Design Document

Online Services for Continuous Evaluation
using Go and Apache Kafka

Andrea Paparella

Version 1.0
April 20, 2024

Contents

1 Introduction **3**
1.1 Purpose . . . 3
1.2 Project Overview . . . 3
1.3 Assumptions and Guidelines . . . 3

2 System Architecture **4**
2.1 High-Level Architecture Diagram . . . 4
2.2 Component Description . . . 4
2.2.1 CLI Application . . . 4
2.2.2 Microservices . . . 5
2.3 Kafka Cluster . . . 5
2.3.1 Topics . . . 6

3 Design Choices **7**
3.1 Language and Frameworks . . . 7
3.2 Kafka Configuration . . . 7
3.3 Data Serialization . . . 7

4 Implementation Details **7**
4.1 Producer Implementation . . . 8
4.2 Consumer Implementation . . . 8

5 Deployment **9**

6 Conclusion **10**
6.1 Summary . . . 10
6.2 Future Work . . . 10

1 Introduction

1.1 Purpose

This document outlines the design and implementation choices for an online service for continuous evaluation, written in Go and using the Apache Kafka distributed event store and stream-processing platform.

1.2 Project Overview

This project involves implementing online services to support university courses that utilize continuous evaluation. The application includes a frontend for handling user requests and a backend for processing these requests. There are three types of users interacting with the service: (1) students who enroll in courses, submit project solutions, and check submission statuses; (2) admins who add new courses and remove old ones; (3) professors who post projects and grade student submissions. The backend is divided into four services based on the microservices paradigm: (1) the users service manages personal data of registered students and professors; (2) the courses service oversees courses, each containing several projects; (3) the projects service handles the submission and grading of projects; (4) the registration service manages grade registration for completed courses. A course is considered completed for a student if all projects for that course are delivered and the sum of the grades is sufficient.

1.3 Assumptions and Guidelines

- Services do not share state but communicate exclusively by exchanging messages/events over Kafka topics, adopting an event-driven architecture.
- Services may crash at any time, potentially losing their state.
- State is implemented using in-memory data structures, and a fault recovery procedure has been developed to resume a valid state of the services.
- It is assumed that Kafka topics cannot be lost.
- For simplicity, there is no authentication system for the different types of users.
- The frontend is implemented through a simple command-line application.
- The sum of the grades is sufficient when is greater than 50.

2 System Architecture

2.1 High-Level Architecture Diagram

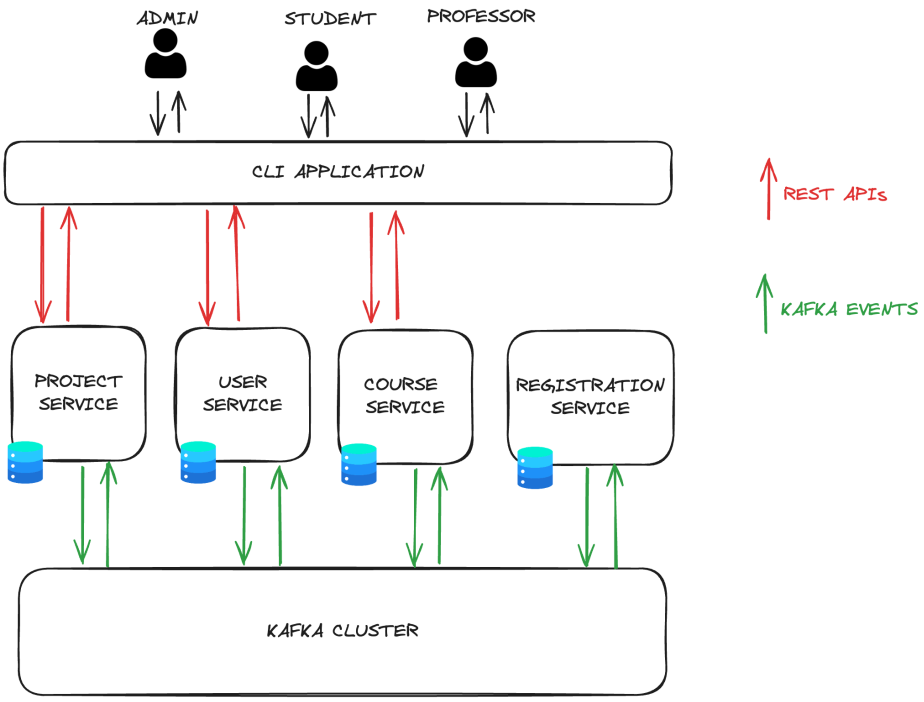


Figure 1: High-level architecture

2.2 Component Description

2.2.1 CLI Application

The CLI Application is a simple frontend app written in Go, utilized by admins, students, and professors to interact with the system. Below is a table listing the commands executable by each user role. These commands call endpoints in the backend to perform the requested actions.

