CISCO SECURE ENDPOINT DATA ENGINEERING PROJECT

LOG GENERATOR FINAL REPORT

Prepared by:

Chinwe Ajieh

David Zarinski

Florian Bache

Ling Lee

APRIL 14, 2023

COMPANY: Cisco Secure Endpoint

CISCO

Table of Contents

List of Figures

List of Table

1.0 Executive Summary

The Cisco Secure Endpoint team is responsible for handling endpoint security for their customers. Large amounts of logging data, collected via endpoint connectors, are sent to the Secure Endpoint team. These logs with varying number of fields are processed real-time to detect anomalies and malicious actors; then reported back to the customer. The team also focuses on developing and maintaining the infrastructure required to handle a continuously evolving amount of customer data. [7].

The goal of this project is to deliver a log generator microservice product that improves upon the log generator that is currently in place, by generating larger volumes of data that is additionally more indicative of a real computers' activity. With a randomized log generator application and machine learning model, Cisco will be able to handle the evolving amount of customer's data and detect malicious actors effectively. The log generator application would generate a variety of log types and will allow for more thorough testing of the Cisco Secure Endpoint solution in a controlled environment. [7].

The source code and additional information can be found in the public repository for this project, available at https://github.com/andymbwu/log-generator.

1.1 Motivations and Objectives

The main aim of this project is to build a microservice application that can generate randomized, usable computer logging/telemetry data in JSON format.

With the growing number of customers and amount of data being generated daily, Cisco Secure Endpoint needs to continually maintain and develop infrastructure required to handle these data. One of the challenges of dealing with customer data is being able to forecast future data requirements and proactively evolve existing infrastructure to handle such forecasts. To be able to forecast future requirements, the team continuously benchmarks their infrastructure by consuming randomly generated data that looks like data received from a customer. Instead of rewriting certain services before they break, Cisco Secure Endpoint wants to be able to forecast the problem. Hence, the need for an application that can randomly generate customer data and simulate real world usage and malicious activity. With this project, Cisco can benchmark the performance of their infrastructure using our log generator application and Machine Learning Model research. [7].

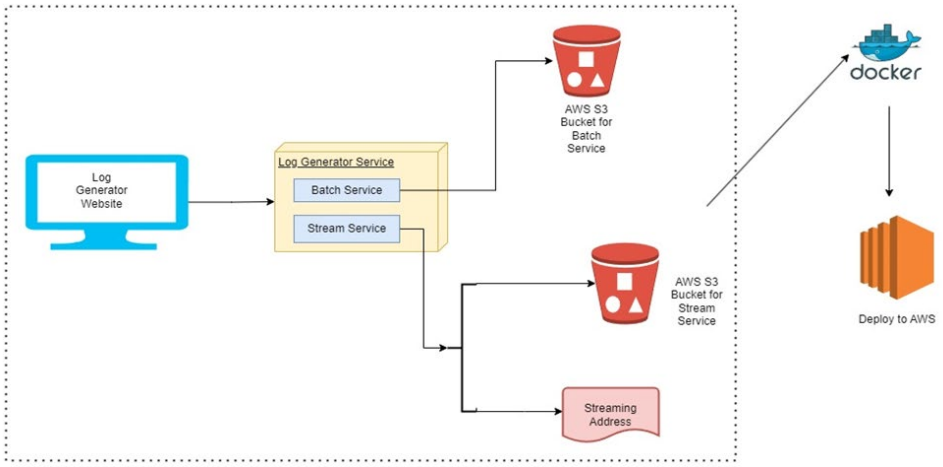


Figure 1.1.1: Conceptual Diagram of the Log Generator Application

1.2 Main Accomplishments

To create a scalable microservice application that can generate log lines in either batch or streaming mode with customizable parameters, we broke the project into three phases. For execution of the three phases, we looked at various topics in Software Development, Cloud Computing and Machine Learning.

Phase 1: Creation of a performant and scalable microservice application that can generate logging data in batch and stream mode.

For this phase, we built a microservice application using Java and Spring Boot framework for the backend and REACT for the frontend. A user can specify various log generation settings using the web application and specify in what mode to generate the logging data. If batch mode is selected, the user gets to indicate the number of log lines to be generated and the application would output the specified logging data in JSON format. If stream mode is selected, the user is prompted to specify an address to stream the logging data. The logs are streamed continuously, in JSON format to the specified address, until the user stops the streaming process. The log generator runs as a multithreaded application.

We explored the following topics in Software development to successfully implement this phase:

- Building microservices applications using Java and Spring Boot [8] [28].
- Exception Handling in Spring Boot [28].
- Generating logging data in batch and stream modes.
- Multithreading in microservice application [8] [30].
- Building real time rest APIs [8] [28] [29].
- Building frontend application using REACT [17].
- Streaming logs to specified file path [23].
- Real time communication between frontend and backend application [5] [15][30].

Phase 2: Log Generation in AWS S3 buckets and deployment of the application to scalable cloud infrastructure.

For this phase, we created two AWS S3 buckets to save the generated logging data. Logs are generated in batch mode and saved in JSON format to AWS S3 bucket. For stream mode, logs can be generated continuously to AWS S3 bucket or at a default buffer size to manage resource utilization and to improve the performance and stability of the log generation process, making it more scalable and efficient. Upon completion of the log generation process, the user can access the AWS object URL to view or download the generated logging data.

We deployed the microservice application to AWS. The log generator application has been dockerized with the frontend and backend as separate containers. Please see figure 1.2.1 on the steps to dockerize a Spring Boot application. Once the application is dockerized, the containerized application is deployed to AWS.

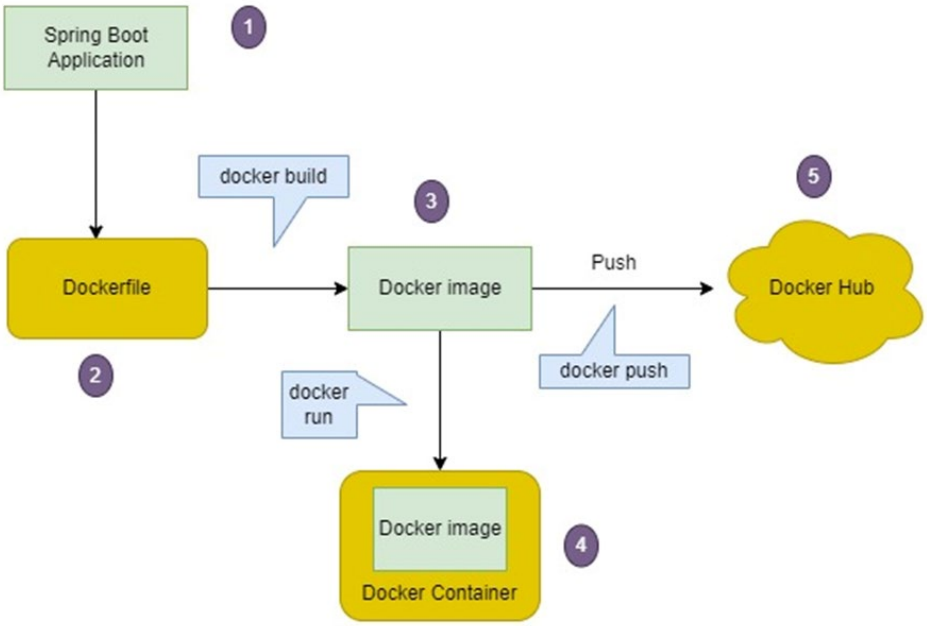


Figure 1.2.1: Steps in Dockerizing a Spring Boot Application [27]

For the implementation of this phase, we successfully completed the following:

- Utilizing AWS S3 buckets to store generated batch and stream files [4] [6] [32].
- Utilized our knowledge of AWS SDK to connect Jenkins, Docker, and the following AWS products: S3, EC2, ECR and ECS [31] [14].
- Configuring a containerized microservice application using Docker [18][27].
- Deployed containerized applications to AWS using Elastic Container Service (ECS) [11][12][13][19][31].
- Building an automated pipeline using Jenkins to automatically deploy an updated version of our application upon a push to our GitHub repository [3][21][25].

Phase 3: Making the generated logs more realistic using machine learning model, adding options for the user to customize the logs generated, developing unit tests and documentation.

In this phase, we researched options to modify the microservice application to generate increasingly realistic data using machine learning model. The goal of this research is to outline the next steps to generate more realistic data by having the model learn from other logs and try to find sequences of events to generate more realistic data. In addition, we implemented some options for the user to be able to customize the logs generated. If the user does not want randomly generated values for the standard fields, they have the option to be able to specify every value that is used. If the user specifies a single value, then every log will use that value. If they specify two or more values, it will be equally distributed between the logs generated. Furthermore, an option was added for the user to add multiple log lines of their choice in JSON format and specify a frequency of that log as an input to the program. The log lines provided by the user can have any fields filled out, and if it is missing the standard fields, then those will be generated randomly. When our generator is generating logs, they will include the custom log lines at the specified frequency. The use case for this is the user will be able to generate log lines that simulate a certain type of compromise or malicious threat actor.

For the implementation of this phase, we researched the following topics in Machine Learning and Deep Learning:

- Developing a machine learning/deep learning model to create a smart event simulator using Python [1] [2].
- Generative models [16].
- Generative deep learning: Teaching machines to paint, write, compose, and play [10]
- Synthesizing Tabular Data using Generative Adversarial Networks [24]
- CTAB-GAN: Effective Table Data Synthesizing [33]
- The Dying ReLU Problem [22]

2.0 Methodology

To fully implement the project, we came up with the following methodology for each phase of the project (See figure 8.3 in Appendix for a detailed UML class diagram of the log generator application):

Phase 1: Random Log Generator

Create a performant and scalable microservice application that can generate logging data in batch and stream mode. (See figure 2.0.1 for a package level illustration of this phase)

- Build a web application using REACT framework as per recommendation by sponsor.
- User can specify the log generation settings. The log generation settings would include:
  - Fields to include and optionally specify values:
    - ✓ Time stamps
    - ✓ Processing Time
    - ✓ Current user ID
    - ✓ Business GUID
    - ✓ Path to File
    - ✓ File SHA256
    - ✓ Disposition
  - Custom logs to include:
    - ✓ Custom log frequency
    - ✓ Custom log fields
  - Log selection mode:
    - ✓ Batch (If batch mode is selected, user to specify a batch size)
    - ✓ Stream (If stream mode is specified, user would be prompted to input a streaming address, and log rate in logs/s)
- User can start and cancel a job at any time.
- User can track job progress and see performance statistics in a chart showing logs generated each second over time.
- User can view and cancel all active jobs.
- User can view a job history page to view all jobs (cancelled, completed, ongoing or failed jobs).
- Frontend Libraries - The following packages and libraries are required to build the log generator application website using react: @chakra-ui/icons, @chakra-ui/react, @emotion/react, @emotion/styled, @stomp/stompjs, @testing-library/jest-dom, @testing-library/react, @testing-library/user-event, formik, framer-motion, react, react-apexcharts, react-dom, react-icons, react-router-dom, react-scripts, react-table, sockjs-client, uuid, and web-vitals.
- Create the microservice log generator application (backend) using Java and Spring Boot framework, in line with what Cisco uses. For this phase, logs can be generated in batch and stream mode to a default file path; it can also be streamed continuously to a user provided address. The logs generated would be in JSON format.

Architecture (Model, Service, Controller and Exception Layers):

Model layer to include:

- AllJobsMetrics- model class containing all variables to view the status of all jobs.
- BatchJobMetrics- model class consisting of all the variables required to view the metrics of a batch job.
- BatchSettings- consist of the variable, number of logs generated in batch mode.
- BatchTracker – consists of variables required to track the metrics of a batch job for asynchronous communication with the frontend.
- ContinueMessage – consists of variables, job ID and message.
- FieldSettings – model class consisting of the field settings for a user to specify.
- JobStatus – consists of the job status (active, completed, failed, cancelled).
- LogMessage – message for the logs generated which would include the log counts and time stamp.
- LogModel- model class containing the variables for the fields a user is required to specify (time stamps, processing time, current user ID, business GUID, path to file, File SHA256, disposition)
- SelectionModel- represents the user specified configuration for a batch or stream job. The field settings, malware settings and log selection mode settings variables would be contained in this class.
- MalwareSettings- will contain the variables for the different malware settings- Trojan, Adware, Ransom.
- StreamJobMetrics - model class consisting of all the variables required to view the metrics of a stream job.
- StreamSettings - consist of the address to stream to for stream mode.
- StreamTracker - consists of variables required to track the metrics of a stream job for asynchronous communication with the frontend.

For the business logic of the log generator application, we have the Service Layer. For this phase, the Service Layer would include the following classes:

- BatchService- contains the logic for asynchronously generating the specified number of logs to local filesystem using LogService, or until a request to stop the batch job.
- BatchTrackerService – contains the logic to continuously send server messages with log/s metrics to a topic specific to each active batch job.
- LogService – handles log generation based on specified fields from the user.
- StatisticsUtilitiesService- handles logic to generate the metrics for all jobs.
- StreamingService- contains the logic for asynchronously generating logs continuously, using LogService, to local filesystem or to user specified address with HTTP requests until job times out or a request to stop stream job.
- StreamTrackerService- contains the logic to continuously send server messages with log/s metrics to a topic specific to each active stream job.

To provide access defined in the service layer to the web application, we have a controller layer. A user's input will be interpreted by the controller which is then transformed and transmitted via endpoints to the user via the web application. The Controller Layer, in this phase would include:

- LogController- controller class to house the endpoints to handle the requests to start a new asynchronous batch or stream job, to stop an existing job, for logs generated to a default file system or user specified streaming address. Also, endpoints to get the metrics of executed jobs.

To handle exceptions in our log generator application, we would have an Exception Layer. The Exception Layer would include the following classes:

- AddressNotFoundException- handles specified exception when a stream address is not found.
- ErrorDetails- a class that contains the variables to describe error message details.
- FilePathNotFoundException- handles specified exception when file path is not found.
- GlobalExceptionHandler- class to handle global exceptions.
- JobNotFoundException- handles specified exception when a job ID is not found.

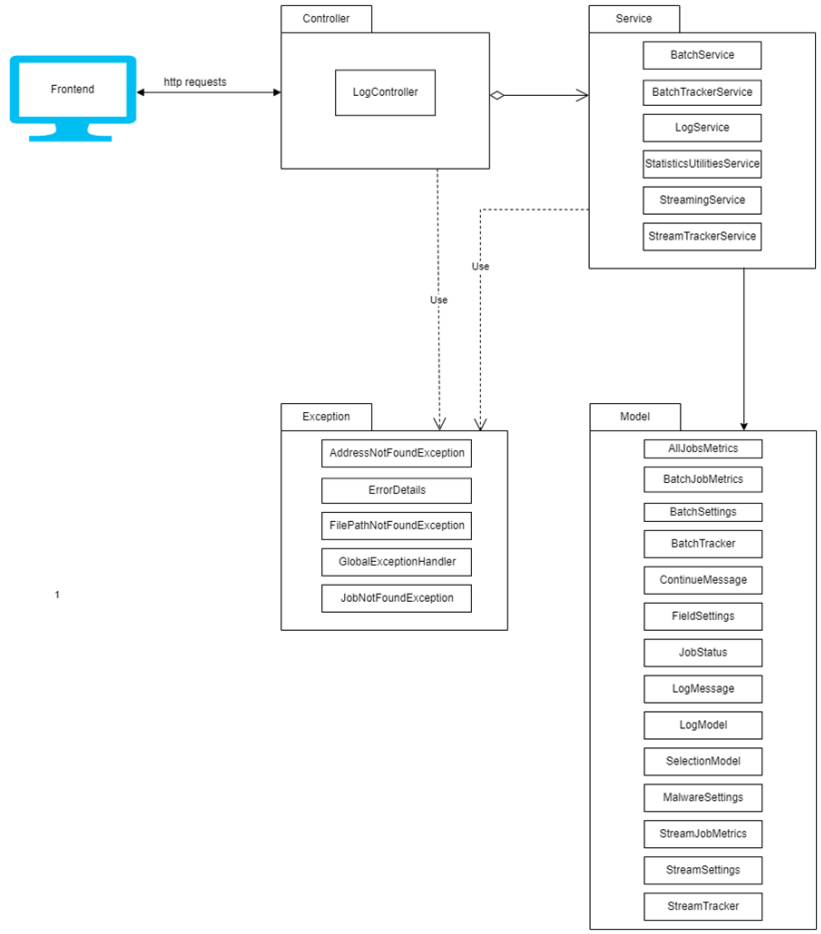


Figure 2.0.1: Package Level Diagram of Phase 1 of the Log Generator Application

Phase 2: Log Generation in AWS S3 buckets and deployment of the application to scalable cloud infrastructure.

Log Generation in AWS S3 buckets

For this part of phase 2, we would add the necessary classes for log generation to AWS S3 bucket.

- Refactor the controller layer so we have two controller classes as follows:
    - LogController: controller class to house the endpoints to handle the requests to start a new asynchronous batch or stream job, to stop an existing job, for logs generated to AWS S3 bucket or user specified streaming address. Also, endpoints to get the metrics of executed jobs.
    - LogToFileController: controller class to house the endpoints to handle the requests to start a new asynchronous batch or stream job, to stop an existing job, for logs generated to a default file system.
- Add three new classes to the existing service layers as follows:
    - AWSLogService- consists of methods to aid the generation of logs in batch and stream mode to AWS S3 bucket.
    - AWSBatchService- class to handle the logic behind generation of logs in batch mode to AWS S3 bucket using the methods in AWSLogService.
    - AWSStreamService- class to handle the logic behind generation of logs in stream mode, continuously, at a specified buffer size and to an address with save option to AWS S3 bucket using the methods in AWSLogService.
- Add a new exception class, AWSServiceNotAvailableException to the existing exception layer. This class would handle Amazon Web Services (AWS) S3 Service not available exception.

See figure 8.1 in Appendix for a package level diagram of phase 2 of the log generator application.

Deployment of the application to scalable cloud infrastructure

To deploy our full stack application, we utilized several technologies to containerize our applications, automate the deployment and undeployment processes and host our applications on the internet to make it accessible to anyone. These technologies and the overall deployment strategy are outlined below:

- Docker was used to prepare Docker images of both the frontend and backend based on their respective Dockerfiles
    - The Dockerfiles used lightweight Linux distributions as their underlying operating system. The images were then packaged to include the source code and all required dependencies were installed. Container ports 8080 (backend) and 80 (frontend) were exposed.
- Amazon Web Services (AWS) was used as our deployment platform to align with Cisco.
    - Elastic Container Repositories (ECR)
        - As part of the configuration process, two repositories will be configured within the ECR for both the front and backend. These repositories will be used to store all versions of the docker images pushed via the Jenkins pipeline.
    - Elastic Compute Cloud (EC2)
        - Provides scalable computing capacity in the cloud. Used to launch virtual machines in the AWS cloud, that will run the Docker containers.
    - Elastic Container Service (ECS)
        - An Elastic Container Service (ECS) cluster is a container orchestration and management tool. An ECS cluster is created and configured to specify how the full stack application will be deployed. Part of the configuration process is specifying compute resources, network configurations and the security settings. Within the ECS, task definitions are

created for both the front and backend. These task definitions link to the specific containers within the ECR and allow for some additional configurations. Once configured the task definitions are used to run tasks, which represent running instances of the multiple containers within the cluster.

- Continuous Integration/Continuous Deployment (CI/CD) tool called Jenkins. This tool uses the pipeline concept to define the various stages required to deploy an application. The pipeline automates the entire deployment process, so whenever a new change is made to our application, a new version of
- The diagram below provides an overview of how the various technologies work together to ultimately deploy a new version of the application whenever an update has been made to our source code stored within our GitHub project repository.

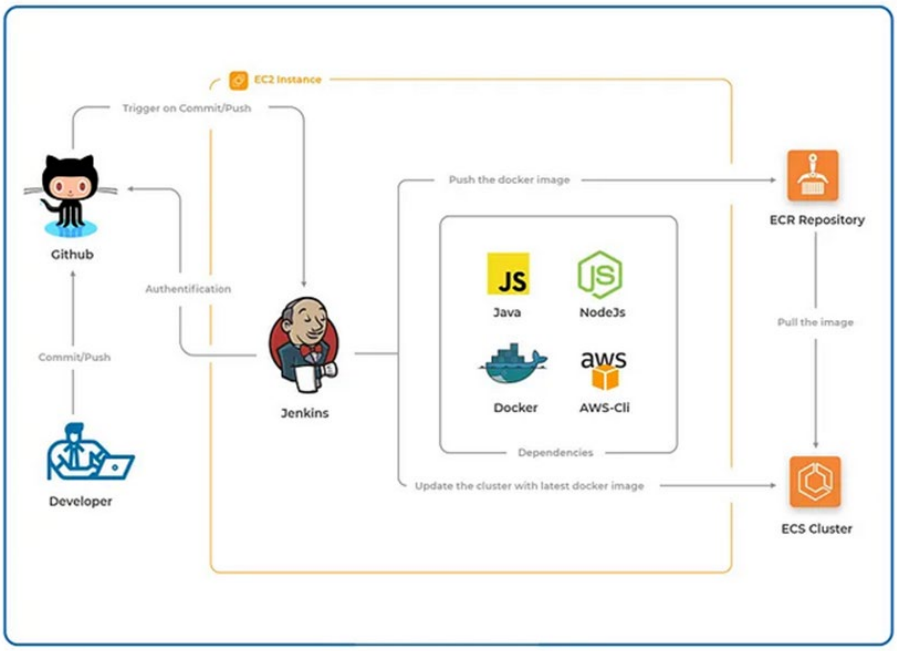


Figure 2.0.2: CI/CD Pipeline with Docker Architecture [3]

Phase 3: Research how to make the generated logs more realistic using deep learning model, add options for the user to customize the logs generated, develop unit, performance tests and documentation.

For this phase, we would conduct research on using Generative Adversarial Networks (GANs) to generate more realistic logs that mimic the patterns and characteristics of real logs for future implementation (see design step for the GAN research). We would add the options for users to customize the logging data to be generated. Also, unit and performance tests will be implemented to evaluate the performance of the log generator application.

Improving the microservice application and implementing options to generate user customizable log lines:

- User can specify additional log generation settings. The log generation settings added include:
  - Fields to include with optional custom values:
    - ✓ Time stamps
    - ✓ Processing Time

        - ✓ Current user ID
        - ✓ Business GUID
        - ✓ Path to File
        - ✓ File SHA256
        - ✓ Disposition
    - Log selection mode:
        - ✓ Stream (If stream mode is specified, user would be prompted to input a streaming address, log rate, and specify whether logs should be saved to s3 as well as streamed)
    - Custom logs to include:
        - ✓ Fields and values
        - ✓ Generation frequency
- User should be able to specify custom values that will be used for the 7 standard fields if the user does not want random values generated.
- User should be able to specify a log line of their choice as input to the program and a frequency of that log for the generator to output.
- User can understand when logs are being uploaded to s3 and when they're available to download
- User can specify whether if logs should be saved locally as well as streamed to an address
- User can specify a maximum log generation rate

Architecture (Model, Service, Controller and Exception Layers):

Changes to model layer:

- BusinessGuid – model class containing list of user specified values for the business id field.
- CurrentUserId - model class containing list of user specified values for the user id field.
- Disposition - model class containing list of user specified values for the disposition field.
- FileSha256 - model class containing list of user specified values for the file sha256 field.
- PathToFile - model class containing list of user specified values for the file path field.
- ProcessingTime - model class containing list of user specified values for the processing time field.
- TimeStamp - model class containing list of user specified values for the timestamp field
- CustomLog – consists of a dynamic map to allow the user to pass in their own custom fields and values, and a desired generation frequency.
- SelectionModel – represents the user specified configuration for a batch or stream job. Model classes for user specified values and custom logs were added to the configuration.
- StreamSettings – added option to specify a maximum log generation rate and option to save logs to s3 as well as stream to address.
- JobStatus – added the 'FINALIZING' status to represent the time during log upload to s3.

Changes to service layer:

- LogService – updated to generate user specified values for the 7 standard fields and to generate user specified custom logs.
- AWSStreamService – updated to adhere to specified log generation rate when streaming to an address with an option to additionally save logs to s3 while streaming to an address.

- StreamingService – updated to adhere to specified log generation rate when streaming to an address with an option to additionally save logs to default location in the file system while streaming to an address.

To validate that the total specified frequency of all custom logs passed as input to the generator does not exceed 100%, we added classes for validating the custom logs:

- ValidCustomLog – custom java annotation applied to the customLogs field in the SelectionModel class to validate the custom logs passed by the user.
- CustomLogValidator – implements the ConstraintValidor interface in the javax validation package. The isValid method contains the business logic for validating the custom logs passed by the user.

