MOOCS RECOMMENDER BASED ON LEARNING STYLES

SERVICE ORCHESTRATOR FOR PARALLEL CLASSIFICATION

Software Requirement Specification
Project ID: 19-089

Liyanage A.Y.K.

IT 16 0327 98

Bachelor of Science Special (Honors) in Information Technology
Specializing in Software Engineering

Department of Software Engineering

Sri Lanka Institute of Information Technology
Sri Lanka

Date of Submission: 2019-05-13

MOOCS RECOMMENDER BASED ON LEARNING STYLES

SERVICE ORCHESTRATOR FOR PARALLEL CLASSIFICATION

Project ID: 19-089

Liyanage A.Y.K.

IT 16 0327 98

Supervisor: Mr. Nuwan Kodagoda

Co-Supervisor: Ms. Kushnara Suriyawansa

Date of Submission: 2019-05-13

Declaration

I hereby declare that the project work entitled “MOOCs Recommender Based on Learning Styles” submitted to the Sri Lanka Institute of Information Technology, is a record of original work done by our group under the guidance of Mr. Nuwan Kodagoda (Supervisor) and Ms. Kushnara Suriyawansa (Co- Supervisor), and this project work is submitted in the fulfillment for the award of the Bachelor of Science (Special Honors) in Information technology Specialization in Software Engineering. The results embodied in this report have not been submitted to any other University or Institute for the award of any degree or diploma. The diagrams, research results and all other documented components were developed by us and we have cited clearly any references we have made.

Name	ID	Signature
Liyanage A.Y.K.	IT16032798	

Supervisor: Mr. Nuwan Kodagoda Signature:

Co-supervisor: Ms. Kushnara Suriyawansa Signature:

Table of Contents

List of figures

List of tables

1 Introduction

1.1 Purpose

This Software Requirement Specification (SRS) document is intended to precisely highlight the expected outcome of the "Service Orchestrator for Parallel Video Classification" component along with the process that will facilitate the fulfillment of the said outcomes and requirements. Furthermore, this document contains information such as functional and non-functional requirements of this components, along with constraints that the component's implementation must adhere to, and interfaces that the implementation will expose so that other components are able to communicate. Given that this component has no direct user interaction, the document is intended to be comprehended by a set of stakeholders that consists of developer(s) and the customer. Hence, the requirements and functionalities stated in this document are listed out and described in such a way that the developer can identify the boundaries of said functionalities and what must be done to implement them.

1.2 Scope

The scope of this component extends across parallel computing and containerization. Containerization involves packaging the video classifier into a Docker container which will be used to spin up multiple instances which will facilitate the first phase of parallel video classification.

Although, this enables parallel processing in hindsight, to make sense out of the outputs produced by these parallel processes, we need a proper architecture to support multiple, independent data streams and their processed outputs which can be used to arrive at a single conclusion.

This architecture can be established by using a message queue to pass work to multiple Docker instances and then to analyze their outputs by a central component which will produce a single conclusion for a given MOOC, classifying it under one label.

1.3 Definitions, Acronyms and Abbreviations

MOOC	Massive Open Online Course
SRS	Software Requirement Specification
EC2	Elastic Computing 2
Containerized Classifier	A Python classifier running within a Docker container

1.4 Overview

From this point onwards, this document has 2 sections. The first upcoming section describes the overall component by elaborating on requirements, software interfaces, constraints and approaches of implementation in a high-level perspective.

The next section describes above areas more informatively.

2 Overall Descriptions

2.1 Introduction

This component can be divided into two high-level functionalities where one functionality is responsible for running the containerized video classifiers while the other functionality analyzes the outputs of multiple containerized video classifiers in order to reach a single conclusion about a video file. The basic idea here is that the same video classifier is made to run on multiple threads and are treated as service workers that do not have any context as to what the MOOC is.

- Containerized Service Workers
  Containerizing is a popular trend among micro services which allows high scalability and portability across platforms by isolating an application and its dependencies into a single execution unit. By doing so, we are able to deploy such containerized applications at a whim without setting up execution environments and dependencies every time we need to scale. With regards to running video classifiers parallelly, a Python & TensorFlow based video classifier will be packaged along with its many dependencies into a Docker image. By running multiple instances of this Docker image effectively provides us multiple service workers ready to classify a part of a video.