User Role	Commands

Admin	`create-course --id=<id> --name=<name>`
	`get-courses`
	`delete-course --id=<id>`
	`create-student --id=<id>`
	`create-professor --id=<id>`

Student	`get-courses`
	`enroll --course-id=<id>`
	`submit-solution --course-id=<id> --project-id=<id> --submission-id=<id> --solution=<solution>`
	`get-course-projects --course-id=<id>`
	`get-project-submissions --course-id=<id> --project-id=<id>`
	`get-submission-grades --course-id=<id> --project-id=<id> --submission-id=<id>`

Professor	`get-courses`
	`create-project --id=<id> --course-id=<course-id> --name=<project-name>`
	`get-sub --course-id=<id> --project-id=<id>`
	`grade --course-id=<id> --proj-id=<id> --sub-id=<id> --grade-id=<grade-id> --grade=<grade>`

Table 1: CLI Commands for Different User Roles

2.2.2 Microservices

There are four decoupled microservices, each one running on a different webserver instance. These are:

- Project Service
- User Service
- Course Service
- Registration Service

Each service, except for the Registration, communicates with the end-user by exposing REST endpoints. Each service keeps in-memory state, thus there's no remote common database. Services keep their state up-to-date by communicating among them through Kafka events; these are handled by the Kafka broker.

Below is a table listing the REST endpoints exposed by each microservice and the end-users that will use them through the CLI app.

Microservice	Endpoint	Method	Description	End-User
Project Service	/courses/:course-id/projects/create	POST	Creates a new project for the given course	Professor
	/courses/:course-id/projects/:project-id/submit	POST	Submit solution for project of a course	Student
	/courses/:course-id/projects/:project-id/submissions/:submission-id/grade	POST	Grade a solution submission of a student	Professor
	/courses/:course-id/projects	GET	Retrieve projects of a course	Student
	/courses/:course-id/projects/:project-id/submissions	GET	Retrieve solution submission of a project	Professor
	/courses/:course-id/projects/:project-id/submissions/:submission-id/grades	GET	Retrieve grade of a solution submitted by a student	Student
User Service	/users/student/create	POST	Creates a new student	Admin
	/users/professor/create	POST	Creates a new professor	Admin
Course Service	/courses	GET	Retrieves all courses	Admin, Professor, Student
	/courses/create	POST	Creates a new course	Admin
	/courses/:course-id/delete	DELETE	Deletes a specific course	Admin
	/courses/:course-id/enroll	POST	Enrolls student in a specific course	Student
Registration Service	Not applicable	N/A	Does not expose REST endpoints	N/A

Table 2: REST endpoints exposed by microservices

2.3 Kafka Cluster

The Kafka Cluster is composed by brokers that handle and store the data exchanged with the micorservices. They manage topics on which microservice publish on or listen to for updates.

2.3.1 Topics

There are seven topics: Course, Enrollment, Grade, Project, Student, Professor and Submission. Below is a table listing the topics used by the producers and consumers, along with the details of who the consumers and producers are.

Topic	Producer	Consumer	Description
Student	User Service	Course Service	Admins create student profiles. Course Service updates its in-memory state to verify that the enrollment operation is performed on students who actually exist.
Professor	User Service	Project Service	Admins create professor profiles. Project Service updates its in-memory state to verify that the grading operation is performed by professors who actually exists.
Course	Course Service	Project, Registration Service	Admins create courses. Course Service updates its in-memory state to verify that operations like creation of projects, solution submission, and grading are carried out for existing courses. Registration Service uses course information to determine whether a student has completed a course.
Enrollment	Course Service	Project Service	Students enroll in courses. This information is propagated to the Project Service to ensure that when a student submits a solution for a project, they are actually enrolled in that project's course.
Grade	Project Service	Registration Service	Professors grade solutions submitted by students for projects. This grade is propagated to the Registration Service, which carries out the logic to determine whether the student has completed a course, i.e., has submitted solutions for all projects of the course and the sum of the grades is sufficient.
Project	Project Service	Registration Service	Professors create projects. These projects are propagated to the Registration Service, which uses them to check whether a student has submitted solutions for all projects in a course.

