Design and Evaluation of a Generic Orchestrator for execution of an AI Pipeline

Tejas Morbagal Harish

Matriculation number: 3200842

May 2021

A thesis submitted in partial fulfillment for the
degree of Master of Science

Institute of Computer Science

Supervisors:
M.Sc. Martin Welß, Fraunhofer IAIS

Examiners:
Prof. Dr. Jens Lehmann, University of Bonn
Dr. Micheal Stadtschnitzer, Fraunhofer IAIS

INSTITUT FÜR INFORMATIK
RHEINISCHE FRIEDRICH-WILHELMS-UNIVERSITÄT BONN

Declaration of Authorship

I, Tejas Morbagal Harish, declare that this thesis titled, 'Design and Evaluation of a Generic Orchestrator for Orchestration of an AI pipeline' has been written independently and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- Where none other than the specified sources and aids were used.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: 26.05.2021

Acknowledgements

I wish to thank everyone, who has been a part of my journey. First and foremost, I would like to thank my thesis supervisor, M.Sc. Martin Welß, for his advice, patience and willingness to help me with any thesis related issues. Also, I'm grateful to Prof. Dr. Jens Lehmann and Dr. Micheal Stadtschnitzer for being my examiners and for the encouragement in my research.

I am thankful to Dr.-Ing. Joachim Köhler, head of the department, NetMedia Department, Fraunhofer IAIS for providing me an opportunity to work at Fraunhofer IAIS for my thesis. Furthermore, I would like to thank Fraunhofer IAIS for helping me providing access to the infrastructure.

I would like to thank my professors at the Informatik department, the teaching and non-teaching staff of the University of Bonn.

I'm grateful to the Fraunhofer IAIS, for helping me with the resources required to implement my thesis. I would also like to thank the AI4EU development team for their cooperation and useful inputs.

I owe my special thanks to my friends especially Ms.Meghana Jayadevan who believed in me and being part of my journey.

Last but not least, I would like to take this moment to thank my family especially my mother, brother for having faith in me and supporting me all the way.

RHEINISCHE FRIEDRICH-WILHELMS-UNIVERSITÄT BONN

Abstract

Institute of Computer Science

Master of Science

by Tejas Morbagal Harish

The IT industry has seen significant adoption of microservice architecture in recent years. Microservices interact with each other using different protocols. SOAP was integrated with an ever-growing set of protocols called WS-I to promote interoperability. This large set of protocols with SOAP quickly became unpopular as it caused a considerable effort on the user side to make it interoperable. JSON replaced the XML as the serialization technology, and REST replaced SOAP as the communication protocol between web services. REST provided more flexibility, was more efficient and faster as compared to SOAP. However, it uses third-party tools such as swagger to auto-generate code for API calls in various languages, lacks support for streaming, needs semantic versioning whenever the API contract changes. This lead to its limited interoperability. As a variant of RPC architecture, Google created gRPC as a new communication protocol that solved most of the SOAP and REST issues. gRPC uses protocol buffers, usually called protobuf, for the serialization of data. It made it possible to clearly define the clean interfaces between services and supported in-built code generation for various programming languages. Automatic code generation makes it possible to use stubs and skeleton to call the services implemented on the server-side.

In this thesis work, we use an open-source framework called Acumos, designed to make it easy to build, share, and deploy AI apps. Acumos has a design studio where users can compose an AI pipeline. Acumos has different orchestrators for different programming languages. However, there is no functionality to execute a generic pipeline that is implemented in multiple programming languages. Using gRPC communication, docker as a containerization tool, Kubernetes as a deployment environment, We propose to design a generic orchestrator capable of running any generic pipeline composed according to AI4EU container specification. This design of a generic orchestrator capable of executing pipelines, it has never seen before, is cutting edge and goes beyond state of the art.

Contents

Chapter 1

Introduction

Microservices have been getting increasingly popular in recent years and many companies are migrating monolithic applications to microservice architecture. A monolithic application involves programming of all of its services and features within a single, indivisible code-base that becomes increasingly difficult to maintain and scale over time. Microservices-based applications solve traditional, monolithic application constraints by breaking the application into multiple independent components called microservices. These individual microservices then use APIs to interact with each other. The APIs allow different microservices developed in various languages and run on different platforms to connect and communicate efficiently. This chapter discusses different APIs for microservices and the reason for using choosing gRPC over other APIs and the docker for containerization in AI4EU Experiments.

1.1 SOAP, WS-I and why it failed

SOAP[1] stands for Simple Object Access Protocol. It is a XML-based protocol for exchanging information in a decentralized, distributed environment.

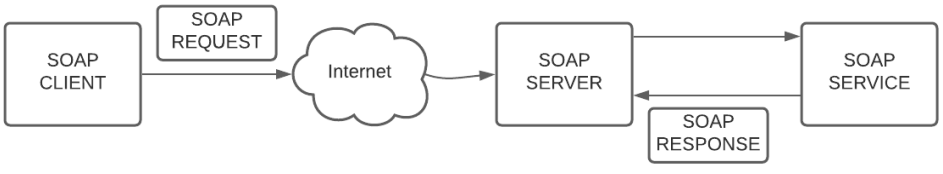

Figure 1.1: Client Server Architecture of SOAP Protocol

The SOAP specification describes a standard, XML-based way to encode requests and responses, including:

- Requests to invoke a method on a service, including in parameters
- Responses from a service method, including return values and output parameters
- Errors from a service

SOAP describes the structure and data types of message payloads by using W3C XML Schema standard issued by the World Wide Web Consortium (W3C). SOAP is a transport-agnostic messaging system; SOAP requests and responses travel using HTTP, HTTPS, or other transport mechanism. [2]

SOAP has the following features:

1. Protocol independence
2. Language independence
3. Platform and operating system independence

Figure 1.1 illustrates the components in the SOAP architecture. In general, a SOAP service remote procedure call (RPC) request/response sequence includes the following steps: [2]

1. A SOAP client formulates a request for a service. This involves creating a conforming XML document, either explicitly or using SOAP client API
2. A SOAP client sends the XML document to a SOAP server. This SOAP request is posted using HTTP or HTTPS to a SOAP Request handler running as a servlet on a Web server.
3. The Web server receives the SOAP message, an XML document, using the SOAP Request handler Servlet. The server then dispatches the message as a service invocation to an appropriate server-side application providing the requested service.
4. A response from the service is returned to the SOAP Request Handler Servlet and then to the caller using the standard SOAP XML payload format.

In general, the SOAP protocol only describes a wrapper around a message, i.e., where to place headers, message payload, how to indicate the request's intention (or the "action"). These basic requirements enable different platforms to process SOAP messages. However, the extracted header and body elements still need to be understood and converted into platform-specific runtime types. WSDL does this function for SOAP protocol. WSDL describes the URL where a service is reachable, the supported operations

or actions at that location, and the format for messages and related type definitions described by XSD Schema. SOAP protocol frames the message, and WSDL describes the application-specific messaging requirements.WSDL (Web Services Description Language) combined with SOAP provided better interoperability between systems that share data.

1.1.1 Why WS* was needed and SOAP failure

The SOAP specification does not define the actual content of message headers and body. An application has the luxury to define the headers and the body for a specified operation. However, there are some standard functions like message routing and addressing, dealing with substantial message payloads, and providing secure and reliable communication. WS* refers to a growing set of standard protocols based on the SOAP specification. Their purpose is to facilitate standard messaging requirements between systems.

Sending XML over HTTP was a simple, straightforward idea. However, each of these WS-I protocols added complexity to that simple idea. Simultaneously, two things started happening:

1. The adoption of open-source frameworks like Ruby on Rails and others became increasingly popular
2. JSON replaced XML as a serialization technology

Applications required a way to provide data to mobile clients. It was simple to use many Web frameworks to look something up in a database and return the results in a serialized format. SOAP was doomed to failure for such prevalent but straightforward scenarios. The other concern with SOAP is strong typing. WSDL accomplishes its magic through XML Schema and strongly typed messages. However, strong typing is a wrong choice for loosely coupled distributed systems. Even for a small change, the type signature changes, and all the clients built according to earlier protocol specifications break. SOAP API also requires a larger bandwidth because of the large size of the XML files and a payload produced by the massive structure of messages. It also requires knowledge and understanding of all protocols used with it, which resulted in a steep learning curve for new users. This resulted in the adoption of REST over SOAP. While SOAP and REST have similarities over the HTTP protocol, SOAP has a rigid set of messaging patterns compared to REST. The rules in SOAP are integral as we cannot achieve any level of standardization without them. REST is an architecture style that does not need processing and is naturally more flexible.

