CloudEvents Plugin for Jenkins

Google Summer of Code 2021 Project Proposal

Shruti Chaturvedi
shrutichaturvedi16.sc@gmail.com
Kanpur, India
GitHub Username: ShrutiC-git

Project Abstract:

Enhance interoperability in the current Jenkins architecture by designing a plugin for Jenkins which emits and consumes CloudEvents. CloudEvents is an industry-adopted standard specification for describing an event and the event data (also referred to as Payload). This project enables Jenkins to produce CloudEvents (as a source) for common Jenkins events like a project-build success or failure event, which other tools (example Azure Event Grid, AWS Kinesis, Apache Pulsar Stream and more) can consume. This project also enables Jenkins to be configured as a sink where external tools (e.g. Amazon Kinesis) can send CloudEvents, and Jenkins can consume those events to perform a Jenkins action (e.g. trigger a Jenkins job build). Implementing this plugin in a workflow will give Jenkins users the ability to use and extend an event-driven Jenkins workflow to systems which use the CloudEvent-spec.

End-Goal of the Project:

The goal of this project is to enable interoperability between Jenkins and external tools like Azure Event Grid, Debezium, Knative, et cetera, which support CloudEvents. CloudEvents specification is a standard spec for describing events. By using the CloudEvents spec for Jenkin events, we are making Jenkins compatible with tools and systems which use CloudEvents spec for their events. This will allow developers to integrate Jenkins in their workflows with tools which use CloudEvents. Additionally, using this standard spec for events will make extending Jenkins capability to other platforms in the future easy.

Project Description:

As development is becoming more complex, developers want tools which can integrate well with other systems they use to make the development and deployment cycles faster. Being a CI tool with CD capabilities, Jenkins greatly enhances development, testing and deployment cycles by integrating with other tools. Currently, Jenkins does not support industry-standard specification for emitting and consuming events. This makes it hard to integrate Jenkins to work with events which support the industry-standard spec for events (CloudEvents). This project will solve that problem by implementing a plugin to emit and consume CloudEvents from Jenkins.

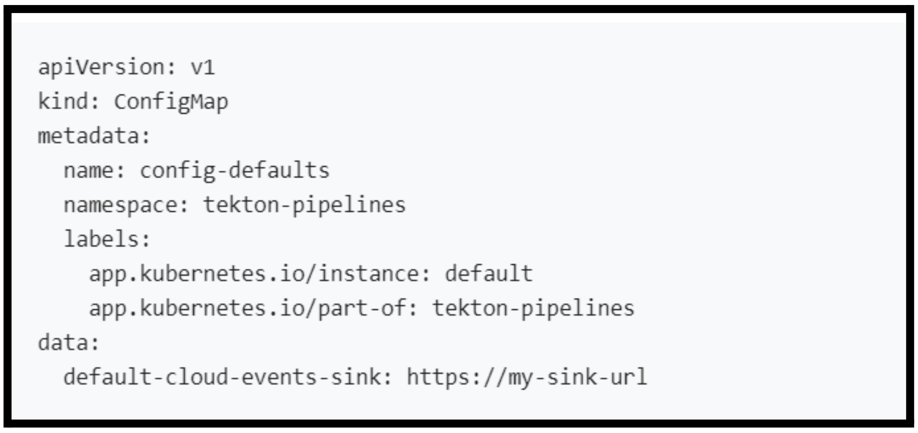
General Understanding

- According to CloudEvents.io, CloudEvents is a spec for describing event data in a standard way. CloudEvents spec is used in managing interoperability between systems which emit and consume events.
- A producer generates a CloudEvent-spec compliant event. A consumer can express interest in this event and consume it without having to worry about the structure of the event or the payload since CloudEvent spec defines a standard way of design for events.
- CloudEvents defines metadata, which contains information like the source of the event, the type of the event and the target to allow efficient routing, and also data about the event (e.g. the build#, Github userID of the user who initiated the event).
- CloudEvents offers a variety of protocols (e.g. HTTP, Kafka) and encodings (JSON, Avro) for serializing events.

- Event-driven tools like Azure Event Grid, Tekton, Keptn (a CD tool), AWS EventBridge, et cetera emit CloudEvent-compliant events. Consider an example of how Tekton, an Open-Source, event-driven, CI/CD tool emits CloudEvent-compliant events:

  ⇒ Run of a Pipeline in Tekton becomes the resource/event-source which emits a ‘Pipeline Succeeded’ CloudEvent. This CloudEvent uses JSON encoding and is sent over HTTP between different systems. The event payload (event data) contains information like the name of the pipeline, steps within the pipeline, environment vars used by the steps, description of the steps and more. **(1)**

  ⇒ Now for this Tekton event, a Sink is configured (fig below). A Sink will enable any system to receive this event and act on it. To configure the sink, a user will have to provide the Sink URL (REST API Endpoint in case of a Request/Response like system) or alternatively provide the Sink topic URL (Apache Pulsar or Kafka streaming like EDA pattern). **(2)**