Table 3: Topics, Producers, and Consumers

N.B. Every microservice that produces on a topic also consumes on the same topic. This is necessary to recover the state during a crash. Indeed, when a microservice crashes, since the state is in-memory, the only way to recover their state is to consume the messages they have previously produced. Therefore, each microservice consumes on a topic that they also produce; these are omitted in the table above in the consumer column to keep the table cleaner and to highlight only the produce-consume message interactions between different microservices.

3 Design Choices

3.1 Language and Frameworks

Go has been chosen as the programming language for three main reasons: speed, ease of building servers, and advanced concurrency capabilities. Goroutines, which are cost-effective in terms of performance, act like virtual threads that can be multiplexed across actual threads. As for the Kafka library, `confluent-kafka-go` by Confluent Inc. has been selected. It offers boilerplate code for creating producers and consumers and is well integrated with the Confluent Cloud, making the deployment of the Kafka cluster easier.

3.2 Kafka Configuration

Topics are created and configured through the Confluent Cloud platform. Each topic has 6 partitions, resulting in a total of 42 partitions. This configuration allows Kafka to parallelize processing by distributing data across multiple brokers. The cluster has been deployed on AWS (us-east-2) within a single availability zone.

Regarding Data Retention Policies, the following default configurations set by Confluent Cloud are relevant:

- *cleanup.policy: delete* — Kafka will delete records that are older than the retention time or if the log exceeds the maximum size.
- *delete.retention.ms: 864000000* — This setting represents how long deleted records are retained before actual deletion, which can be important for allowing consumers to catch up.
- *retention.ms: 604800000* — This represents the duration (in milliseconds) for which Kafka will retain log messages. Set to 604800000 ms (7 days), after which old messages will be eligible for deletion based on time.
- *retention.bytes: -1* — This indicates that there is no size limit for data retention, so data will be retained indefinitely based on size.

The replication factor for every topic is 3, which represents the total number of replicas, including the leader. This enables automatic failover to these replicas when a server in the cluster fails, ensuring that messages remain available.

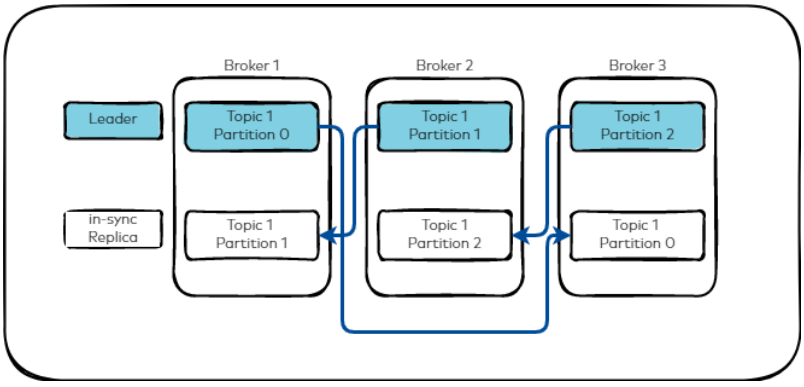


Figure 2: A generic topic with three partitions and how they might be replicated across three brokers[1].

3.3 Data Serialization

To enable communication between the CLI application, the microservices, and the Kafka brokers, messages are serialized into the JSON format.

4 Implementation Details

The code for the instantiation of producers and consumers, along with the code to handle their events, has been decoupled from the microservices. Since it's shared code, it was wiser to create a separate package importable as a

library in Go. Thus, any update to its code would be propagated to every microservice at once. This has been open-sourced and can be found on GitHub.. This acts like a wrapper to the functions provided by confluent-kafka-go, allows actions on data such as add and delete, and introduces fault tolerance. The package can be easily imported in the following way.

```
import ( "github.com/engpap/kafka-wrapper-go/pkg" )
```

Listing 1: Importing the wrapper

Follow the implementation details of the producer and consumer of this package.