- Service Orchestrator and Analyzer
  While we have a set of service workers to classify videos parallelly, it is equally important to divide a single video file into multiple small chunks which will provide multiple smaller video files/chunks that each service worker can work on. Ideally, a video chunk will be classified by a single service worker. By doing so, we can achieve an efficient, fast and parallelized workflow. To achieve this, a service orchestrator is paramount. Once each video chunk is classified, we will be left with multiple labels/classifications. A label or a classification simply implies the prediction that the video classifier produced for the video chunk it analyzed. This way, if we divide a video into 10 chunks, we will be left with 10 predictions; thus, it is the analyzers responsibility to go through the predictions given for video chunks that belong to the same parent video file and arrive at a conclusion about the overall video.

- Message Queue Integration
  In order to achieve the persistent communication between the service works and the orchestrator & analyzer, a message queue can be used. This is a popular approach for running highly de-coupled and scalable systems. Since the orchestrator can simply push messages that contain information about video chunks, its operation does not depend on the service workers and vice versa. Also, we can scale up or scale down the service works independently since the crucial data is preserved in the message queue until they are consumed.

2.2 Product Perspective

2.2.1 System Interfaces

- Docker Daemon
- Docker Network
- Host Network
- Disk Mount

2.2.2 User Interfaces

This component contains a set of self-managed backend tasks, thus offers no user interfaces. Its functionality is also only exposed to other components of the system and not to the end user.

2.2.3 Hardware Interfaces

- Linux Based Server
  A Linux based (preferably CentOS 7) server is key for running Docker. Since this component is fully a backend process, a CLI based Linux server can give maximum head room for processing.

2.2.4 Software Interfaces

- Python 3.7
- Docker
- RabbitMQ

2.2.5 Communication Interfaces

- A 100Mbit or higher network interface to download video files fast enough such that it will not end up being a bottleneck.

2.2.6 Memory Constraints

- 8GB per server at minimum to efficiently run multiple instances of a Docker image.

2.2.7 Operations

- The functionalities are fully automated such that they can run without any user intervention.

2.2.8 Site Adaption Requirements

- Given that this component is confined to a server as a background task, no site adaption is required.

2.3 Product Functions

This component functionality is fully confined to the backend servers. The main functionality offered by this component is logically marking MOOC videos into small chunks of videos and feed them to containerized machine-learning classifiers to independently classify the said video chunks. Finally, the predictions given to video chunks of the same video file will be analyzed together to provide a single classification for the entire video.