1.2 Why REST was not interoperable

REST[3] stands for Representational State Transfer. REST is not a protocol or a standard but a set of constraints on architecture design. Developers implementing an API can do it in a wide range of ways. It involves a client-server architecture where back-end data is made available to clients through the JSON or XML messaging format. According to Roy Fielding [4], an API qualifies as "RESTful" when it meets the following constraints:

1. Uniform Interface: An API must expose specific application resources to API consumers.
2. Client-Server Independence: The client and server are independent of one another. The client knows the URIs of the resources only.
3. Stateless: The server does not save any data about the client request. The client saves this "state data" on its end (via a cache).
4. Cacheable: Application resources exposed by the API need to be cacheable.
5. Layered: The architecture is layered, allowing different components to be maintained on different servers.

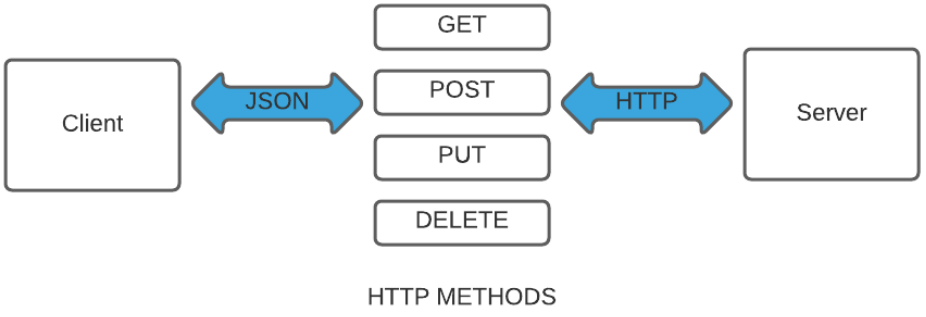

FIGURE 1.2: Client Server Architecture of REST API

Fig 1.1 explains the working of REST API. The REST API architecture is based on HTTP protocol, and it is most commonly used for building web applications and connecting microservices. In REST, the client should be able to get a piece of data called a resource when linked to a specific URL. Each URL is a request, while the data sent back to the client is a response. The response is usually in JSON format but can be in other forms as well. Clients can access these resources via a standard interface that accepts different HTTP commands like GET, POST, DELETE, and PUT.

1.2.1 Problems with REST

The following reasons makes REST not easily interoperable

1. While creating RESTful services, users need to follow a standard practice of writing a client library and update the client library whenever there is a change in API contracts. This takes a extra effort from the user side.
2. Streaming is complex, and it is almost impossible in most languages.
3. Duplex streaming is not possible.
4. Difficult to get multiple resources in a single request.
5. Requires semantic versioning whenever the API contract needs to be changed.
6. REST APIs use third-party tools such as Swagger to auto-generate the code for API calls in various languages. This was the main reason, AI4EU Experiments decided to use gRPC instead of REST.
7. There is no standard data format for serialization. REST can use JSON, XML, etc.
8. The URI for entities are not standardized, The developer has the choice to design the URI. For example to access a book entity with id 12345 it could be mybook-app/api-layer/book/12345 or restapi/books/12345 or book-service/entities/book/12345

1.3 Motivation for using gRPC

As a predecessor of REST, RPC stands for Remote Procedure Call. RPC enables a user to call a function on a remote server in a specific format and get a response in the same form. It does not matter what format the server executing the request uses, if it is in a local server or a remote server.

As a variant of the RPC architecture, Google created gRPC to enable faster communication and enable transmission of data between microservices and other systems that need to interact.

Figure 1.3 shows the communication between the gRPC server and its multiple gRPC clients. A client application can call a service implemented on the server application on a different application as it is a local object. On the server-side, the server implements this interface and starts a gRPC server to handle client requests. There is a stub that

provides a way to call the same methods implemented in the server on the client-side. gRPC servers and clients can communicate in a variety of languages supported by gRPC.

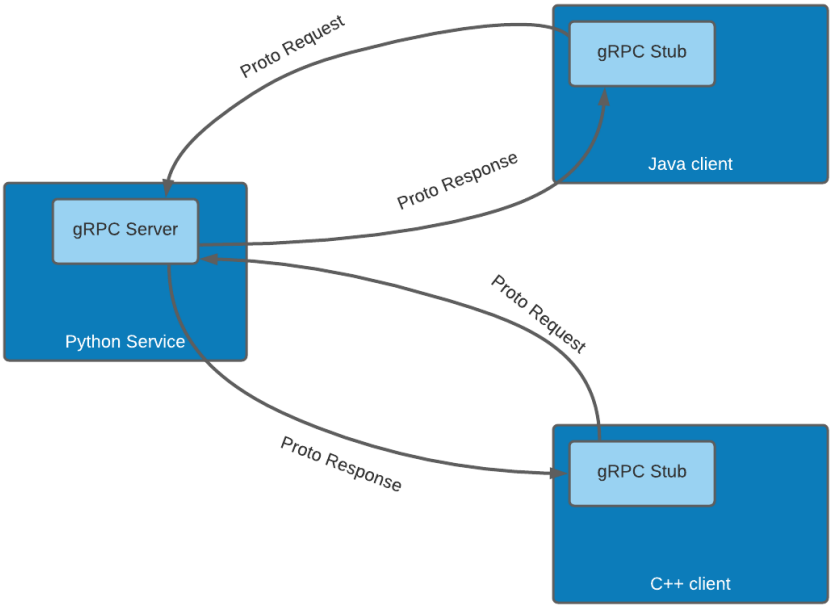

FIGURE 1.3: Client Server Architecture of gRPC Protocol [5]

Compared to REST and SOAP, gRPC solves the problem of inter-operability in the following ways:

1. gRPC uses protobuf (Protocol buffers) instead of JSON as its messaging format. Protobuf makes gRPC faster, lighter, and more efficient. In comparison with REST, it has a standard data format.
2. It is built on HTTP 2 Instead of HTTP 1.1.
3. gRPC has native built-in Code Generation. So it is capable of generating ready-to-use stubs and skeletons. Built-In Code Generation is one of the main reasons AI4EU experiments decided to use gRPC for serialization.

1.4 The role of Docker as a containerization tool

Docker [6] solves a critical problem in a distributed environment like microservices. Because microservices are self-contained, independent application units that satisfy a particular business requirement, they are treated as small, independent applications. What will happen if there are a considerable number of microservices created for an

application? Moreover, what if several microservices are built with different technology stacks? The development team will soon be in trouble as developers have to manage even more environments than traditional monolithic applications. The solution to this problem is using microservices and containers to encapsulate each microservice. Docker helps manage those containers. A Docker container is a standard unit of software that packages up code and all its dependencies to execute quickly and reliable from one computing environment to another. In AI4EU experiments, each AI node in a pipeline is a Docker container. gRPC services running inside a Docker container can be implemented in any language.

1.5 AI4EU Experiments and Acumos

In AI4EU Experiments, we propose to build reusable building blocks similar to microservices. An extensive AI application is broken down into multiple smaller blocks, reused across different applications. These reusable blocks are implemented as Docker Containers, and gRPC protocol is used for communication between the containers. These blocks are then connected to build an AI Pipeline.

After containers are deployed in a suitable environment, execution of the pipeline has to be handled by a Run-time Orchestrator. The orchestrator controls the flow of messages between containers and returns the status code and status message. Kubernetes is our target deployment environment as it works well with docker containers and has numerous advantages concerning scalability, availability, and flexibility.

The primary motivation to use gRPC is to support automatic code generation on the fly. It enables the orchestrator to communicate with models or AI blocks that it has no prior knowledge about. The models or the pipeline nodes act as gRPC servers that will be running in a docker container, and the services will be listening on a specified port. The orchestrator acts as a gRPC client that uses generated stubs by the proto compiler to call the services on gRPC servers, running in multiple Docker containers.

AI4EU experiments use the Acumos platform [7] to onboard the AI resources. The models are onboarded as a dockerized URI using the Acumos platform. These models can be shared publicly, within teams, or kept private. The design studio AcuCompose provides a visual frontend to create a composite solution. The models onboarded in the platform before can be connected to compose a pipeline. After the pipeline is composed, the user needs to export the solution to the local system. The solution package will have the Kubernetes deployments files, the protobuf definitions for all the pipeline nodes, a Kubernetes client script, and an orchestrator client responsible for executing a pipeline.