4.1 Producer Implementation

Each microservice instantiates the producer within its controller. Then, a controller function is responsible for producing the message through the `ProduceMessage` function to the specified topic. To do this, the topic, action type, and data must be specified.

The Partition is set to `kafka.PartitionAny`, which allows Kafka to decide the optimal partition based on the current partition load.

The message contains headers, one of which is `action_type`. This header specifies the type of action the message represents, which can be used by consumers to handle different types of messages appropriately. For instance, a consumer might handle an `add` action differently from a `delete` action to update its in-memory state.

The data is encoded in JSON format.

```
func ProduceMessage(producer *kafka.Producer, actionType string, topic string, data any) error {
    // convert data into a byte array
    byteData, err := json.Marshal(data)
    if err != nil {
        return err
    }
    // send the message to the Kafka topic
    err = producer.Produce(&kafka.Message{
        TopicPartition: kafka.TopicPartition{Topic: &topic, Partition: kafka.PartitionAny},
        Key:            []byte("data"),
        Value:          byteData,
        Headers:        []kafka.Header{{Key: "action_type", Value: []byte(actionType)}},
    }, nil)
    return err
}
```

Listing 2: ProduceMessage Function

4.2 Consumer Implementation

Each microservice that wants to subscribe to a topic instantiates a consumer listening on that topic. When instantiating the consumer, a handler function is passed as input. This function is run as *goroutine* and has the purpose of updating the in-memory state of the microservice running as consumer. The state will be updated based on `add` or `delete` actions and will be used for any operation that the microservice needs to carry out on data generated by other microservices.

The consumer group is set to `evaluation-sys-<microservice-name>`. This means that every microservice will have a different group, which allows for unique partitions bundled to the microservice. This is necessary in order to have consumers from different micorservices consuming the events on the same topic in parallel. If a single group were created for all the microservices and there were multiple microservices subscribed to a topic, an event on that topic would be consumed only by one microservice, and not by all the subscribers. The `auto.offset.reset` configuration is set to `earliest`, meaning that new members of the group will start reading the topic from the earliest message, unless offset data is already committed for the group.

```
func consumerMessageHandler(c *kafka.Consumer, handler MessageHandler) {
  run := true
  for run {
    ev, err := c.ReadMessage(100 * time.Millisecond)
    if err != nil {
      // Errors are informational and automatically handled by the consumer
      continue
    }
    // Commit
```

```
    c.CommitMessage(ev)
    fmt.Printf("Consumed event from topic %s: key = %-10s value = %s\n",
      *ev.TopicPartition.Topic, string(ev.Key), string(ev.Value))
    // Convert JSON to map
    var data interface{}
    err = json.Unmarshal(ev.Value, &data)
    if err != nil {
      fmt.Printf("Failed to unmarshal data: %s", err)
    }
    headers := ev.Headers
    actionType := ""
    for _, header := range headers {
      if string(header.Key) == "action_type" {
        actionType = string(header.Value)
        break
      }
    }
    // check that action_type is either "add" or "delete"
    if actionType != "add" && actionType != "delete" {
      fmt.Printf("Invalid action type: %s\n", actionType)
      continue
    }
    // execute the callback function
    handler(actionType, data)
  }
}
```

Listing 3: consumerMessageHandler Function

If a consumer crashes and later restarts, it loses its in-memory state.
With the current `group.id` and `auto.offset.reset` configuration, it can recover messages it hasn't processed yet (i.e., the messages sent while it was down), starting from the last committed offset. However, the messages that were sent while it was up cannot be recovered with this configuration since they have already been committed. Thus, to achieve fault tolerance, the function `resetPartitionsOffset` has been designed to seek to the beginning of each assigned partition. This allows the service to reprocess all messages from the start, enabling the consumer to read from the beginning of the partition in order and correctly restore its state.