Unit testing the log generator microservice application:

To develop a test suite for our log generator microservice application, we utilized JUnit 5 to write our test cases, Mockito to mock collaborators and verify interactions for behavior-based testing, MockWebServer to simulate making post requests to a web server in stream mode, and MockMvc to enable support to make requests to our own endpoints and validate the response.

Test suite to include:

- AWSBatchServiceTest
    - Verify interactions with all collaborators.
- AWSStreamServiceTest
    - Verify interactions with all collaborators.
- BatchServiceTest
    - Validate that the batch job tracker objects have the correct log line count.
    - Validate that the batch job tracker objects have the correct job status.
    - Validate that the log file generated is valid JSON.
    - Verify the interactions with the SelectionModel collaborator.
    - Verify the interactions with the LogService collaborator.
- StreamingServiceTest
    - Validate that the stream job tracker objects have the correct log line count.
    - Validate that the stream job tracker objects have the correct job status.
    - Validate that the log file generated is valid JSON when streaming to file or s3.
    - Verify the interactions with the SelectionModel collaborator.
    - Verify the interactions with the LogService collaborator.
    - Verify an error is returned if an address is not available to stream to.
- LogServiceTest
    - Validate that the random disposition value generated is between 1 and 4.
    - Validate that the random file sha256 generated is a valid UUID.
    - Validate that the random file path generated is a valid Linux or windows file path.
    - Validate that the random business id generated is a valid UUID.
    - Validate that the random user id generated is a valid UUID.

- Validate that the random processing time generated is between 0 and 1000 seconds.
- Validate that the random time stamp generated is a time stamp in the past.
- Validate that custom values for 7 standard fields are always chosen if specified.
- Validate the process of picking a custom log or random log to generate.
- Validate the log that is generated when a custom log is chosen to be generated.
- Validate the log that is generated excludes fields that the user specifies.
- Validate the log that is generated when a random log is chosen to be generated.

- LogControllerTest
  - Verify the status and response body returned when requesting to start a new batch or stream job with valid or invalid SelectionModel configurations.
  - Verify the status and response body returned when requesting to stop a batch or stream job.
  - Verify the status and response body returned when requesting for metrics for jobs.
- LogsToFileControllerTest
  - Verify the status and response body returned when requesting to start a new batch or stream job with valid or invalid SelectionModel configurations.

2.1 Design

The project progressed well compared to the timeline proposed. Phase 1 was completed with a basic microservice running locally. Additionally, work on phase 2 (deployment to scalable cloud infrastructure) has been completed on schedule. Backend functionality was developed and tested which saves generated logs to an S3 bucket on AWS (Amazon Web Services). Additionally, the application was containerized for deployment, and the team has been researching different deployment options on AWS, with deploying to ECS with self-managed EC2 instances being tentatively selected.

Log Generator Timeline			DURATION (days)
DESCRIPTION	START DATE	END DATE	
Phase 1 – Basic microservice running locally Additional features will be continually added	January 16, 2023	February 25, 2023	39
Midterm Presentation	February 27, 2023	March 3, 2023	6
Midterm Report	February 27, 2023	March 10, 2023	13
Phase 2 – Deployment to scalable cloud infrastructure	February 25, 2023	March 25, 2023	30
Phase 3 – Use data to make generated logfiles more realistic	February 25, 2023	April 8, 2023	43
Final Presentation	April 8, 2023	April 14, 2023	6
Final Report	April 8, 2023	April 14, 2023	6

Table 2.1.1: Log Generator Timeline

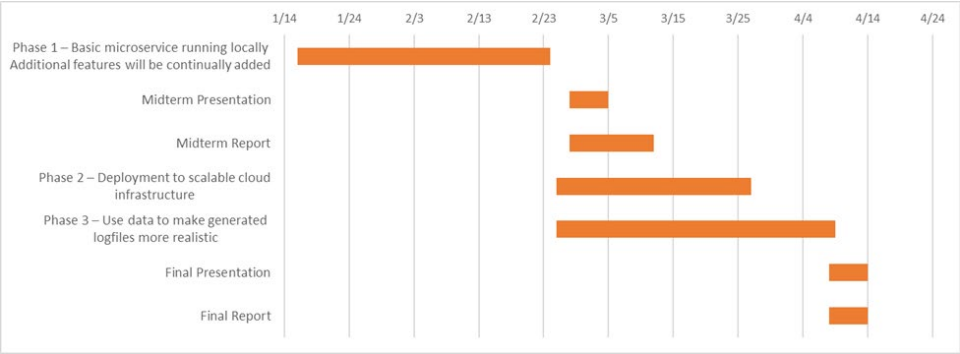


Figure 2.1.2: Project timeline

Server Phase 1 and Phase 3 Updates

The backend microservice application was designed by separating the classes by layer, making it easy to locate a class by type. As shown in our UML diagram, our application consists of the controller layer, the service layer, and the model layer. In addition, the config package was made to place our Spring Boot configuration classes, and the exception package was created to locate our custom exception classes.

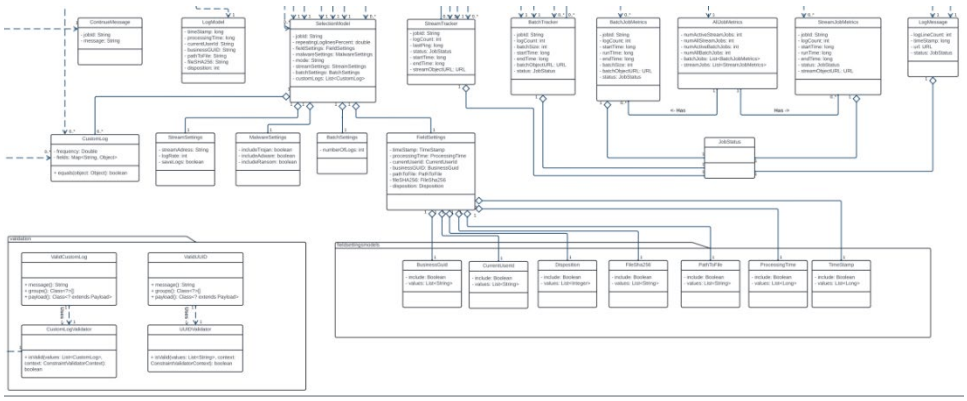


Figure 2.1.3: Model Package Class Diagram

The model layer consists of 22 classes. One of the key classes, the SelectionModel class, was designed for the user to provide configurations to specify how a log line should be generated. The configuration first requires the user to specify whether to generate the logs in batch or stream mode, a requirement of the project. If batch mode is selected, the user must additionally specify how many logs they would like to generate. If stream mode is selected, the user must specify whether they would like to continuously generate logs to a file, by omitting the address to stream to, or if they would like to continuously generate logs sent to an address through HTTP requests, by providing the address to stream to. As per requirements, the user can specify which fields they would like to optionally have included in the log lines generated. The logs generated are represented by the LogModel class, which includes 7 standard optional fields: businessGuid, currentUserId, disposition, fileSha256, pathToFile, processingTime, and TimeStamp.

During phase 3, the scope of the project was altered due to tight time constraints; Instead of developing a machine learning model to generate more realistic logs, the user should be able to provide their own custom values for each of the 7 optional fields, or the user should be able to specify their own custom logs in JSON format in the user interface with a specified frequency of appearance. To allow the user to specify their own values for the 7 optional fields, 7 classes were added to the SelectionModel configuration class: BusinessGuid, CurrentUserId, Disposition, FileSha256, PathToFile, ProcessingTime, and TimeStamp. The values specified by the user will be serialized into a List<T> attribute in each respective class. To allow the user to specify their own custom logs, the CustomLog class was developed as part of the SelectionModel configuration specified by the user, and contains a dynamic Map<String, Object> attribute to allow the user to pass in their own fields and values and desired generation frequency. The fields specified can be any field at all, including the 7 optional fields that are generated by our server. In addition, validations were added to the server to ensure that the total frequency for all CustomLog's does not exceed 100%.

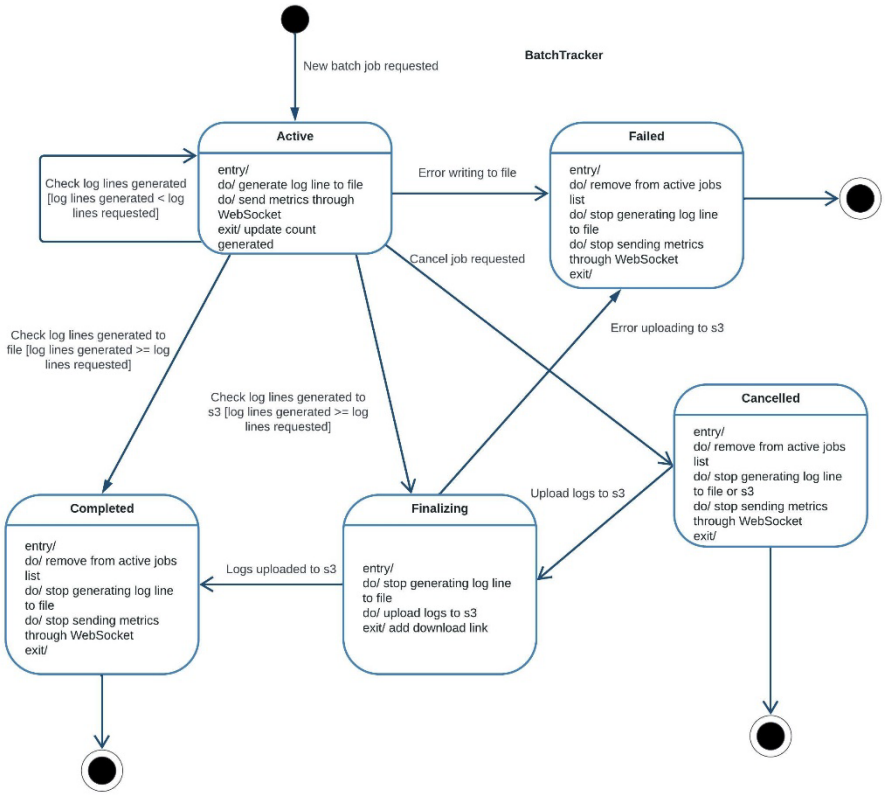


Figure 2.1.4: BatchTracker state transition diagram

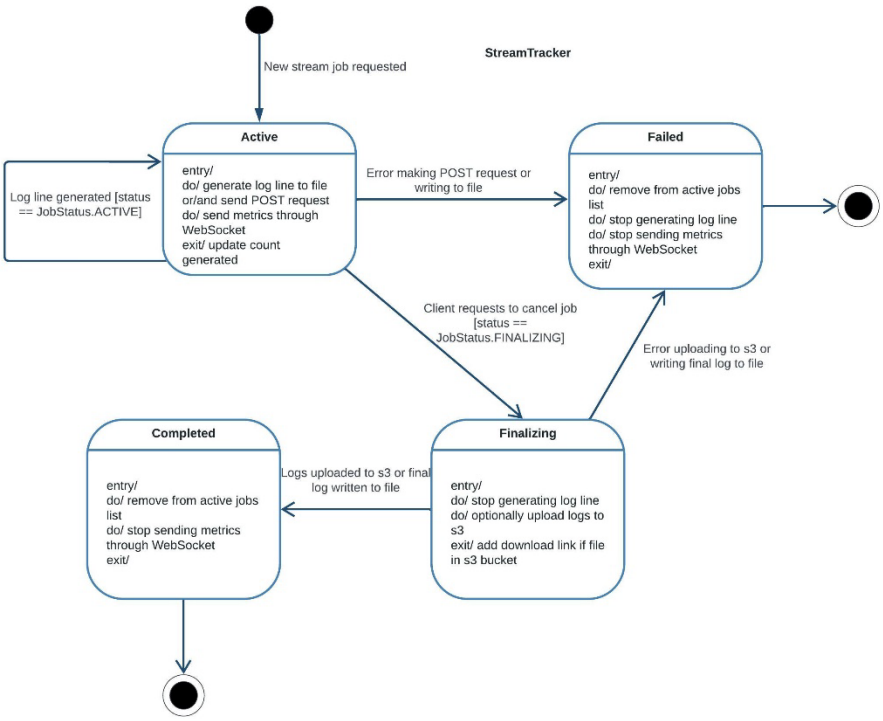


Figure 2.1.5: StreamTracker state transition diagram

As the project progressed, the requirement to provide metrics for a batch or stream job was requested by the sponsor. To fulfill this new requirement, the BatchTracker and StreamTracker classes were designed to keep track of the metrics for each batch job and stream job respectively. Some key metrics include the number of logs lines currently generated, the start time and end time of the job, as well as the status of the job. As shown in the state transition diagrams in figure 2.1.4 and figure 2.1.5, when a batch or stream job is requested, a new job tracker object is created, it is given a unique job id, and the status is set to 'ACTIVE'. If a batch job is cancelled prematurely, the status is switched to 'CANCELLED' and will remain in that status if the destination of the logs is the local file system. If an error occurred during the log generation process, the status becomes 'FAILED'. Finally, if a batch job is completed successfully or is cancelled prematurely, and the destination of the logs generated is in a s3 bucket, the status will be changed to 'FINALIZING' until the logs have been uploaded successfully, at which point the status changes to 'COMPLETED'. Similarly, once a stream job has been requested to stop, the status of the job tracker is set to 'FINALIZING' until the final log has been written to file or the logs have been uploaded to s3 bucket, at which point the status changes to 'COMPLETED'. Alternatively, if an error occurs during upload to s3, the status of the job tracker object is changed from 'FINALIZING' to 'FAILED'. The log generation process for a particular job will continue if the status of its respective job tracker object is 'ACTIVE'. The log generation process will be explained in further detail during discussion of the service layer.

Another key usage of the job tracker objects is to generate metrics upon request for each batch or stream job. These metrics are represented by the BatchJobMetrics, StreamJobMetrics, and AllJobMetrics classes, which currently include metrics such as the number of log lines currently generated, the run time in seconds, the status of the job, the number of active stream and batch jobs, the URL to the file in s3 if applicable, and the number of

all batch or stream jobs. These metrics will be displayed on the ‘Active Jobs’ and ‘History’ pages in the front end in tabular format.

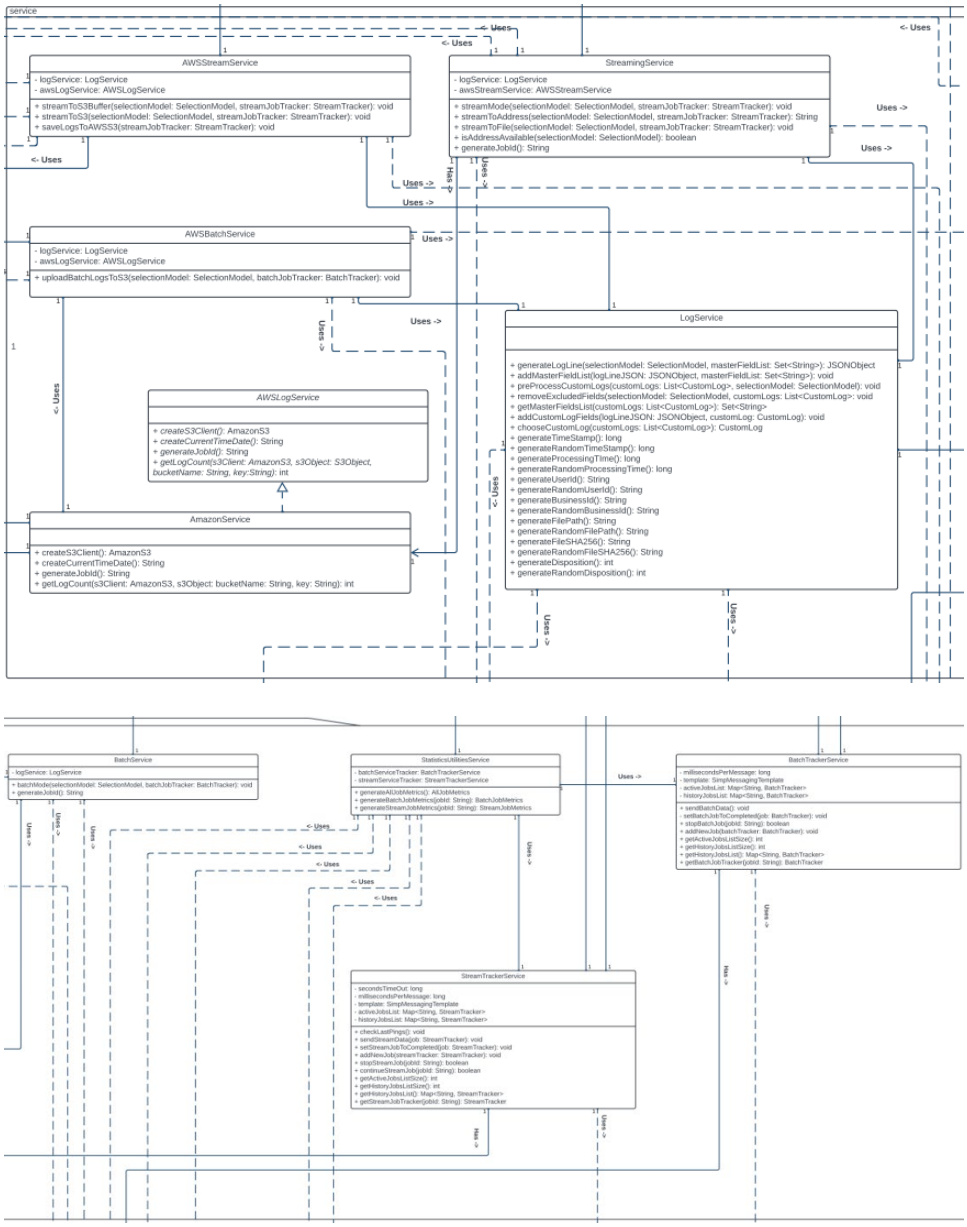


Figure 2.1.6: Service Package Class Diagram

The service layer consists of 9 concrete classes. The first step was developing the LogService class, which is responsible for taking the user specified configuration in the SelectionModel and generating a log line which is returned as a JSONObject to the caller. To adhere to SOLID principles, the logic to generate each field was separated into individual methods. The LogService class generates the values for these fields randomly as part of phase 1. To generate a random file path, currently a static list of folder names and extensions are randomly

selected from and concatenated together. The goal of phase 3 is to improve upon this class to generate log lines that are more realistic.

As part of phase 3, the LogService class was modified to optionally allow the user to specify values for each field and/or provide custom logs to generate with a specified frequency. If the user did not specify custom values for the 7 fields and did not provide any custom logs, then randomly generated logs will be returned to the caller, as before. If the user specifies values for any of the 7 fields and did not provide any custom logs, the LogService class will randomly choose one of the specified values for each respective field and generate random values for the fields that did not specify custom values. If the user specifies custom logs to generate and does not provide specific values for the 7 fields, the LogService class will first create a list of each unique field from all custom logs. Then the LogService class will decide whether to return a custom log or to randomly generate a log, depending on the frequency specified for each custom log. If a custom log is chosen, the LogService class adds all the fields and values specified for the chosen custom log to the generated log. Next, it will add the 7 optional fields with randomly generated values to the generated log if values for these fields were not specified in the chosen custom log. Finally, it will add all other unique fields from all custom logs with null values to the generated log if values for those fields were not specified in the chosen custom log. If a custom log is not chosen, the generated log will contain null values for each unique field from all custom logs and randomly generated values for the 7 optional fields. If the user specifies custom logs to generate and provides specific values for the 7 fields, the main difference is the LogService class will choose one of the specified values instead of randomly generating one. Therefore, when custom logs are provided, all logs will have the same fields, a set of every unique field including the 7 optional fields and custom log fields.

Following the completion of the LogService class, the next logical step was to develop the BatchService class to generate a log file to fulfil the requirement of being able to generate logs in batch mode. The BatchService class is responsible for writing the log lines generated from the LogService class to a file in the local file system as part of phase 1. The BatchService class simply calls the LogService class to generate log lines, as per the logic for generating logs described above, for as long as the number of logs generated is less than the number of log lines requested, or until the client requests that the batch job is stopped prematurely. Each time a log line is generated, the corresponding log count metric in the BatchTracker object for the respective batch job is incremented. If the batch job is completed without being prematurely stopped by the client, the status of the job is set to ‘COMPLETED’ in the BatchTracker object. Alternatively, if an exception occurs, such as a problem with writing the log line to file, then the status of the job is set to ‘FAILED’.

The StreamingService class fulfils the requirement for being able to continuously generate logs to file or an address until the client requests to stop. Like the BatchService class, the logic simply calls the LogService class to generate log lines to a file or sends them in the body of a POST request to the user specified address, depending on the configuration in the SelectionModel as explained in the discussion of the SelectionModel class, until the client requests the stream job be stopped. Additionally, upon each log line that is generated, the corresponding metric in the StreamTracker object is incremented, and the status is switched to ‘FAILED’ if an exception occurs such as being unable to write to file or when a POST request fails. If the client requests the stream job to stop, the status of the job is set to ‘COMPLETED’ in the StreamTracker object, ending the stream job. The StreamingService class also provides a utility method to first check whether the user specified address is available to accept POST requests before starting the job.

As part of phase 3, the StreamingService class was updated such that if the user specifies to save the logs while making post requests to the user specified address in the SelectionModel, the StreamingService class will save the logs locally as well. Furthermore, an additional configuration was added to the SelectionModel class to allow the user to specify a maximum log generation rate, which the StreamingService class will attempt to adhere to by delaying log generation if it is too fast. Finally, to improve the performance of a stream job that streams to an address, the StreamingService class was updated to generate a batch of 10 logs before making a POST request to the specified address rather than making a POST request with a single log.

At this stage, the group encountered a challenge. To ensure that the application was scalable, we needed to make changes to convert the StreamingService and BatchService classes to run asynchronously. Through our research, we learned that by configuring Spring Boot we could expose the thread pool task executor and enable Spring's asynchronous method execution capability [15]. By applying the '@Async' annotation to the appropriate method, the log generation methods of the above classes were implemented to run asynchronously, ensuring that our application could generate log lines for multiple requests concurrently.

We next considered the challenge of generating and displaying metrics for the batch and stream jobs and defining a communication protocol between the front end and the back end. We first considered using HTTP requests to communicate our metrics, the disadvantage in this was that a large number of requests would be required to generate a real-time graph at the front end. Upon further research, we learned that by configuring Spring Boot we could use Spring's WebSocket project to establish low latency and high frequency messages to communicate our metrics [30].

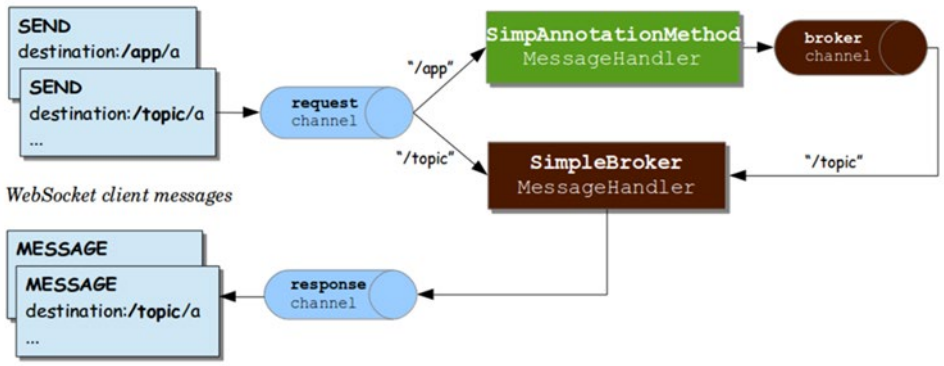


Figure 2.1.7: Spring Boot WebSocket Components [30]

For our use case of returning metrics from the back end to the front end for each job, we decided that the front end would subscribe to a destination that is made unique for each job by appending the job id of that job to the destination, and the backend would periodically send messages with metrics to a topic for each active job through the broker channel. The message would then be processed in the front end and a graph displaying the logs/s metric would be generated.