The user then needs to execute the Kubernetes client to have the models deployed in the local namespace of a Kubernetes cluster and then execute the orchestrator client, which will run a pipeline composed in the user's local environment.

An important aspect here is understanding AI4EU experiments go beyond the original Acumos implementation by using gRPC, Docker, and Kubernetes. In this master thesis work, the problem of designing a generic orchestrator that can run any generic pipeline composed according to the AI4EU container specification is addressed. This design of a generic solution that can execute any generic pipeline goes beyond the the state of the art.

1.6 Problem Statement

Design and evaluation of a generic orchestrator that executes AI pipelines never seen before. An AI pipeline, in this context, consists of nodes (docker containers) and edges (information flow), thus forming a graph. After a pipeline has been composed and deployed, execution needs to be controlled by a so-called runtime-orchestrator which dispatches the message flow between the participating nodes following the topology specification (blueprint.json). It must do the required initialization of the nodes and call each model according to the flow using gRPC/protobuf communication. The following tasks are completed in the course of this master thesis:

1. Design of both simple and advanced pipeline scenarios that serve as proof of concept for the implementation. These scenarios should describe the detailed information flow between the nodes.
2. Specify the strategy for orchestration.
3. Design and implement the orchestration subsystem as well as the necessary algorithms and test it with scenarios from step 1
4. Describe and review the results, again based on the scenarios from step 1

1.7 How the Thesis is Organised

The thesis is organised into following chapters:

- Introduction and approach(Chapter 1)

- In chapter 2 we cover theoretical background regarding different fields of AI and need for a AI pipeline, gRPC and Protobuf, Acumos AI Platform, Design Studio of Acumos, Deployment Environment i.e Docker and Kubernetes.
- The methodology we developed is explained in chapter 3
- In chapter 4, we discuss about the different experiments involving simple pipelines, advanced pipeline and the challenges faced in designing the orchestrator for different scenarios.
- Related Work and other platforms with similar approaches(Chapter 5)
- Chapter 6 concludes with future work details.

Chapter 2

Theoretical Background

Here we discuss in brief the different fields of AI, starting with symbolic AI, machine learning, neural networks, and Artificial Intelligence and the need for hybrid pipelines. We then discuss gRPC and protobuf, deployment environment, and Acumos AI platform, which is used by the AI4EU project.

2.1 Different Fields of AI and Need for AI pipelines

This section gives a basic idea about different fields of AI, hybrid AI and the need for pipelines. AI4EU experiments platform is not limited to machine learning or deep learning models. Any AI tool from any AI area can be used for building pipelines.

2.1.1 Symbolic AI

In the present world, artificial intelligence is primarily about artificial neural networks and deep learning. However, this was not always the case. As a matter of fact, for the more significant part of its history, the field was dominated by symbolic artificial intelligence, also called "classical AI," "rule-based AI," and "good old-fashioned AI."

Symbols represent other things. Symbols play a critical role in the thought and reasoning process of humans. We use symbols to define things (such as dog, car, and bike) and people (such as teacher, lawyer, salesperson). Symbols can represent abstract concepts such as bank transactions or things that do not physically exist, such as web pages and blog posts. They can also describe actions such as running or states of being inactive and busy. Symbols can be organized into hierarchies. For example, a car is made of doors, windows, tires, seats. They can also be used to describe other symbols like a dog

with a long tail or a velvet jacket. Communication through symbols is one of the main reasons that makes humans intelligent. Hence, symbols have also played a significant role in the creation of artificial intelligence.

Symbolic artificial intelligence exhibited early progress at the dawn of AI and computing. Visualizing the logic of rule-based programs, communicating, and troubleshooting were easy. Symbolic artificial intelligence is appropriate for settings where the rules are precise, and it is easy to obtain input and transform it into symbols. Rule-based systems still account for many computer programs today, including those used to create deep learning applications. Symbolic AI was intended to produce general, human-like intelligence in a machine, whereas most modern research focuses on specific sub-problems. Research into general intelligence is now studied in the sub-field of artificial general intelligence.

Some believe that the significance of symbolic AI is diminishing. Nevertheless, this assumption is far from the truth. Rule-based AI systems are still essential in today's applications. Many leading scientists believe that symbolic reasoning will continue to remain a vital component of artificial intelligence.

2.1.2 Artificial Neural Networks

Artificial Neural Networks[8] are inspired by nature, basically from neural networks in the brain. It is an abstraction of a biological neural network. Plenty of research occurred in the area of artificial neural networks. Those rapid research works unfolded many different network capabilities and gave multiple variants of these networks. Basic neural network architecture can be seen in the figure 2.1. Neural Networks are developed to exploit the architecture of the human brain to perform tasks that conventional algorithms can solve.

2.1.2.1 Components

A typical neural network has the following components:

- **Neurons**: They receive input, combine the input with their internal state (known as activation) and an optional threshold using an activation function, and produce output using an output function. The inputs can be any external data, such as numbers, images, and documents.
- **Connections and weights**: The network consists of connections; the connection takes the output of one neuron and provides it as input to another neuron. Each

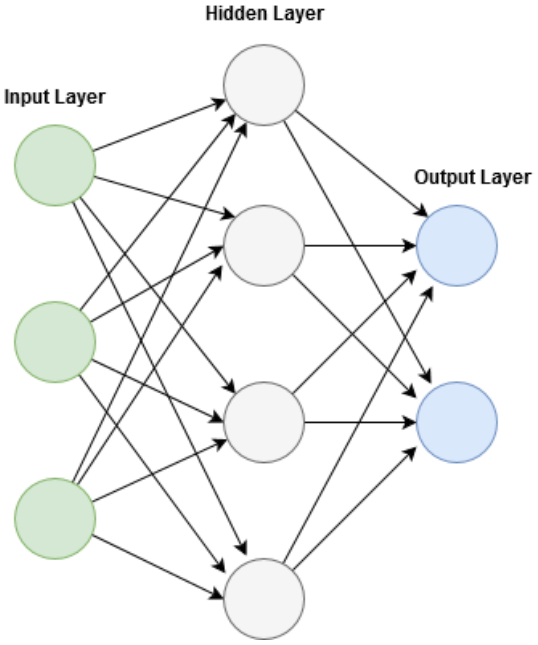

FIGURE 2.1: Artificial Neural Network basic architecture with input, hidden and output neurons

connection is assigned a weight, and a neuron can have multiple inputs and output connections[9].

- **Activation Function and bias**: The activation function computes the input to a neuron from the outputs of previous neurons and their connections as a weighted sum, and a bias is added to the result. The activation function provides a smooth transition as input values change.

2.1.2.2 Working

Artificial neural networks use various layers of mathematical processing. An artificial neural network contains neurons anywhere from dozens to millions arranged in a sequence of layers. The input layer receives information from various data sources. The information goes through one or more hidden neurons from the input neuron. The input information is transformed into an intermediate state by a hidden neuron which the output neuron can use. The network starts learning about the information once the data flow through each neuron. The network responds to the information that was given and processes the information at the output neuron.

Neural networks need to be trained with a considerable amount of information through training for learning. For example, to teach an artificial neural network how to differentiate a car from a bus, the training set should be thousands of images of buses that the network would begin to learn. Once it is trained with a significant amount of data(bus

images in this case), the network will classify future data. During the training period, the output is compared to the ground truth data label. If they are the same, then the model learning is finished. It uses backpropagation to adjust its learning by going back through the layers to tweak the weights if it is incorrect.

2.1.3 Machine Learning

Machine Learning is a branch of artificial intelligence where systems can learn from data, recognize patterns and enable decisions with minimal human intervention. Machine learning is essential because it enables enterprises to view trends in data such as customer behavior, operational business patterns that support the development of new products and aid decision making. Classical machine learning is categorized by how an algorithm learns to predict more accurately. There are four basic approaches:

1. **Supervised Learning:** In supervised learning, labeled training data is used, and the variables are correlated. The input and the output of the algorithm are specified. The algorithm infers a function that maps the input data to the output label.

2. **Unsupervised Learning:** In unsupervised learning, unlabeled training data is used, and the algorithm tries to discover the hidden patterns in the data.

3. **Semi-Supervised Learning:** This machine learning approach combines supervised and unsupervised methods. An algorithm is fed with labeled training data primarily, but the model is free to explore the data independently and develop its understanding of the data set.

