IE-Sprint-1-Report

Ana Almeida 102618, Daniel Carvalho 102556

May 2025

1 Introduction

In our project we have 3 main directories currently. We have the directory **terraform**, which stores our terraform project. We only have one terraform project, because every element of our project (kafka brokers, microservices, etc.) will be included through terraform modules (which are present in the modules directory, along with all scripts required). This allows us to inject dependencies on modules that depend on them automatically, for example, the RDS module for the microservices. However, since we have the problem of only having at maximum 9 EC2 instances running on AWS, some of them might need to be commented and run on another account. To use, we also need to have a **terraform.tfvars** file, that stores some variables which are used by the terraform project. This information includes the number of Kafka brokers, username, password and name of the database, docker user and 2 docker tokens, one with pull permissions passed to EC2 instances and one with create permissions, used to create docker images in local computer (in place of the tokens, password may be used, but these allow more safety since the one passed through network has only pull permissions). There exists a **terraform.tfvars.example** file which can be used as a template for the mentioned variables file.

Besides that we also have the **microservices** directory, which houses our implementations of the microservices.

Lastly, we have the **test** folder, that holds a LaaSSimulator .jar file provided by the professors, but updated to our current purchases shape (may have future changes, to include more randomness in the discount coupons) and a folder named **microservicesAPIs**, which contains multiple files, each corresponding to an automatic test for a microservice.

To run the project, an environment with Terraform and Docker installed is needed. We use a feature of docker called **buildx** that allows the host to build the docker images for several architectures at once. We use it to build for both arm64 and amd64, which is later pulled to an EC2 instance with the correct architecture, this allows anyone to build the image for the microservices and still work.

To boot the project, we have a script at root called **init_project.sh** that creates a buildx builder if not created and runs the terraform project (to automate, it is using -auto-approve). To destroy the terraform project, we have the script **destroy_project.sh** at root.

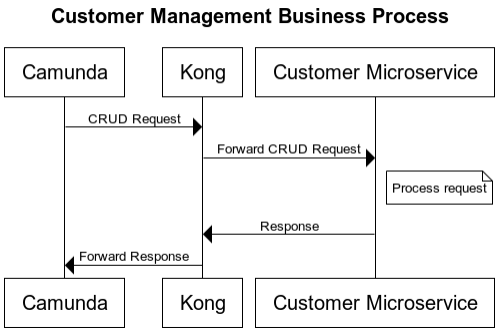
2 Information flows

In this section, we will present the information flows for each of our business processes. Since we still haven't started using Camunda and Kong and don't know their entire capabilities, the microservices we built and are present may have to change in the future so it can fit our use case (whether it is adding more endpoints, changing current ones, or changing the database tables).

To build the information flows, we used Sequence Diagrams made with this website. The bigger images may be a bit blurry due to the size.

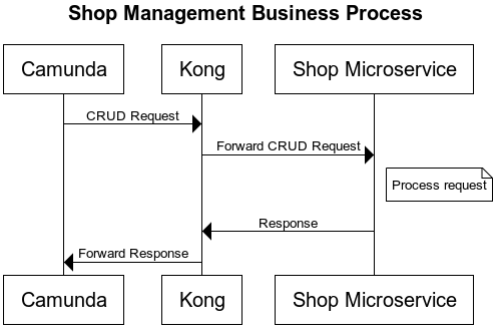
In each diagram, every time a note saying "Process request" is present, it means that the microservice is interacting with the database. We did not add it to save space.