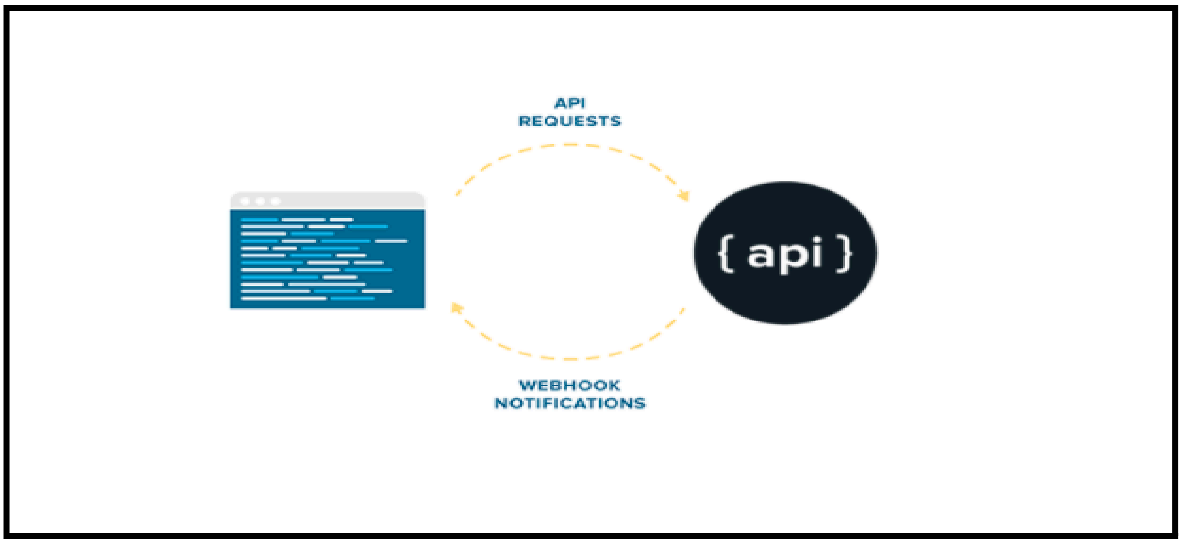
```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-defaults
  namespace: tekton-pipelines
  labels:
    app.kubernetes.io/instance: default
    app.kubernetes.io/part-of: tekton-pipelines
data:
  default-cloud-events-sink: https://my-sink-url
```

  ⇒ A post request will be sent to the Sink’s API, upon which the Sink will receive this CloudEvent as it is emitted *(the retrieval of this Event by the Sink may vary by architecture)*. **(3)**

- Several tools which can consume Events, like Azure EventGrid are called Sinks. These tools have SDK Clients to configure a Sink where CloudEvents from a Source can be sent. For example, **Azure provides a client library for [Azure Event Grid]**. This client can be configured from a Source application with Access Keys and Topic URL to send events to Event Grid from whereon Event Grid can be configured to trigger Azure-specific actions.

- One way to configure a sink would be using the Request/Response architecture similar to the architecture used by Webhooks. This, however, is not the most conducive architecture for an event-driven system. We'll look at EDA-sink implementation in sections ahead: ***High-level implementation***; ***Detailed-implementation***

- Similar to the diagram below, a Source which emits a CloudEvent will send a CloudEvent-complaint event to the sink's REST API endpoint or a topic URL depending upon the architecture used. The sink will this event to perform a task. The source only cares about the exposed endpoint where it would send the events to. Architecture of the sink (what processed/receives the event) is independent of the Source.



- In a more complex system, this architecture can also be implemented using either a Broker-like pattern (consider Pub/Sub method) or an Event-Mediator pattern where the Broker will perform an intermediating role between the Source emitting Cloud Events and the Sink consuming these events. These are EDA Designs pattern implementations.

Project Deliverables:

- Jenkins as a Source and integration with external tools (*Phase I*)
- Global Plugin Config as part of Jenkins as a Source (*Phase I*)
- Jenkins as a Sink (*Phase II*)
- Global Plugin Config as part of Jenkins as a Sink (*Phase I + II*)

Implementation in High-Level Terms:

Jenkins needs to be implemented as a **Source emitting CloudEvents** and as a **Sink consuming CloudEvents**. We will start by implementing the architecture which is used by tools like Tekton, Debezium. Here are the high-level steps, which we will look into detail in the '**A Closer Look**' section ahead:

1. **Jenkins as a Source (HL):**
   - **Jenkins events will be wrapped as CloudEvents using CloudEvents Java SDK**. We will then be configuring external tool's Sink-clients for cloudevents emerging from Jenkins; this is where Jenkins events will be sent. A common example would be using ***Azure Event Grid client library for Java.***
     ⇒ *This Azure Event Grid Java SDK will be used to serialize and publish events to EventGrid (sink), which is compatible*

with CloudEvents. To use EventGrid, users will have to create a topic to which our SDK Client will send the CloudEvent to in Azure portal/console.

- The jenkins plugin will have **support/integration for common tools which consume CloudEvents** like Amazon Kinesis, Google Cloud Pub/Sub, Azure EventGrid and more depending upon possible integrations needed. It can also work by supplying a Sink URL/Topic URL for Event Sinks outside of these tools. This will be configured as part of the Global Plugin Config, adding UI changes to accept input from the User depending upon the external 3rd-party Client that is being used (or a Sink URL in case a custom Sink is being used).

- **Global Plugin Config as Part of Jenkins as a Source:** This plugin should allow users to efficiently send events to integrations/tools this plugin offers support for. In order to do that, the **plugin will also need UI changes to accept user-input to configure the Sink**. Referring to the example above, the user would have to enter *Azure Subscription Keys* and *topic info* to send events from Jenkins to Event Grid.

2. Jenkins as a Sink (HL):

- The general implementation of a Sink without an event-driven architecture pattern would be a simple Request/Response system similar to the architectural pattern used by the Generic Webhook Trigger Plugin. This will involve creating a REST API Endpoint and configuring it to receive POST requests with the CloudEvent. Followed by polling to catch events as they happen.