2.3.1 Overall Design

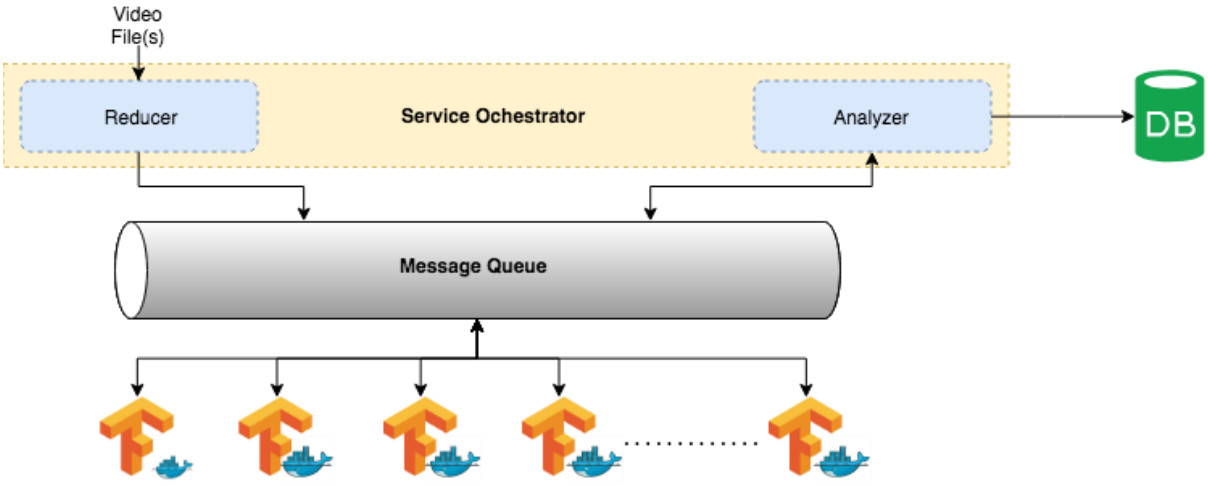

Figure 2.1: High Level Design

The figure 2.1 shows how the containerized classifiers are listening to a message queue in order to receive details about video chunks that have to be classified. At the same time, orchestrator is sending messages to the message queue that contains details about the video chunks such as start and end times, parent video file's name, parent MOOC course's name. At the same time, the analyzer is waiting till containerized classifiers send messages to the message queue that contains details such as its prediction/classification on the video chunk along with all the original details that the orchestrator sent.

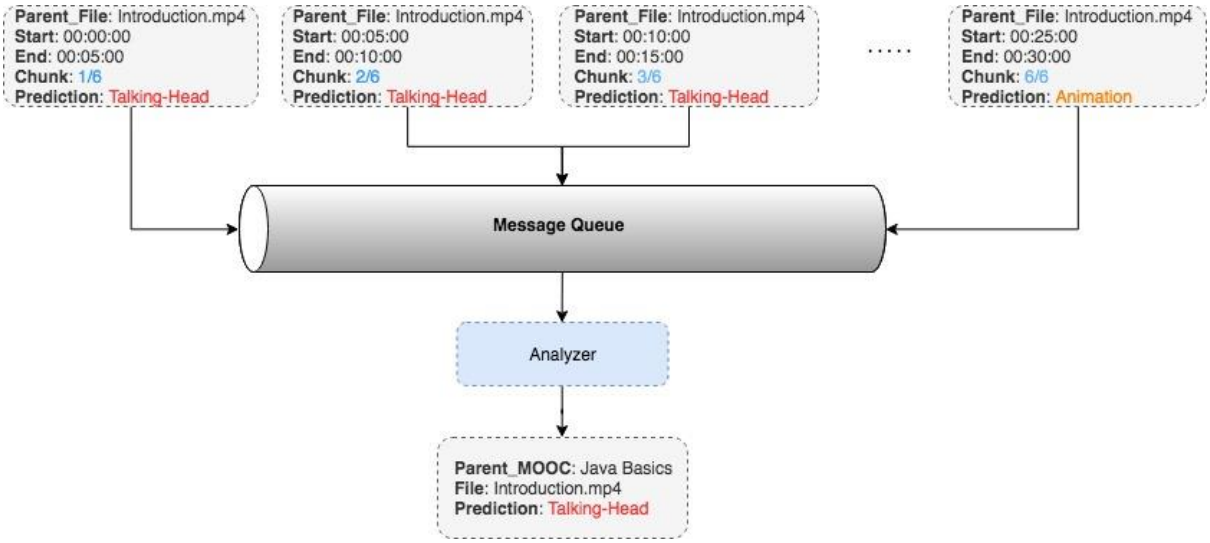

Figure 2.2: Message Queue

Once the analyzer has intercepted all the messages that belong to a single video file, it will provide a final classification for the entire video file as shown in figure 2.2. To arrive at the final prediction, the percentage of all different predictions that the video chunks will be weighted and the prediction with highest percentage will be considered as the final prediction.

2.3.2 Use Case Scenarios

Table 2-1: User Case -- Logically mark video file into chunks

User Case Name	Logically mark video file into chunks
Pre-Condition	A MOOC course with compatible video files has been downloaded
Post-Condition	Message queue contains messages where each message represents a logical video chunk.
Actor	Orchestrator (Backend System)
Main Success Scenario(s)	1. Analyze the video's length and select a suitable video chunk duration. 2. Create messages that contains the MOOC name, video file name, start and end times that match the chunk duration chosen in previous so that all the chunks cover the whole video. 3. Push the messages to message queue.