4. **Reinforcement Learning:** Reinforcement learning is the training of machine learning models to make a sequence of decisions. In an uncertain, potentially complex environment, the agent learns to achieve a goal by facing a game-like situation. The algorithm performs a trial and error to come up with the problem's solution, and each action gets a reward or penalty. The end goal is to maximize the total reward. Although the designer sets the reward policy as a game rule, he gives the model no hints or suggestions for solving the game. It is up to the model to determine the solution for the task by maximizing the reward, starting from arbitrary trials and finishing with sophisticated tactics and skills.

2.1.4 Hybrid AI

There is an emerging need for hybrid AI in today's AI world. It would be ideal to have intelligent systems that can provide human-like expertise such as domain knowledge, uncertain reasoning, and adaptation to a noisy and time-varying environment. These hybrid solutions are crucial in tackling practical computing problems. The integration of various learning and adaptation techniques, overcome individual limitations, and achieve synergetic effects through hybridization or fusion of these techniques, has contributed to many new intelligent system designs in recent years. Hybrid AI denotes a sub-category of AI which employs, in parallel, a combination of methods and techniques from artificial intelligence subfields.

The search engine of Google is an example of hybrid AI that combines state-of-the-art deep learning techniques such as Transformers and symbol-manipulation systems such as knowledge-graph navigation tools. AlphaGo, one of the landmark AI achievements of the past few years, is another example of combining symbolic AI and deep learning.

The AI4EU experiments provide an ideal platform for working on hybrid solutions by combining various models in a pipeline.

2.2 gRPC and Protobuf

Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data.[10] It is language-independent and uses an Interface Definition Language (IDL) for defining the shape of a given message and the properties within that message. Protobuf message definitions have values but no methods; they are data-holders. Protobuf messages are defined in .proto files.[11] protoc is a compilation tool for protocol buffers that can compile the code in various supported languages. All the properties in protobufs are assigned default values, so if the user does not define an integer, the integer field will default to 0. Fields with string data types will default to empty strings, booleans to false, and similar for other data types.

gRPC is a group of technologies that allow communication with other applications using Remote procedure calls (RPCs). Remote procedure calls are function calls that users make in their codebase, but they execute on another machine, then return in the user's application. The RPC abstracts the network from the user, letting them call methods as if they were local code.

Protocol Buffers is used by gRPC for data serialization between clients and servers. Users need to define the interface in a .proto file containing the services used in applications.

Making use of the gRPC plugin for protoc (or a plugin as part of build script for a given platform), users can generate code that will give them the methods required to call a given service, all with native typing in the language of the user's choice.

We can define a gRPC service in our .proto file like this:

```
//Define the used version of proto:
syntax = 'proto3';

//Define a message to hold the features input by the client :
message Text{
    string query = 1;

}
//Define a message to hold the predicted price :
message Prediction{
    float review      = 1 ;
}

//Define the service :
service Predict{
    rpc classify_review(Text) returns (Prediction){}
}
```

gRPC supports multiple data transfer methods: Request and Response similar to typical REST service, Server-side streaming, Client-side streaming, Bidirectional streaming. protocol buffers and gRPC protocol are used for communication between the models/nodes in AI4EU pipelines.

2.3 Docker

Docker [12] is an open-source platform for developing, shipping, and running applications. Docker enables us to separate our applications from the infrastructure so that software can be delivered quickly. Docker enables the IT teams to manage the infrastructure similar to the managing of the applications. The delay between writing a code and running it in production can be significantly reduced by using Docker. Docker enables to package and run of an application in an isolated environment called a container. Because of Docker's isolation and security of containers, multiple containers can

run on a single host.The containers run directly with the host machine's kernel and are lightweight as compared to virtual machines. The docker platform manages the lifecycle of the containers in the following ways [6]:

- Development of software application and its supporting components using containers.
- The container is the unit for distributing and testing the software application.
- Deployment of the application and solutions to the production environment. The deployment remains the same whether it is a local data center, a cloud provider, or a hybrid environment.

2.3.1 Docker Architecture

Docker works in client-server architecture. The Docker client communicates with the Docker daemon. The daemon, which acts as a server here, performs the building, running, and distributing the Docker containers. The Docker client and daemon can run on the same system, or a Docker client can communicate with a remote Docker daemon. [6]

2.3.2 Docker Objects

1. **Docker Image:** An Docker image is a read-only template with instructions for creating a Docker container.
2. **Docker Container:** A container is a runnable instance of an image. Users can create, start, stop, move, or delete a container using the Docker API or client interface i.e CLI.
3. **Docker Service:** Services allow containers to scale across multiple Docker daemons, which all work together as a swarm with multiple managers and workers.A service is used to define the desired state, such as the number of replicas of the service that must be available at any given time.

The role for docker as containerization tool in AI4EU experiments is explained in section 1.4.

2.4 Kubernetes

Kubernetes[13] is an open-source container orchestration platform that automates the manual processes involved in deploying, managing, and scaling containerized applications.

A Kubernetes cluster comprises a group of worker machines called nodes. Containerized applications are made to run on these nodes. Every cluster has at least one worker node. The worker nodes host the Pods that are the components of the application. The control plane controls the worker nodes and the Pods in the cluster, and it runs across multiple machines in a typical production environment. A cluster runs multiple nodes, providing fault tolerance and high availability. A Kubernetes cluster can be installed as a managed Kubernetes, i.e., DigitalOcean, Amazon Elastic Kubernetes, and others. It can be installed by using minikube to run a cluster locally. A Kubernetes cluster can also be installed and configured manually. In the following subsection, we will be explaining the important objects in Kubernetes.

2.4.1 Pod

A pod is the most basic unit of a Kubernetes cluster and represents one or more application containers and shared resources for those containers. A pod usually contains one or more running containers. Containers running in a pod share the same network, storage, and lifecycle. This means that they can communicate directly and will both be stopped and started at the same time.

2.4.2 ReplicaController and ReplicaSets

The Replica Controller and Replicaset help to run multiple instances of a pod in a Kubernetes cluster, thus providing high availability.They ensure that at any given time, the desired number of pods specified is in the running state. If a pod stops or dies, they create another one to replace it. The difference is Replicaset uses set-based selectors, and Replica Controller uses equity-based selectors.

2.4.3 Deployment

Deployments are Kubernetes objects that manage the pods. The first thing a Deployment does is to create a replica set. The replica set creates pods according to the number

specified in the replica option. Deployment is a higher-level concept that manages replica sets efficiently and provides declarative updates to the pods.

2.4.4 Services

Kubernetes service is an abstract way to expose an application running on a set of Pods as a network service. Kubernetes service enables developers not to use unknown service discovery mechanisms. Kubernetes provides pods with their IP addresses and a unique DNS name for a set of pods, and it is also responsible for balancing the load between the pods.

Kubernetes is our target deployment environment in AI4EU experiments. For each node/model in the pipeline, the Kubernetes client will create a ReplicaSet which in turn will create a pod. The Kubernetes client will also create a node-type service by which the gRPC services become reachable.

2.5 AI4EU Container Specification

In this section, we discuss the AI4EU container specifications [14] to be followed for AI4EU experiments. The containers should define their public service methods using protobuf v3 and expose these methods via gRPC. All these technologies are open source and freely available. Because the goal is to have re-usable building blocks to compose pipelines, the main reason to choose the above technology stack is to achieve the highest level of interoperability:

- Docker is today the defacto standard for server-side software distribution, including all dependencies. It is possible to onboard containers for different architectures (`x86_64`, GPU, ARM, HPC/Singularity)
- gRPC and protobuf are a proven specification and implementation for remote procedure calls supporting a broad range of programming languages. It is optimized for performance and high throughput.

Please note that the tools and models are not limited to deep learning models. Any AI tool from any AI area like reasoning, semantic web, symbolic AI, and deep learning can be used for pipelines as long as it exposes a set of public methods via gRPC.

2.5.1 Defining the protobuf interface

The public service methods should be defined in a file called model.proto:

- It should be self contained, thus contain the service definitions with all inputs and output data structures and no imports can be used
- The full feature set of protobuf v3 can be used (except import)
- A container can define several methods, but all in the .proto file
- we should take into account that all interacting proto-files must be in the same package, or use no package at all

2.5.2 Create the gRPC docker container

Based on model.proto, we can generate the necessary gRPC stubs and skeletons for our choice's programming language using the protobuf compiler protoc and the respective protoc-plugins. Then create a short primary executable that will read and initialize the model or tool and starts the gRPC server. This executable will be the entry point for the docker container. The gPRC server must listen on port 8061. If the model also exposes a Web-UI for human interaction, optional, it must listen on port 8062. The file tree of the docker container should look like below. In the top-level folder of the container are the files model.proto and license.json.The license file is not mandatory and can be generated after onboarding with the License Profile Editor in the AI4EU Experiments Web-UI using this link.