2.1 Customer Management Business Process



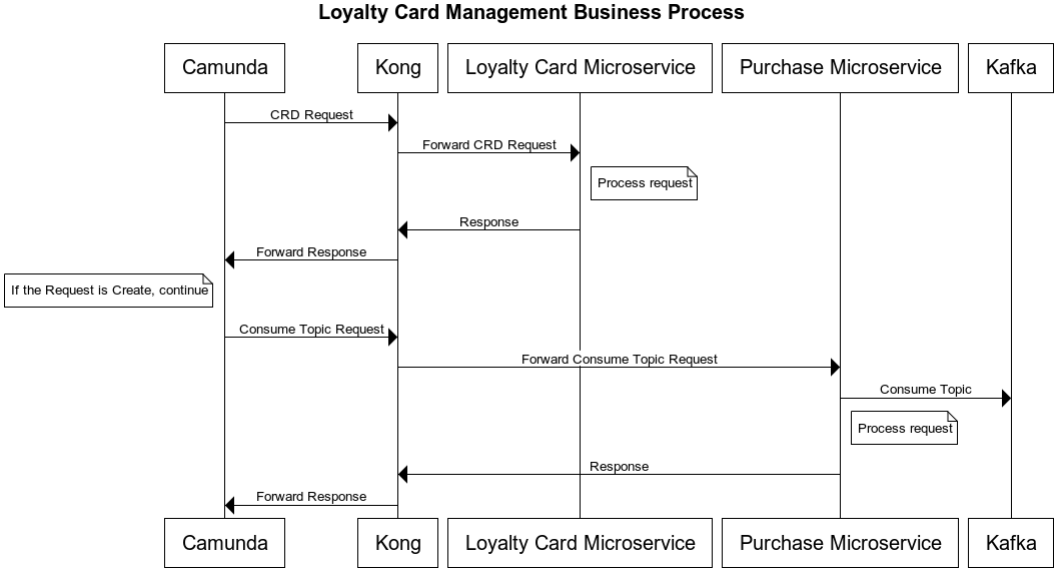
One of the simpler business processes. The business process in this case simply receives the data of a client and creates it, or receives an identifier of the customer and updated/deleted/read.

2.2 Shop Management Business Process



This business process behaves similarly to the one before, but for shops.

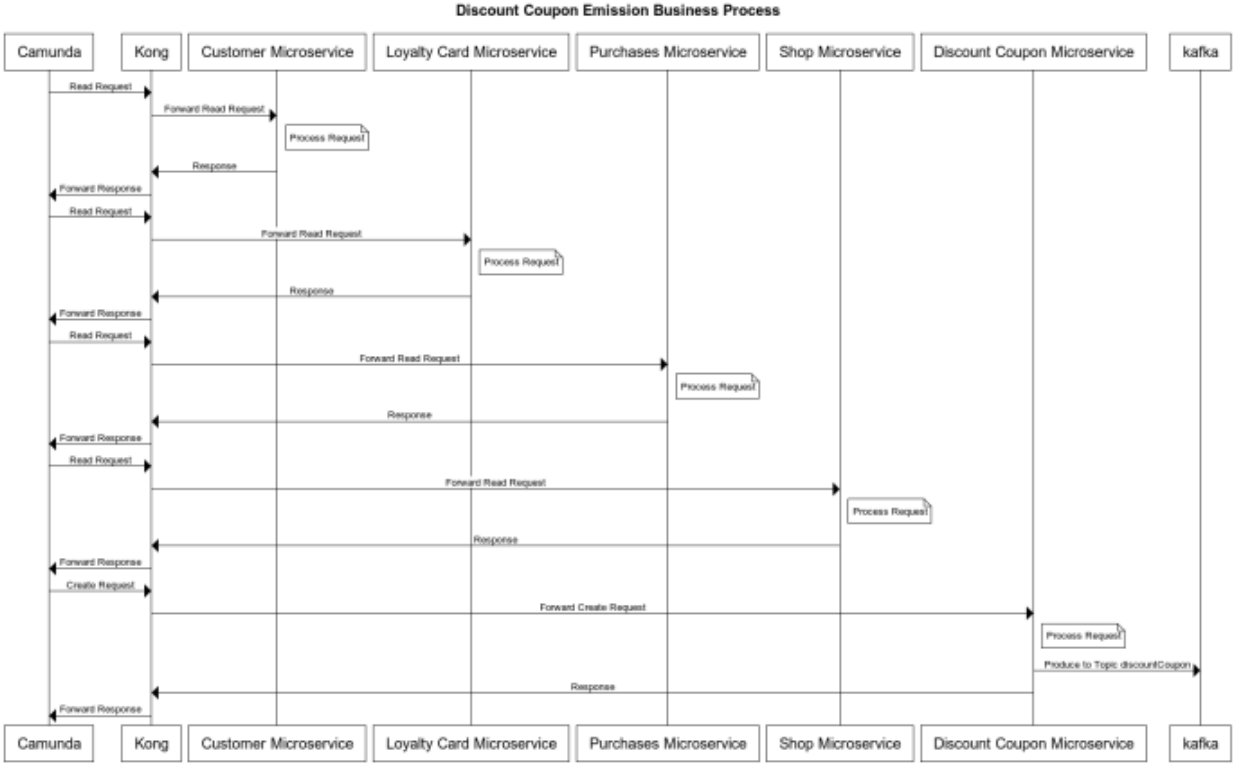
2.3 Loyalty Card Management Business Process