The decision to use the WebSocket protocol for communication led to the design and implementation of the BatchTrackerService and StreamTrackerService classes. These classes contain two ConcurrentHashMap's with the job id as the key and the BatchTracker and StreamTracker objects as the values respectively. The first

HashMap consists of all jobs created and the second HashMap consists only of active jobs. When a new batch or stream job is requested, a new BatchTracker or StreamTracker object is created and added to both HashMap's. The primary responsibility of these classes is to periodically send logs/s metrics to a topic for each active batch or stream job by iterating over the HashMap of active jobs. A message will be sent through the web socket for each job until the status of the job has been changed from 'ACTIVE' or 'FINALIZING', at which point the job tracker object is removed from the HashMap of active jobs and messages are no longer sent. These classes additionally contain methods to change the status of a job tracker object from 'ACTIVE' to 'CANCELLED' or 'FINALIZING' when a specific job is requested to be stopped by the client. Changing the status to 'CANCELLED' or 'FINALIZING' will stop the BatchService and StreamingService classes from generating logs, and in the StreamingService class, the status is changed to 'COMPLETED' after it finishes writing to file. In the AWSBatchService and AWSStreamService classes developed as part of phase 2, the status will be changed from 'FINALIZING' to 'COMPLETED' after successfully uploading to s3, or alternatively from 'FINALIZING' to 'FAILED' if there are issues uploading to s3. Thus, when a job is requested to be cancelled, or when a job is completed, or when a job fails, metrics for that job will no longer be sent through the web socket. The HashMap which contains all jobs created is used to generate additional metrics that will be displayed in the 'Active Jobs' and 'History' pages in the front end.

The last service class implemented as part of phase 1 is the StatisticsUtilitiesService class, which simply contains logic to generate metrics described in the BatchJobMetrics, StreamJobMetrics, and AllJobMetrics classes by accessing the HashMap containing the job tracker objects of all jobs in the StreamServiceTracker and BatchServiceTracker classes.

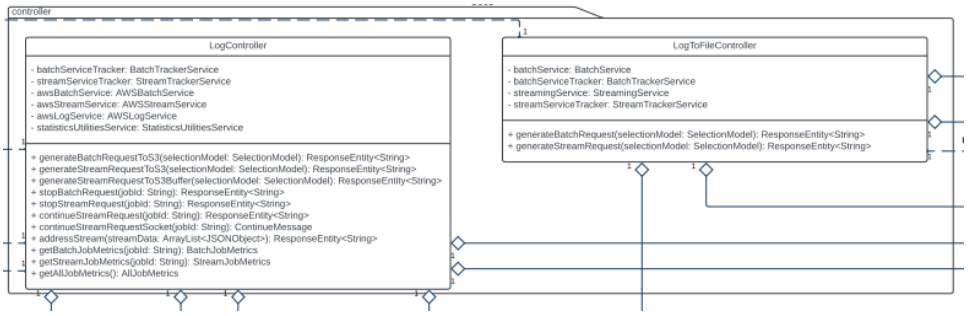


Figure 2.1.8: Controller Package Classes

The controller layer consists of two controller classes, LogController and LogToFileController. These classes have 11 HTTP end points. Firstly, LogToFileController consists of two end points "/api/v1/generate/batch" and "api/v1/generate/stream" to start a new asynchronous batch or stream job that generates logs to the local file system respectively. As part of phase 2, the endpoints "/api/v1/generate/batch/s3", "/api/v1/generate/stream/s3", "/api/v1/generate/stream/s3/Buffer", and "/api/v1/generate/stream/s3/toAddress" were added to LogController start a new asynchronous batch or stream job that generates logs to a s3 bucket. Each end point accepts a SelectionModel configuration, creates a new job tracker object and adds them to either the BatchServiceTracker or StreamServiceTracker classes to begin sending metrics through a web socket, and starts the asynchronous log generation methods. They return

the job id for the requested job immediately so that the client can use it to request to stop or continue a job, or request for metrics for that job.

The LogController class additionally contains two end points "/api/v1/generate/batch/stop/{jobId}" and "/api/v1/generate/stream/stop/{jobId}" to request to stop a batch or stream job respectively, which will update the status of the job tracker object specific to that job id to "CANCELLED" or "FINALIZING" depending on the type of job and the final destination of the logs, effectively ending the log generation process and ending the communication of metrics through the web socket.

Finally, the end points "/api/v1/generate/stats/batch/{jobId}", "/api/v1/generate/stats/stream/{jobId}", and "/api/v1/generate/stats" can be used to request for metrics generated by the StatisticsUtilitiesService for a specific job or for all jobs.

Server Phase 2

For generating logs to AWS (Amazon Web Services) S3 in batch and stream mode, we created two AWS S3 buckets (batch and stream bucket) in an AWS account. The buckets would contain objects of generated logs for each executed job. Logs generated in stream mode are saved in an object in a stream folder in the stream S3 bucket. Also, logs generated in batch mode are saved in an object in a batch folder in the batch S3 bucket.

In our Spring Boot application, we created three service classes in the service layer to handle the logic for log generation in AWS S3 bucket, namely:

- AWSLogService- This class consists of methods to aid the generation of logs in batch and stream mode to AWS S3 bucket. It has four methods:
  - public AmazonS3 createS3Client() : This method is used to create an instance of Amazon S3 client using the account's access and secret keys.
  - public String createCurrentTimeDate() : This method is used to create the current time and date as a String to append to file path. This is used as part of the key for an object containing a generated log job.
  - public String generateJobId() : This method is used to generate random job ID in UUID format. For each executed job, a job ID is generated which can be used to stop, continue, or keep track of the job.
  - public int getLogCount(AmazonS3 s3Client, S3Object s3Object, String bucketName, String key) : This method is used to get the log counts from an S3 bucket object. After the logs are generated in the AWS S3 bucket, the log lines in the object are counted to get the number of logs generated for a particular job.
- AWSBatchService- This class handles the logic behind generation of logs in batch mode to AWS S3 bucket. This class has an aggregation relationship with the AWSLogService and LogService classes. It consists of one method,
  - public void upLoadBatchLogsToS3(SelectionModel selectionModel, BatchTracker batchJobTracker): This method generates, populates logs (using a StringBuilder) in batch mode and saves it to AWS S3 bucket. The generateLogLine method of the LogService class generates the log lines; the AWS bucket name for batch mode is initiated and the methods in the

AWSLogService classes are used to aid the process of uploading the batch files to AWS S3 bucket. The object is set to public and the URL of the S3 object is set to the batch tracker. If the job is completed, the job status is set to completed else failed.

- AWSStreamService- This class handles the logic behind generation of logs in stream mode, continuously and at a specified buffer size to AWS S3 bucket. It consists of two methods, namely:
  - public void streamToS3 (SelectionModel selectionModel, StreamTracker streamJobTracker) : This method generates logs continuously in stream mode. The generateLogLine method of the LogService class generates the log lines and it is populated using a StringBuilder. The content of the StringBuilder object is converted to a byte array. An ObjectMetadata object is created, and its content length is set to the length of the byte array. A ByteArrayInputStream object is created from the byte array which is used as the content of the PutObjectRequest. The methods in the AWSLogService classes are used to create an instance of the Amazon S3 client. The object of the PutObjectRequest is uploaded to the stream S3 bucket using the method of the created S3 client. The object is set to public and the URL of the S3 object is set to the stream tracker. If the job is completed, the job status is set to completed else failed.
  - public void streamToS3Buffer (SelectionModel selectionModel, StreamTracker streamJobTracker) : This method generates, and streams logs at a specific buffer size. To control the amount of memory used by the log generator application while streaming logs, this method generates, populates logs (using a StringBuffer) at a default buffer size and saves it to AWS S3 bucket. Specifying a buffer size can help prevent excessive memory usage while streaming logs which could cause the application to slow down or crash.

To provide access defined in the AWS service classes to the web application, we refactored the controller class from phase 1. So, we have two methods in the controller layer in this phase, namely:

- LogController: This controller class houses the endpoints to handle the requests to start a new asynchronous batch or stream job, to stop an existing job, for logs generated to AWS S3 bucket or user specified streaming address. It also houses the endpoints to get the metrics of executed jobs.
- LogToFileController: This class houses the endpoints to handle the requests to start a new asynchronous batch or stream job, to stop an existing job, for logs generated to a default file system. It relies on the service classes from phase 1 of the project.

To handle exceptions when Amazon Web Services (AWS) S3 Service is not available, a new exception class, AWSServiceNotAvailableException was added to the existing exception layer in phase 1. This class contains the method to handle exceptions when AWS S3 service e.g. when access to an Amazon S3 client is not available or when a wrong/invalid access or secret key is provided.

Deployment Phase 2

Configuration and Setup

Prior to running the pipeline which is meant to deploy updated versions of the application to AWS, some configuration tasks were required.

- Jenkins

  Jenkins is installed locally and must be configured. During installation several essential plug-ins are required to be installed. These include: CloudBees AWS credentials, Docker plugins and AWS plugins. Once the plug-ins are installed, several credentials must be set up to ensure proper authentication while running stages of the Jenkins pipeline. Two pipelines were then configured to automate the process of deploying and undeploying our web application. For the deployment pipeline, this involved specifying the project URL, which branch to build and the path to the Jenkins file stored in the main branch of our GitHub project repository. The Undeployment pipeline required the same configurations, but the Jenkins pipeline code was stored directly in Jenkins, rather than a Jenkinsfile within source control.

- AWS

  AWS was configured to prepare the static aspect of deployment. After this configuration has been completed once, it doesn't require any modifications when an update is made to the web application and the Jenkins pipeline is run. Listed below are the AWS components that were required to be configured:

    - Elastic Container Repositories (ECR)
        - Two repositories are created to store docker images pushed to ECR via the Jenkins pipeline (i.e., lg-backend and lg-frontend)
    - Elastic Container Service (ECS)
        - Create ECS cluster and specify the following: self-managed EC2 instances, the security group used and the Amazon Virtual Private Cloud (VPC) to be used. The VPC defines the virtual network being used and provides control over the IP address ranges, subnets, routing tables and network gateways
        - Create task definitions for both the frontend and backend. These task definitions specify memory allocation, port mappings, container reference and environment variables such as AWS credentials and EC2 instance URLs.
        - Create services for both the front-end and back-end. These services specify which task (instance of a task definition) to run, which revision of the task definition to use and the desired number of tasks to run
    - Elastic Compute Cloud (EC2)
        - Allocate elastic IPs for both the frontend and backend. This ensures URLs remain the same and do not change when a new version of the application is deployed.
        - Create security group to define the following: which VPC to use and any inbound traffic rules.
        - Create EC2 instances for both the frontend and backend. When creating the EC2 instances it is required to specify instance type (t2.micro which is free to use under AWS free tier account), network settings (associate the VPC, subnet and security configured in previous steps) and associate elastic IPs.

Deployment Pipeline

After configuring the various products, the deployment pipeline can be run to automatically deploy the full stack application to AWS. The figure below shows the various stages of the deployment. Each of these stages will be discussed in detail.

Pipeline LogGenPipeline

Stage View

Figure 2.1.9: Jenkins Pipeline for the Log Generator

- Stage 1 and 2: Checkout SCM and GitHub repository
    - The first two stages of the pipeline checkout our Source Code Management (SCM) system (i.e., GitHub repository) and run the Jenkinsfile script that is located within the main branch. The script specifies using the 'prod' branch for subsequent stages, so it's the branch that is cloned.

- Stage 3, 4, 5, and 6: Building frontend and backend images and pushing them to ECR
    - The next stages of the pipeline include creating docker images for both the front and backend and pushing both images to AWS Elastic Container Registry (ECR). The images are created via the use of the Dockerfiles which specify the build instructions and ensure all dependencies are included. For the React frontend this is done using 'npm' and for the Spring Boot backend 'maven' is used. The ECR stores the containerized applications and allows them to be consumed by other AWS services.

- Stage 7 and 8: Deploying frontend and backend
    - At this stage the EC2 instances are booted up and a one minute sleep is utilized to ensure they are fully running until moving on to the next deployment steps. The ECS task definitions are then updated to use the latest Docker images pushed the ECR for both the frontend and backend. The ECS services are updated to change their desired number of running tasks from zero to one. This update to the services triggers them to use the latest task definition to launch tasks within the now running EC2 instances.

Once the pipeline has completed the full stack application is now fully deployed. The frontend and backend can now be accessed by their respective URLs as specified by their associated elastic IPs configured in the earlier configuration steps.

Website

The developed front end for the microservice is a website developed using React. The home page contains all the optional user settings for log generation, with options to select the desired log fields, which malware should be included, and settings for stream mode or batch mode. Upon starting a job, the page updates to display the job statistics which are continually updated with information from the server using a socket. A button to cancel the job is available. Additionally, a chart is displayed showing the number of logs generated each second, in stream mode it allows the user to see the load being generated and streamed to the server under test. In batch mode it allows the user to see the speed at which the job is progressing.

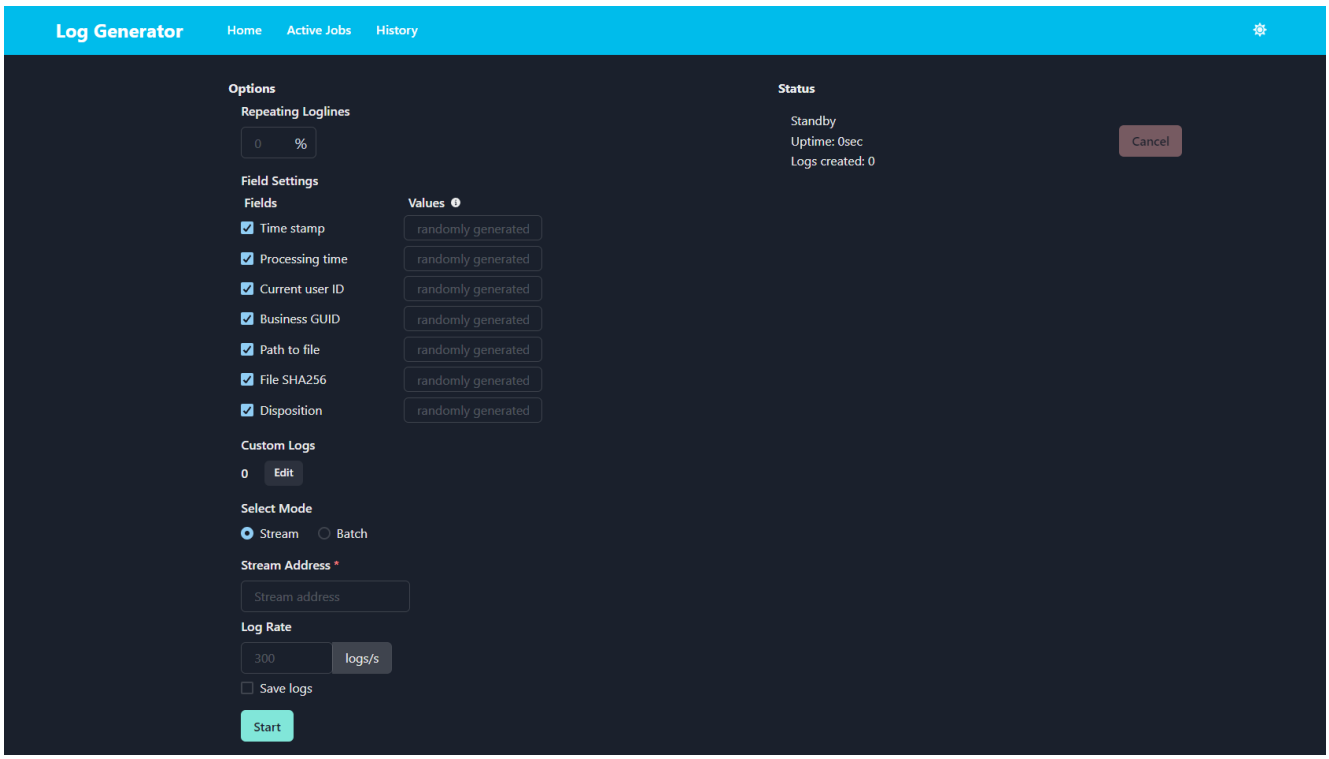


Figure 2.1.10: Home page

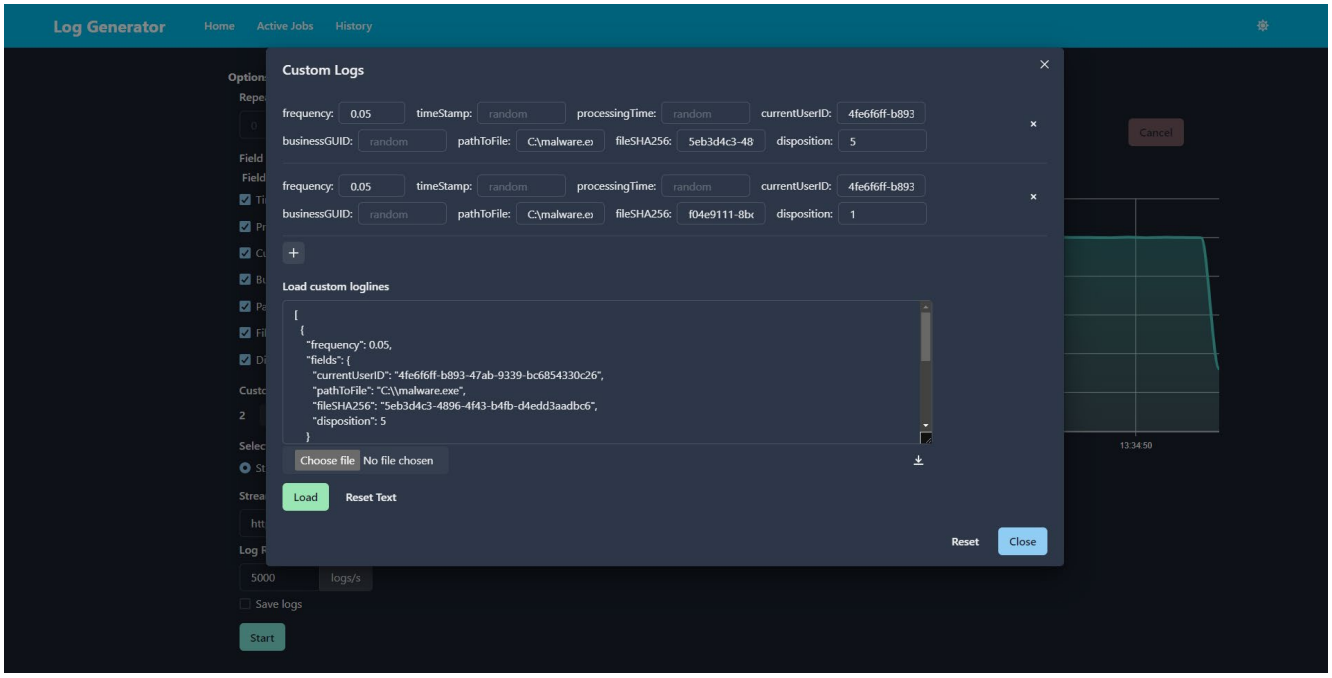


Figure 2.1.11: Custom Logs menu

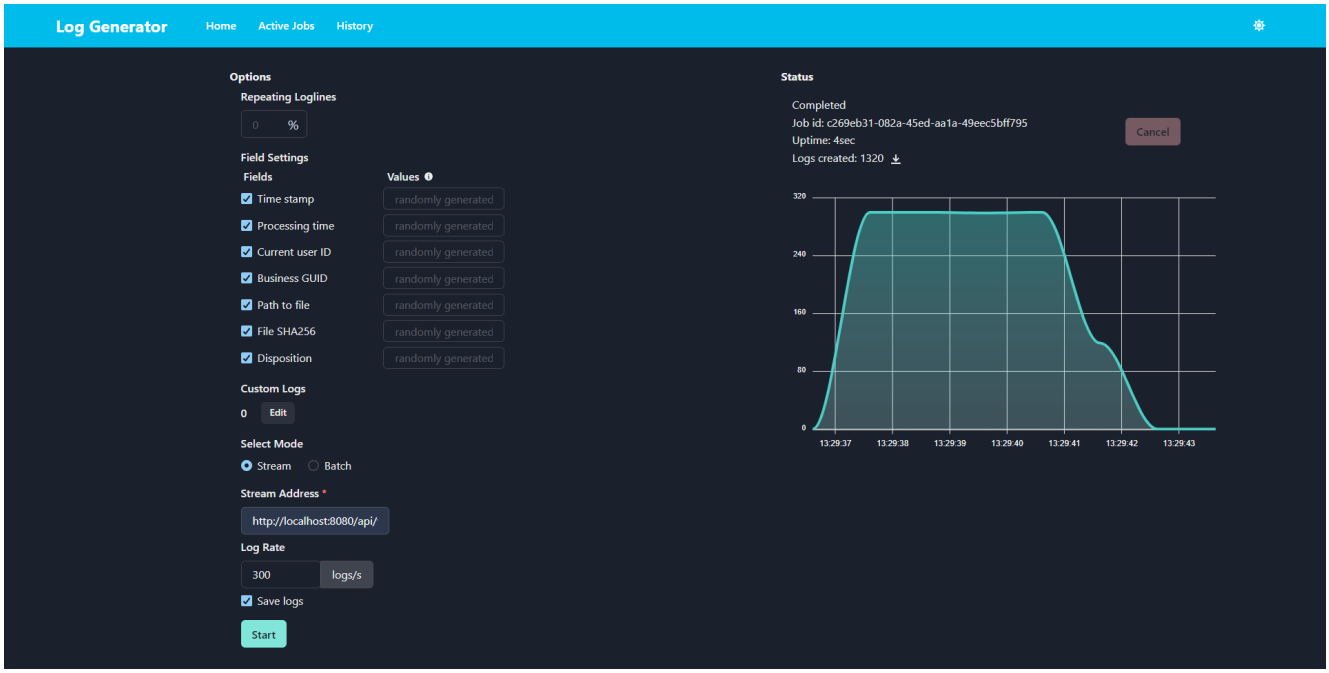


Figure 2.1.12: Stream mode example

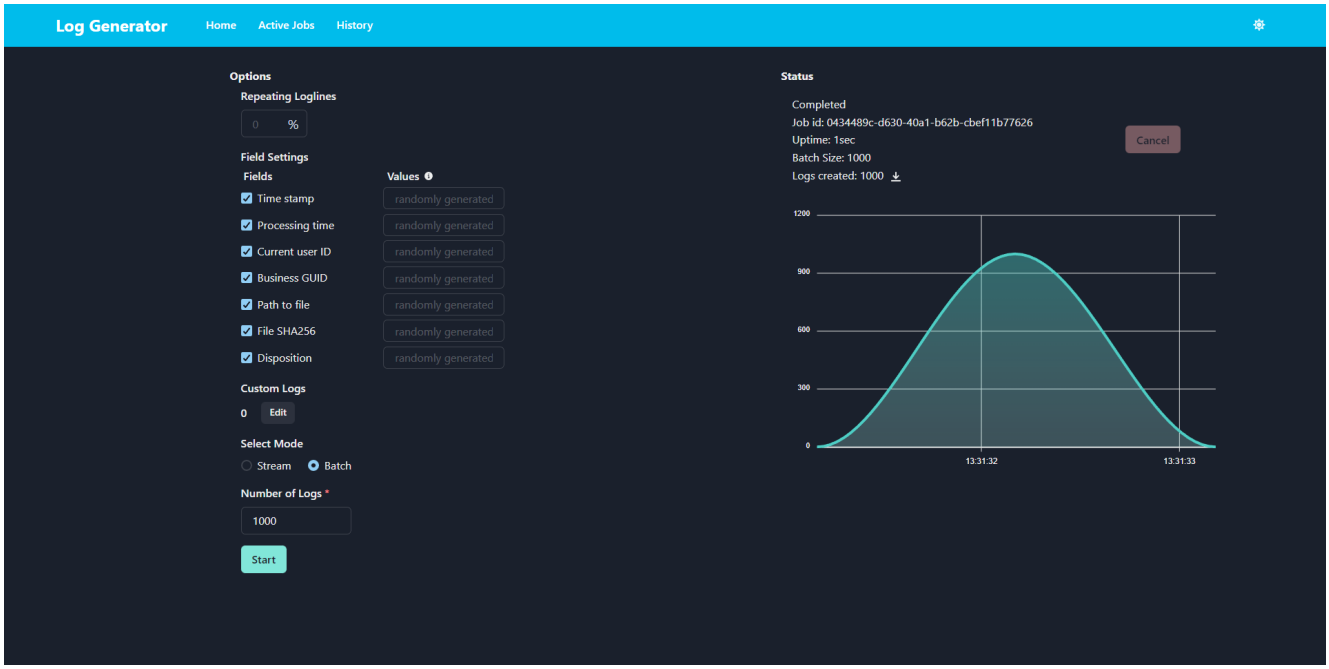


Figure 2.1.13: Batch mode example

An Active Jobs page is also available which shows continually updating information for each active job, with an option to cancel and download logs if available.