Important recommendation: For security reasons, the application in the container should not run as root, which is the default. Instead, an unprivileged user should be created that runs the application. Here is an example code snippet from a Dockerfile:

```
RUN useradd app
USER app
CMD ["java", "-jar", "/app.jar"]
```

This will also allow the docker container to be converted into a Singularity container for HPC deployment.

2.5.3 On boarding

The final step is to on board the model. There are several was to on board a model into AI4EU Experiments but currently the only recommended way is to use "On-boarding dockerized model URI":

1. Upload docker container to a public registry like Docker Hub or to a private registry
2. Start the on boarding process
3. Upload the protobuf file
4. Add license profile
5. Enter the docker image URI

2.5.4 First Node Parameters (e.g. for Data brokers)

Generally speaking, the orchestrator dispatches the output of the previous node to the following node. A particular case is the first node, where no output from the previous node exists. To design a generic orchestrator, the first node must define its services with an "empty" input datatype. In this case, it should be google.protobuf.Empty. Typically this concerns nodes of type Data broker as the usual starting point of a pipeline.

```
syntax = "proto3";
import "google/protobuf/empty";

message NewsText {
string text = 1;
}

service NewsDatabroker {
rpc pullData(google.protobuf.Empty) returns(NewsText);
}
```

2.5.5 Scalability, GPU Support and Training

The potential execution environments range from Minikube on a Laptop over small Kubernetes clusters to big Kubernetes clusters and even HPC and optional GPU acceleration. It is possible to support all those environments with a single container image taking into account some recommendations:

- let the model be flexible with memory usage: use more memory only if available.
- let the model be scalable if more CPU cores are available (allow for concurrency).
- Some AI frameworks like PyTorch, or Tensorflow can be used to work with or without GPU with the same code.
- even training is possible if the model exposes the corresponding methods in the protobuf interface.

Chapter 3

Methodology

In this chapter, we illustrate our approach for solving the task of designing a generic serial orchestrator which is capable of executing any AI pipeline which is composed in Visual editor of Acumos following the AI4EU container specification.

First, we introduce all important sections which is required to design a generic solution to execute a pipeline and the final section shows the overall picture and how the concepts in the earlier sections are integrated into designing a solution.

3.1 gRPC Generated-code and use of stubs

The design of a generic orchestrator is done using python. This section gives details about how built-in code generation of gRPC python is made use in our approach. gRPC python depends on the protobuf compiler (protoc) to generate code. It uses a plugin to supplement the generated code by plain protoc with gRPC-specific code. For a .proto service description containing gRPC services, the plain protoc generated code is synthesized in a `_pb2.py` file, and the gRPC-specific code is contained in a `_pb2_grpc.py` file. The `_pb2_grpc.py` python module imports the `_pb2.py` . [15]

Consider the following example

```
service start_orchestrator {
rpc executePipeline(PipelineConfig) returns(PipelineStatus);
}
```

When the service is compiled, the gRPC protoc plugin generates code similar to the following `_pb2_grpc.py` file:

```
import grpc
import orchestrator_pb2 as orchestrator__pb2

class start_orchestratorStub(object):
    """Missing associated documentation comment in .proto file."""
    def __init__(self, channel):
        """Constructor.
        Args:
            channel: A grpc.Channel.
        """
        self.executePipeline = channel.unary_unary(
            '/start_orchestrator/executePipeline',
            request_serializer=orchestrator__pb2.PipelineConfig.SerializeToString,
            response_deserializer=orchestrator__pb2.PipelineStatus.FromString,
            )

class start_orchestratorServicer(object):
    """Missing associated documentation comment in .proto file."""

    def executePipeline(self, request, context):
        """Missing associated documentation comment in .proto file."""
        context.set_code(grpc.StatusCode.UNIMPLEMENTED)
        context.set_details('Method not implemented!')
        raise NotImplementedError('Method not implemented!')


def add_start_orchestratorServicer_to_server(servicer, server):
    rpc_method_handlers = {
            'executePipeline': grpc.unary_unary_rpc_method_handler(
                    servicer.executePipeline,
                    request_deserializer=
                    orchestrator__pb2
                    .PipelineConfig.FromString,
                    response_serializer=
                orchestrator__pb2.
                PipelineStatus.SerializeToString,
            ),
    }
    generic_handler = grpc.method_handlers_generic_handler(
```

```
            'start_orchestrator', rpc_method_handlers)
    server.add_generic_rpc_handlers((generic_handler,))


 # This class is part of an EXPERIMENTAL API.
class start_orchestrator(object):
    """Missing associated documentation comment in .proto file."""

    @staticmethod
    def executePipeline(request,
            target,
            options=(),
            channel_credentials=None,
            call_credentials=None,
            insecure=False,
            compression=None,
            wait_for_ready=None,
            timeout=None,
            metadata=None):
        return grpc.experimental.unary_unary
        (request, target, '/start_orchestrator/
        executePipeline',
            orchestrator__pb2.
            PipelineConfig.SerializeToString,
            orchestrator__pb2.PipelineStatus.FromString,
            options, channel_credentials,
            insecure, call_credentials, compression, wait_for_ready,
            timeout, metadata)
```

3.1.1 Code Elements

The gRPC generated code starts by importing the grpc package and the plain _pb2 module, synthesized by protoc, which defines non-gRPC-specific code elements, like the classes corresponding to protocol buffers messages and descriptors used by reflection. [15]

For each service in the .proto file, three primary elements are generated:

1. Stub: used by the client to connect to a gRPC service. In our example above it is `start_orchestratorStub`
2. Servicer: used by the server to implement a gRPC service. In our example it is `start_orchestratorServicer`
3. Registration Function: `add_to_server` function used to register a servicer with a grpc.Server object. In our example it is `add_start_orchestratorServicer_to_server`

3.1.1.1 Stub

The generated Stub class is used by the gRPC clients. It has a constructor that takes a grpc.Channel object and initializes the stub. For each method in the service, the initializer adds a corresponding attribute to the stub object with the same name. Depending on the RPC type (unary or streaming), the value of that attribute will be callable objects of type UnaryUnaryMultiCallable, UnaryStreamMultiCallable, StreamUnaryMultiCallable, or StreamStreamMultiCallable. [15]

3.1.1.2 Servicer

For each service, a Servicer class is generated, which serves as the superclass of a service implementation. For each method in the service, a corresponding function in the Servicer class is generated. Override this function with the service implementation. Comments associated with code elements in the .proto file appear as docstrings in the generated python code. [15]

3.1.1.3 Registration Function

For each service, a function is generated that registers a Servicer object implementing it on a grpc.Server object, so that the server can route queries to the respective servicer. This function takes an object that implements the Servicer, typically an instance of a subclass of the generated Servicer code element described above, and a grpc.Server object. [15]

3.1.1.4 Use of `_pb2 and _pb2_grpc` in Orchestrator

In a pipeline Scenario, Each node will have a input protobuf message, output protobuf message and services associated to it. `_pb2` files will be used to generate the right input and output messages. `_pb2_grpc` will be used to create stubs which in turn will allow

the orchestrator to call the right service for each node. Let us consider the following example:

```
syntax = "proto3";
import "google/protobuf/empty.proto";

message AudioFileJob {
 string file_name = 1
 string work_dir = 2;
 }

service AudioFileBroker {
 rpc getAudioFileJob(google.protobuf.Empty) returns(AudioFileJob);
 }
```

The above protobuf file has a message AudioFileJob and service AudioFileBroker. The protoc complier for any protobuf file generates pb2 and `pb2_grpc` files. These generated files are used by the orchestrator to create request message and call the service using stub as shown below.

```
# create a stub (client)
stub = pb2_grpc.AudioFileBrokerStub(channel)

# create a valid request message
request = pb2.AudioFileRequest(work_dir="Local_pipeline/")
# make the call
response = stub.getAudioFileJob(request)
```

3.2 Toplogy and Node Information of the Pipeline

Topology, order of nodes, DNS names, and other pipeline information designed in the visual editor of Acumos will be known by two files: blueprint.json and dockerinfo.json. The blueprint.json file is generated by the design studio of Acumos when a composite solution is created, and the Kubernetes client generates dockerinfo.json. Both the JSON files are included in the downloaded solution package. The Kubernetes python script in the solution package will create the Kubernetes deployments and services for each node in the specified namespace of a Kubernetes cluster. The Kubernetes python script will also write DNS names in the dockerinfo.json.