In this business process, we can either add or remove a association between a shop and a customer, with this being done through a loyalty card. In this process, the update is not present (the CRD in the image without U) because the data used to build the loyalty cards are the identifiers of a customer and a shop, which never change. The only way

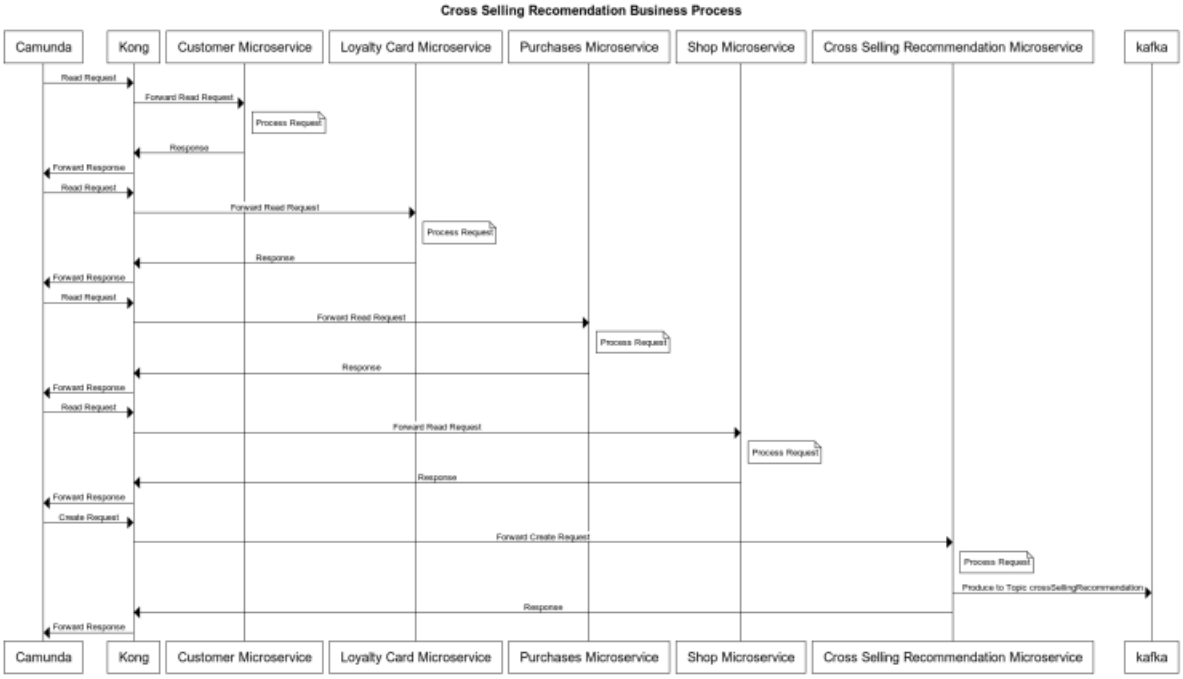
a loyalty card can be changed is by adding or removing associations between stores. Every time a new association is made, the business process makes the Purchase microservice consume a topic with the name LoyaltyCardID-ShopID where purchases will be sent, consumed and stored in the database.