- The implementation of sending events to Sink in an EDA pattern instead of a Request/Response system will need some more thought to be compliant with the Event-Driven Architecture. **There are two commonly adopted approaches for implementing EDA: Event-Streaming and Pub/Sub.** The methods described here use Event Streaming over Pub/Sub and Request/Response Architecture because the **project needs real-time or almost real-time processing of events and it can be achieved efficiently with Event-Streaming EDA pattern**.

    a. First and the preferred approach is implementing **Apache Kafka** as a streaming solution. Here we will have to implement Confluent REST Proxy to send events to a Kafka topic through HTTP. The source will be given the URL of the topic. If our topic is "cloudEvents", the URL for the Sink would be https://<Kafka-host>:*<port-where-Kafka-is-running>/topic/cloudEvents.*
    The next step will be implementing a Kafka Consumer to consume and process events. Details here.

    b. **Another approach is to have an HTTP Sink implementation for Apache Flume or Apache Nifi**. The Sink will take events sent as POST requests to the HTTPSource URL. Depending upon the data received (source of the event/event type), a Jenkins event (job-build) can be triggered using Jenkins REST API Client. Details here

EVENT-DRIVEN ARCHITECTURE VS REQUEST/RESPONSE SYSTEMS

EDA	REQ/RES
Loosely Coupled	**Strongly coupled**
Asynchronous Ops	**Synchronous Ops**
Easy to Scale	**Difficult to Scale**
Event-Streaming allows real-time processing	**Polling for real-time which wastes compute and network.**
Resilience achieved easily	**Resilience will need more configuration.**
Managing infra can be hard	**Infra Management is easy**
Difficult to set up the architecture.	**Easy and quick setup; a REST Endpoint config can be easily added.**
Will require setting up one or more servers to run kafka cluster or Pub/Sub Broker.	**No additional infrastructure will be required to use the Req/Res System**

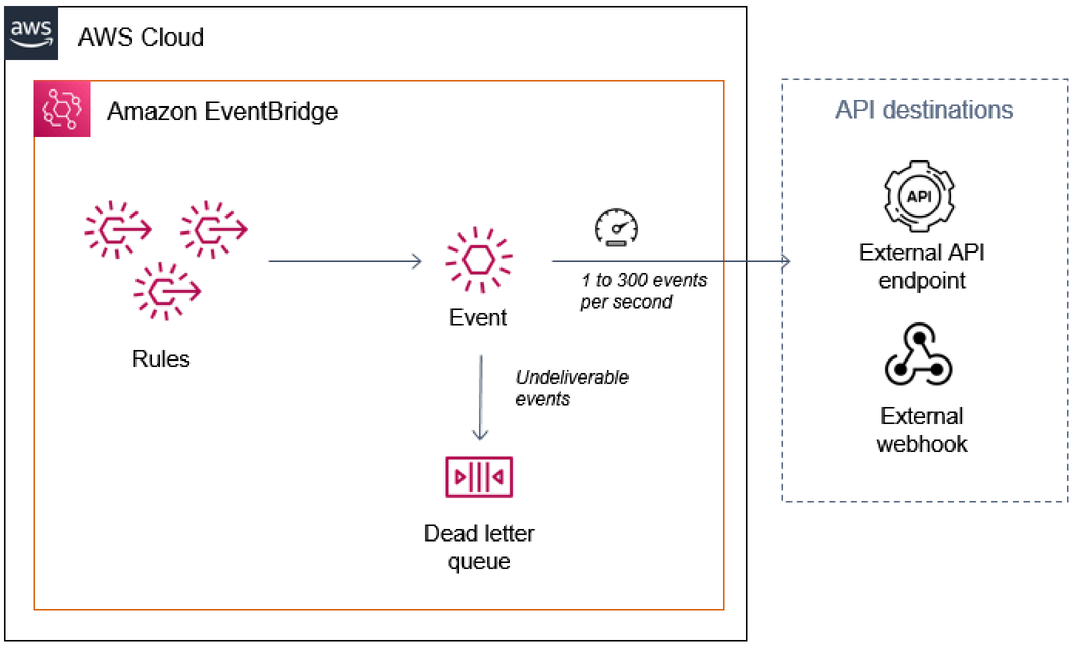
NOTE: *Between the above two implementation suggestions for event-streaming EDA pattern, Apache Kafka is preferred. It offers a resilient and scalable architecture whereas Flume is more functional for usage with Hadoop or other Big Data architecture. Since the Events transported will need scalability over storage given the fact that each single event can only hold 64 KB of data inside of it. However, Flume or NiFi can also be used by following an architecture pattern similar to Request/Response along with setting up an HTTP Source, JDBC Connector (or in-memory connector) and an HTTP Sink.*

- **Global Plugin Config as Part of Jenkins as a Sink (not applicable for Kafka)**: This task will include creating a REST API endpoint.

a. The creation of an external REST API Endpoint will be required to receive requests from external systems in a Request/Response system. For other architectures, topicURL (Kafka) or Source URL (Flume) is needed. *Example, a lifecycle-start Choria event (which is CloudEvent compliant) would send a POST request to our designed API endpoint (/cloudevents) from where we will receive and process this singleton request.* This is the creation of a protected REST API endpoint for external tools to use, where they will send a POST request with the CloudEvent.

b. Ideally, this endpoint will be also protected with an API Key or a username/password to not over-burden our system. Raises another concern about an EDA system.

c. Consider an example of using AWS EventBridge as an Event Source. EventBridge invokes the REST API endpoint it is configured with and delivers the event as a payload within the request. Figure below:

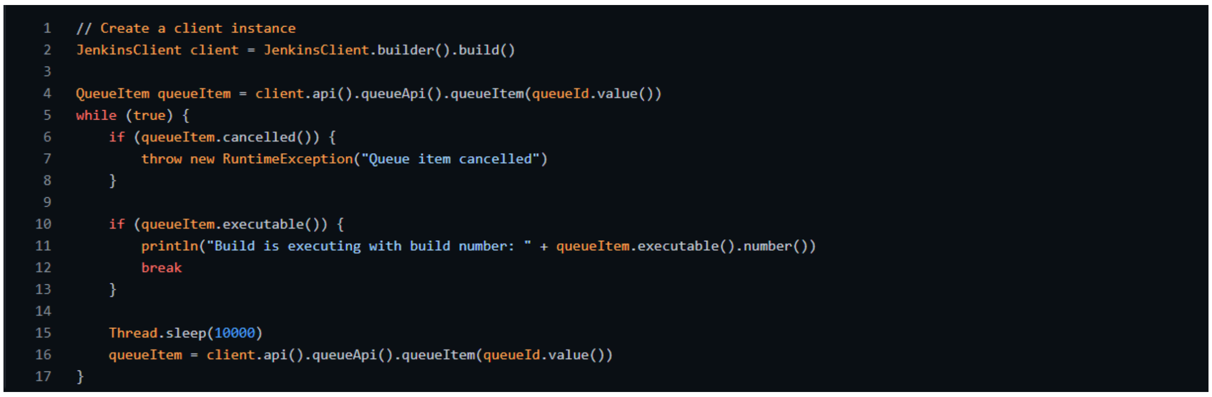


NOTE: *In the example above, a Kafka REST Proxy URL can also be given instead.*
A Request-Response System is not very adept in handling event-driven system design. Although we can use this system, an event-streaming EDA design will be better as it allows a scalable and resilient real-time processing system.

A Closer Look

=> **Jenkins as a Source:**

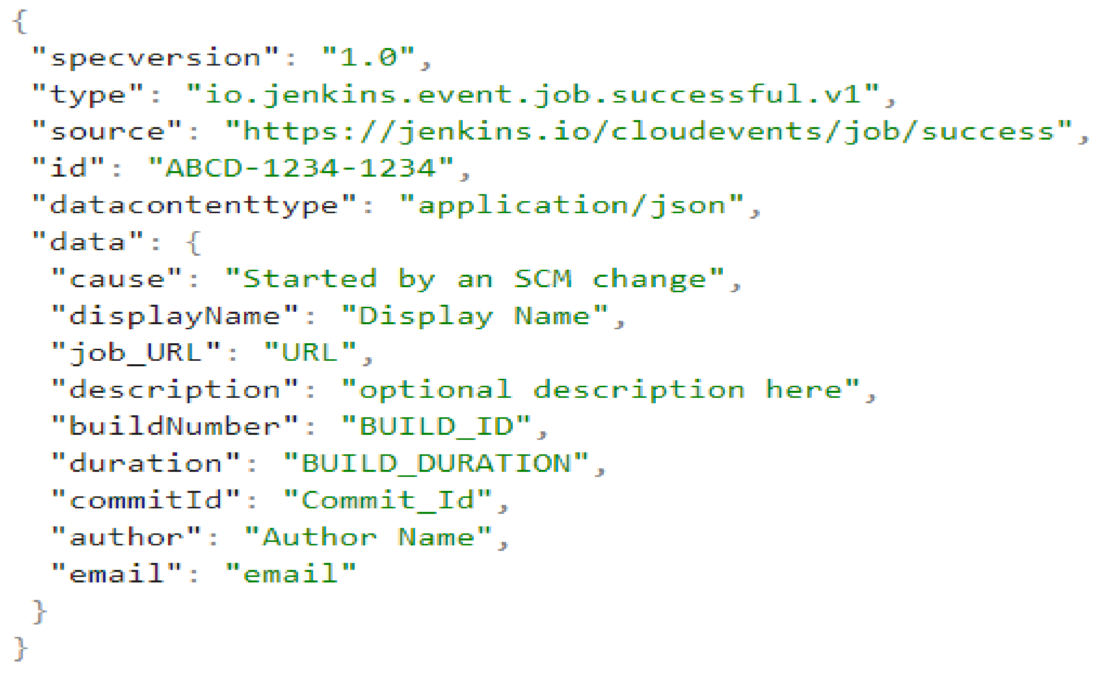
A. Start by implementing a Jenkins REST API Client which would be responsible for polling the Queue on the QueueAPI to look for status of queue item (tested on /queue/api/json and /queue/item/<queueitem#>/api/json). We would then use the JobsAPI to get the current build info on the item being built, and keep polling it until the job stops building. An example of doing this from Jenkins below:



```
// Create a client instance
JenkinsClient client = JenkinsClient.builder().build()

QueueItem queueItem = client.api().queueApi().queueItem(queueId.value())
while (true) {
    if (queueItem.cancelled()) {
        throw new RuntimeException("Queue item cancelled")
    }

    if (queueItem.executable()) {
        println("Build is executing with build number: " + queueItem.executable().number())
        break
    }

    Thread.sleep(10000)
    queueItem = client.api().queueApi().queueItem(queueId.value())
}
```

B. Using buildInfo(), we will be able to extract information essential to understand the status of a build. For example, if building is *False*, and the result is *Success* it would emit a ‘job successful’ type of