Table 2-2: User Case -- Classify a logical video chunk

User Case Name	Classify a logical video chunk
Pre-Condition	A message queue crowded with messages that represent logical video chunks.
Post-Condition	A message that contains the prediction for the logical video chunk should be sent to the message queue.
Actor	Containerized Classifier (Backend System)
Main Success Scenario(s)	1. Retrieve a message that represent a logical video chunk from message queue. 2. Extract the part of video that the start and end time indicates from the actual video file in the shared storage device. 3. Run the extracted video through the classifier. 4. Push the prediction of the video style of the extracted video to the message queue.

Table 2-3: User Case -- Provide final prediction for a video file

User Case Name	Provide final prediction for a video file
Pre-Condition	A logical video chunks that makes up the video file have been classified and the prediction data is available in the message queue.
Post-Condition	Prediction for the video file must be stored in the database.
Actor	Analyzer (Backend System)
Main Success Scenario(s)	1. Retrieve messages that represent the predictions given for all logical video chunks of an individual video file. 2. Calculate the percentage of unique predictions. 3. Store the unique prediction with highest percentage in the database.

2.4 User Characteristics

The orchestrator deals with video files of different formats and sizes while the analyzer mainly deals with JSON data. Containerized classifier contains a TensorFlow based classifier that is provided by another component while there's a service worker within the container that communicate with the message queue, extract parts of videos and control the execution of the classifier. For the message queue "RabbitMQ" will be used. Since the classifier is written in Python, other modules of this component will also be written in Python.

Therefore, the developers should have an understanding about Python, messages queues, and JSON data.

2.5 Constraints

This component is the most resource consuming aspect of the overall system. Also, given that this component is a core part of the system, tight constraints are introduced to maintain stability and efficiency.

- The UI must be validated using HCI techniques and should prove to be user friendly.
- The UI must at least have 4GB of memory to smoothly run in the client side.
- The system must be cloud-ready and should ideally be implemented in AWS Cloud.
- Each server node must contain at least 8GB of RAM.
- Each server node must be connected to a shared storage device.
- Each server node must run CentOS 7 and if the system is implemented in AWS, the EC2 machine image should be Amazon Linux 2 which is based on CentOS 7.
- Python 3.7 must be used for this component.
- Containerization has to be done by using Docker.
- Messages must be JSON data.
- A NoSQL database must be used, preferably MongoDB.
- Video files must be deleted after classification of each video file.
- Only 1 message queue must be maintained.
- Each server node should have access to internet and the message queue.
- Each server node should have at least 100Mbit network and internet connection.

2.6 Assumptions and Dependencies

- TensorFlow based video classifier.

2.7 Apportioning of Requirements

2.7.1 Essentials Requirements

- Classify video files parallelly by splitting them into logical video chunks of smaller sizes.
- At the end, arrive at a single prediction for a given video file.
- Increase or decrease number of containerized classifier instances based on number of video files remaining.

2.7.2 Desirable Requirements

- Download lower resolution videos given that the resolution is still good enough for the video classifier to run effectively.

3 Specific Requirements

3.1 External Interface Requirements

3.1.1 User Interfaces

This component is fully confined to the backend and does not expose any functionality to the user directly. Therefore, this component does not contain any user interface.

3.1.2 Hardware Interfaces

- 100Mbit network interface.
- SSD based EBS volume of 50GB or more.
- At least one EC2 instance with below specifications.
  - 8GB RAM
  - 8GB internal storage for the operating system
  - 4 or more vCPUs
- Another EC2 instance for running the DB with below specifications.
  - 16GB RAM
  - 25GB SSD based storage
  - 4 or more vCPUs

3.1.3 Software Interfaces

- Amazon Linux 2 operating system
- Python 3.7
- Docker
- RabbitMQ
- TensorFlow
- MongoDB

3.1.4 Communication Interfaces

- Since server sided infrastructure is created in AWS Cloud, all communication interfaces are managed by AWS and created when the servers are created.

3.2 Functions

This section depicts the functional requirements of the component from the perspective of developers.