3.2.1 Topology Information

Topology information is generated by the Design Studio editor, Acucompose, in a blueprint.json file. AcuCompose enables pipeline composition by using the models onboarded in the AI4EU experiments platform like in figure 4.1. A typical blueprint.json generated after pipeline composition is as below.


```
  {
"name": "simplepipeline",
"version": "v1",
"input_ports": [
  {
    "container_name": "csvdatabroker1",
    "operation_signature": {
      "operation_name": "get_next_row"
    }
  }
],
"nodes": [
  {
    "container_name": "csvdatabroker1",
    "node_type": "MLModel",
    "image": "https://cicd.ai4eu-dev.eu:7444/csvdatabroker:v2",
    "proto_uri": "org/acumos/csvdatabroker/1.0.1/csvdatabroker-1.0.1.proto",
    "operation_signature_list": [
      {
        "operation_signature": {
          "operation_name": "get_next_row",
          "input_message_name": "Empty",
          "output_message_name": "Features"
        },
        "connected_to": [
          {
            "container_name": "houseprice1",
            "operation_signature": {
              "operation_name": "predict_sale_price"
            }
          }
        ]
```

```
        }
      ]
    },
    {
      "container_name": "houseprice1",
      "node_type": "MLModel",
      "image": "https://cicd.ai4eu-dev.eu:7444/houseprice:v1",
      "proto_uri": "org/acumos/c6bc5abe-6422-47e9-9dcf-c01c1cbb7c48/houseprice/1.0.0/
      "operation_signature_list": [
        {
          "operation_signature": {
            "operation_name": "predict_sale_price",
            "input_message_name": "Features",
            "output_message_name": "Prediction"
          },
          "connected_to": []
        }
      ]
    }
  ],
  "probeIndicator": [
    {
      "value": "false"
    }
  ]
}
```


blueprint.json will have important information such as the input ports, node name, docker image of the node, proto URI, the adjacent nodes, input message name, output message name, and the service the node implements. The above example is of a simple AI pipeline with two nodes. The first node is the CSV data broker, and the last node is the prediction node with predicts the house's sales price given a set of attributes by the first node. These information from the blueprint.json is parsed to extract all the information needed by the orchestrator. Object oriented programming is used for storing all the node information in classes.

3.2.2 The DNS name and port of nodes in a pipeline

The information such as the Domain name system(DNS) name and port of the nodes in the pipeline will be generated in dockerinfo.json. This information is crucial as the orchestrator will use them to call the services implemented for each node. Our target deployment and execution environment for AI4EU experiments in this scope of master thesis is Kubernetes. The Kubernetes client will use docker URI's to create deployment and service YAML files for each node which will be a part of the solution package when a pipeline solution is downloaded. The solution package will have the deployment and service YAML files for the orchestrator server as well. Once the solution files are downloaded, the user is expected to extract them in his local system, where a Kubernetes cluster is setup. The user runs a deployment script that deploys all the pods in a Kubernetes cluster. The Kubernetes services give the orchestrator the DNS names and the ports for accessing the microservices implemented in the Kubernetes pods. In our case, a Nodeport service is used to expose a service running on a set of Pods as a network service to the orchestrator. A typical dockerinfo.json will look as below.

```
{
  "docker_info_list": [
    {
      "container_name": "csvdatabrokerempty1",
      "ip_address": "csvdatabrokerempty1",
      "port": 30016
    },
    {
      "container_name": "housepricemodel1",
      "ip_address": "housepricemodel1",
      "port": 30027
    },
    {
      "container_name": "orchestrator",
      "ip_address": "orchestrator",
      "port": 30009
    }
  ]
}
```

The above file has the container name, DNS name of the node as the IP address, and port information of the nodes and the orchestrator. Kubernetes takes care of the resolution

of the domain name system, i.e., DNS for each node. Kubernetes will also take care of port mapping of node port to a container-specific port for each AI nodes in the pipeline. AI4EU container specification suggests port 8061 as a default container port and 8062 as an optional port for the web service.

3.3 Graph Data Structure and the Traversal

The idea of building pipelines in AI4EU experiments follows the principles of Graph Theory. Each AI block in a pipeline represents a node, and the vertices make the connections. The Graph Data Structure stores all node information, and the adjacent list stores adjacent node details for each node in the pipeline. After the graph's construction for the pipeline, a graph traversal method is used to get the traversal path. The orchestrator uses this traversal path to call the services of the node in the order according to the topology. This implementation enables the orchestrator to execute pipelines with parallel paths from a node as long as there are no cycles between the pipeline nodes. These pipelines are referred to as Directed Acyclic Graphs(DAG) according to Graph Theory. In our case, Breadth-First Search(BFS) graph traversal is used because the orchestrator needs to traverse the nodes at depth d before traversing the nodes at depth d+1. An adjacency list is used to store the nodes connected to a source node. An adjacency list is a collection of unordered lists used to represent a finite graph. Each list inside the adjacency list will contain the set of neighbors of a vertex in the graph.

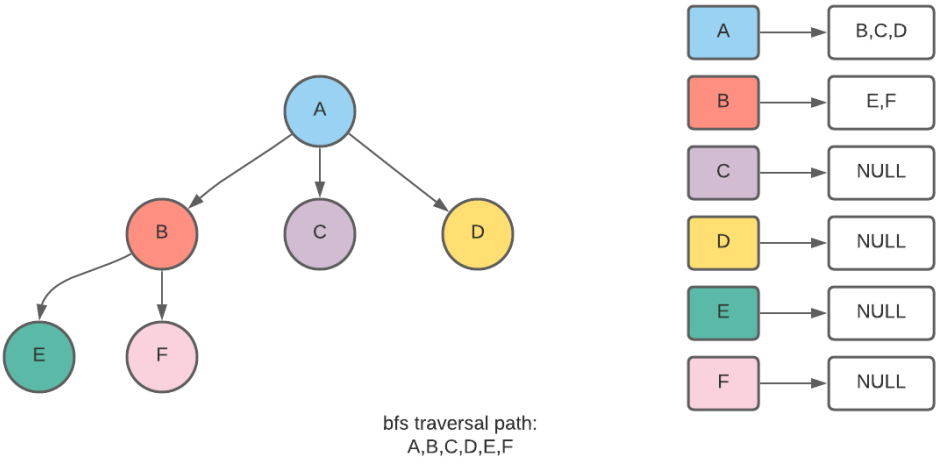

FIGURE 3.1: Graph Data structure and Adjacency List

Consider the pipeline scenario in figure 3.1; there are six nodes connected in a directed acyclic way. The figure also shows the adjacency list for each node in the graph. BFS is a traversing algorithm that starts traversing from the source or starting node and traverses

the graph layerwise, thus exploring the neighbor nodes (nodes directly connected to the source node). The algorithm moves towards the next-level neighbor nodes after traversing the previous levels. As the name BFS suggests, the algorithm traverses the graph breadthwise by first moving horizontally ,visiting all the nodes of the current layer, and then moving to the next layer

3.4 Protobuf Merging and Automatic Generation of Stubs

The merging of multiple protobuf files is an essential preprocessing step before the orchestrator starts executing a generic pipeline. Each node in the pipeline has a separate protobuf file. The orchestrator needs to communicate with all of them. After several attempts, the only solution that worked was the merging of protobuf files. This step considers multiple protobuf definitions for all nodes in a pipeline and consolidates a single protobuf definition for a pipeline. It is an essential step as we take care of duplicate messages between nodes. The user comments are ignored, and there is explicit importing of the default google.protobuf.empty for the starting node. This step also prepares a map that connects the service name and the remote procedure call(RPC) name. The generic orchestrator will later use this map to call the right services using the automatic generation of stubs.

To explain this better, let us consider the protobuf definitions of a simple two node pipeline.

