


aws_ssm_parameter.producer_kafka_client_transaction_timeout_ms
aws_ssm_parameter.producer_kafka_client_security_protocol
aws_ssm_parameter.producer_kafka_client_sasl_mechanism
aws_ssm_parameter.producer_kafka_client_compression_type
aws_ssm_parameter.producer_kafka_client_client_dns_lookup
aws_ssm_parameter.producer_kafka_client_auto_create_topics_enable
aws_ssm_parameter.producer_kafka_client_acks
aws_ssm_parameter.consumer_kafka_client_session_timeout_ms
aws_ssm_parameter.consumer_kafka_client_security_protocol
aws_ssm_parameter.consumer_kafka_client_sasl_mechanism
aws_ssm_parameter.consumer_kafka_client_request_timeout_ms
aws_ssm_parameter.consumer_kafka_client_max_poll_interval_ms
aws_ssm_parameter.consumer_kafka_client_enable_auto_commit
aws_ssm_parameter.consumer_kafka_client_client_dns_lookup
aws_ssm_parameter.consumer_kafka_client_basic_auth_credentials_source
aws_ssm_parameter.consumer_kafka_client_auto_offset_reset
aws_ssm_parameter.consumer_kafka_client_auto_commit_interval_ms
aws_iam_role_policy_attachment.glue_policy_attachment
aws_iam_role.glue_role
aws_iam_policy.snowflake_s3_access_policy
aws_iam_policy.glue_s3_access_policy
aws_secretsmanager_secret_version.schema_registry_cluster_api_key_java_client
module.flink_api_key_rotation
time_rotating.api_key_rotations
time_static.api_key_rotations
confluent_api_key.resouce_api_key
data.confluent_flink_region.env
confluent_api_key.flink_sql_statements_runner_api_key
confluent_flink_compute_pool.env
aws_secretsmanager_secret_version.flink_compute_pool
confluent_service_account.flink_sql_statements_runner
confluent_role_binding.flink_sql_statements_runner_env_admin
data.confluent_organization.env
confluent_environment.env
aws_secretsmanager_secret.flink_compute_pool
confluent_service_account.kafka_cluster_api
confluent_role_binding.kafka_cluster_api_environment_admin
confluent_kafka_cluster.kafka_cluster
data.confluent_schema_registry_cluster.env
confluent_role_binding.schema_registry_developer_write_all_subjects
confluent_service_account.schema_registry_cluster_api
confluent_role_binding.schema_registry_developer_read_all_subjects
aws_secretsmanager_secret.kafka_cluster_api_key_java_client
aws_secretsmanager_secret_version.kafka_cluster_api_key_java_client
aws_secretsmanager_secret.kafka_cluster_api_key_python_client
aws_secretsmanager_secret_version.kafka_cluster_api_key_python_client
module.kafka_cluster_api_key_rotation
time_rotating.api_key_rotations
time_static.api_key_rotations
confluent_api_key.resouce_api_key
confluent_kafka_topic.airline_flight
confluent_kafka_topic.airline_flyer_stats
confluent_kafka_topic.airline_skyone
aws_secretsmanager_secret.schema_registry_cluster_api_key_java_client
confluent_kafka_topic.airline_skyone_avro
confluent_kafka_topic.airline_sunset
confluent_kafka_topic.airline_sunset_avro
module.schema_registry_cluster_api_key_rotation
time_rotating.api_key_rotations
time_static.api_key_rotations
confluent_api_key.resouce_api_key
aws_secretsmanager_secret_version.schema_registry_cluster_api_key_python_client
aws_secretsmanager_secret.schema_registry_cluster_api_key_python_client
module.snowflake_service_user_rsa_key_pairs_rotation
aws_secretsmanager_secret_version.secrets
aws_secretsmanager_secret.secrets
aws_iam_role.lambda_execution_role
time_rotating.rsa_key_pair_rotations
time_static.rsa_key_pair_rotations
data.aws_caller_identity.current
aws_iam_policy.lambda_policy
aws_iam_role_policy_attachment.attach_lambda_policy
aws_lambda_function.lambda_function
aws_lambda_invocation.lambda_function
snowflake_grant_privileges_to_account_role.warehouse
snowflake_account_role.security_admin_role
snowflake_grant_privileges_to_account_role.user
snowflake_user.user
snowflake_warehouse.apache_flink
snowflake_grant_account_role.user_security_admin
data.aws_secretsmanager_secret.admin_service_user
data.aws_secretsmanager_secret_version.admin_service_user
snowflake_grant_privileges_to_account_role.database
snowflake_database.apache_flink
snowflake_grant_account_role.user_account_admin
snowflake_account_role.account_admin_role
snowflake_grant_privileges_to_account_role.schema
snowflake_grant_privileges_to_account_role.integration_grant
snowflake_schema.apache_flink_schema
snowflake_stage.flyer_stats
snowflake_external_table.flyer_stats
aws_s3_object.warehouse
snowflake_storage_integration.aws_s3_integration
snowflake_stage.skyone_airline
snowflake_external_table.skyone_airline
random_uuid.s3_bucket
aws_s3_bucket.iceberg_bucket
snowflake_stage.sunset_airline
snowflake_external_table.sunset_airline
aws_iam_policy.snowflake_s3_glue_role_access_policy
snowflake_stage.flight
snowflake_external_table.flight
aws_iam_role_policy_attachment.snowflake_s3_glue_policy_attachment
aws_iam_role.update_snowflake_s3_glue_role

> **To fully view the image, open it in another tab on your browser to zoom in.**

When you update the Terraform Configuration, to update the Terraform visualization, use the `terraform graph` command with Graphviz to generate a visual representation of the resources and their dependencies. To do this, run the following command:

```
terraform graph | dot -Tpng > .blog/images/terraform-visualization.png
```

4.0 Hands-On Kickoff: Practical Examples for Rapid Learning

4.1 Flink Applications Powered by Java on a locally running Apache Flink Cluster in Docker

Let's go!

4.2 Flink Applications Powered by Python on a locally running Apache Flink Cluster in Docker

Let's go!

4.3 Flink Applications Powered by Python on Confluent Cloud for Apache Flink

Let's go!

5.0 Powering Analytics with Apache Iceberg tables in Snowflake

After running the Flink applications, you can now leverage the data stored in the Apache Iceberg tables in Snowflake for analytics. Click here to learn more about using **Non-Managed Apache Iceberg tables** in Snowflake. This blog post will help you understand the trade-offs, operational characteristics, and integration patterns of using Apache Iceberg tables in Snowflake. These terraform configurations:

- setup-aws-s3 - creates the necessary Snowflake IAM roles and policies in AWS to get started.
- setup-snowflake-service_user - grants the necessary privileges to the Snowflake resources to get started.
- setup-snowflake-objects - creates the necessary resources in Snowflake to get started.

6.0 Resources

Apache Flink's Core is Dataflow Programming

What is Apache Flink? — Architecture

Apache Flink Use Cases

Building Apache Flink Applications in Java

J3's techStack Lexicon

Unlocking Schema Registry Access: Granting Confluent Service Account Permissions with Terraform