3.2.1 Orchestrator

3.2.2 The orchestrator should mark a single video file into logical video chunks.

Table 3-1: Functions of Orchestrator

Description	The orchestrator should logical break down a large video file into small sections so that multiple service-workers can work on the large video file at different sections of it.
Sequence of Operations	1. Orchestrator goes through a video file and calculate its length 2. Orchestrator then calculates the start and end times of each logical video chunk in order to divide the large video file into small parts 3. Orchestrator then pushes messages containing details of each logical video chunk to the message queue.
Validations	Video file format should be supported by the classifier within the service worker.
Input	Video file

Output	Messages (in JSON format), each containing details about a logical video chunk.
	{
	“ParentFile”: “.....”,
	“Start”: “…..”,
	“End”: “…..”,
	“Chunk”: “…..”
	}

Exception/Error Handling	If the video file format is unsupported, log the error and notify the system administrator via an email.

3.2.3 Service Worker

3.2.4 Service worker should intercept a message in message queue.

Table 3-2: Functions of Service Worker in Message Queue

Description	The service worker should subscribe to the message queue and intercept a message in the message queue.
Sequence of Operations	1. Service worker subscribe to the message queue 2. Callback function within the service worker is called when a message is available 3. Message is read, and is converted to a Python dictionary for internal use (Even though the message is read, it is not acknowledged until the classification is successfully finished)
Validations	1. Message should contain a valid JSON string. 2. Message should contain all the necessary data.
Input	RabbitMQ Message
Output	A Python dictionary representing the JSON data within the message. It should contain following keys. [ParentFile, Start, End, Chunk]
Exception/Error Handling	If the message does not contain a valid JSON string, push the message back to message queue with an error message attached to it.

3.2.5 Service worker should trigger the classifier and retrieve its prediction.

Table 3-3: Prediction Function of Service Worker

Description	The service worker should call the classifier function after intercepting a message from the message queue. In return, the classifier will analyze the logical video chunk represented in the message and provide a prediction/classification.
Sequence of Operations	1. Service worker invokes the classifier with the Python dictionary created in *function 3.2.3.1* as an input 2. Classifier returns the prediction back to the service worker 3. Service worker pushes prediction details to the message queue 4. Service worker should acknowledge the initial message that was used to create the Python dictionary
Input	Python dictionary containing following keys. [ParentFile, Start, End, Chunk]
Output	A message containing following JSON data as a string.

	{
	"ParentFile": ".....",
	"Start": "…..",
	"End": "…..",
	"Chunk": "…..",
	"Prediction": "...."
	}

Exception/Error Handling	If the classifier crashes without providing a prediction, service worker should not acknowledge the initial message.

3.2.6 Analyzer

3.2.7 Analyzer should predict the video style of the original video file.

Table 3-4: Functions of Analyzer

Description	The orchestrator should logical break down a large video file into small sections so that multiple service-workers can work on the large video file at different sections of it.
Sequence of Operations	4. Orchestrator goes through a video file and calculate its length 5. Orchestrator then calculates the start and end times of each logical video chunk in order to divide the large video file into small parts 6. Orchestrator then pushes messages containing details of each logical video chunk to the message queue.
Validations	Video file format should be supported by the classifier within the service worker.
Input	Video file

Output	Messages (in JSON format), each containing details about a logical video chunk.
	{
	"ParentFile": ".....",
	"Start": "…..",
	"End": "…..",
	"Chunk": "….."
	}

Exception/Error Handling	If the video file format is unsupported, log the error and notify the system administrator via an email.

3.3 Performance Requirements

Given that the main reason of implementing this component is to make video classification efficient and fast, it is paramount that all the functionalities related to this component achieve a high level of performance in multiple aspects while minimizing bottlenecks as much as possible. Hence, below are the performance requirements which are expected from this component.

3.3.1 Network Performance Requirements.

- Network latency must be kept to a minimum to ensure messages are exchanged among the orchestrator, analyzer and the containerized classifier instances via the message queue optimally.
- Messages must only contain the most crucial information to keep the size of a message down to a minimum.