The first node in this pipeline example is a data broker node. The protobuf definition for this node has a empty message,a features message and a `get_next_row` service

```
syntax = 'proto3';

import "google/protobuf/empty.proto";

message Features {
    float MSSubClass     = 1 ;
    float LotArea        = 2 ;
    float YearBuilt      = 3 ;
    float BedroomAbvGr   = 4 ;
    float TotRmsAbvGrd   = 5 ;
}

service get_next_row {
```

```
    rpc get_next_row(google.protobuf.empty) returns(Features);
}
```

The second node is a prediction model which has the following interface.

```
syntax = "proto3";

message Features {
    float MSSubClass     = 1 ;
    float LotArea        = 2 ;
    float YearBuilt      = 3 ;
    float BedroomAbvGr   = 4 ;
    float TotRmsAbvGrd   = 5 ;
}

message Prediction {
    float salePrice      = 1 ;
}

service Predict {
    rpc predict_sale_price(Features) returns (Prediction);
}
```

From the above pipeline example, we can see a duplicate message in the form of message Features. These duplicate messages are inevitable as we expect the output message to match the subsequent node's input message. This matching of messages between adjacent nodes is a prerequisite for a valid connection in the design studio of Acumos, and it avoids users connecting random unrelated models in a pipeline. To overcome this problem, we take care of duplicate messages and write common duplicate messages only once in the consolidated protobuf.

The logic to write a consolidate protobuf for a pipeline is designed by using python dictionaries. A dictionary with key-value pair is created for messages and services. Other non-essential lines of code in individual protobuf files like user comments are ignored. A different dictionary for mapping the service names with the remote procedure call names is also created. This dictionary of service with the RPC name is significant for orchestrator as it can make the required mapping from stubs to call the services of a particular node. For the service `predict_sale_price` in the above example, following RPC to stub mapping will be created.

```
  {'predict_sale_price': 'PredictStub'}
```

The consolidated pipeline.proto for the above pipeline will look as below:

```
syntax = "proto3";
import "google/protobuf/empty.proto";

message Features {

    float MSSubClass     = 1 ;
    float LotArea        = 2 ;
    float YearBuilt      = 3 ;
    float BedroomAbvGr   = 4 ;
    float TotRmsAbvGrd   = 5 ;
}

message Prediction {

    float salePrice      = 1 ;
}

service get_next_row {

    rpc get_next_row(google.protobuf.Empty) returns(Features);
}
service Predict {

    rpc predict_sale_price(Features) returns (Prediction);
}
```

The consolidated pipeline.proto will have the clean structure with messages and the services for the nodes of the pipeline without duplicates. The protoc compiler will then be used to generate the stubs and skeletons i.e gRPC Generated-code explained in section 3.1.

3.5 Dynamic Linking of Nodes of the pipeline

A generic orchestrator has to have the capability of dynamically making connections between nodes given the blueprint JSON file with topology and node information. Dynamic linking of nodes makes use of the breadth first search(BFS) traversal output and the list of all node class instances which has all the node properties. This list is obtained by parsing the blueprint JSON file. The design of the logic to dynamically link nodes of the pipeline is shown in figure 3.2.

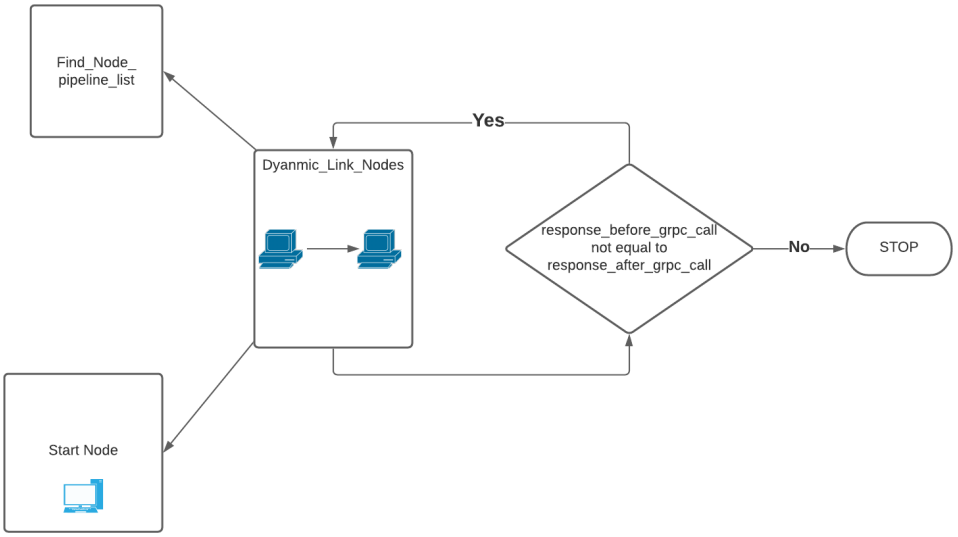

FIGURE 3.2: Dynamic Linking of Nodes

3.5.1 Find Node in pipeline List

This function is responsible for finding the current node index in a pipeline list that stores all the node information. A python list of Node objects is used to store the node information. Each element in this python list gives a node class instance that stores the nodes' individual properties. After parsing the blueprint file and storing details in the pipeline list, the orchestrator constructs a Graph for the nodes and makes the relevant connections. The BFS traversal algorithm's output will guide the orchestrator in the order of nodes to be called. BFS output is another python list. The orchestrator needs to find the index of the current node it is processing in the pipeline list. This function will return an index that the orchestrator can use to get all the current node information.

3.5.2 Start Node

This function takes the index returned by the Find Node function above in section 3.5.1. The function is responsible for creating a gRPC request, a stub that will be used to call the current node's service. The logic behind connecting nodes in AI4EU experiments is that the previous node's output will be the input of the subsequent node. A special case here is the starting node where the output from the previous node does not exist. AI4EU container specification specifies that the first node should have an empty request message. The orchestrator creates a default empty request with google.protobuf.Empty for the first node. For the next subsequent nodes, the previous node's output is dispatched by the orchestrator.

3.5.3 Dynamic linking of nodes

This recursive function will dispatch the messages from the source node to all the subsequent nodes until a termination condition is reached. The relevant stub for the current node is used to call the service the node implements. This recursive function's termination condition is when the response before the gRPC service call is the same as the response after the gRPC service call. This termination condition denotes the previous node is exhausted and does not have valid messages to return. To explain this, let us consider an example where a node implements the following service.

```
    service Predict {
    rpc predict_sale_price(Features) returns (Prediction);
}
```

The `predict_sale_price` service has a input message Features and output message Prediction. The output prediction message has the following members:

```
    message Prediction {
    float salePrice      = 1 ;
}
```

Since the member field salePrice is a datatype float, When the orchestrator creates a response message, it has a default value of the data type, which is 0.0f in the case of float. This is stored in `response_before_call` object. The orchestrator then calls the service using the stub and the output of the `predict_sale_price` is captured in the `response_after_call` object. If the call is successful, the `response_after_call` object will have a float value other than the default.

3.6 Pipeline Execution

In this section, we explain the execution of pipeline by using a flow diagram in figure 3.3. The getstubs function in figure 3.3 is responsible for reading the protoc complier generated `_pb2_grpc file` and getting all the stubs for the nodes. The Find stub for the node function stores the right stub for all the nodes in the pipeline. This association of the right stub for each node is made by using the `rpc_service_map` which is given by the module mergeproto which is explained in section 3.4