2.4 Discount Coupon Emission Business Process



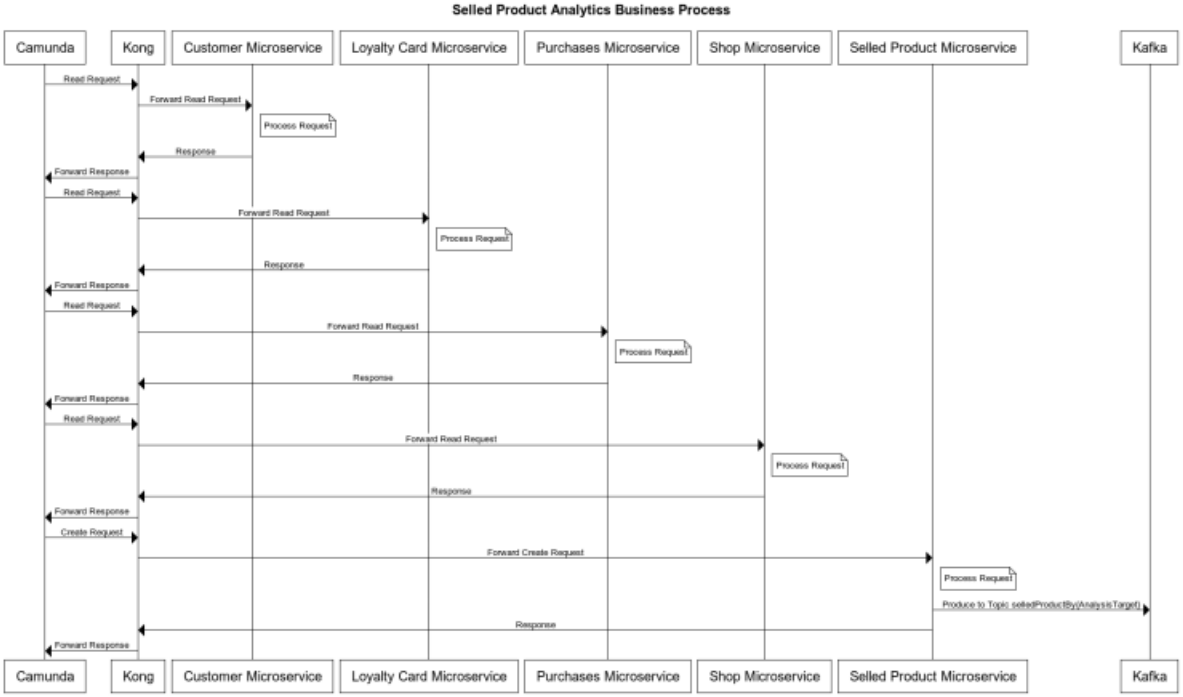
In this business process, we read the relevant data we need to make a decision about the emission of a discount coupon for a specific customer. For that we need information about the customer, its loyalty card and associations, its purchases and the information of the shops associated. With that, through rules in the business process, we can decide to emit or not the discount coupon. When the coupon is emitted, we simply send it to a Kafka topic named discountCoupon.

2.5 Cross Selling Recommendation Business Process



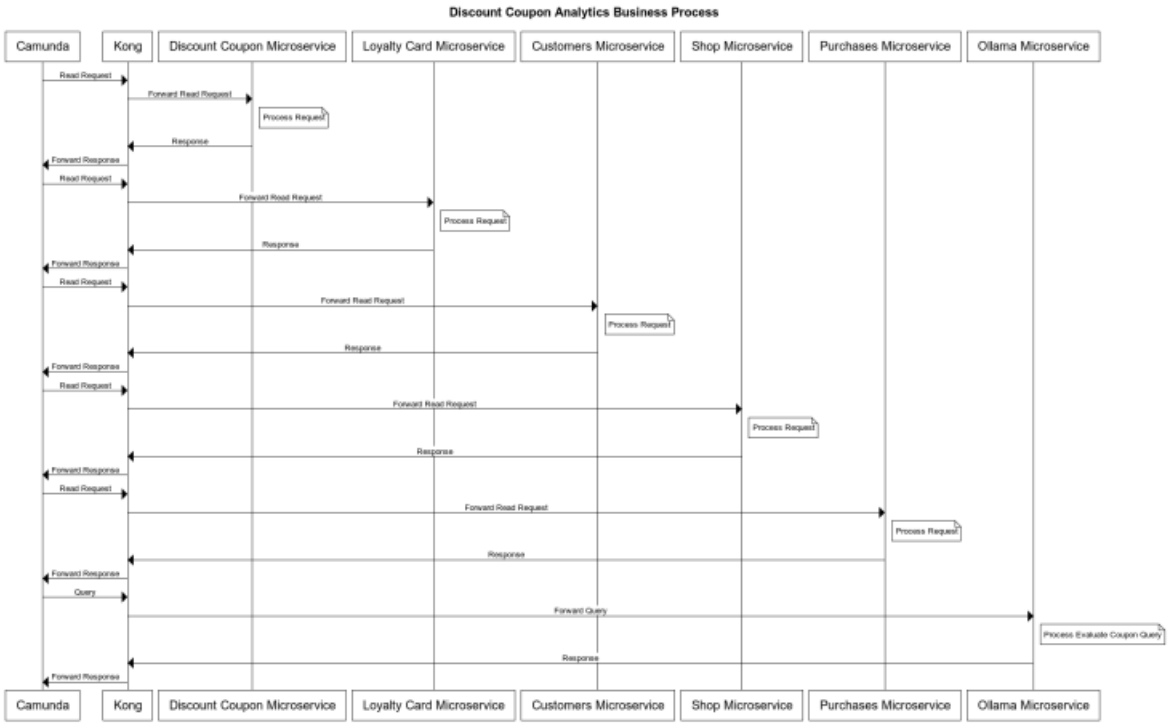
In this business process, we read the relevant data we need to make a decision about the cross-selling recommendation of a customer. For that we need information about the customer, its loyalty card and associations, its purchases and the information of the shops associated. With that, the business process applies some rules to decide or not to create cross-recomendations and creates them in the respective microservice. Those recommendations are then also produced to a topic called crossSellingRecommendations.