3.3.2 Disk Performance Requirements.

- Video files should be read off of a solid-state shared storage device to ensure no bottleneck is created when transferring files between the classifier and the storage device.
- Each MOOC course must be classified only once.

3.3.3 Service-worker Performance Requirements.

- No two containerized classifier instances should classify the same logical video chunk.

3.4 Logical Database Requirements

3.4.1 Data Format

All data should be stored as JSON documents. MongoDB must be used in a cluster configuration to facilitate this.

3.4.2 Data Criteria

All crucial details related to a MOOC must be stored in a separate database.
Therefore, following aspects are considered crucial.

- MOOC Name
  Name of the MOOC/Course
- MOOC URL
  Direct URL to view the course at the course providers' site.
- Video Style Classification
  A prediction as to what video production style the MOOC belongs to overall.

3.4.3 Indexing

A single JSON document represent a single unique MOOC. Therefore, all documents should be indexed by the MOOC URL in order to uniquely identify courses with similar names provided by different course providers.

3.4.4 Data Accessibility

No user has direct access to data. Only the MOOC Recommender component should have access to data.

3.4.5 Data Availability

To ensure that classification data is not lost, MongoDB must be run in a cluster configuration with 1 master node and 2 slave nodes. This is done to prevent the need to re-classify MOOCs in case of a data loss.

3.5 Design Constraints

3.5.1 Standards Compliance

The overall design must be compliant with containerization of applications. Furthermore, the Docker-images must be compliant with Docker standards and best practices to ensure high performance and security.

3.5.2 Data Constraints

- JSON must be used to represent data to maintain interoperability among components.

3.6 Software System Attributes

- This section overlays the metrics that can be observed in the functional system while operating.

3.6.1 Reliability

The ability of the system to keep classifying videos while keeping failures to a bare minimum. Given that the overall architecture is based on distributed computing and all sub-components can run independent of each other, a high level of reliability can be achieved.

- A message queue is used to convey tasks to containerized classifiers and every message will remain in the message queue until a response has been given by a containerized classifier instance. This ensures that another instance can take up the work if a given instance crashes during the classification process.
- By using a suitable auto-scaling mechanism, the number of containerized classifier instances can be kept at a desire number. This ensures that a new instance is started as soon as one crashes in order to keep the processing power balanced.

3.6.2 Security

Since we are dealing with copyrighted content it is important that the information about MOOC courses as well as their video files remain securely without being exposed to outside.

- All EC2 instances will be created within the same Virtual Private Cloud (VPC) provided by AWS. This provides the ability to route all communications across servers internally instead of over the internet.

3.6.3 Maintainability

Give the importance of this component along with its performance sensitive nature, the system, its architecture, codebase and connected infrastructure must be maintainable to ensure future improvements as well as easy troubleshooting and seamless execution.

- Develop the orchestrator, analyzer and containerized classifier as de-coupled and independent modules so that each of them can be maintained without affecting the functionality of another.
- Create and maintain cloud infrastructure via code using CloudFormation or Terraform to maintain consistency and transparency.
- Enforce Python coding standards in the code base.

3.6.4 Scalability

Since the component is aimed at achieving high levels of performance and efficiency, the containerized classifier instances must scale up or down based on the number of video files that have to be classified. This allows the system to use most of its resources when needed while additional server nodes can be shut down when the demand is low.

- Use EC2 auto scaling policies to scale the number of server nodes (EC2 instances) based on resource usage.
- Use container scaling policies to utilize most of the processing power in a single server node by running multiple containerized classifiers parallelly.

3.6.5 Availability

In order to keep an updated list of MOOC recommendations, it is important that the system keeps classifying new MOOC videos as they come.

- By using EC2 auto scaling policies, even if a server node crashes, a new one will be spun up right away to make the system highly available.
- By utilizing availability zones provided by AWS, server nodes can be scattered across different geographical locations to ensure continuous availability.

3.7 Organizing Specific Requirements

3.7.1 System Mode

The component should be designed to run by itself without any manual intervention from time to time. All sub-components will be invoked by the orchestrator and the orchestrator itself runs automatically based on the availability of new, unclassified MOOC videos. Therefore, the whole component can be considered fully automated.

4 References

5 Appendix