CloudEvent with JSON encoding which uses HTTP protocol in the example here. The current suggested architecture is using HTTP Protocol for transporting CloudEvents. Most Sinks with CloudEvents support use HTTP protocol to receive events. Here is what the event payload defined with CloudEvent Spec can look like:



```
{
 "specversion": "1.0",
 "type": "io.jenkins.event.job.successful.v1",
 "source": "https://jenkins.io/cloudevents/job/success",
 "id": "ABCD-1234-1234",
 "datacontenttype": "application/json",
 "data": {
  "cause": "Started by an SCM change",
  "displayName": "Display Name",
  "job_URL": "URL",
  "description": "optional description here",
  "buildNumber": "BUILD_ID",
  "duration": "BUILD_DURATION",
  "commitId": "Commit_Id",
  "author": "Author Name",
  "email": "email"
 }
}
```

** all variables will be extracted from the REST Client.*

** source and/or id can be UUIDs, which is the suggested pattern.*

***NOTE:** This data or event payload is a basic example of how we can track and send data. If we were to add more functionality to our build, we can start to incorporate that data into the payload as well.*

C. The above event will be built and sent using CloudEvents Java SDK (if not using an external client in which case the CloudEvent will be transported by the Sink client of that tool). As we saw earlier, since each external tool has either their **external Sink URL** (links to DataDog) or **Sink API Client** (links to Azure), we will need to use that particular system to allow users to interact with the Sink of their choice.

D. A user using this plugin will have the option to choose from available Sink providers. The users can either provide a custom Sink URL for their custom sink OR provide information necessary to register an external sink tool. For example, to use Azure Event Grid as a Sink would require us to implement their Sink API Client.

E. Users would have to enter information like their Azure Key Credentials, EndpointURL, et cetera through the plugin. The plugin will allow for taking the info specific to the integration from the user.

Example of Events within Jenkins:

(The *Design Doc* provided with this project is used for reference)

Resource	**Event**	**Event Type**
Job Execution	Started	io.jenkins.event.job.started.v1
	Running	io.jenkins.event.job.running.v1
	Succeeded	io.jenkins.event.job.successful.v1
	Failed	io.jenkins.event.job.failed.v1

Resource	**Event**	**Event Type**
Job in Queue	EnteredQueue	io.jenkins.event.queue.entered.v1
	QueueItem# Change	io.jenkins.event.queue.changed.v1
	Into Build	io.jenkins.event.queue.left.v1
	Cancelled	io.jenkins.event.queue.cancelled.v1

Similarly, a Job step execution tracking the actions in the build can be used to emit CloudEvents about the class of the step. Also, **the class**

returned from the API endpoint can be used to notify when a particular action/step occurred (example, a step which checked out a file occurred in the build--for the SCM Checkout Step referred to in the Design Doc).

> ***NOTE:*** *These are just suggestions on what CloudEvent emittance can look like. It will need more collaboration with the team to understand the importance of various events within and outside Jenkins, and understand how different plugins send information to the REST API*.

=> **Jenkins as a Sink**:

1. (Not applicable for the Kafka Design Pattern) We will begin by creating a REST API endpoint which will listen to POST requests. External tools which emit CloudEvents, for example, Tekton CloudEvents. Tekton emits TaskRun or PipelineRun events. Each source (Task or Pipeline Run) from Tekton needs to be configured with a Sink which would process these requests.

    - This endpoint will need to be protected similar to other REST endpoints Jenkins uses (by instantiating a Jenkins client or using Crumb-ID). *Take reference from DataDog, Zendesk et cetera. They have REST API Endpoints for receiving events.* The plugin should ideally also allow for edits in configuration of the API Endpoint. For example, editing the rate-limiting features, security credentials et cetera.

NOTE: *We are configuring a single API endpoint instead of having multiple endpoints. This will allow a central system of processing the stream of events. The stream solution we use will tag the appropriate event, and Jenkins will process it accordingly. This is a strategy which can be re-designed to instead have different channels exposed by different API endpoints where external tools can send requests to.*

2. A central stream system to manage these events becomes critical for a complex system where multiple sources or external tools enter a variety of different events. EDA is a better handler than Request/Response System for configuring the Sink. Here, an event-streaming method using services like Kafka is the ideal solution although few other architectures are listed too.

    - One approach is to **implement an event-streaming queue service (Apache Kafka like systems)**, which will handle the transportation of events in a loosely-coupled manner.

      ⇒ For example, using **Apache Kafka as a streaming service between the Source and the Sink is the most preferred method of implementing the Streaming EDA method**.

      a. Here, we will have to make use of KAFKA-REST which "provides a RESTful interface to a Kafka cluster. It makes it easy to produce and consume messages, view the state of the cluster, and perform administrative actions without using the native Kafka protocol or clients". By using this proxy, **we will be able to use Kafka native REST endpoints on /topic/topicName and configure topics to send the CloudEvent via a POST request at the endpoint**(for example /topic/job).

b. Next, we would have to implement a Kafka Consumer which will subscribe to a topic and for each event in the consumer, the plugin will trigger a jenkins event as needed.