2.6 Selled Product Analytics Business Process



In this business process, we want to do analytics on the purchases done until the request in one dimension, which can be location, customer, discount coupons, products, etc. For this, we need to gather all the information we may need to do that analysis, which may imply using specific microservices (for example, an analysis done in terms of a single product may not need data from customers). This analysis is then sent to its microservice and stored in the database, so it may later be used to compare analysis, and produced to a kafka topic specific for that type of analysis (for example, a analysis in location would be sent to the topic SelledProductByLocation).

2.7 Discount Coupon analysis using Artificial Intelligence Business Processes



In this business process, we want to do a quantitative analysis of the purchases that used a specific discount coupon and see if the customer values the coupon by using it. For that we need to gather all the information about the coupon, loyalty card, customers, shops and purchases. After gathering some analytics, the business process then sends them to ollama to support the decision if the coupon is useful or not.

3 Implementation

In this section, we will talk more in detail about our implementation and decisions taken in each portion of the project.

3.1 Kafka

In our solution, we created a Kafka cluster with 3 brokers. Since we have such a reduced number of brokers, we used the replication factor to be the same as the number of brokers. Besides that, every topic is created with a default replication factor and the number of partitions equal to the number of brokers. This allow us to have a huge fault tolerance, durability, and improved read availability (due to having one partition leader for each topic, since it is equal to the number of brokers). However, if we started to scale the number of clusters beyond three, it would be better if the replication factor did not increase with the number of brokers, but maintained a lower value like 3, since this means each partition tolerates 3 broker failures, while also avoiding using too much storage for redundancy. Each of our brokers is running on a t2.small instance which has 8 GB storage which would be our maximum memory usage for our Kafka cluster (Ignoring space occupied by the dependencies installed and metadata of ZooKeeper and Kafka), due to replication. To start the cluster itself, after building the EC2 instances, we have a terraform null resource that executes remotely a script to run and configure ZooKeeper and Kafka. We start by

repeating the running of ZooKeepers until all members of the cluster are connected and a leader is elected, then we start the Kafka server of the leader, followed the Kafka Server of the followers.

3.2 RDS

For the database, we use the RDS service of AWS, with 20GB of storage (due to credit constraints) and MySQL. The instance class is db.t4g.micro. The database created, and its user and password is passed through the terraform.tfvars file.

3.3 Ollama

In the Ollama microservice, we are running llama3.2 model, in a t2.mediuminstance with a volume of 24GB. We have port 11434 open to receive queries for the model.

3.4 Microservices

Every microservice (excluding Ollama) follows the same process in terraform. They receive the variables they need to connect to the database and to kafka (if needed), and we execute locally the creation of docker image and the following push for arm64 and amd64 architectures. Then, every microservice, when booting, runs the same script whcih installs docker, pulls the respective image and runs it. While running, it puts the variables Quarkus needs to work correctly in environment variables, which take precedence over its respective in application.properties. This means we can put RDS and Kafka hostnames in the environment for the use of the application.

For all requests made to the microservices, we use JSON for the body.

3.4.1 Customer

We have basic CRUD endpoints for this service that update the database. We have 2 GETs, one for all Customers, the other for a specific one through ID, a Post where a customer is passed in the body, a Delete of a specific instance through ID, and an Update through ID, where the customer comes through the body. The body, when needed, should have a Long fiscal number, a String address, a String postal code and a String name.

3.4.2 Shop

We have basic CRUD endpoints for this service that update the database. We have 2 GETs, one for all Shops, the other for a specific one through ID, a Post where a shop is passed in the body, a Delete of a specific instance through ID, and an Update through ID, where the shop comes through the body. The body, when needed, should have a String address, a String postal code, and a String name.