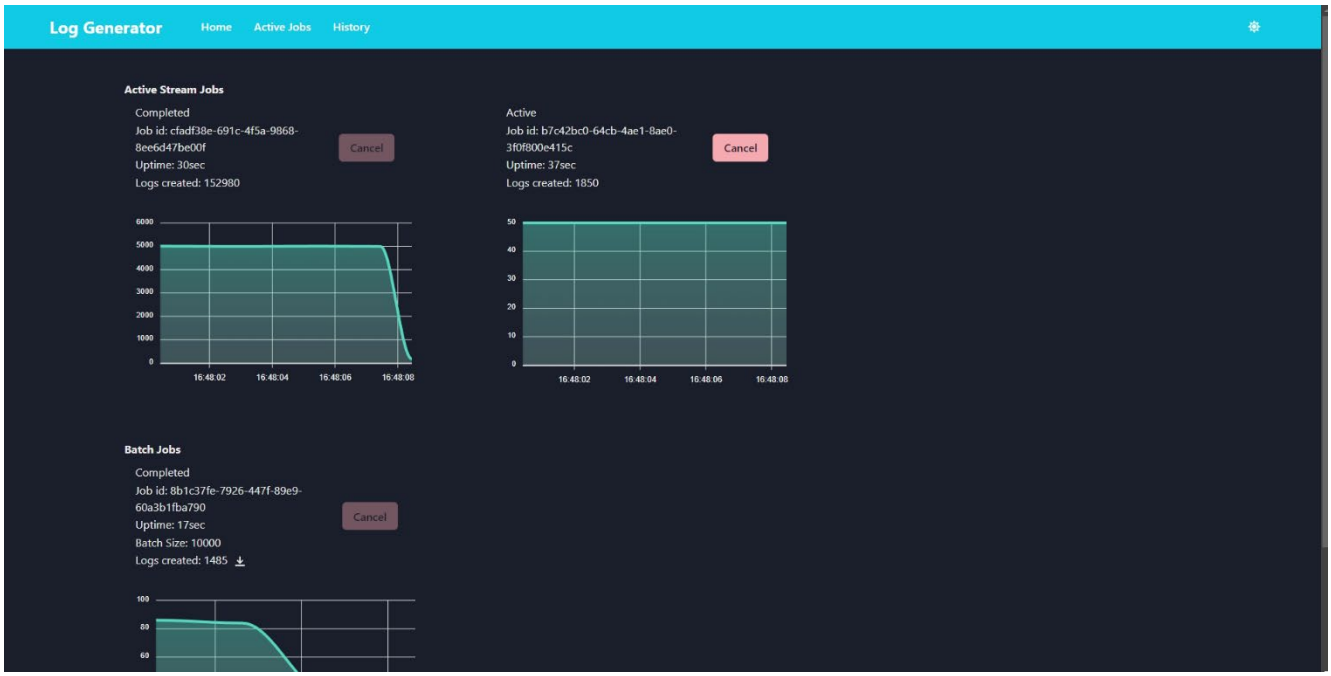


Figure 2.1.14: Active Jobs page

A history page is available which shows a table with all completed and cancelled jobs.

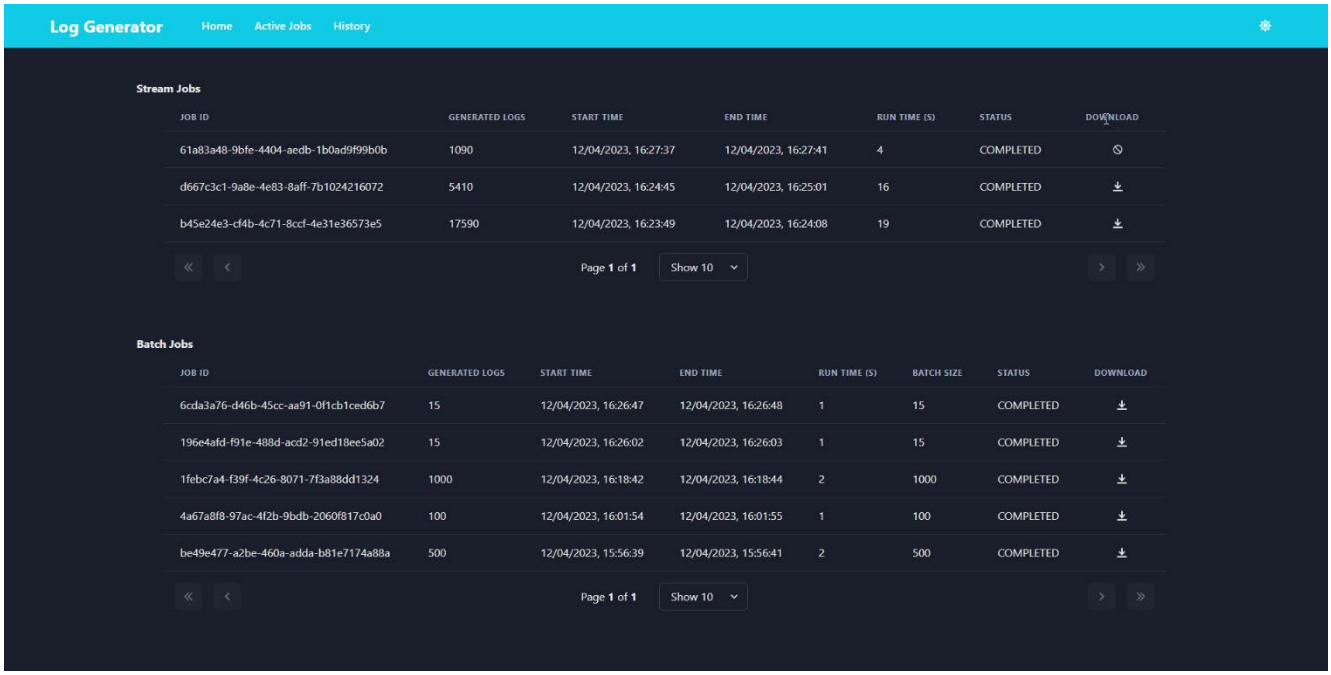


Figure 2.1.15: History page

Finally, light mode is also available to suit user preferences.

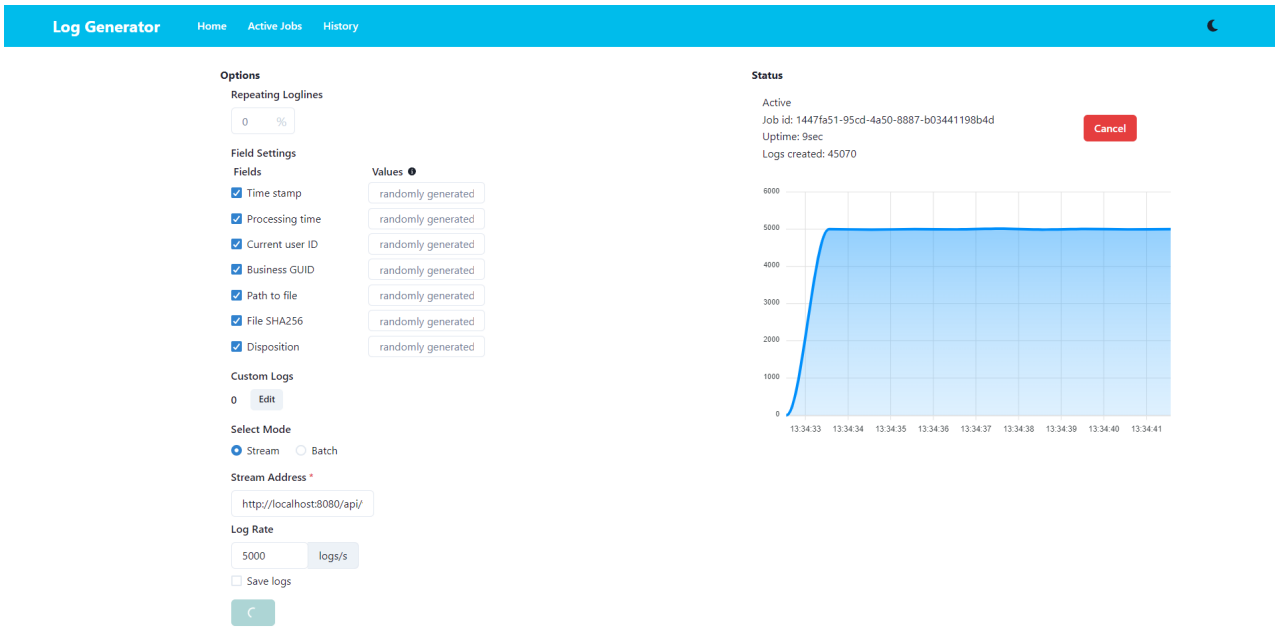


Figure 2.1.16: Light mode feature

Unit Testing

Unit tests were developed for the microservice application over the final couple weeks of the project using JUnit5 and Mockito, as per requirements. Due to time constraints, unit tests were not developed for every class, and we could not achieve 100% code coverage. However, a total of 164 unit tests were developed to test the core functionality of our application. Specifically, unit tests were developed for the LogService, BatchService, StreamingService, AWSBatchService, AWSStreamService, LogController, and LogToFileController classes.

The LogService class contains the core functionality for generating a single log, thus 56 unit tests were developed to validate the logs generated. These tests ensure that the standard fields generated adhere to the required format. To provide a few examples, the tests ensure that the LogService generates FileSha256, BusinessGuid, and CurrentUserId values that respect the UUID format, that the PathToFile values are valid file paths, and that the TimeStamp value reflects a time in the past. We test to ensure that when a user passes custom values for the 7 standard fields, the LogService always generates one of these values instead of a random value. Similarly, we test to ensure that when no custom values for the 7 standard fields are specified, a random value will be generated. We test to ensure that when a user passes in a list of custom logs that together sum to a frequency of 100%, a custom log is always generated. Similarly, we test to ensure that if no custom logs are passed, then a random log will always be generated. We assert that when custom logs are passed with values for the 7 standard fields, and if a custom log is chosen to be generated, the values for the 7 standard fields match the ones passed with the custom logs. Finally, we test to ensure that when custom logs are passed with varying fields, the log generated always contains the set of the 7 standard fields and each unique field in the custom logs.

The BatchService class contains the core functionality for orchestrating the log generation process for batch jobs to the local file system, thus 29 unit tests were developed to validate the process. As some aspects of testing the BatchService class was difficult to do with state-based testing, Mockito was used to mock certain collaborators and to perform behavioral based testing. To provide a few examples of state-based testing, the tests ensure that the log count in the BatchTracker objects reflect the batch size specified by the user with various configurations, such as specifying log lines to repeat at different frequencies. We also test to ensure that when a batch job is completed, the status attribute in the BatchTracker is ‘COMPLETED’. Similarly, we test to ensure that if the BatchTracker cannot write the logs to a file, an exception is thrown and the status attribute in the BatchTracker is ‘FAILED’. We also test to ensure that the file generated contains data that is valid JSON by matching it with a regular expression. To provide a few examples of behavior-based testing, we mocked certain collaborators such as the SelectionModel and LogService classes to validate the method calls with different configurations, such as ensuring the correct number of LogService.generateLogLine() methods are called with different batch sizes specified.

The StreamingService class contains the core functionality for orchestrating the log generation process for streaming jobs to the local file system and to a user specified address, with the option to save the logs locally as well, thus 55 unit tests were developed to validate the process. Mockito was used to mock certain collaborators when state-based testing was not possible. For example, the StreamTracker object was mocked to control how many logs are generated by changing the status to ‘COMPLETED’ after a specified number of StreamTracker.getStatus() calls were made. Furthermore, to test making POST requests to a web server, a MockWebServer class was set up in the test class that allows us to make post requests with programmed responses. To provide a few examples of state-based testing, the tests ensure that the StreamTracker object’s

log count attribute reflects the expected number of logs generated. We also ensure that an exception is thrown if there was an error writing to file and a call was made to set the StreamTracker object's status attribute to 'FAILED'. We also test to ensure that the file generated contains data that is valid JSON by matching it with a regular expression. To provide a few examples of behavior-based testing, we mocked certain collaborators such as the SelectionModel and LogService classes to validate the method calls with different configurations, such as ensuring the correct number of LogService.generateLogLine() methods are called when streaming to a file or to an address. Finally, we ensure that the StreamingService class correctly identifies when an address is available to stream to or not.

The LogsToFileController class contains 2 endpoints to start a batch job that generates logs to the local file system or a streaming job that generates logs to the local file system or to an address, thus 5 unit tests were developed to test the response. To simulate making requests to these endpoints, Mockito was used to mock the collaborators such as the BatchService, StreamingService, BatchTrackerService and StreamTrackerService classes, as we only wish to test the controllers and not the services. The MockMvc class was used for server-side Spring MVC test support and allows us to make requests to our endpoints and validate the response. For example, we test to ensure that if a valid SelectionModel is passed, then an OK status is returned, and the response body contains a job id. We test to ensure that if an invalid SelectionModel is passed, then a BadRequest status is returned, and the response body contains an error message. Finally, we test to ensure that if an invalid address is provided for streaming, then a 404 status is returned, and the response body contains an error message explaining the problem.

The LogController class contains 9 endpoints to start a batch job or stream job that generates logs to an s3 bucket, or to start a stream job to a specified address with the option to save the logs to s3, to request to stop a specific job, or to obtain metrics for a job. Thus, 18 unit tests were developed to test the response of these end points. Similarly, to the unit tests developed for LogsToFileController, Mockito was used to mock the collaborators to only test the controllers and MockMvc was used to allow us to make requests to our endpoints and validate the response. For example, we test to ensure that an OK status is returned, and the response body contains a job id if a valid SelectionModel is passed, or a BadRequest status is returned with an explanation of the error if an invalid SelectionModel is passed. We test to ensure that an OK response is returned when a job is stopped successfully, and a 404 status is returned if the job was not found. Finally, we test to ensure that when metrics are requested, the response body contains the expected JSON properties and values.

Research on making the generated logs more realistic using deep learning model.

For this phase, we would explore using Generative Adversarial Networks (GANs) to generate more realistic logs that mimic the patterns and characteristics of real logs.

Using Generative Adversarial Networks (GANs)

To generate real logging data that have the same distribution as the input log, we can use Tabular Generative Adversarial Network (TGAN); while to generate logging data based on certain features like specific file path, we can use Conditional Generative Adversarial Network (CTGAN).

We would use the logs generated from our log generator application and real logs wrangled from open sources as input to the chosen Generative Adversarial Network (GAN) model. The GAN can learn the patterns and

characteristics of the generated logs and generate more realistic logs that follow similar patterns and characteristics.

The general steps we would explore to train the Generative Adversarial Network (GAN) model are as follows:

1. **Prepare the data:** Generate a large dataset of logging/telemetry data using the log generator application, in JSON format. Load the data from the object URL and write the log data to a CSV file. Combine cleaned, wrangled real logging/telemetry data with the generated log data. Load the combined log dataset from the output CSV file into a pandas dataframe.

   Example Code Snippet

   ```
   # Make a GET request to the object URL and load the response
   response = requests.get('https://batch-s3-log-generator.s3.us-east-2.amazonaws.com/batch/20230324_111749.json')
   data = json.loads(response.content)

   # Open a CSV output file and write the headers
   with open('log_generator.csv','w',newline='') as csv_file:
   writer = csv.writer(csv_file)
   writer.writerow(data[0].keys())

   # Loop through each logline in the logging/telemetry data and write it to the CSV file
   for logline in data:
       writer.writerow(logline.values())

   # Combine the generated data with the cleaned,wrangled data
   # Load the data
   modelData = pd.read_csv('log_generator.csv')
   ```

2. **Preprocess the logging data:** Convert the logging data into a suitable format for the GAN to consume. Normalize the numerical variables using mode-specific normalization [24] and encode the categorical variables using one hot encoding. To effectively sample values from a multimodal distribution, values of a numerical variable can be clustered using a Gaussian Mixture Model (GMM) for TGAN [24], while for CTGAN, the Variational Gaussian Mixture Model (VGM) can be used for each continuous column independently [31]. The goal of normalization is to transform features to be on a similar scale. This improves the performance and training stability of the model [26]. Also, most deep learning models require categorical variables to be converted into numeric data to evaluate it. Shuffle and split the logging data into a training set and a test set.

   Convert the preprocessed training logging data to tensors suitable as input for the GAN model. Create a dataloader for the data to load it in batches during training, to ensure efficient use of memory and parallel processing during training.

3. **Design the TGAN architecture:** The GAN architecture would consist of two networks, namely: a generator network and a discriminator network. In GAN, the discriminator D tries to distinguish whether the logging

data is from the real distribution, while the generator G generates synthetic logging data and tries to fool the discriminator [24]. Both networks can be implemented as simple feedforward neural networks with fully connected layers.

Generator:

The generator would take in a random noise vector as input and generate synthetic logging data as output. The architecture would consist of a chosen number, n of fully connected linear layers with activation functions between them. Choosing at least one hidden layer would enable the model to learn more complex, non-linear relationships between input and output. The first fully connected layer would transform the input noise vector to a hidden representation of size H_d, the second fully connected layer would transform the hidden representation to another hidden representation of size H_d while the nth fully connected layer would transform the hidden representation to the output, same size as the logging data columns. ReLU or Leaky ReLU activation functions, to introduce non-linearity can be chosen for the hidden layers and a Tanh activation function for the output layer [24], to help the generator output values in the desired range, which is useful for generating more realistic logs. The output layer would have the same dimension as the input layer.

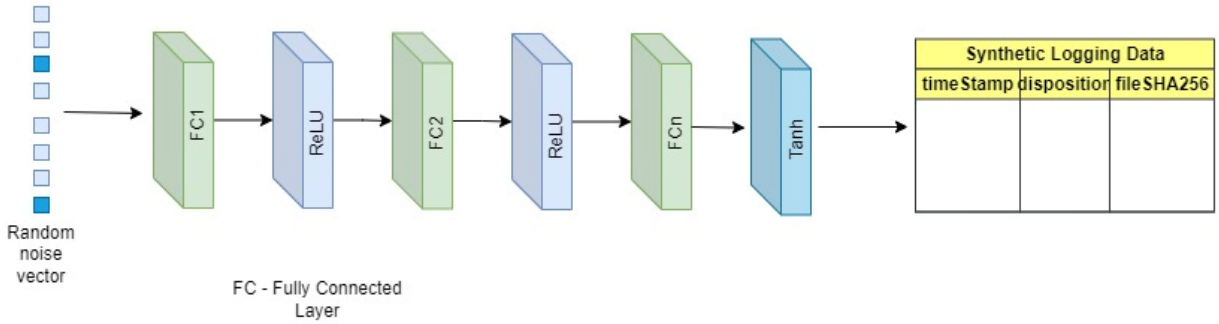


Figure 2.1.17: Architecture of the Generator

Discriminator:

The discriminator D would take either the real logging data or synthetic logging data generated by the generator G, as input and predict whether the input logs are real or fake. The architecture would also consist of a chosen number, n of fully connected linear layers with activation functions between them. The first fully connected layer would transform the input data (real logging data or synthetic logging data) to a hidden representation of size H_d, the second fully connected layer would transform the hidden representation to another hidden representation of size H_d while the nth fully connected layer would transform the hidden representation to the output of size 1.

ReLU activation functions, to introduce non-linearity, can be chosen for the hidden layers; however, there is the issue of “dying ReLU problem”. This refers to the scenario when many ReLU neurons only output values of 0 [22], leading to gradients of zero during backpropagation. Consequently, the weights of these neurons stop updating, and they become "dead" or inactive. The “dying ReLU problem” does not happen all the time since the optimizer considers multiple input values each time; however, we can choose to use Leaky ReLU activation function to address the dying ReLU problem and improve training stability [22].

The output layer would have a single output with a sigmoid activation function to predict the probability of the input logging data being real.

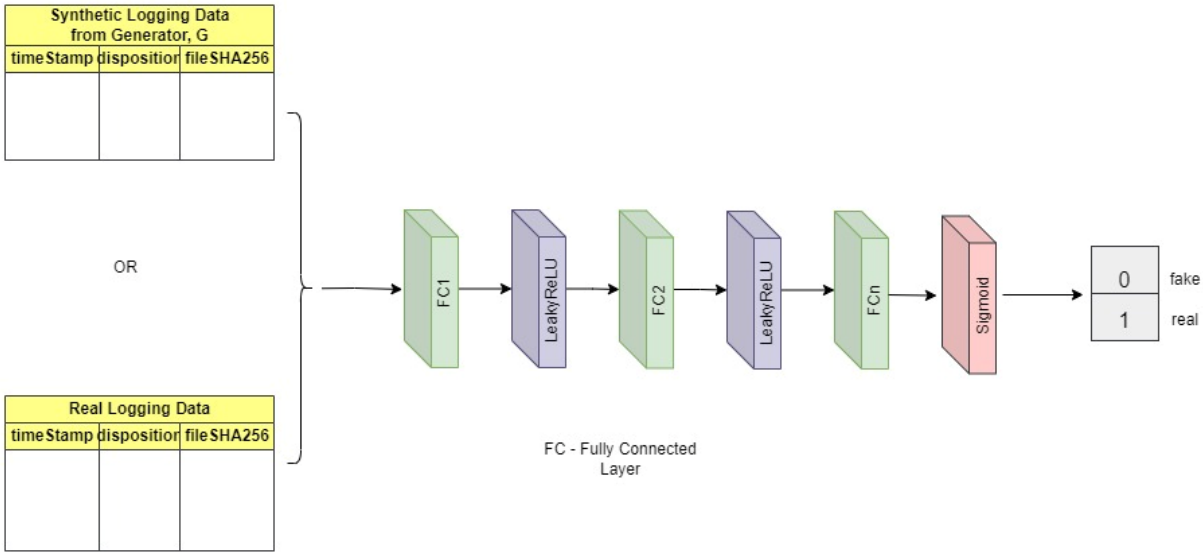


Figure 2.1.18: Architecture of the Discriminator

4. **Define loss function and optimizer:** We would optimize the discriminator using Binary Cross Entropy loss (BCE), which is normally used for binary classification [10]. The loss function is used to measure the difference between the predicted probabilities and the actual labels (0 for fake, 1 for real). We would optimize our models using Adam optimizer. The optimizer is responsible for updating the weights of the neural networks during training. We optimize the generator so that it can fool the discriminator as much as possible [24].

5. **Train the GAN model:** Training the TGAN involves alternating between training the discriminator, D and the generator, G.
   Train the discriminator: Train the discriminator, D to learn how to tell the difference between the original and synthetic logging data, outputting values near 1 for the true logging data and values near 0 for the fake logging data [10]. Calculate the Binary Cross Entropy loss for the real logging data from the dataset and the synthetic logging data from the generator, G differently. The goal of the discriminator is to maximize the probability of accurately classifying real logging data and synthetic logging data. The weight of the discriminator should be updated at each point, to correctly distinguish between the two logging data.

   Train the generator: Train the generator to generate more realistic log data and update its weights to minimize the error between the synthetic logging data and the real logging data.
   As the training progresses, the generator, G becomes better at generating fake/synthetic logging data that closely resemble the real logging data, while the discriminator, D tries to improve its ability to differentiate between them. This adversarial process helps the generator, G, to learn the underlying patterns in the real logging data, even though it does not take the log data as input directly. Crucially, the weights of the discriminator can be frozen while we are training the combined model, so that only the generator's weights are updated [36]. If we do not freeze the discriminator's weights, the discriminator will adjust so that it is

more likely to predict generated synthetic log data as real, which is not the desired outcome. We want synthetic logging data generated from the generator to be predicted close to 1 (real) because the generator is strong, not because the discriminator is weak [10].