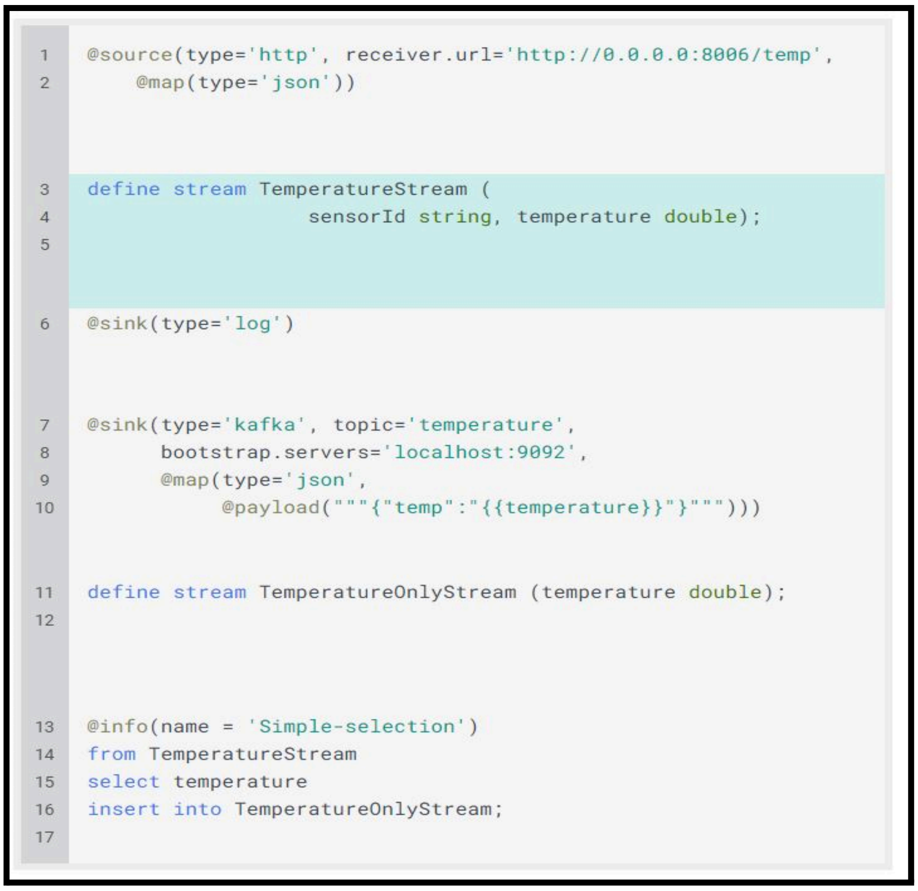
This infrastructure will require setting up Kafka Cluster/s.

- Another infrastructure (as talked earlier) is having an **HTTP Source implementation of Apache Flume which uses a JDBC channel (to allow persistence) to send events to an HTTP Sink implementation of Apache Flume.**
    a. The Source will receive POST requests from external clients with CloudEvent payload in the request as JSON.
    b. Then, send it to a JDBC Channel.
    c. The HTTPSink will receive events from the channel.

  This infrastructure will require setting up a Flume agent (JVM Daemon) which will be running the whole architecture. It will also need defining properties for the Source/Sink.

- Alternatively, **Apache Nifi can be also used here in place of Flume**, which follows almost the same design architecture with more scalability and a reactive-architecture with a robust support community. Nifi is not specific to Big Data unlike Flume thus allows for data routing between disparate systems with more ease. Nifi however is more memory intensive and will require a more in-depth research.

⇒ **Reference can be taken from an Open-Source tool called Siddhi**. This tool has offerings for a variety of business cases starting from basics like configuring Source and Sink to Data Processing and logging. Here's an example of configuring an HTTP Source sending payload to Log Sink and a Kafka Sink;



```
@source(type='http', receiver.url='http://0.0.0.0:8006/temp',
    @map(type='json'))

define stream TemperatureStream (
                  sensorId string, temperature double);

@sink(type='log')

@sink(type='kafka', topic='temperature',
      bootstrap.servers='localhost:9092',
      @map(type='json',
           @payload("""{"temp":"{{temperature}}"}""")))

define stream TemperatureOnlyStream (temperature double);

@info(name = 'Simple-selection')
from TemperatureStream
select temperature
insert into TemperatureOnlyStream;
```

-

Http Source to receive requests on, a Stream to collect incoming data, a Kafka topic for stream to send data to and a Kafka Sink which will receive from the Stream. The payload can be a CloudEvent specific payload. *We can implement Siddhi with kafka if the mentors agree to using Siddhi.*

NOTE: *A simple infrastructure of polling the API endpoint to listen to POST requests (Request/Response) can be designed for the sink, however, given the fact that multiple events can be triggered within a short period of time, and across multiple sources, an event queue or a an event streaming system is a better design approach in terms of parallel and scalable architecture. The acceptance waiting and bonding phase will involve researching and developing test architectures.*

Proposed Project Schedule:

Note: The dates in the GSoC timeline takes precedence over this document

Time Period	Tasks
March 29 - April 13 (Application Period) **Involves:** • **Research Architecture (Jenkins as a Sink)**	1. Deep dive into Jenkins infrastructure: Jenkins REST API Client; Jenkins-generated payload as Jenkins builds a project; understanding the different build stage classes and actions; testing *jobs* and *queues* endpoint and analyzing data returned; understand architecture of the Generic Webhook Trigger; looked at developing an external plugin for extending Jenkin capabilities; read Jenkins glossary to understand Jenkins specific terms. 2. Deep-dive into Event-Driven Architecture: looking at tools and services which employ CloudEvents for the EDA architecture. Examples are: Choria, Debezium, Azure EventGrid, commerTools, Kinesis 3. Beginners exploration into the two most commonly employed EDA patterns: pub/sub method and Event-streaming architecture.
April 14 - May 17 (Acceptance Waiting) **Involves:** • **Test Architecture (Jenkins as a Sink)** • **Research and test integrations (Jenkins as a**	**1. (Jenkins as a Sink | Global Plugin Config)** - Start by implementing a test architecture using the Event-streaming EDA method for Jenkins events which follow the CloudEvents specification; Event-streaming method over Pub/Sub will allow better handling of multi-channel events. - Begin with testing Apache Kafka by implementing the Kafka REST proxy to receive event data from external systems, Kafka topic and a Kafka Consumer.