3.4.3 Loyalty Card

We have basic CRD endpoints (update we do not have) for this service that update the database. We have 3 GETs, one for all Loyalty Cards, one for a specific one through ID and the other for a specific pair between Costumer and Shop, a Post where a loyalty card is passed in the body and 2 Deletes of a specific Loyalty Card through ID and one for a specific pair between Customer and Shop. The body, when needed, should have a Long Customer ID and a Long Shop ID.

3.4.4 Purchase

We have 3 endpoints in this microservice, 2 GETs to get all purchases or a specific one through ID and one POST endpoint where a String Topic name, where the microservice starts consuming a Kafka topic with that name to get purchases, storing them in the database.

3.4.5 Cross Selling Recommendation

For this microservice, we use 2 database tables, one to hold the mapping of a recommendation to a loyalty card id and the other between a recommendation id to shops (since we can have 2 or more associated to a single recommendation). For endpoints, we have 2 GETs, one to get all recommendations, the other to get a recommendation by id, 1 Post where we send a recommendation through the body, and we send it to a Kafka Topic if created, and

1 Delete, where we delete a recommendation based on ID. In the body, if needed, should have a Long Loyalty Card ID and a List of Longs Shops IDs.

3.4.6 Discount Coupon

In this microservice we also have 2 tables, one for mapping between Discount Coupons IDs and Shop IDs and one where a discount is associated with a Loyalty Card and other data. For endpoints, we have 2 GETs, one to get all coupons, the other to get a coupon by id, 1 Post where we send a coupon through the body, and we send it to a Kafka Topic if created, and 1 Delete, where we delete a coupon based on ID. In the body, if needed, should have a Long Loyalty Card ID and a List of Longs Shops IDs.

3.4.7 Selled Product

In this microservice we have 2 GET endpoints, one to get all analytics and the other to get a analytic through ID, 1 Post, where we receive a analytic through the body and if created send to the respective Kafka Topic, and 1 Delete for an analytic through ID. In this microservice, we use several fixed Kafka Topics since a business process that provides analytics is not used constantly, so we can manage with only one topic for each type to distinguish. In the body, we need s String type of analysis with one of the following values: LOYALTY_CARD, CUSTOMER, DISCOUNT_COUPON, SHOP, PRODUCT, POSTAL_CODE, a String type value to get the value corresponding of that analytics (for example a customer id), a String timestamp and the Double of the result.

4 Tests

To test our project to see if it is working, we need to make sure our microservices endpoints work correctly, our kafka cluster is resilient enough and is receiving messages.

4.1 Microservices

Inside the test folder, we have several shell script files. Each one first needs the microservice to be up and running, with its address added to the script, and then run. What each test does is do a series of calls to each endpoint of the service and check if it is working and responding promptly and correctly (microservice purchase get by id cannot be tested automatically due to the absence of purchases, which can only be added through the consumption of Kafka topics). After having put the address, we only need to do the command, if on the root of the project: **./tests/microservicesAPIs/{fileName}.sh**

4.2 Kafka

To verify whether Kafka is correctly transmitting messages, we can begin by ensuring that the purchases microservice is running. Then, we create a topic through its endpoint, using the format **{number}-{name}**. If the previously described purchase test has been executed, this step is already completed. Next, transfer the simulator JAR file to a running instance—ideally one of the Kafka brokers—using the following command:

```
scp -i /path/to/key.pem /path/to/yourfile/LaaSSimulator2025-0.0.1-shade.jar\
ec2-user@your-ec2-public-ip:/destination/path/
```

After connecting to the instance via SSH, run the simulator with:

```
java -jar LaaSSimulator2025-0.0.1-shade.jar \
  --broker-list kafka01.example.com:9092,kafka02.example.com:9092,kafka03.example.com:9092 \
  --throughput 100
```

Here, `--broker-list` specifies the addresses of all Kafka brokers, and `--throughput` defines the approximate number of messages to produce per minute. Optionally, the `--filter-prefix` flag can be used to restrict production to topics with a specific prefix.

By leaving the simulator running, we can evaluate Kafka's performance and throughput. Additionally, sending data to multiple topics helps verify the correct behavior of the Kafka cluster. To assess resiliency, we can deliberately shut down one of the brokers during simulation and confirm that the system continues operating as expected.