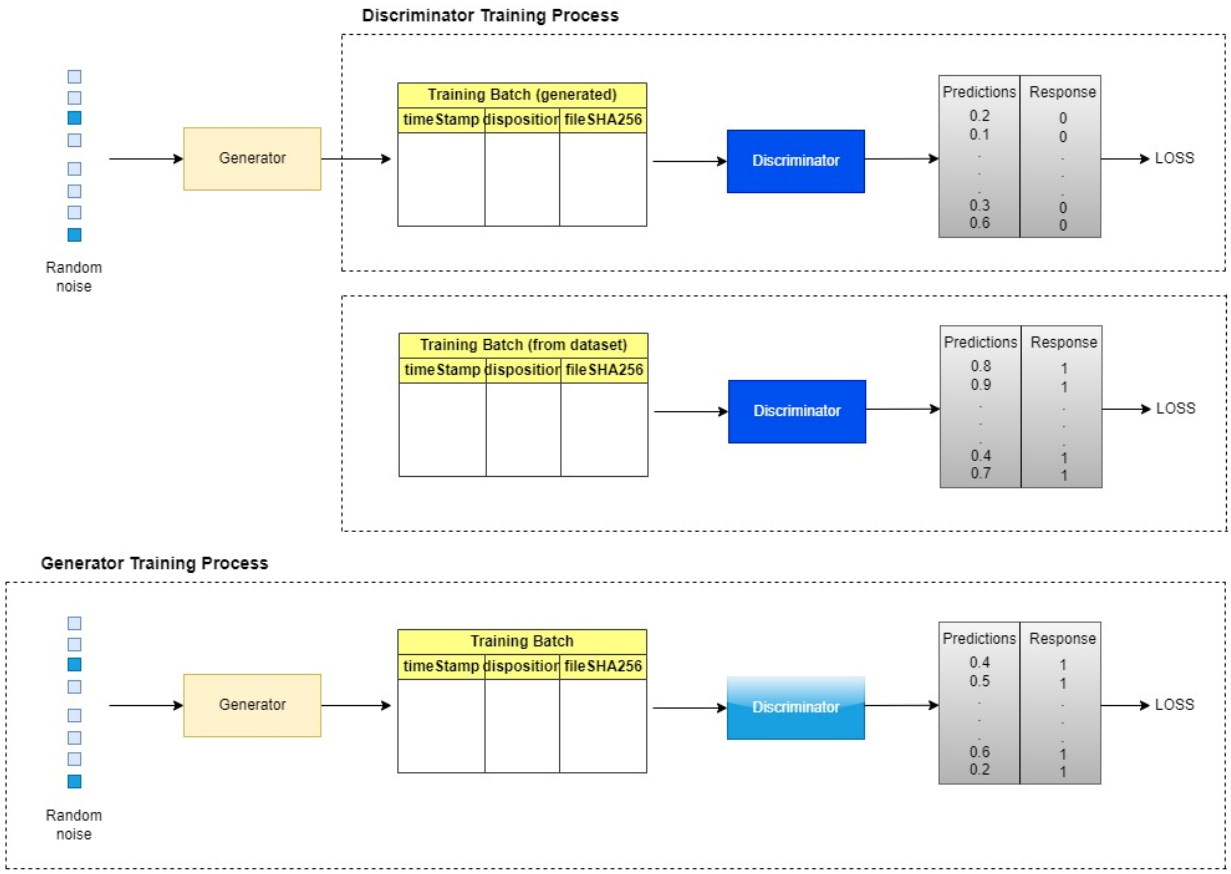


Figure 2.1.19: Training a GAN model [10]

6. **Fine-tune the GAN:** Fine-tune the GAN by adjusting hyperparameters, tweaking the architecture, and continuing to train until a satisfactory result is achieved.

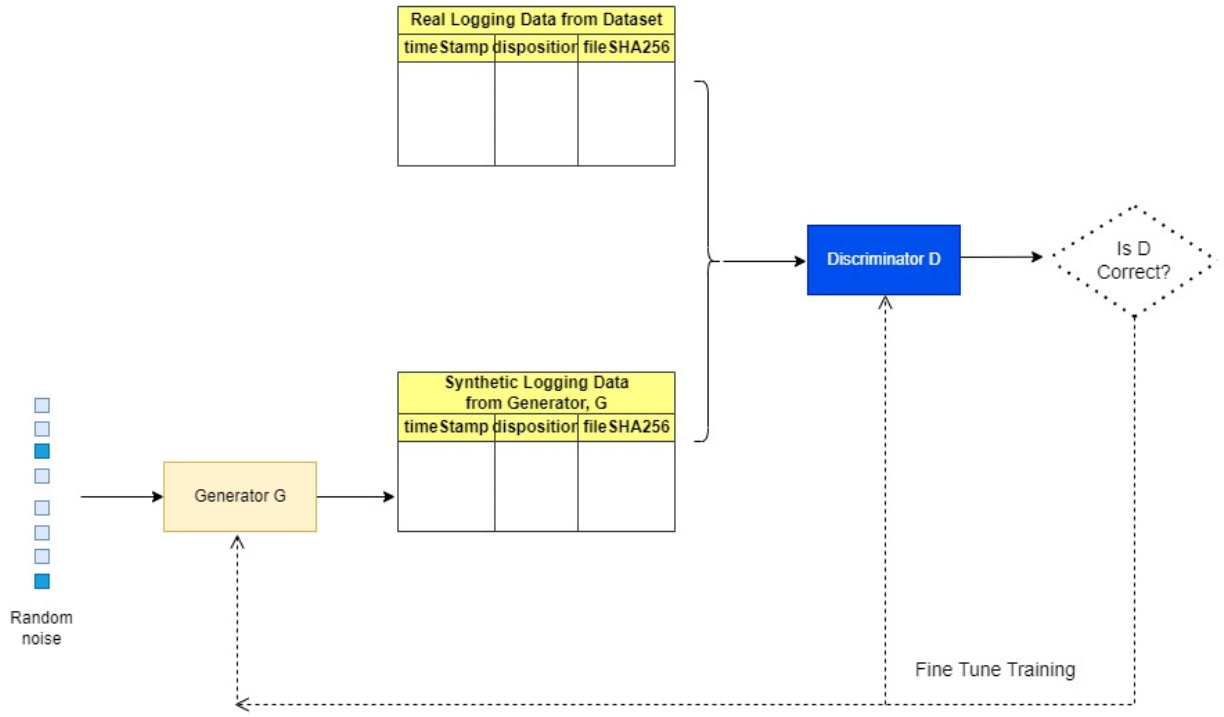


Figure 2.1.20: TGAN Training Pipeline [20] [33]

7. **Evaluate the results:** Evaluate the performance of the GAN by examining the generated logging data and comparing them to the real logging data from the dataset. The generated logging data can be evaluated based on its machine learning efficacy.
   Evaluate Generated Logs based on Machine Leaning Efficacy: To evaluate the machine learning efficacy of the generated synthetic logging data, the data can be evaluated using machine learning algorithms like Decision Tree Classifier, Linear Support Vector Machine (SVM), Random Forest Classifier, Multinomial Logistic Regression and Multi-Layer Perceptron (MLP) [33]. Train the above models using the synthetic logging data and real training logging data separately and evaluate performance on the real test logging dataset, for instance in predicting the disposition of the file path. The machine learning performance can be measured using the accuracy, F1-score, and area under the ROC [33]. The aim of this design is to test how close the machine learning utility is when we train a machine learning model using the synthetic logging data compared to the real logging data [33].

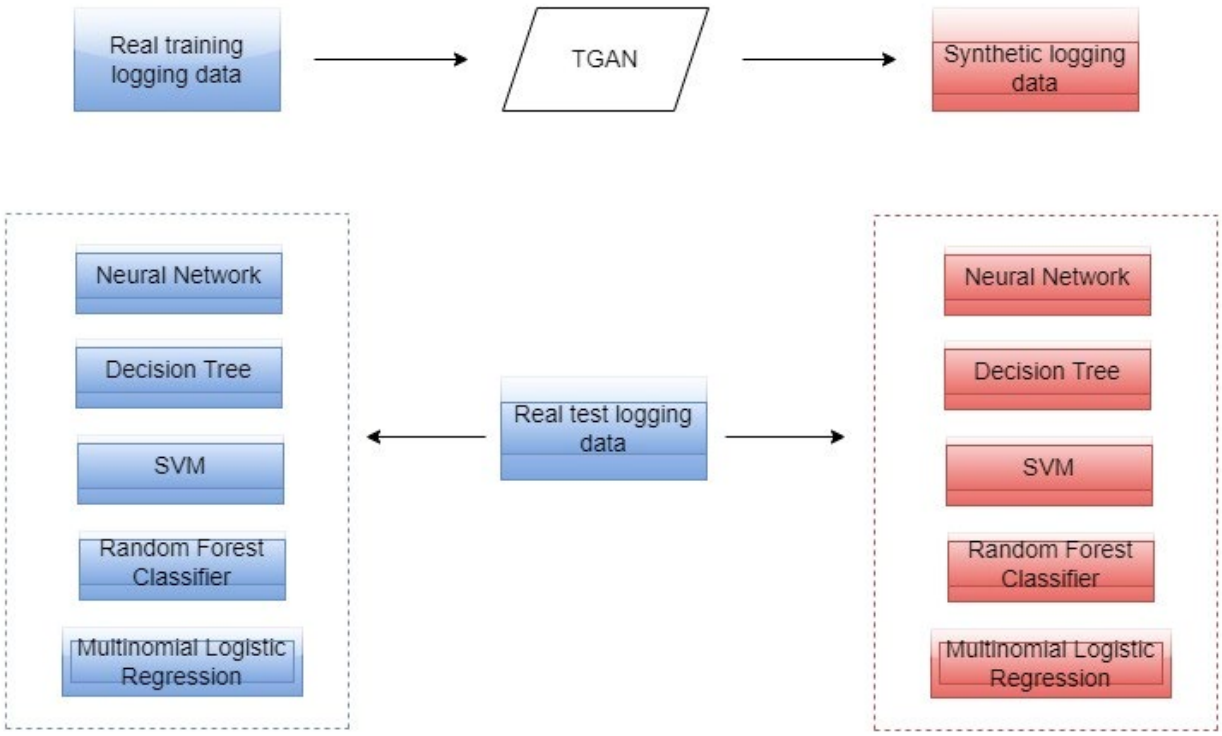


Figure 2.1.21: Evaluating TGAN [24]

8. **Use the trained GAN:** Once the GAN is trained and its performance is satisfactory, the trained GAN can be used to generate more realistic log files. To achieve this, write a function that takes the number of samples to generate as input and use the trained generator to generate the realistic logging data. Convert the generated logging data in csv format to json format.

Example Code Snippet

```
csv_file = 'generated_log.csv'
json_file = 'generated_log.json'

# Read the CSV file then store the header and log data
with open(csv_file,'r') as csvfile:
  csv_reader = csv.DictReader(csvfile)
  header = csv_reader.fieldnames
  data = [row for row in csv_reader]

# Write the JSON file using the logging data from the CSV file
with open(json_file,'w') as jsonfile:
  json.dump(data, jsonfile, indent = 4)
```

3.0 Product Scope and Functionality

Product Functionality

The log generator microservice application consists of the following features:

1. Batch mode to generate a specified number of logs.
2. Stream mode to stream to an end point at a specified log rate (log/s).
3. Customizable fields.
4. Custom logs.
5. Uploading logs to s3 bucket.
6. Front end website created with React.
7. Active jobs page to track all running jobs.
8. History jobs page to see all past jobs.

Configuring the log lines generated

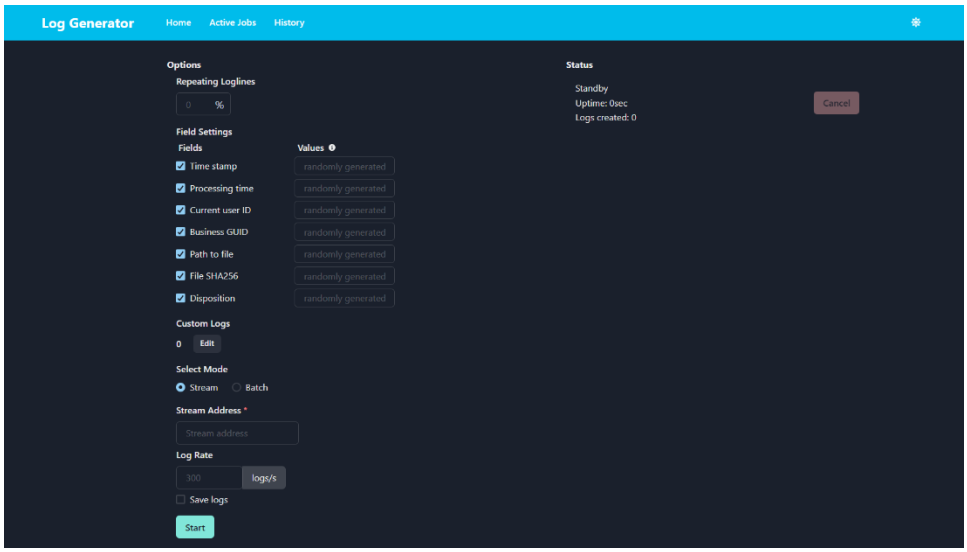


Figure 3.1: Home page

Repeating Loglines

The user can configure the log lines generated on the left-hand side of the home page. From top to bottom, the first option is to specify at what frequency a log line should be repeated. The value provided should be a value between 0% and 100% and represents the probability that a log will be duplicated in the logs that are generated.

Field Settings

The 'Field Settings' section is used to first specify which of the 7 standard fields should be optionally included in the generated logs by checking the checkbox field adjacent to the field name. If a field is specified to be excluded, then the logs generated will never have that field, even if custom values were specified for it. Each field will be randomly generated unless custom values are specified to be used in the text box adjacent to the name of the field. If one custom value is specified, then all logs generated will use that value instead of randomly generating

a value. If two or more custom values are specified, then these values will be evenly distributed among the logs that are generated. To specify multiple values, the values should be separated with a single comma.

Custom Logs

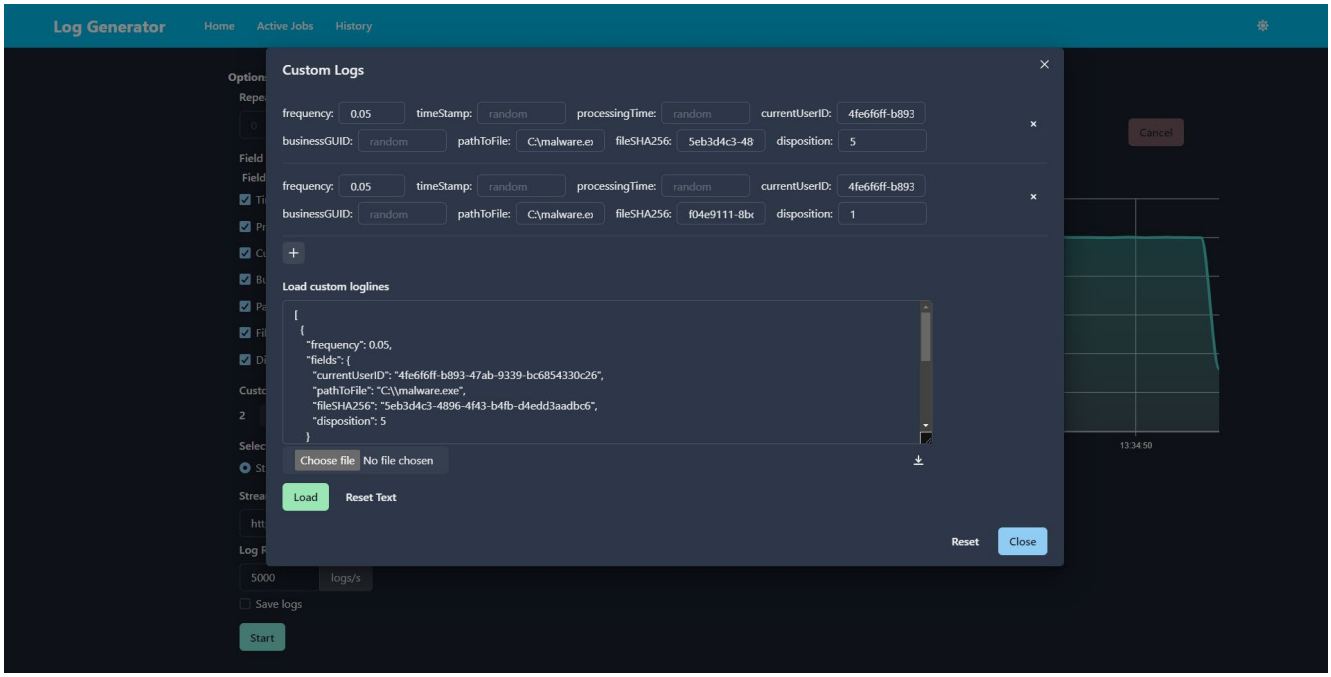


Figure 3.2: Custom Logs

The user can specify their own custom log lines to be generated at a specified frequency by clicking on 'Edit' and opening the 'Custom Logs' view. In this view, the user can add their own custom logs directly through the user interface or load custom log settings from text or a JSON file. To specify multiple custom logs, the user can add more custom logs by clicking the plus icon in the user interface or load custom loglines with additional JSON objects with the 'frequency' and 'fields' properties.

Each JSON object must contain 'frequency' and `fields` values to denote the desired generation frequency of the log and the field values it will contain. If a field isn't included in `fields`, or the value isn't specified, it will be generated randomly or with the given field values in Field Settings. Any number of custom logs can be specified, however, the total frequency of all the custom logs passed must not exceed 100%.

By using the text area or loading custom log settings from a file, the user can specify custom logs with their own additional fields. After configuring the custom logs to be generated, the user can download the custom logline settings for later use by clicking the download button. The configuration can then be later loaded to duplicate the configuration so that they do not have to configure it manually again.

When custom logs are passed, every log that is generated will have the same set of fields, consisting of the 7 standard fields that were not specified to be excluded and each unique field in all custom logs. If the total frequency of the custom logs specified is less than 100%, there is a chance the log generator will generate a log with random values. The log generator will generate logs based on the following conditions:

1. No custom logs passed, and no custom values passed for the 7 standard fields
    a. The logs generated will have random values for the 7 standard fields that are specified to be included.
2. Custom logs are specified, and a random log is chosen to be generated
    a. The logs generated will have random values for the 7 standard fields that are specified to be included, unless custom values for these fields were specified instead.
    b. The unique fields in all custom logs will be added with null values.
3. Custom logs are specified, and a custom log is chosen to be generated
    a. The logs generated will have the chosen custom log's fields and values.
    b. The 7 standard fields that are specified to be included will have randomly generated values, unless custom values for these fields are specified instead.
    c. If the chosen custom log already specifies values for the 7 standard fields, these values will take priority over the above.
    d. The unique fields in all other custom logs will be added with null values.

Selecting a Mode

The user can select to either generate logs in batch mode or stream mode by using the two radio buttons under 'Select Mode'.

Stream Mode

If stream mode is selected, the user will be prompted to input an address to stream to, a log generation rate in logs/s, and they can also specify whether they would like to additionally save the logs generated to s3. Note that by specifying –1 or 0 for log rate, the maximum rate available will be used.

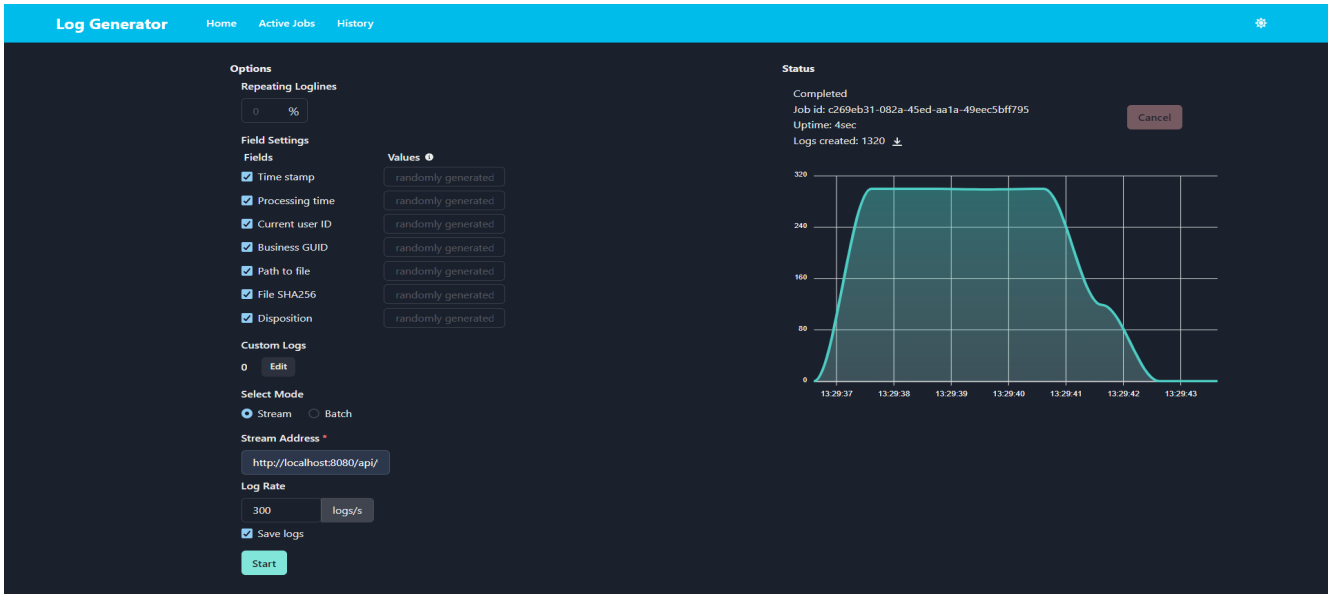


Figure 3.3: Home Page Stream Mode

Configuring Stream Address Endpoint

In stream mode, the log lines will be streamed to the address provided through HTTP POST requests. The endpoint will receive an array of up to 10 logs in JSON format with each request. The frequency of requests will be approximately the log rate specified divided by 10. All logs sent by a given job will have the same fields and contain information such as a timestamp, file path, and processing time. The endpoint should return a success or error response based on the outcome of the request. Streaming will be stopped upon receiving an error response or being cancelled. If the user specifies to additionally save the logs to s3, the same logs that were streamed to the address will be saved to a file locally and uploaded to s3 after the user cancels the stream job.

```
POST /api/endpoint HTTP/1.1
Content-Type: application/json


[
  {
    "timeStamp":1082495877,
    "disposition":2,
    "currentUserID":"083550e9-1289-46af-be89-268ae1fd7ca9",
    "fileSHA256":"b6cdd8fa-848d-4a19-8da5-4d27425a5fa2",
    "pathToFile":"C:\\Program Files\\f9e3489b-06f9-49ff-b7a9-0f1e1ed100f1.json",
    "businessGUID":"f147180d-8961-459f-af10-121c593a0fa2",
    "processingTime":153
  },
  {
    "timeStamp":332981314,
    "disposition":4,
    "currentUserID":"5071c9ee-ec9f-429a-ba2a-e8c28707c3cf",
    "fileSHA256":"f0d789f7-7439-4d21-a811-fff027b90076",
    "pathToFile":"C:\\home\\ba8f8032-5ecb-408e-9218-e89f14ac4084.xlsx",
    "businessGUID":"142d09de-ce38-4b3c-8119-b04f9db82199",
    "processingTime":761
  },
  ...
]
```

Figure 3.4: Example POST Request in Stream Mode

Batch Mode

In batch mode, the log generator will generate the specified number of log lines and upload the file to s3. Once completed, a download icon will appear enabling the user to download and view the logs.

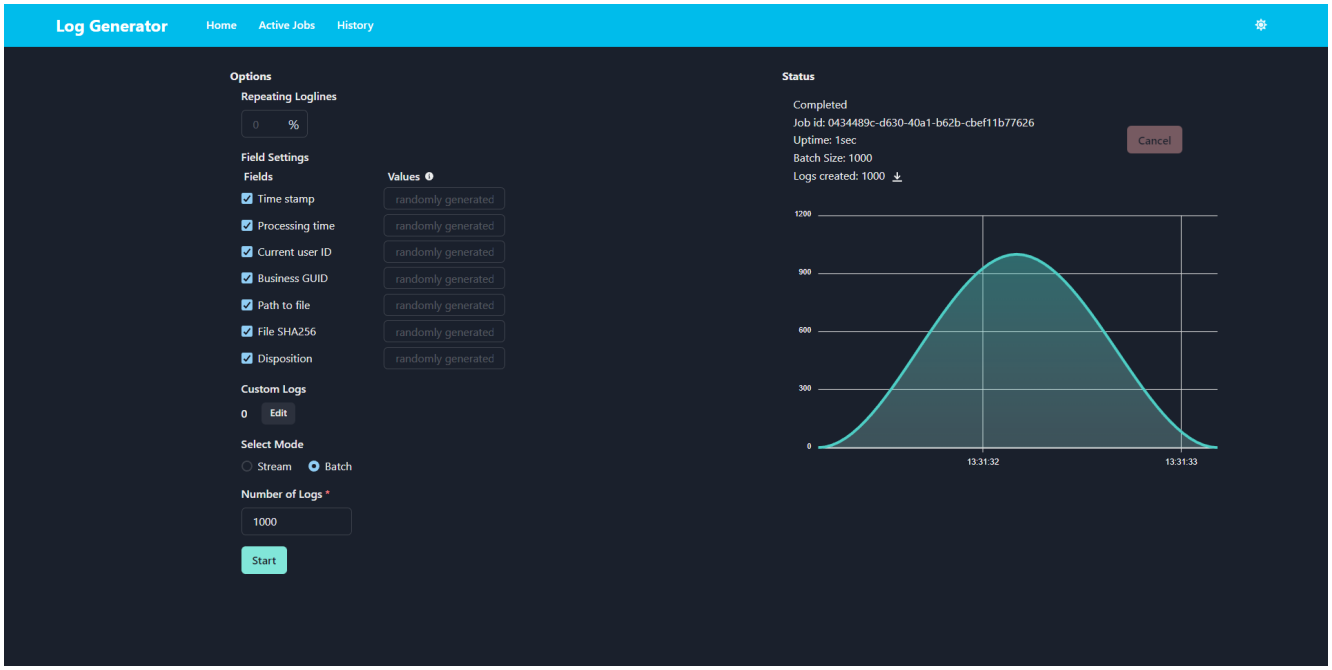


Figure 3.5: Home Page Batch Mode

Starting Job

Once the user is satisfied with their configuration for generating log lines, they can press the 'Start' button at the bottom of the home page. The job can then be tracked on the right-hand side of the page. The user can track the job progress and cancel the job at any time. Once the job is complete, if the logs were saved to S3, a download icon can be clicked to download and view the logs.