Source) - **Data for CloudEvents (Jenkins as a Source)**	After the consumer is written, I will poll the topic to get event data and process it to produce a Jenkins action. (More Preferred) - Also test Siddhi integration with Jenkins configured with HTTPSource and a Kafka or consequently an HTTP Sink. (Next preferred method) - Begin with testing Apache Nifi VS Apache Flume architecture configured with HTTP Source, Sink and a persistence channel.(Less Preferred) - Test a simple Request/Response System without using any external streaming or queueing system. (Least Preferred) **2. (Jenkins as a Source)** - Research possible integrations with tools and services which consume CloudEvents and fit into the Jenkins infrastructure. Starting point is to test with Azure Event Grid which spans a wide variety of Event Handlers. ⇒ For example, if a Jenkins build has succeeded, Jenkins will send a CloudEvent to Azure topic (which the user will provide as a part of the plugin), and Azure EventGrid will be configured to take events coming into the topic to push to Azure Functions, or any event handler. - Also research the data which is passed by Jenkins REST API when building with a variety of plugins and

	running a variety of Jobs/Pipelines. For example, what data is emitted for a Gerrit Trigger, and how this data can be wrapped in a CloudEvent-compliant event which is useful. 3. More interaction with the community to understand the need for integrations within this plugin.
May 17 - June 7 (Community Bonding) **Involves:** • **Discussing Sink Architecture (Jenkins as a Sink)** • **Protocol for Event Transfer (Global)** • **Discussing integration needs (Jenkins as a Source)** • **Data state (Jenkins as a Source)**	Crucial time to connect with the community and hear suggestions on: ⇒ The Event-Driven Architecture to use for implementation of HTTP Event Sink. ⇒ Talk about if Request/Response would be a better design over EDA. ⇒ Possibly hear ideas about the desired protocol for Event transfer. The protocol used in this proposal doc is HTTP. ⇒ Data wrapped into CloudEvents for build jobs. ⇒ Integrations for using Jenkins as a source and use cases between these integrations. ⇒ Evaluate the need for types of integrations by connecting with the community.
June 7 - July 16 (Coding Phase I) **Involves:** • **Design Event-Spec (as Source)** • **Represent Event as CloudEvent (as Source)** • **UI Changes (as Source)**	1. **(Jenkins as a Source)** Build event-specification for Jenkins events. Primary event sources are: a. **Job Execution** (events emitted: job started, running, cancelled, succeeded, failed): listening and polling the Jobs API and using buildInfo() b. **Job Step Execution** (events emitted: step started, step finished): listening and polling the Jobs API and listening for Actions as they are started. A similar pattern to listen

• **API Endpoint (as Sink)**	for Builds in the Queue can be used for Steps of a Pipeline. c. **Job Queue Execution** (events emitted: enteredInQueue, changedPositionInQueue, leftQueue): Polling the queueAPI and listening Queue and QueueItem. 2. **(Jenkins as a Source)** Wrap data into a structured CloudEvent which uses JSON serialization and HTTP binding. The structured format is easier to integrate with event-streaming platforms. 3. **(Jenkins as a Source)** Implement external tool Sink Clients (refer to this example). 4. **(Global Plugin Config | Jenkins as a Sink)** Build the REST API endpoint for Sources to send POST requests to (depending on architecture) or configure Topic Sink URL. 5. **(Global Plugin Config | Jenkins as a Source)** Add UI changes as part of the plugin to accept User-input to configure the Sink for our Source. --- A step for each development task will be rigorous testing to ensure functionality works as desired.
July 16 - August 16 (Coding Phase II) **Involves:** • **Implementing the URL for external tools to send events to**	1. **Extend the REST API** endpoint as part of phase 1 to process the requests received for systems which need API endpoint (**Request/Response or a Flume-like Sink**). a. One implementation which uses an API endpoint can also be designed similar to the Generic WebHook

(as a Sink) • **Regular feedback from the mentors to streamline UX (as a Sink)**	Trigger which listens for POST requests as they are made, and processes the request. This approach however could be blocking execution and is less preferred. 2. **Implement the route of Event-Streaming architecture which is the preferred method over Request/Response and Pub/Sub** (it allows both scalability and resilience) for this scenario: ⇒ first, **implement Kafka REST Proxy producer, which will ingest the POST request sent to the Proxy API endpoint** (/topics/<topicDesigned>). This proxy API will receive events from an external system. This is also the URL which will be given as Sink to other tools. ⇒ then create a **Kafka topic where the proxy can send the data** using the NewTopic Module. For example, a topic named *cloudEvents* can be generated by specifying a partitionId (scalability) and replicationFactor (resiliency). ⇒ then implement a **Kafka Consumer to subscribe to the topic we created.** ⇒ then the **consumer will poll the topic** until there are events the topic is receiving. ⇒ then **extract the CloudEvent Data and produce a Jenkins Action** (build the project or

	more). 3. Another **sink implementation using the streaming architecture will be using Apache Flume HTTP Clients**. ***Find details here***
August 16 - August 23 (Final Submission) **Involves:** • **Testing (Global)** • **Deploying the Plugin (Global)**	1. Testing and more rigorous testing 2. Evaluating the architecture from the perspective of a tool which interacts with Jenkins through CloudEvents. 3. Ensure the UI workflow for the plugin works for a User who wants to send and receive CloudEvents. 4. Build and deploy the Plugin to the marketplace. 5. Touch base with the community and mentors to make sure the business case is solved efficiently keeping in mind that the Plugin is suitable for complex architecture.