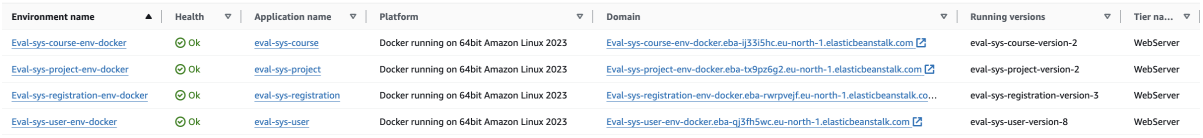
```
func resetPartitionsOffset(c *kafka.Consumer) {
  var assignedPartitions []kafka.TopicPartition
  for {
    assignedPartitions, _ = c.Assignment()
    if len(assignedPartitions) > 0 {
      break
    }
    c.Poll(100)
  }
  // for each assigned partition, seek to the beginning of the partition
  for _, partition := range assignedPartitions {
    fmt.Printf("Topic: %s, Partition: %d\n", *partition.Topic, partition.Partition)
    c.Assign([]kafka.TopicPartition{{Topic: partition.Topic, Partition: partition.Partition,
    Offset: kafka.OffsetBeginning}})
  }
}
```

Listing 4: resetPartitionOffest Function

5 Deployment

The four microservices have been containerized using Docker and deployed on Amazon Web Services (AWS). Each microservice has its own Dockerfile and runs as a separate Elastic Beanstalk application, each within its own EC2 instance.

The Kafka Cluster, the collection of servers that work together to handle data streams for the Kafka system, has been deployed through Confluent Cloud.



Environment name	Health	Application name	Platform	Domain	Running versions	Tier na...
Eval-sys-course-env-docker	Ok	eval-sys-course	Docker running on 64bit Amazon Linux 2023	Eval-sys-course-env-docker.eba-ij33i5hc.eu-north-1.elasticbeanstalk.com	eval-sys-course-version-2	WebServer
Eval-sys-project-env-docker	Ok	eval-sys-project	Docker running on 64bit Amazon Linux 2023	Eval-sys-project-env-docker.eba-tx9pz6g2.eu-north-1.elasticbeanstalk.com	eval-sys-project-version-2	WebServer
Eval-sys-registration-env-docker	Ok	eval-sys-registration	Docker running on 64bit Amazon Linux 2023	Eval-sys-registration-env-docker.eba-rwrpvejf.eu-north-1.elasticbeanstalk.co...	eval-sys-registration-version-3	WebServer
Eval-sys-user-env-docker	Ok	eval-sys-user	Docker running on 64bit Amazon Linux 2023	Eval-sys-user-env-docker.eba-qj3fh5wc.eu-north-1.elasticbeanstalk.com	eval-sys-user-version-8	WebServer

Figure 3: AWS Environments

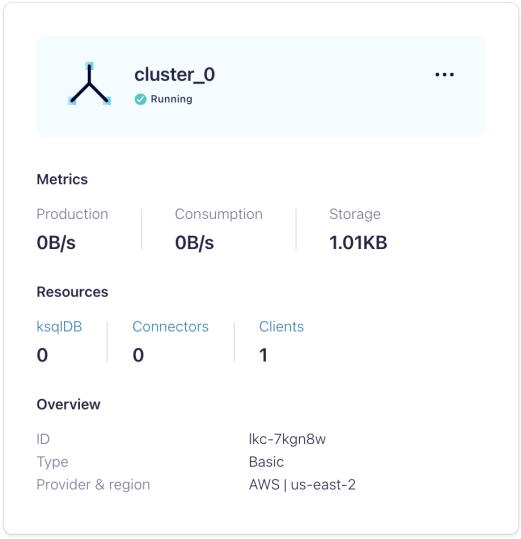


Figure 4: Confluent Cloud Kafka Cluster

6 Conclusion

6.1 Summary

The Online Services for Continuous Evaluation system was developed using Go and Apache Kafka. The frontend is a simple CLI application, while the backend system has been broken down into four decoupled microservices, each running on a separate web server deployed on AWS. The microservices produce and consume events on predefined topics that are managed by Kafka brokers deployed on Confluent Cloud.

6.2 Future Work

As a follow-up, an authentication system and a stricter authorization policy for granting access to the specific functions of the system should be developed, as currently, any user can potentially act like an admin.

References

[1] Confluent Documentation. "Kafka Replication and Committed". In: (2024). URL: https://docs.confluent.io/kafka/design/replication.html#:~:text=Kafka%20guarantees%20that%20a%20committed,the%20message%20to%20their%20log..