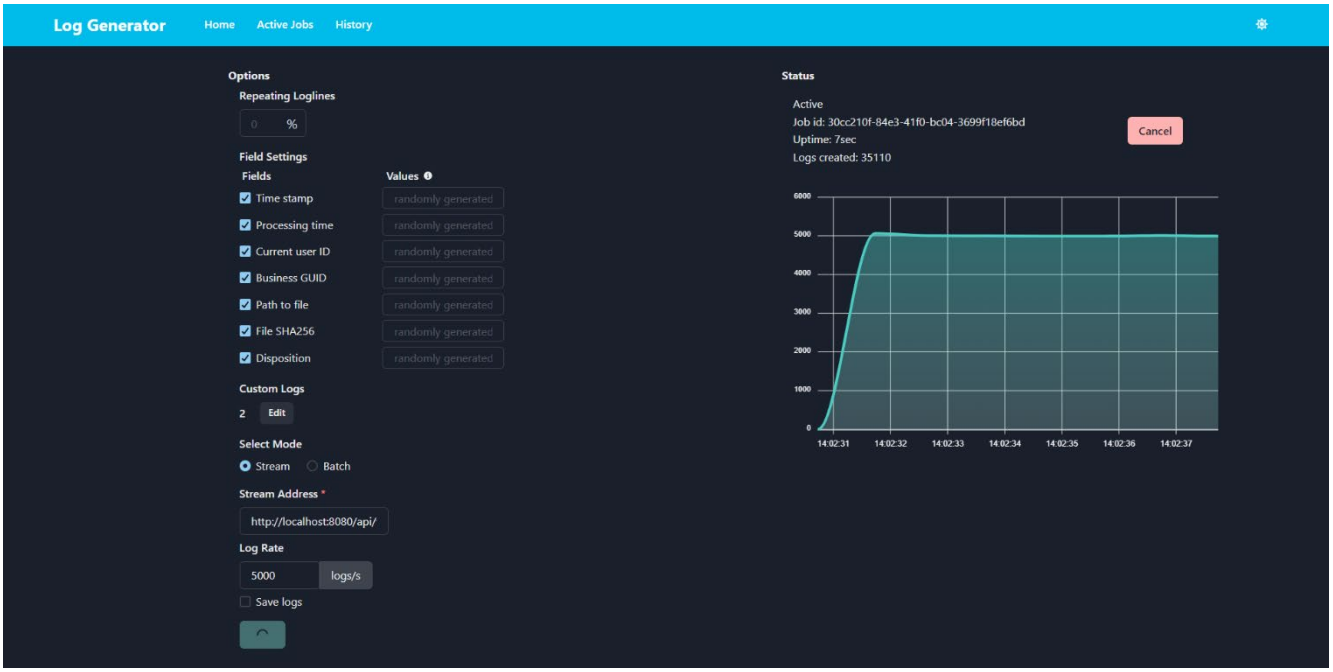


Figure 3.6: Starting a Job

Active Jobs Page

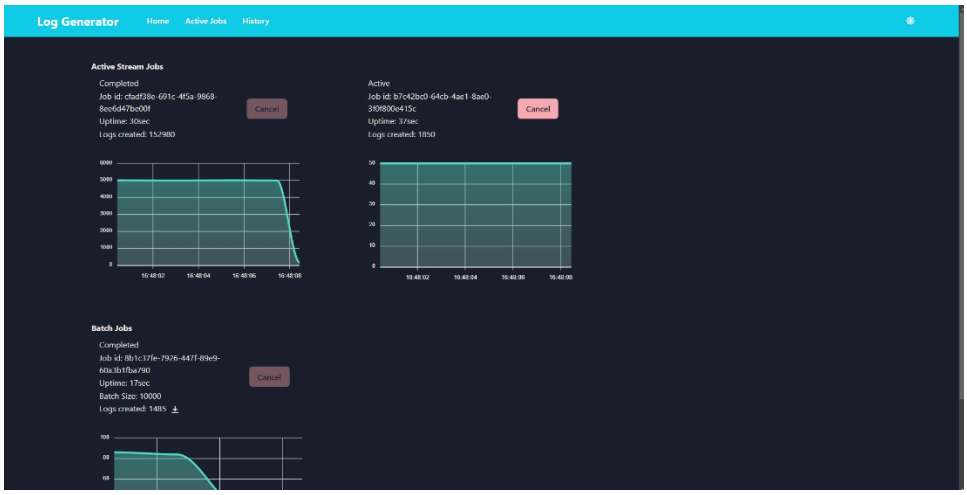


Figure 3.7: Active Jobs Page

Users can navigate to the 'Active Jobs' page to view metrics for all jobs that are currently generating logs, with stream jobs shown above the batch jobs. Each active job can be identified by the job's unique id, and metrics for each job include the status of the job, the job's uptime, and total logs currently generated. Additionally, a graph with the job's log generation rate is displayed with real-time updates.

To request to cancel a batch job early, or to end a stream job, the user can click on the 'Cancel' button next to the job's metrics. Cancelling a batch job will stop the log generation process for that specific job before the specified batch size is reached and will begin the upload process to s3. Cancelling a stream job will stop the application from streaming logs to the specified address, and if logs were specified to be saved, then the upload process to s3 will begin. If logs were uploaded to S3, a download icon will appear and allow users to download and view the logs.

History Page

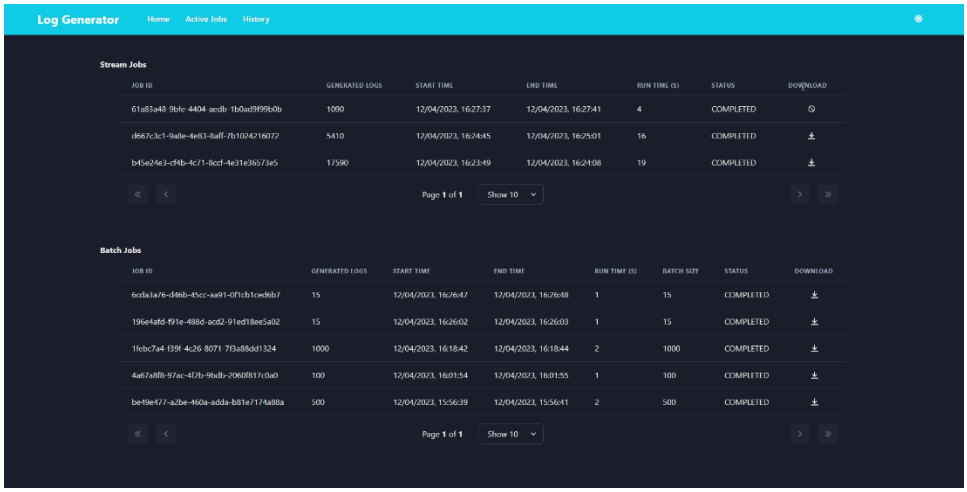


Figure 3.8: History Page

Once the user has started at least one batch or stream job, they can navigate to the 'History Page' to view metrics for the jobs that have been requested in the past, with stream jobs shown above the batch jobs. Each row represents a single job, which can be identified by the job's unique identifier. Here, the user can view the total number of logs that were generated, the start time of the job, the end time of the job, the run time of the job, the status of the job, as well as a link to download the log file from s3 if available. Additionally for batch jobs, the specified batch size will be displayed.

Limitations of product

- Implementing a machine learning model to generate data that is more realistic.
    - The main limitation of the product compared to the original product scope is that we were unable to implement a machine learning model to generate log lines that reflect real world data based on a certain type of compromise or threat actor due to time constraints. This part of the scope of the product was altered to instead perform research and outline the steps to how we could use machine learning to generate more realistic logs as part of our report. To get around the intelligent machine learning aspect of the application, we implemented additional options for the user to customize the logs generated by the microservice application, which is explained in detail in sections 2.0 and 2.1.

Limitations of deployment

- Webhook and automatic trigger to initiate pipeline
    - During the research aspect of phase 2 as discussed in our midterm report it was decided that we would like our deployment pipeline to be triggered via webhooks each time a new update has been pushed to the 'prod' branch of our project GitHub repository. While implementing this approach it was discovered that the EC2 instance type 't2.micro' that comes with the free tier subscription of AWS couldn't handle running the Jenkins pipeline and would crash since it was very limited in its computational power. Therefore, it was decided to scrap the webhook approach, install Jenkins locally and trigger the pipeline manually after a new push has been made to the 'prod' branch of our GitHub repository.

- Rolling change on source code update
    - The 't2.micro' EC2 instances utilized to deploy both the frontend and backend are unable to handle a rolling change because their resources are already maxed out to run the applications. Therefore, we were required to interrupt service and take our application offline prior to initiating the pipeline to deploy a new version of the application. This interruption of service would cause a decrease in the user experience of the application. This issue can be rectified easily by utilizing more powerful EC2 instances not included in the AWS free tier subscription that can handle a rolling change update to the application.
    - Due to this limitation, we developed a separate Jenkins pipeline to take our application offline when required. As seen in the figure below, this pipeline has two stages: Stopping ECS services and Stopping EC2 instances. First, the ECS services are updated to set their desired number of tasks from one to zero. Then both EC2 instances are stopped, not terminated, to ensure they remain usable for future use.

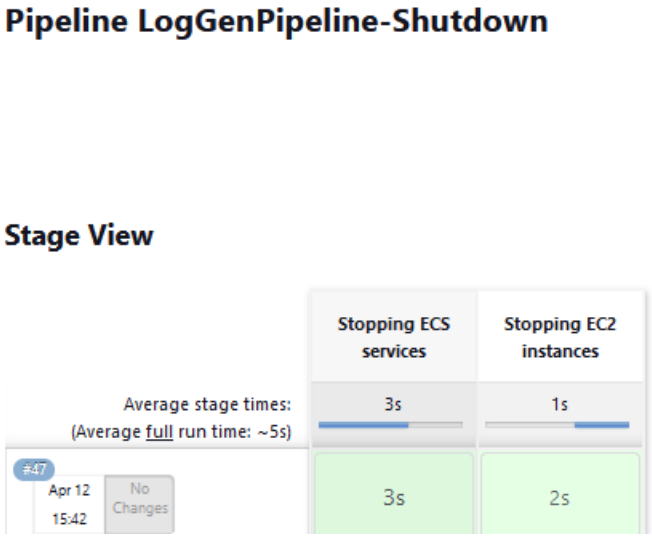


Figure 3.9: Undeployment pipeline

- Limited performance of current deployment
    - Upgrading to higher performing EC2 instances
        - The 't2.micro' specifications include 1 vCPU and 1 GiB of memory. This low performing EC2 type limits the performance of our web application. By upgrading our application to EC2 types with greater computational power we could greatly increase the performance (i.e., increase the number of logs/second output overall and increase the number of concurrent users that would experience acceptable application performance)
    - Changing deployment strategy to use Fargate + Load balancer
        - An alternative approach to deploying the application which wasn't covered under the AWS free tier subscription involves utilizing Fargate with a load balancer. This approach allows you to run containers without having to manage the underlying infrastructure, making the deployment easier to manage and easily scalable. When traffic is increased or decreased Fargate scales the number of EC2 instances running your application code up or down. This ensures the resources required for your application to perform are utilized. The load balancer distributes incoming traffic across multiple containers running the application which ensures high availability and makes the deployment extremely fault tolerant.

4.0 Technical Specifications of the Final Product

Final Product: Log Generator Microservice Application

Technical Specifications:

a. Frontend: React
b. Backend: Java 19 and Spring Boot 3
c. Hosting: ECS with self-managed EC2 instances
d. API: RESTful API
e. Responsive Design: Supports desktop devices
f. Browser Capability: Chrome, Firefox
g. Performance: Maximum 173,900 logs/s with a single batch job

In the original proposal, the following specifications were submitted:

h. Frontend: React
i. Backend: Java and Spring Boot to build the microservice application, Python to develop a machine learning model
j. Hosting: AWS EC2 or AWS Fargate
k. API: RESTful API
l. Responsive Design: Supports desktop devices
m. Browser Capability: Chrome
n. Performance: 500 logs/s

The systems capabilities were tested on locally with an Intel® Core™ i9-9900KS Processor, 8 cores 16 threads @5.00GHz. On this system, the log generator microservice application was able to achieve a maximum log rate of 155,267 logs/s with a single stream mode job, and up to 2,878 logs/s with 50 concurrent stream jobs. With a single batch mode job, the system generated on average 112,720 logs/s, and with 50 concurrent batch jobs, the system generated on average 3,404 logs/s. When both batch and stream jobs were run concurrently with an equal number of jobs, the system achieved 107,633 logs/s for stream mode and 77,389 logs/s for batch mode with one of each type. The system achieved 8,567 logs/s for stream mode and 6,457 logs/s for batch mode with 10 of each type run concurrently.

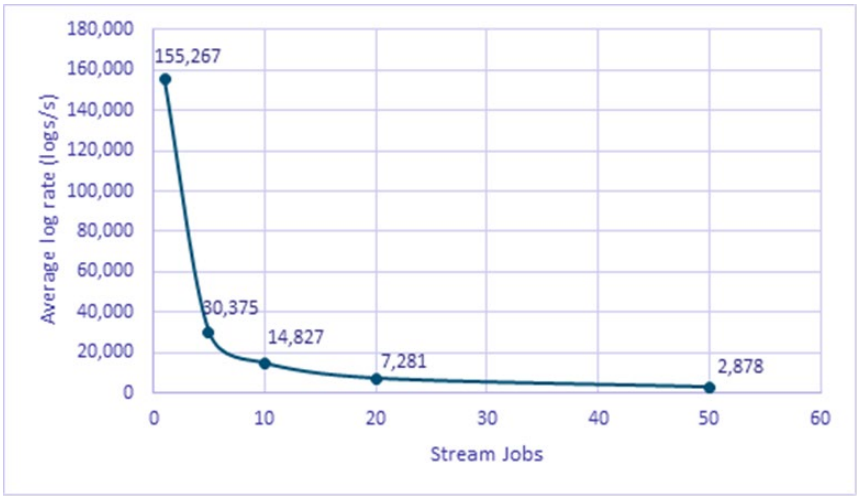


Figure 4.1. Average Log Rate, Stream Mode Only

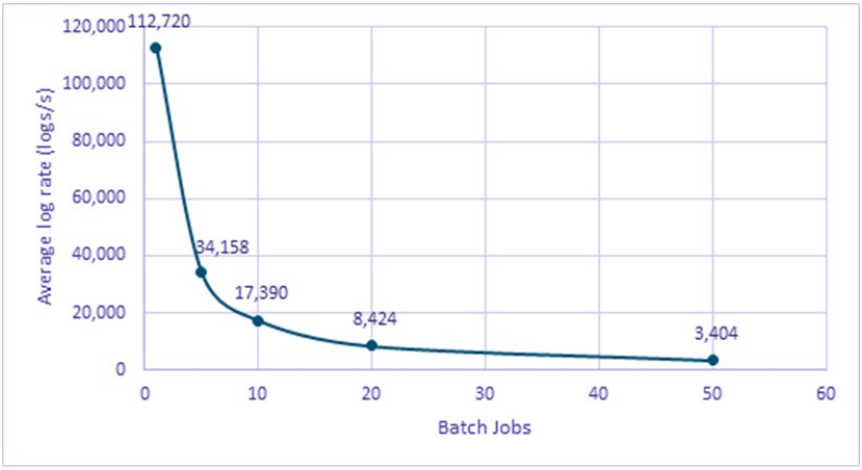


Figure 4.2. Average Log Rate, Batch Mode Only

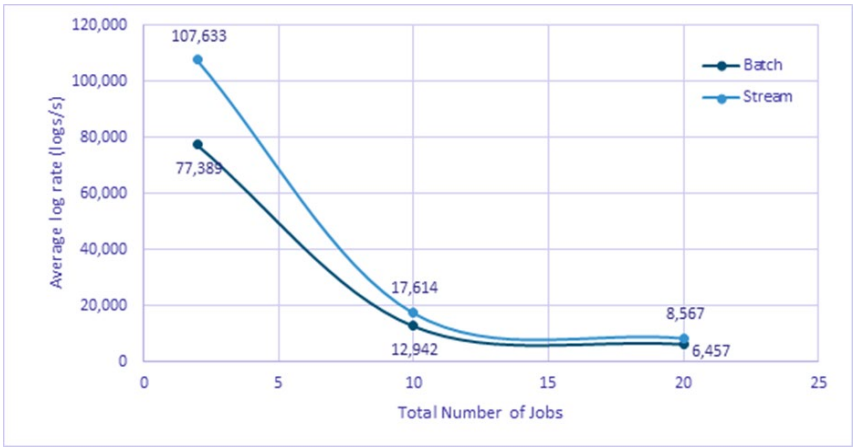


Figure 4.3. Average Log Rate, Both Batch and Stream Mode

5.0 Measures of Success and Validation Test

As discussed in our midterm report, the primary evaluation method for our project is user satisfaction based on our industry sponsor. In addition, based on discussions with the project sponsor, a key performance metric we would like to hit is being able to send at least 500 requests per second in a single stream mode job. Another performance metric we would like to hit is being able to run at least 5 stream mode jobs and 5 batch mode jobs at the same time with acceptable speeds, at least 300 logs per second.

In addition to the above performance metrics, additional evaluation metrics for our project include:

- Ability to generate more realistic log lines that simulate a certain type of malicious threat actor.
- Unit testing for key parts of the application.
- Detailed documentation and README

Performance Testing

Tested locally with an Intel® Core™ i9-9900KS Processor, 8 cores 16 threads @5.00GHz.

Stream mode

To improve performance, stream logs are sent to the address with 10 logs in each request. Requests were being sent to an endpoint on the server for testing, causing some performance loss. The overall log rate of the server maintains are high rate of above 140,000 logs/s with up to 50 jobs and is split evenly between all streaming jobs. The results of the performance testing shown in Figures 5.1-5.2 show that we were able to achieve our original goal of sending at least 500 requests per second in a single stream mode job, as 10 logs are sent with each request, meaning the server sends 15,000 requests per second with a single stream job.

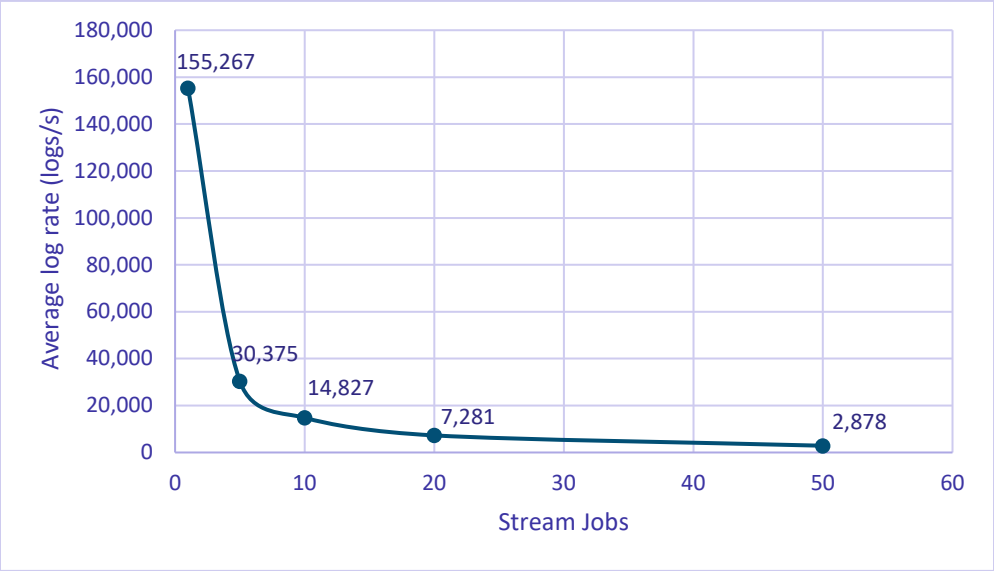


Figure 5.1: Average Log Rate, Stream Mode Only

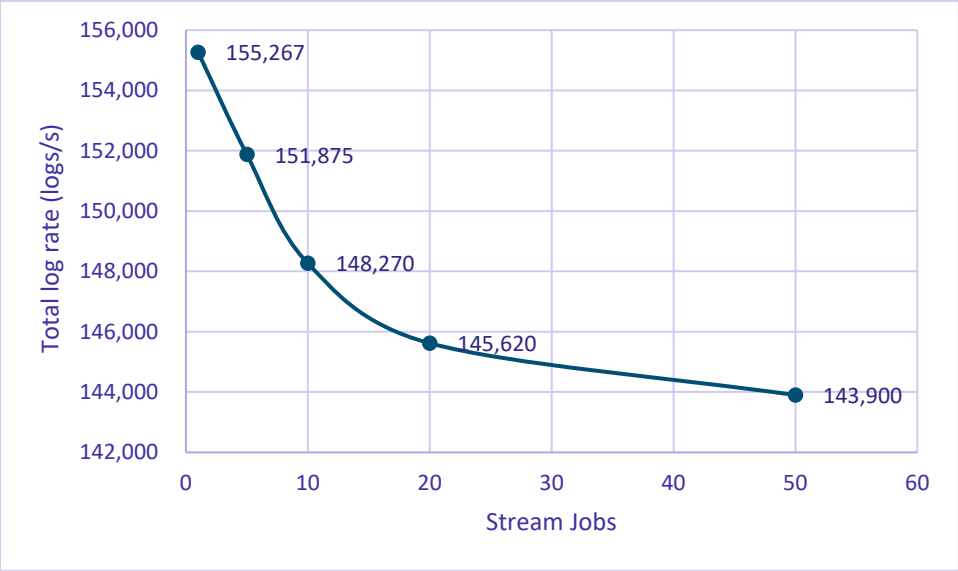


Figure 5.2: Total Log Rate, Stream Mode Only

Batch mode

Batch mode was saving logs locally in this test. The Total log rate remained at approximately 170,000 log/s from 5-50 concurrent batch jobs. Similarly to stream mode, the log rate was split evenly across all active jobs.