Foreseeable Challenges:

- The one big challenge is understanding the architecture needed for streaming events parallely in a scalable and resilient manner from various Event Sources or generators to the Sink.

- Another challenge would be understanding CloudEvent payload which can be consumed by different tools.

Workaround: The best workaround is talking with the mentors and the community to discuss possible architectural patterns to understand trade-offs. Also, do more mocking and testing to understand data that is generated as a result of running a Job, and how different Sinks (outside of Jenkins) can use this payload.

Future Plans (Post GSoC):

- I would keep enhancing the CloudEvents architecture to support multi-directional events and tools. EDA has enhanced developer and user-experience. I would be committed to ensuring that the current plugin is able to handle workloads simultaneously while not losing information.

- An important point which came up during discussion with a community member was the lack of continued support for projects. As a future plan, I would like to be committed to this project and make sure that the plugin fits user needs, and meets industry standards.

- Jenkins is an absolutely amazing tool, which I have had the chance to use in 2 projects. Post this project, I would be connected with the Jenkins community and be developing for Jenkins as the tech ecosystem grows. I would love developing tools which will enhance interoperability between Jenkins and external tools.

- Encourage developers to incorporate Jenkins in their workflows by presenting the power of developing with Jenkins through various platforms and projects I am a part of.

* * Conflicts of Interest or Commitment:

No conflicts which would hinder me to develop and deliver this project. I am committed and excited to work on this project.

Why do I Want to Work on a Jenkins Project:

Ever since I have started coding, I have been fascinated by CI/CD automation tools. Jenkins is a really special CI tool for me. It was being used as an automated test tool in the first professional project I was a part of. We had a Gerrit infrastructure for Code Review, with resources for Web, Mobile, Server Config and more. Jenkins was being used to test the stability of our web and mobile apps, and also the health of our API, and pushing results to our Slack App. I was so fascinated by how vast yet easy to use Jenkins was. It was always so fun visiting the console and looking at the stability of our projects. It was allowing the developer team to be able to spend more time developing over having to worry about build stability. It simplified our workload. And the coolest thing was the availability of plugins to extend Jenkins to external systems (like Gerrit). As a User, we had developer freedom to configure the test pipelines as we wanted with custom variables or a custom environment. It was and still is such an amazing tool, and this project came at the right time allowing me to discover and build Jenkins. Jenkins is a holistic Software System, and has a variety of challenging and fun solutions to think about and/or work on. I want to be an insider of the jenkins community. As a user of Jenkins, I have had the chance of using this truly diverse CI/CD tool, and now I want to help build it even further. As I was researching for this project, I was so curious and interested in learning about how Jenkins is built by an amazing community. This project will help me work on a tool I am really curious about and inspired by!

Relevant Background Experience:

- As a Jenkins User: I have used **Jenkins as an automated test and build tool** for 2 different projects. The first project was testing the stability of Android App Bundle. This project employed Gradle Invocation to build and test the artifact from a Gerrit Push Trigger. The second project was using Docker-plugins to build and publish Docker Image to a central repo with the Jenkins Build-Number being used as the tag. Working on these projects has given me a general idea of **workflow implementation on Jenkins and how plugins can be configured to incorporate additional Build Steps into Jenkins**.

- I have **designed a smaller-scale Home-Automation MQTT pub/sub IoT project using C# and Arduino**. An integral part of the project was configuring topics to which a Publisher can send “events” to as they occur. The subscribers were subscribed to relevant topics, and would trigger an action accordingly. This project used MQTT as the transportation protocol. This **project laid a good idea of an EDA system, and how it can be designed for processing of events without blocking**.

- I was working on a **Microservice-based project**, which is a part of a startup in Toronto (Connexa.ai). I **designed a resilient and distributed data ingestion and streaming platform using AWS managed services**. Primarily, AWS Firehose for ingesting data from an Amplify Client App. This data consists of information on current open user interactions, and needs to be processed right away. The solution I designed included using AWS Firehose for ingesting data into an S3 bucket, and using Glue alongside Glue Crawlers to analyze the data. This project will help me brainstorm strong infra ideas for Jenkins EDA system.

Relevant Language Skills:

- Java (Intermediate)
- Go (Beginner/Basic)
- Jenkins (Intermediate)
- Networking (Intermediate)
- Git/GitHub (Intermediate/Advanced)

Extra Personal Info:

I’m a Junior (Undergrad) at Kalamazoo College, Michigan majoring in CS, returning back to college this Spring (April’21 - June’21) after a health break. I am from Kanpur, India, where I am also currently located, doing remote study and work in the light of COVID-19. I will be in Kanpur, India for the whole duration of the project. I am a Cloud Architect and Developer, with a focus on delivering applications that scale. I have worked with multi-industrial tech spaces, and am known to have taken and worked successfully on complex projects. I am extremely passionate about CI/CD tools which enhance developer workflows and user-experience. I recently delivered a talk for GitHub Satellite, which is centered around DevOps architecture through GitHub Workflows, which can be found on YouTube. This talk was a result of some months of discovering DevOp architecture through GitHub (or similar tools), and developing applications which require iterative development. I am always interested in, and inspired by how CI/CD tools are built as they are a complete tech solution which enhances other tools and systems. I also recently developed for IBM Call for Code, and our team of 3 won the first prize for predicting wildfires in various territories in Australia and making a production-ready ML Pipeline. Above all, I am a curious learner, passionate and an innovative thinker who loves to brainstorm ideas with the team, and deliver. I am always ready to challenge myself, after all, that's the only CI/CD way to enhance human-knowledge!