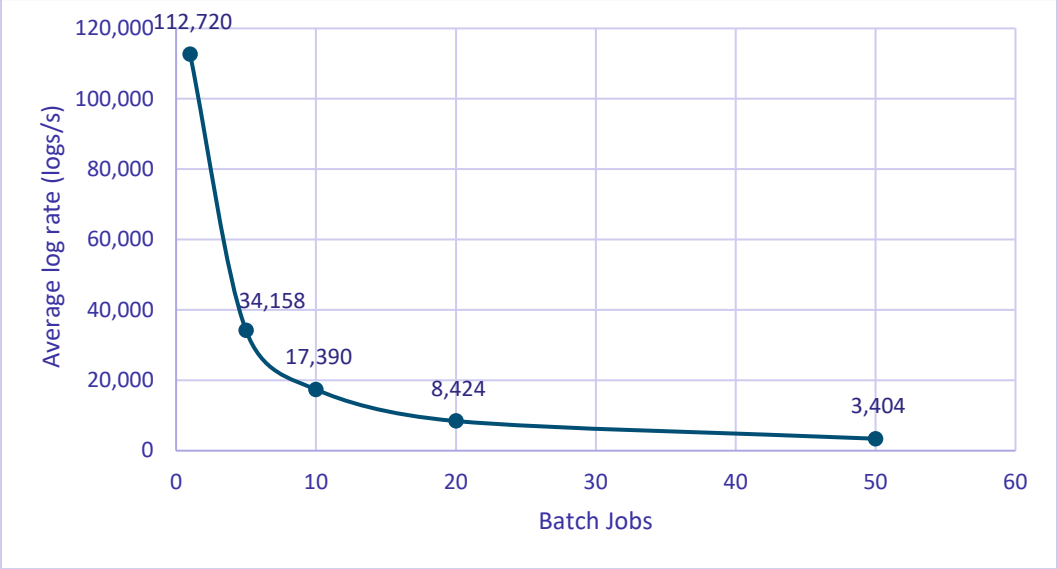


Figure 5.3: Average Log Rate, Batch Mode Only

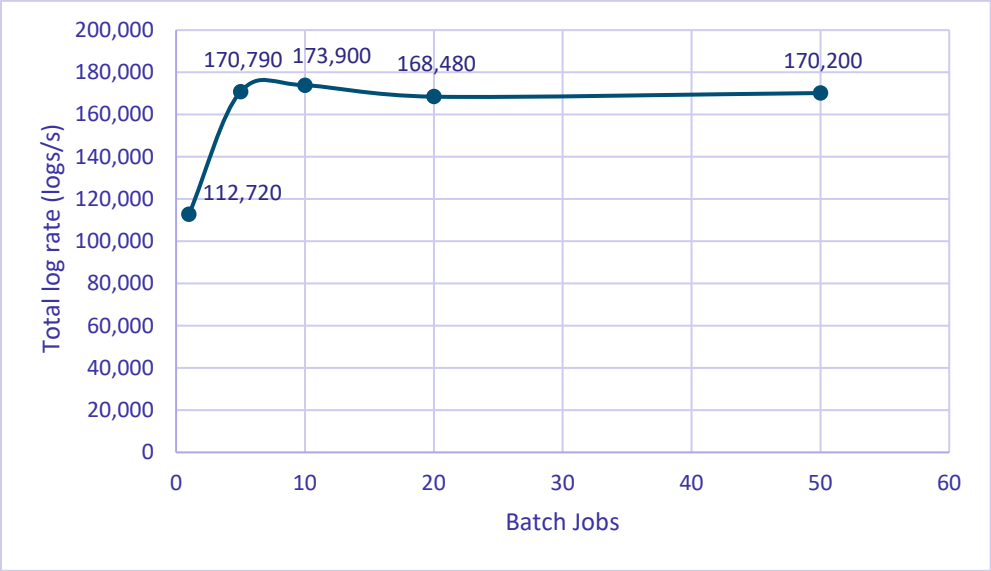


Figure 5.4: Total Log Rate, Batch Mode Only

Batch and Stream modes

Finally, batch and stream jobs were both running concurrently with an even number of jobs. The Total log rate remained at above 150,000 log/s with 2-20 jobs running at the same time. The stream mode average log rate was higher than batch mode throughout the test, being around 35% higher. Within the batch and stream modes, the log rate was split evenly between active jobs. The results of the performance testing shown in Figure 5.5 that we were able to achieve our original goal of being able to run at least 5 stream mode jobs and 5 batch mode jobs at the same time with at least 300 logs per second. When 5 batch and stream jobs were run concurrently on the server, the batch jobs obtained an average log rate of 12,942 logs/s and the stream jobs obtained an average log rate of 17,614 logs/s.

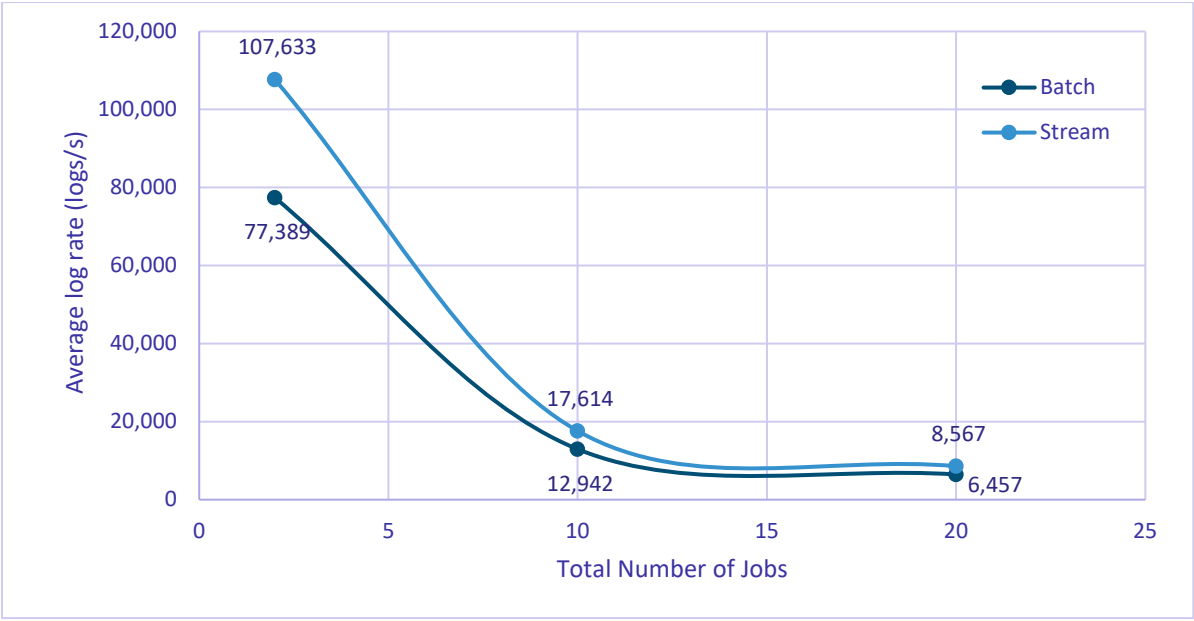


Figure 5.5: Average Log Rate, Both Batch and Stream Mode

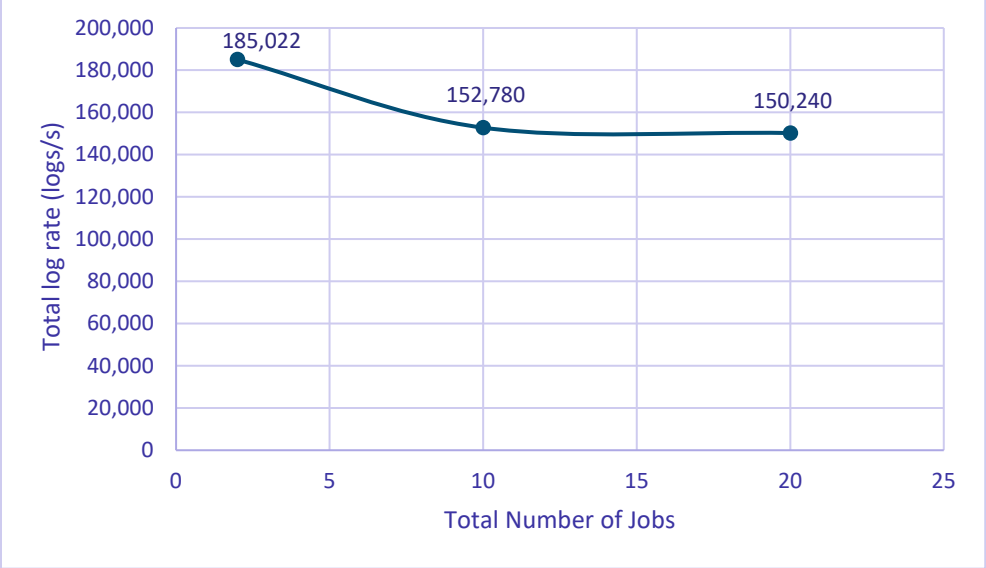


Figure 5.6: Total Log Rate, Both Batch and Stream Mode

In addition to achieving our original goals for performance, we developed a test suite with a total of 164 unit tests distributed across our service and controller classes, as described in sections 2.0 and 2.1 and in Figure 5.7 below, to test the core functionality of our application. Detailed documentation for our application was written within the java files and a README was uploaded to our GitHub repository with detailed instructions and gifs that show how to use the application, including how to customize the log lines generated by the microservice, how to configure the application to use batch or stream mode, and an overview of the user interface. Furthermore, the README file provides instructions with how to deploy the full stack application locally, as well as how to deploy the full stack application using AWS ECS with self-managed EC2 instances using Jenkins pipeline.

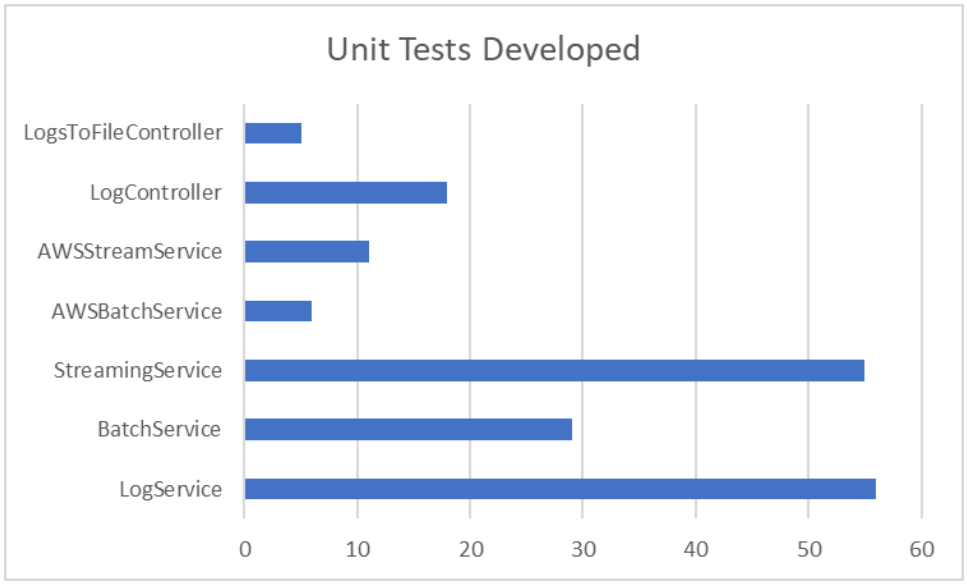


Figure 5.7: Test Suite Distribution

Finally, in terms of the scope of the product that was developed when compared to what was outlined in the proposal, we were mostly successful. As outlined in the proposal, we developed a performant and scalable microservice application in phase 1 that can be run in either batch or stream mode, the user can customize the batch size for batch more and the address to stream to for stream mode, the user can ask the program to stop generating log lines for either mode, the application runs as a multithreaded application, and the logs generated are in JSON format. In phase 2 we deployed the service to AWS and the final output of the generated log file is stored in AWS s3. In phase 3, we were unable to modify the microservice application to generate realistic data by applying machine learning. However, we performed research to explore using Generative Adversarial Networks (GANs) to generate more realistic logs that mimic the patterns and characteristics of real logs and outlined future steps to implement the GAN in our report. In addition, we added options to allow the user to generate customizable logs that can simulate a certain type of compromise or malicious threat actor.

6.0 List of Tools, Materials, Supplies and Cost

AWS Free Tier

To ensure we keep the costs of the deployment to a minimum we utilized a deployment strategy that would remain free within the limitations of the free tier subscription but is also an industry standard.

- S3: no costs incurred
- EC2
    - Elastic IPs
        - AWS charges $0.005 per Elastic IP address not attached to a running instance per hour. Since there is also a limitation on how many hours you can have EC2 instances running monthly before incurring fees, we were forced to undeploy our application and stop our EC2 instances. This resulted in us being charged for the two elastic IPs that were idle (one for each the frontend and backend). The charges incurred were: $5.19 USD in March 2023 and a forecasted amount of approximately the same charges in April 2023.
- ECS: no costs incurred.
- ECR: no costs incurred.

Our initial estimate was to incur no charges for this project. Due to the idle elastic IPs to ensure we remain with the same frontend and backend URLs we incurred a total project cost of less than $15 CAD.

Other tools and resources

To conclude, no other tools or resources utilized during this project resulted in any additional fees. The other tools utilized during the project include:

- Jenkins
    - Software to automate the deployment process.
- GitHub
    - Software to act as our source control management tool for the project.
- Docker
    - Software used to create docker images of our application to be consumed by AWS products.
- Preferred IDEs (i.e., IntelliJ, VS Code, etc.)
    - Software used to develop our application.
- Jira
    - Software used to aid in project management and task tracking.
- Confluence
    - Software used to store meeting minutes and track action items.
- Discord, Teams and Webex
    - Software used for virtual meetings, to discuss project topics and share content related to the project.
- React
    - Software framework utilized to build our frontend application.

- Node Packages
    - The following packages and libraries were used to build the log generator application website using react: @chakra-ui/icons, @chakra-ui/react, @emotion/react, @emotion/styled, @stomp/stompjs, @testing-library/jest-dom, @testing-library/react, @testing-library/user-event, formik, framer-motion, react, react-apexcharts, react-dom, react-icons, react-router-dom, react-scripts, react-table, sockjs-client, uuid, and web-vitals.
- Spring Boot
    - Software framework utilized to build our backend application.
- Maven
    - Software used to build automation and dependency management of our backend application.
- Draw.io and LucidChart
    - Software used to create UML, pipeline, and package diagrams.
- Junit 5 and Mockito
    - Software utilized for testing our application.

7.0 References

[1] A. Ng, “Deep Learning Specialization,” Coursera. [Online]. Available: https://www.coursera.org/specializations/deep-learning. [Accessed: 08-Mar-2023].

[2] A. Ng, “Machine Learning Specialization,” Coursera. [Online]. Available: https://www.coursera.org/specializations/machine-learning-introduction. [Accessed: 08-Mar-2023].

[3] A. V. Carrales, “How to create a CI/CD pipeline with Jenkins, containers, and Amazon ECS,” Medium, 22-Jun-2022. [Online]. Available: https://betterprogramming.pub/how-to-create-a-ci-cd-pipeline-with-jenkins-containers-and-amazon-ecs-af4eaec75b8b. [Accessed: 09-Mar-2023].

[4] “AWS documentation - docs.aws.amazon.com.” [Online]. Available: https://docs.aws.amazon.com/index.html. [Accessed: 08-Mar-2023].

[5] Baeldung, “Intro to WebSockets with spring,” Baeldung, 22-Jun-2022. [Online]. Available: https://www.baeldung.com/websockets-spring. [Accessed: 08-Mar-2023].

[6] B2 Tech, “Upload file to AWS S3 Bucket in Spring Boot [crash course],” YouTube, 15-Aug-2021. [Online]. Available: https://www.youtube.com/watch?v=c3POiw8rHoQ. [Accessed: 08-Mar-2023].

[7] C. Ajieh, D. Zarinski, F. Bache, and L. Lee, “Cisco Secure Endpoint Data Engineering Project - Log Generator Project Proposal”.

[8] C. Darby, “Learn hibernate and spring (as a total beginner) tutorial,” Udemy. [Online]. Available: https://www.udemy.com/course/spring-hibernate-tutorial/. [Accessed: 08-Mar-2023].

[9] “Class ConcurrentHashMap,” *ConcurrentHashMap (java 2 platform SE 5.0)*. [Online]. Available: https://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/ConcurrentHashMap.html. [Accessed: 09-Mar-2023].

[10] D. Foster, Generative deep learning: Teaching machines to paint, write, compose, and play. Beijing: O'Reilly, 2019.

[11] “Deploying a react spring boot app with Docker Part I,” YouTube, 07-Jul-2022. [Online]. Available: https://www.youtube.com/watch?v=x5W5oDzqND0. [Accessed: 10-Mar-2023].

[12] “Deploying a react spring boot app with Docker Part II,” YouTube, 12-Jul-2022. [Online]. Available: https://www.youtube.com/watch?v=4ugChIR9sS8. [Accessed: 10-Mar-2023].

[13] “Deploying a react spring boot app with Docker Part IV,” YouTube, 03-Aug-2022. [Online]. Available: https://www.youtube.com/watch?v=89l2o70GdO8. [Accessed: 10-Mar-2023].

[14] “Developer guide - AWS SDK for Java 2.x - AWS SDK for Java 2.X.” [Online]. Available: https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/home.html. [Accessed: 08-Mar-2023].

[15] Eduardo, Sanket Daru, and Ged, “Async Rest API using Spring Boot,” Sanket Daru, 04-May-2021. [Online]. Available: https://sanketdaru.com/blog/polling-model-async-rest-spring-boot/. [Accessed: 08-Mar-2023].

[16] G. Blumenstock, Y. Fan, and Y. Tian, “GENERATIVE MODELS.” [Online]. Available: https://humboldt-wi.github.io/blog/research/information_systems_1819/generativemodels/. [Accessed: 10-Mar-2023].

[17] “Getting started,” React. [Online]. Available: https://reactjs.org/docs/getting-started.html. [Accessed: 08-Mar-2023].

[18] “How to install Docker on Windows 11 | or windows 10 #docker,” YouTube, 07-Nov-2022. [Online]. Available: https://www.youtube.com/watch?v=TA0R6yeHDqw. [Accessed: 10-Mar-2023].

[19] “How to deploy a Docker app to AWS ECS,” YouTube, 18-Dec-2021. [Online]. Available: https://www.youtube.com/watch?v=YDNSItBN15w. [Accessed: 10-Mar-2023].

[20] I. Ashrapov, “Review of GANS for Tabular Data,” Medium, 04-Jun-2020. [Online]. Available: https://towardsdatascience.com/review-of-gans-for-tabular-data-a30a2199342. [Accessed: 10-Apr-2023].

[21] “Jenkins practical beginners’ course for devops | CI/CD pipeline with Jenkins | Learn devops tools EP1,” YouTube, 16-May-2021. [Online]. Available: https://www.youtube.com/watch?v=wHtIWvb5nzo. [Accessed: 09-Mar-2023].

[22] K. Leung, “The Dying Relu Problem, Clearly Explained,” Medium, 23-Sep-2021. [Online]. Available: https://towardsdatascience.com/the-dying-relu-problem-clearly-explained-42d0c54e0d24. [Accessed: 10-Apr-2023].

[23] L. Gupta, “Spring webclient (with examples),” HowToDoInJava, 01-Dec-2022. [Online]. Available: https://howtodoinjava.com/spring-webflux/webclient-get-post-example/. [Accessed: 10-Mar-2023].

[24] L. Xu and K. Veeramachaneni, “Synthesizing Tabular Data using Generative Adversarial Networks - arxiv.org.” [Online]. Available: https://arxiv.org/pdf/1811.11264.pdf): [Accessed: 09-Apr-2023].

[25] Linux. [Online]. Available: https://www.jenkins.io/doc/book/installing/linux/. [Accessed: 09-Mar-2023].

[26] “Normalization |Machine Learning |Google Developers,” Google. [Online]. Available: https://developers.google.com/machine-learning/data-prep/transform/normalization. [Accessed: 06-Apr-2023].

[27] R. Fadatare (Java Guides), “Building microservices with Spring Boot and Spring Cloud. Section 22: Dockering Spring Boot Application Step by Step,” Udemy, 26-Jan-2023. [Online]. Available: https://www.udemy.com/course/building-microservices-with-spring-boot-and-spring-cloud/. [Accessed: 08-Mar-2023].

[28] R. Fadatare (Java Guides), “Building microservices with Spring Boot and Spring Cloud,” Udemy, 26-Jan-2023. [Online]. Available: https://www.udemy.com/course/building-microservices-with-spring-boot-and-spring-cloud/. [Accessed: 08-Mar-2023].

[29] R. Fadatare (Java Guides), “Building real-time rest APIs with Spring boot - blog App,” Udemy, 12-Jan-2023. [Online]. Available: https://www.udemy.com/course/building-real-time-rest-apis-with-spring-boot/. [Accessed: 08-Mar-2023].

[30] Section 26: Websocket Support. [Online]. Available: https://docs.spring.io/spring-framework/docs/4.3.x/spring-framework-reference/html/websocket.html. [Accessed: 08-Mar-2023].

[31] Stavroula Bourou, A. El Saer, T.-H. Velivassaki, A. Voulkidis, and T. Zahariadis, “A Review of Tabular Data Synthesis Using GANs on an IDS Dataset.”.

[32] “Working with Amazon S3 objects.” [Online]. Available: https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/java_s3_code_examples.html. [Accessed: 08-Mar-2023].

[33] Z. Zhao, A. Kunar, R. Birke, H. Van der Scheer, and L. Y. Chen, “CTAB-GAN: Effective Table Data Synthesizing.” [Online]. Available: https://arxiv.org/pdf/2102.08369v1.pdf. [Accessed: 11-Apr-2023].

8.0 Appendix

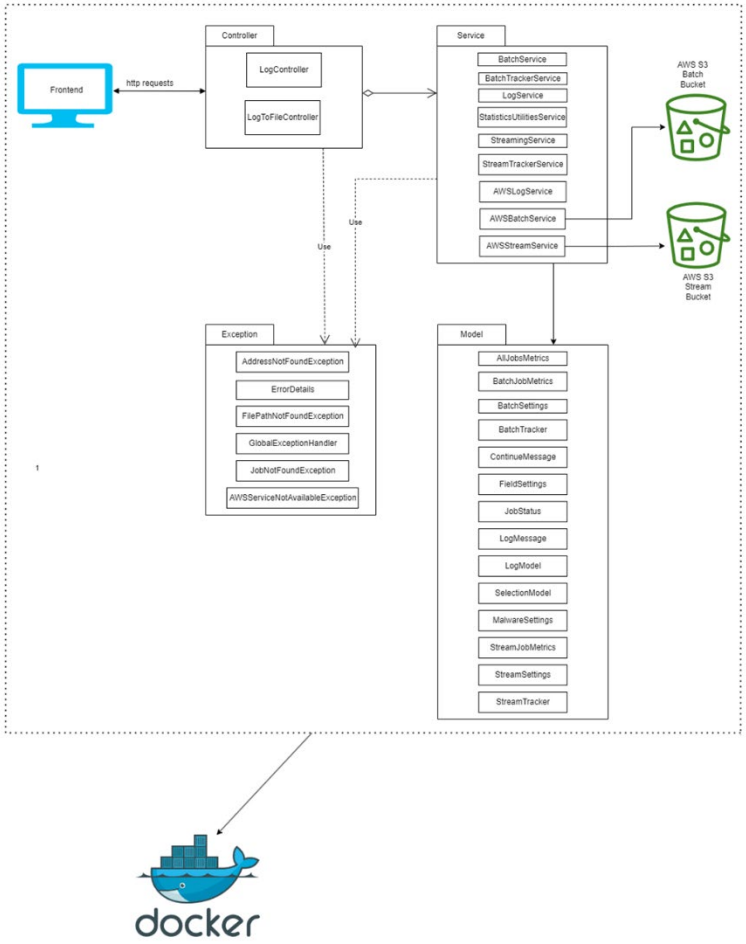


Figure 8.1: Package Level Diagram of Phase 2 of the Log Generator Application

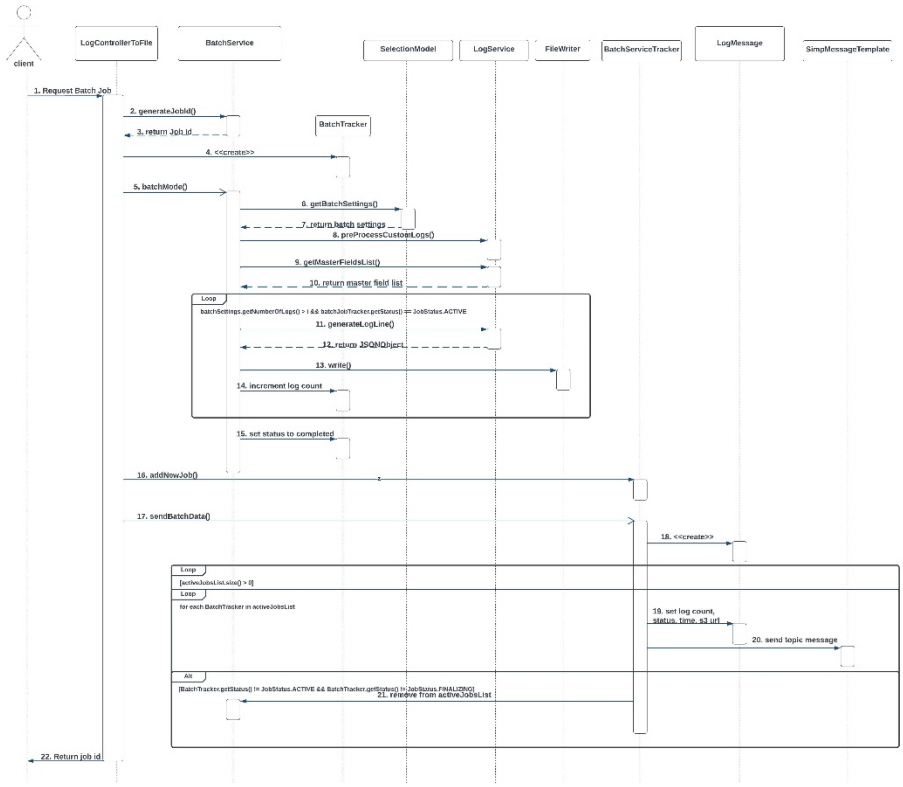
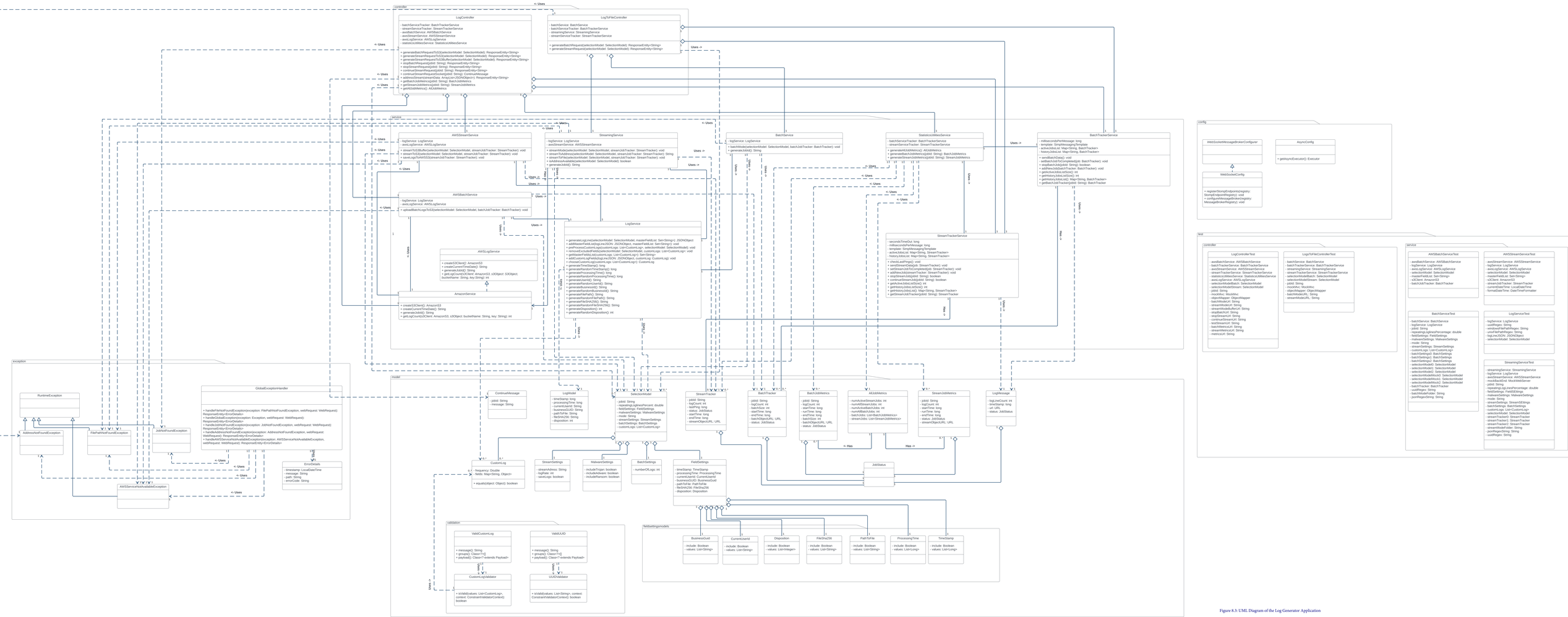


Figure 8.2: Batch job sequence interaction diagram


controller
LogController
- batchServiceTracker: BatchTrackerService
- streamServiceTracker: StreamTrackerService
- awsBatchService: AWSBatchService
- awsStreamService: AWSStreamService
- awsLogService: AWSLogService
- statisticsUtilitiesService: StatisticsUtilitiesService
+ generateBatchRequestToS3(selectionModel: SelectionModel): ResponseEntity<String>
+ generateStreamRequestToS3(selectionModel: SelectionModel): ResponseEntity<String>
+ generateStreamRequestToS3Buffer(selectionModel: SelectionModel): ResponseEntity<String>
+ stopBatchRequest(jobId: String): ResponseEntity<String>
+ stopStreamRequest(jobId: String): ResponseEntity<String>
+ continueStreamRequest(jobId: String): ResponseEntity<String>
+ addStreamData(streamData: Array<List<JSONObject>>): ResponseEntity<String>
+ addStreamData(streamData: Array<List<JSONObject>>): ResponseEntity<String>
+ getBatchJobMetrics(jobId: String): BatchJobMetrics
+ getStreamJobMetrics(jobId: String): StreamJobMetrics
+ getAllJobMetrics(): AllJobMetrics
LogToFileController
- batchService: BatchService
- batchServiceTracker: BatchTrackerService
- streamingService: StreamingService
- streamServiceTracker: StreamTrackerService
+ generateBatchRequest(selectionModel: SelectionModel): ResponseEntity<String>
+ generateStreamRequest(selectionModel: SelectionModel): ResponseEntity<String>
service
AWSStreamService
- logService: LogService
- awsLogService: AWSLogService
+ streamToS3Buffer(selectionModel: SelectionModel, streamJobTracker: StreamTracker): void
+ streamToS3(selectionModel: SelectionModel, streamJobTracker: StreamTracker): String
+ streamToFile(selectionModel: SelectionModel, streamJobTracker: StreamTracker): void
+ isAwsServiceAvailable(selectionModel: SelectionModel): boolean
+ getJobStatus(jobId): String
StreamingService
- logService: LogService
- awsStreamService: AWSStreamService
+ streamMode(selectionModel: SelectionModel, streamJobTracker: StreamTracker): void
+ streamToAddress(selectionModel: SelectionModel, streamJobTracker: StreamTracker): String
+ streamToFile(selectionModel: SelectionModel, streamJobTracker: StreamTracker): void
+ isAddressAvailable(selectionModel: SelectionModel): boolean
+ getJobStatus(jobId): String
AWSBatchService
- logService: LogService
- awsLogService: AWSLogService
+ uploadBatchLogsToS3(selectionModel: SelectionModel, batchJobTracker: BatchTracker): void
AWSLogService
+ createS3Client(): AmazonS3
+ createCurrentTimeData(): String
+ generateJobId(): String
+ getLogCountS3Client(: AmazonS3, s3Object: S3Object, bucketName: String, key: String): int
AmazonService
+ createS3Client(): AmazonS3
+ createCurrentTimeData(): String
+ generateJobId(): String
+ getLogCountS3Client(: AmazonS3, s3Object: bucketName: String, key: String): int
LogService
+ generateLogLine(selectionModel: SelectionModel, masterFieldList: Set<String>): JSONObject
+ addMasterFieldList(logLineJSON: JSONObject, masterFieldList: Set<String>): void
+ removeExcludedFields(logLine: JSONObject, selectionModel: SelectionModel): void
+ removeExcludedFields(selectionModel: SelectionModel, customLogs: List<CustomLog>: void
+ getMasterFieldList(customLogs: List<CustomLog>): Set<String>
+ addCustomLogFields(logLineJSON: JSONObject, customLog: CustomLog): void
+ chooseCustomLog(customLogs: List<CustomLog>): CustomLog
+ generateRandomUserId(): String
+ generateRandomTimeString(): long
+ generateProcessingTime(): long
+ generateRandomProcessingTime(): long
+ generateUserId(): String
+ generateRandomUserId(): String
+ generateBusinessId(): String
+ generateRandomBusinessId(): String
+ generateFilePath(): String
+ generateRandomFilePath(): String
+ generateFileSHA256(): String
+ generateRandomFileSHA256(): String
+ generateDisposition(): int
+ generateRandomDisposition(): int
BatchService
- logService: LogService
+ batchMode(selectionModel: SelectionModel, batchJobTracker: BatchTracker): void
+ generateJobId(): String
StatisticsUtilitiesService
- batchServiceTracker: BatchTrackerService
- streamServiceTracker: StreamTrackerService
+ generateAllJobMetrics(): AllJobMetrics
+ generateBatchJobMetrics(jobId: String): BatchJobMetrics
+ generateStreamJobMetrics(jobId: String): StreamJobMetrics
BatchTrackerService
- millisecondsPerMessage: long
- template: SimpMessagingTemplate
- activeJobsList: Map<String, BatchTracker>
- historyJobsList: Map<String, BatchTracker>
+ sendBatchData(): void
+ addNewJob(batchTracker: BatchTracker): void
+ stopBatch(Job(jobId: String): boolean
+ addNewJob(batchTracker: BatchTracker): void
+ getActiveJobsListSize(): int
+ getHistoryJobsListSize(): int
+ getHistoryJobsList(): Map<String, BatchTracker>
+ getBatchJobTracker(jobId: String): BatchTracker
StreamTrackerService
- secondsTimeOut: long
- millisecondsPerMessage: long
- template: SimpMessagingTemplate
- activeJobsList: Map<String, StreamTracker>
- historyJobsList: Map<String, StreamTracker>
+ checkLastPing(): void
+ addNewJob(streamTracker: StreamTracker): void
+ setStreamJobToCompleted(job: StreamTracker): void
+ addNewJob(streamTracker: StreamTracker): void
+ stopStream(Job(jobId: String): boolean
+ continueStream(Job(jobId: String): boolean
+ getActiveJobsListSize(): int
+ getHistoryJobsListSize(): int
+ getHistoryJobsList(): Map<String, StreamTracker>
+ getStreamJobTracker(jobId: String): StreamTracker
exception
RuntimeException
GlobalExceptionHandler
+ handleFileNotFoundException(exception: FilePathNotFoundException, webRequest: WebRequest): ResponseEntity<ErrorDetails>
+ handleGlobalException(exception: Exception, webRequest: WebRequest): ResponseEntity<ErrorDetails>
+ handleJobNotFoundException(exception: JobNotFoundException, webRequest: WebRequest): ResponseEntity<ErrorDetails>
+ handleAddressNotFoundException(exception: AddressNotFoundException, webRequest: WebRequest): ResponseEntity<ErrorDetails>
+ handleAWSServiceNotAvailableException(exception: AWSServiceNotAvailableException, webRequest: WebRequest): ResponseEntity<ErrorDetails>
AddressNotFoundException
FilePathNotFoundException
JobNotFoundException
AWSServiceNotAvailableException
ErrorDetails
- timestamp: LocalDateTime
- message: String
- path: String
- errorCode: String
model
ContinueMessage
- jobId: String
- message: String
LogModel
- timeStamp: long
- processingTime: long
- currentUserId: String
- businessGUID: String
- pathToFile: String
- fileSHA256: String
- disposition: int
SelectionModel
- jobId: String
- repeatingLogLinesPercent: double
- fieldSettings: FieldSettings
- malwareSettings: MalwareSettings
- mode: String
- streamSettings: StreamSettings
- batchSettings: BatchSettings
- customLogs: List<CustomLog>
StreamTracker
- jobId: String
- logCount: int
- batchPing: long
- status: JobStatus
- startTime: long
- endTime: long
- streamObjectURL: URL
BatchTracker
- jobId: String
- logCount: int
- batchSize: int
- startTime: long
- endTime: long
- batchObjectURL: URL
- status: JobStatus
BatchJobMetrics
- jobId: String
- logCount: int
- batchSize: int
- startTime: long
- endTime: long
- batchObjectURL: URL
- status: JobStatus
AllJobMetrics
- numActiveStreamJobs: int
- numActiveBatchJobs: int
- numAllStreamJobs: int
- numAllBatchJobs: int
- batchJobs: List<BatchJobMetrics>
- streamJobs: List<StreamJobMetrics>
StreamJobMetrics
- jobId: String
- logCount: int
- startTime: long
- endTime: long
- status: JobStatus
- streamObjectURL: URL
LogMessage
- logLineCount: int
- timeStamp: long
- url: URL
- status: JobStatus
CustomLog
- frequency: Double
- fields: Map<String, Object>
+ equals(object: Object): boolean
StreamSettings
- streamAdress: String
- logRate: int
- saveLogs: boolean
MalwareSettings
- includeTrojan: boolean
- includeAdware: boolean
- includeRansom: boolean
BatchSettings
- numberOfLogs: int
FieldSettings
- timeStamp: TimeStamp
- processingTime: ProcessingTime
- currentUserId: CurrentUserId
- businessGUID: BusinessGuid
- pathToFile: PathToFile
- fileSHA256: FileSha256
- disposition: Disposition
JobStatus
validation
ValidCustomLog
+ message(): String
+ groups(): Class<?>[]
+ payload(): Class<? extends Payload>
CustomLogValidator
+ isValid(values: List<CustomLog>, context: ConstraintValidatorContext): boolean
ValidUUID
+ message(): String
+ groups(): Class<?>[]
+ payload(): Class<? extends Payload>
UUIDValidator
+ isValid(values: List<String>, context: ConstraintValidatorContext): boolean
fieldSettingsmodels
BusinessGuid
- include: Boolean
- values: List<String>
CurrentUserId
- include: Boolean
- values: List<String>
Disposition
- include: Boolean
- values: List<Integer>
FileSha256
- include: Boolean
- values: List<String>
PathToFile
- include: Boolean
- values: List<String>
ProcessingTime
- include: Boolean
- values: List<Long>
TimeStamp
- include: Boolean
- values: List<Long>
config
WebSocketMessageBrokerConfigurer
WebSocketConfig
+ registerStompEndpoints(registry: StompEndpointRegistry): void
+ configureMessageBroker(registry: MessageBrokerRegistry): void
AsyncConfig
+ getAsyncExecutor(): Executor
test
controller
LogControllerTest
- awsBatchService: AWSBatchService
- awsStreamService: AWSStreamService
- awsStreamService: AWSStreamService
- streamTrackerService: StreamTrackerService
- statisticsUtilitiesService: StatisticsUtilitiesService
- awsLogService: AWSLogService
- selectionModelBatch: SelectionModel
- selectionModelStream: SelectionModel
- mockMvc: MockMvc
- mockMvc: MockMvc
- objectMapper: ObjectMapper
- batchModeUrl: String
- streamModeUrl: String
- stopBatchUrl: String
- stopStreamUrl: String
- continueStreamUrl: String
- testStreamUrl: String
- batchMetricsUrl: String
- streamMetricsUrl: String
- metricsUrl: String
LogsToFileControllerTest
- batchService: BatchService
- batchServiceTracker: BatchTrackerService
- streamingService: StreamingService
- streamTrackerService: StreamTrackerService
- selectionModelBatch: SelectionModel
- selectionModelStream: SelectionModel
- jobId: String
- mockMvc: MockMvc
- objectMapper: ObjectMapper
- batchModeURL: String
- streamModeURL: String
service
AWSBatchServiceTest
- awsBatchService: AWSBatchService
- awsStreamService: AWSStreamService
- awsLogService: AWSLogService
- selectionModel: SelectionModel
- masterFieldList: Set<String>
- s3Client: Amazons3
- batchJobTracker: BatchTracker
AWSStreamServiceTest
- awsStreamService: AWSStreamService
- awsLogService: AWSLogService
- selectionModel: SelectionModel
- masterFieldList: Set<String>
- s3Client: Amazons3
- streamJobTracker: StreamTracker
- currentDateTime: LocalDateTime
- formatDateTime: DateTimeFormatter
BatchServiceTest
- batchService: BatchService
- logService: LogService
- logToFileService: LogToFileService
- awsBatchService: AWSBatchService
- fieldSettings: FieldSettings
- malwareSettings: MalwareSettings
- logLine: String
- streamSettings: StreamSettings
- customLogs: List<CustomLog>
- batchSettings1: BatchSettings
- batchSettings2: BatchSettings
- selectionModel1: SelectionModel
- selectionModel2: SelectionModel
- selectionModel3: SelectionModel
- selectionModelMock0: SelectionModel
- selectionModelMock1: SelectionModel
- selectionModelMock2: SelectionModel
- batchJobTracker: BatchTracker
- uuidRegex: String
- batchModelFolder: String
- jsonRegexString: String
LogServiceTest
- logService: LogService
- awsLogService: AWSLogService
- windowsFilePathRegex: String
- unixFilePathRegex: String
- logLineJSON: JSONObject
- selectionModel: SelectionModel
StreamingServiceTest
- streamingService: StreamingService
- logService: LogService
- awsStreamService: AWSStreamService
- mockBackEnd: MockWebServer
- jobId: String
- repeatingLogLinesPercentage: double
- fieldSettings: FieldSettings
- malwareSettings: MalwareSettings
- mode: String
- streamSettings: StreamSettings
- batchSettings: BatchSettings
- customLogs: List<CustomLog>
- selectionModel: SelectionModel
- streamTracker0: StreamTracker
- streamTracker1: StreamTracker
- streamTracker2: StreamTracker
- streamModelFolder: String
- jsonRegexString: String
- uuidRegex: String
<- Uses
Uses ->
1..1
0..*
<- Has
Has ->
Figure 8.3: UML Diagram of the Log Generator Application

8.1 API Documentation

All fields are **REQUIRED** unless otherwise mentioned.

GeneratorSettings Object

This object provides all the settings required for log generation.

Fixed Fields

Field Name	Type	Description
repeatingLoglinesPercent	number	Percent chance a logline should repeat, expressed as a decimal. Can be from 0 to 1 inclusive.
fieldSettings	FieldSettings Object	An object that specifies which fields to include in the logs and optionally their values.
customLogs	array of CustomLog objects	An array of objects that specify custom log data to include.
mode	string	The mode in which to send the logs. Must be "Batch" or "Stream".
streamSettings	StreamSettings Object	Settings used for stream mode. OPTIONAL if stream mode not selected.
batchSettings	BatchSettings Object	Settings used for batch mode. OPTIONAL if batch mode not selected.

GeneratorSettings Object Example

```
{
  "repeatingLoglinesPercent": 0,
  "fieldSettings": {
    "timeStamp": {
      "include": true,
      "values": []
    },
    "processingTime": {
      "include": true,
      "values": []
    },
    "currentUserID": {
      "include": true,
      "values": []
    },
    "businessGUID": {
      "include": true,
      "values": [
        "Cisco"
      ]
    },
    "pathToFile": {
      "include": true,
      "values": []
    },
    "fileSHA256": {
```

```
        "include": true,
        "values": []
      },
      "disposition": {
        "include": true,
        "values": []
      }
    },
    "customLogs": [
      {
        "frequency": 0.05,
        "fields": {
          "processingTime": "15",
          "pathToFile": "C:\\Users\\malware.exe",
          "disposition": "5"
        }
      }
    ],
    "mode": "Batch",
    "streamSettings": {
      "streamAddress": "http://localhost:8080/api/v1/generate/stream
/toAddress",
      "logRate": 300,
      "saveLogs": false
    },
    "batchSettings": {
      "numberOfLogs": 100
    }
  }
```

FieldSettings Object

This object provides the settings for fields in the generated logs.

Fixed Fields

Field Name	Type	Description
timeStamp	FieldSetting Object	An object that specifies whether to include the timestamp field in the logs and optionally its values. Randomly generated as timestamp of the log entry in Unix epoch format.
processingTime	FieldSetting Object	An object that specifies whether to include the processing time field in the logs and optionally its values. Randomly generated as the time it took to process the log entry in milliseconds.
currentUserID	FieldSetting Object	An object that specifies whether to include the current user ID field in the logs and optionally its values. Randomly generated as the GUID of the current user associated with the log entry.
businessGUID	FieldSetting Object	An object that specifies whether to include the business GUID field in the logs and optionally its values. Randomly generated as the business GUID associated with the log entry.

pathToFile	FieldSetting Object	An object that specifies whether to include the path to file field in the logs and optionally its values. Randomly generated as the file path associated with the log entry.
fileSHA256	FieldSetting Object	An object that specifies whether to include the file SHA256 field in the logs and optionally its values. Randomly generated as the SHA256 hash of the file associated with the log entry.
disposition	FieldSetting Object	An object that specifies whether to include the disposition field in the logs and optionally its values. Randomly generated as the disposition code associated with the log entry.

FieldSettings Object Example

```
{
  "timeStamp": {
    "include": true,
    "values": []
  },
  "processingTime": {
    "include": true,
    "values": []
  },
  "currentUserID": {
    "include": true,
    "values": []
  },
  "businessGUID": {
    "include": true,
    "values": [
      "Cisco"
    ]
  },
  "pathToFile": {
    "include": true,
    "values": []
  },
  "fileSHA256": {
    "include": true,
    "values": []
  },
  "disposition": {
    "include": true,
    "values": []
  }
}
```

FieldSetting Object

This object specifies the settings for a given field.

Fixed Fields

Field Name	Type	Description
include	boolean	If generated logs should include the field.
values	array	Empty array if values should be randomly generated, or array with values to randomly choose from for logs.

FieldSetting Object Example

```
{
    "include": true,
    "values": [
      "Cisco",
      "Amazon"
    ]
  }
```

CustomLog Object

Fixed Fields

Field Name	Type	Description
frequency	float	The frequency at which to include the custom log data in the logs.
fields	CustomFields object	An object that specifies the custom log data to include in the logs.

CustomLog Object Example

```
 {
  "frequency": 0.05,
  "fields": {
    "processingTime": "15",
    "pathToFile": "C:\\Users\\malware.exe",
    "disposition": "5"
  }
}
```

CustomFields Object

All fields in this object are **optional**. If a field isn't specified, it will be randomly generated. New fields can be added to the CustomFields object as required.

Variable Fields

Field Name	Type	Description
timeStamp	number or string	The timestamp of the log entry.
processingTime	number or string	The time it took to process the log entry.
currentUserID	number or string	The ID of the current user associated with the log entry.

businessGUID	number or string	The business ID associated with the log entry.
pathToFile	string	The file path associated with the log entry.
fileSHA256	string	The SHA256 hash of the file associated with the log entry.
disposition	number or string	The disposition code associated with the log entry.
any	number or string	Any field can be added to the custom log.

CustomFields Object Example

```
{
  "timeStamp": 1680830124,
  "processingTime": 15,
  "currentUserID": "4891",
  "businessGUID": "Cisco",
  "pathToFile": "C:\\Users\\malware.exe",
  "fileSHA256": "20a74582-aa8e-42ac-a8a3-6796ff62f161",
  "disposition": 5,
  "newField": "newFieldValue"
}
```

StreamSettings Object

This object provides all the settings for stream mode.

Fixed Fields

Field Name	Type	Description
streamAddress	string	The address to send generated logs to.
logRate	integer	The rate at which to send the logs in streaming mode, expressed as logs/s.
saveLogs	boolean	If generated and streamed logs should also be saved.

StreamSettings Object Example

```
 {
    "streamAddress": "http://localhost:8080/api/v1/generate/stream
/toAddress",
    "logRate": 300,
    "saveLogs": false
  }
```

BatchSettings Object

This object provides all the settings for batch mode.

Fixed Fields

Field Name	Type	Description
numberOfLogs	number	The number of logs to generate.

BatchSettings Object Example

```
{
    "numberOfLogs": 1237
}
```

Configuring Stream Address Endpoint

Log lines will be streamed to the address provided by sending HTTP POST requests. The endpoint will receive an array of up to 10 logs in JSON format with each request. The frequency of requests will be approximately the log rate specified divided by 10. All logs sent by a given job will have the same fields and contain information such as a timestamp, file path, and processing time. The endpoint should return a success or error response based on the outcome of the request. Streaming will be stopped upon receiving an error response or being cancelled.

Stream Post Request Example

```
POST /api/endpoint HTTP/1.1
Content-Type: application/json


[
  {
    "timeStamp":1082495877,
    "disposition":2,
    "currentUserID":"083550e9-1289-46af-be89-268ae1fd7ca9",
    "fileSHA256":"b6cdd8fa-848d-4a19-8da5-4d27425a5fa2",
    "pathToFile":"C:\\Program Files\\f9e3489b-06f9-49ff-b7a9-
0f1e1ed100f1.json",
    "businessGUID":"f147180d-8961-459f-af10-121c593a0fa2",
    "processingTime":153
  },
  {
    "timeStamp":332981314,
    "disposition":4,
    "currentUserID":"5071c9ee-ec9f-429a-ba2a-e8c28707c3cf",
    "fileSHA256":"f0d789f7-7439-4d21-a811-fff027b90076",
    "pathToFile":"C:\\home\\ba8f8032-5ecb-408e-9218-e89f14ac4084.xlsx",
    "businessGUID":"142d09de-ce38-4b3c-8119-b04f9db82199",
    "processingTime":761
  },
  ...
]
```