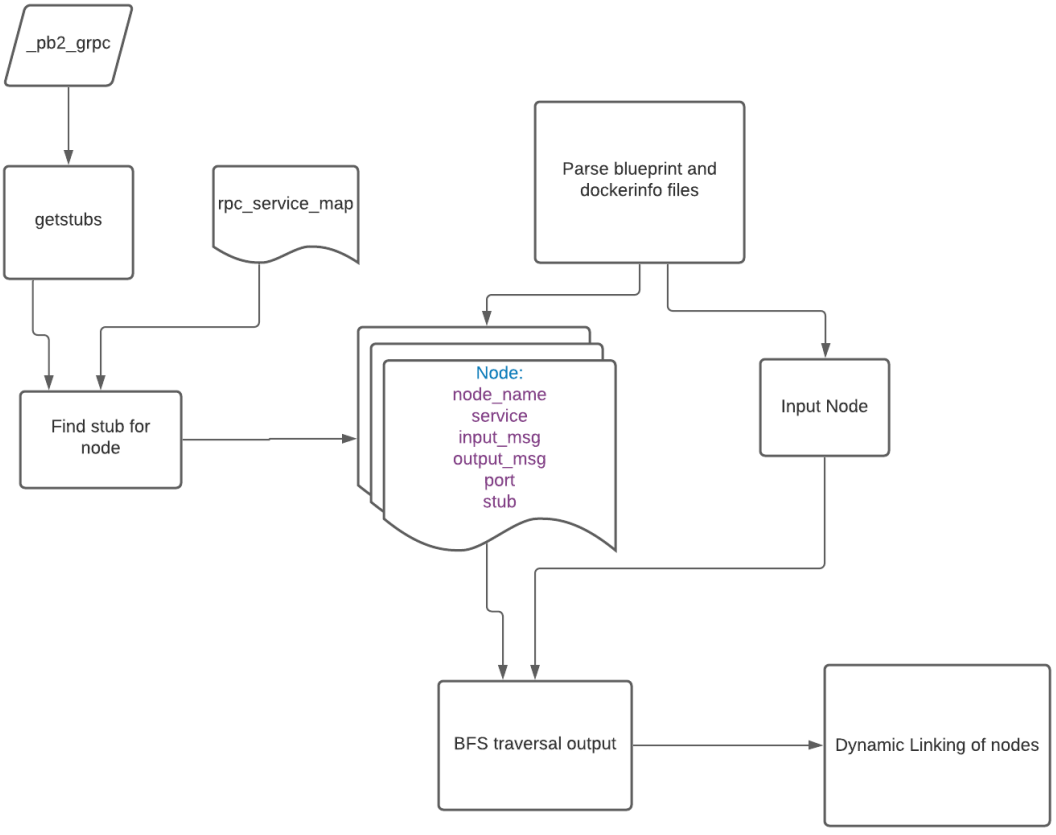

FIGURE 3.3: Execution of the pipeline

The orchestrator parses blueprint and dockerinfo files to get all node information and the input node name. This information will be used to get a BFS traversal output. The orchestrator then dispatches the messages between the nodes by dynamically linking them according to BFS traversal path.

3.7 Use of gRPC for triggering the orchestrator

The basic idea is to have an orchestrator running in a pod in a Kubernetes cluster, just like the other nodes in a pipeline. In this way, a generic orchestrator can handle multiple requests sequentially. However, the problem was how to trigger the orchestrator for each pipeline execution from outside the Kubernetes pod? We solve the problem of triggering the orchestrator by using the client-server architecture of gRPC.

The orchestrator exposes a grpc service like below:

```
message PipelineConfig {
  text blueprint = 1;  //complete blueprint.json as text
  text dockerinfo = 2; //complete dockerinfo.json as text
  bytes protos-zip = 3; //zip file of protos as binary
}

message PipelineStatus {
  int32 statusCode = 1;
  text statusText = 2;
}

rpc executePipeline(PipelineConfig) returns(PipelineStatus)
```

The above protobuf interface for the orchestrator has an input message named Pipelineconfig, which has fields for the blueprint file to be passed as text, dockerinfo file to be passed as text, and a zip file of protobufs for all the nodes passed as binary. The output message is named PipelineStatus and has fields for status code and text description of a status. StatusCode is similar to HTTP status code; for instance, 200 is "OK/Success," and statusText contains "success and the result" or the detailed exception text. For each pipeline run, the orchestrator excepts these three files, i.e., blueprint.json, dockerinfo.json, and a zip file containing all the individual protobuf interfaces of the nodes.

3.8 The flow of the generic orchestrator

In this section, we explain the orchestrator's overall flow using all the concepts explained in the previous section. The flow diagram is shown in figure 3.4. A user first composes a

pipeline in Acumos Visual studio editor AcuCompose, validates the composite solution, and downloads the solution package. Solution.zip contains the following files and folders:

1. blueprint and dockerinfo JSON files
2. A microservice folder which will have all the protobuf definitions of the nodes
3. A deployment folder which has all the necessary Kubernetes artifacts for the nodes and the orchestrator
4. Kubernetes client script
5. an orchestrator client script.

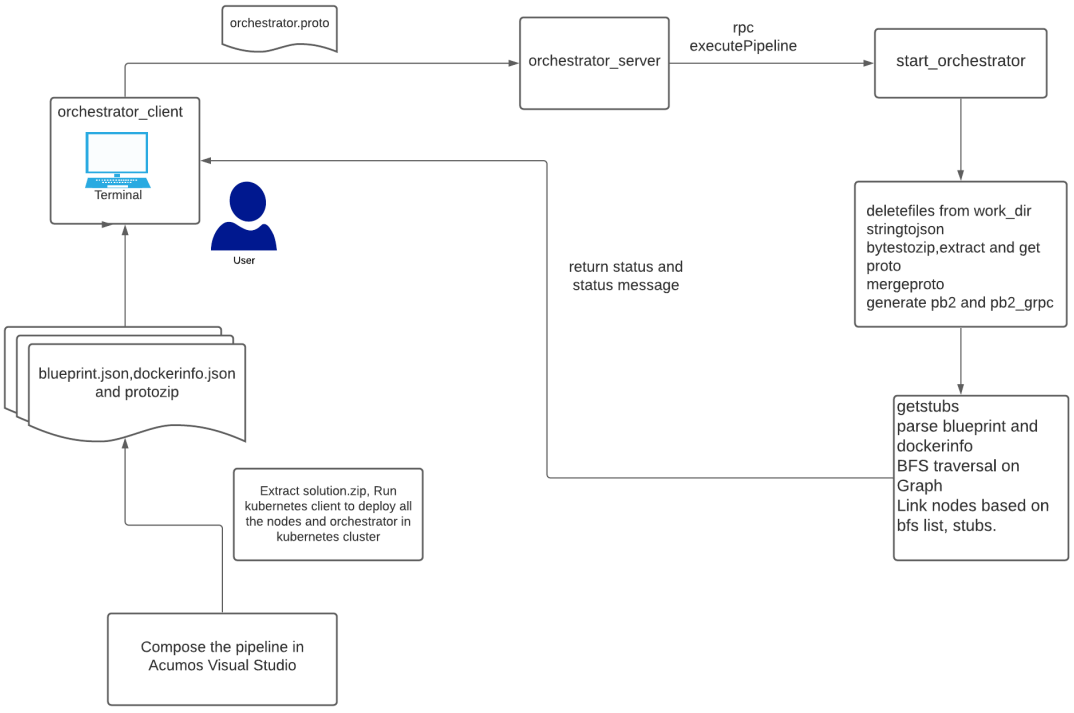

FIGURE 3.4: Triggering the orchestrator

The users executes the Kubernetes client script which deploys all the nodes and the orchestrator server in the Kubernetes cluster. The output of the Kubernetes client script gives the IP address and the port of the orchestrator server.

3.8.1 Orchestrator Client

The orchestrator client will scan all the protobuf files in the solution package's microservices folder and converts them into a zip file format. The orchestrator client then

converts the blueprint and dockerinfo files as text, the zip file of the protobuf files to binary, and sends it over to the orchestrator server according to the protobuf interface as explained in section 3.7. The IP address and port given as output from the Kubernetes client script will be used by the orchestrator client to make a gRPC connection to the orchestrator server.

3.8.2 Orchestrator Server

The Orchestrator server deserializes the blueprint and dockerinfo files back to JSON files and the nodes' protobuf definitions. The orchestrator will then execute a remote procedure call to reach the gRPC service executePipeline. All files received from the client and those generated for the current pipeline are stored in a directory named `work_dir`. The idea behind creating the working directory in the orchestrator's container is to store all the relevant files for a particular pipeline in a single directory. All the files in this working directory will be deleted before each pipeline run, and the new files for the current pipeline will again be generated. In this way, we delete the working directory before each run to have the last run's files for diagnostic means. The gRPC service executePipeline will then call the generic serial orchestrator module. The generic serial orchestrator will do the following:

1. Merges the individual protobuf files of all the nodes and generates the stubs automatically as explained in section 3.4.
2. Parses the JSON files to get the node properties as explained in section 3.2.
3. Stores the node information in a Graph Data Structure and uses a Breath first search traversal to get the traversal path. This is explained in section 3.3.
4. Finally dispatches the messages from one node to another according to the pipeline topology by dynamically linking the nodes in the pipeline. This is explained in section 3.5.

Finally, after dispatching the messages to all the nodes, the orchestrator will return a status code and status message. If the execution was successful, the status code returned is 200. In case of failure, the orchestrator server will return an appropriate exception code and exception message to the client. The output of the orchestrator can be viewed by logging into the bash of the orchestrator Kubernetes pod.

Chapter 4

Experiments

In this chapter, we explain the different experiments conducted throughout the scope of this master thesis. The design of a generic orchestrator capable of running any pipeline built according to AI4EU container specification required some simple and complex scenarios to understand the complications and solve real-world challenges. This chapter demonstrates the various experiments conducted and challenges faced during the design of the generic orchestrator.

Before composing the pipeline in AcuCompose, the individual nodes need to be implemented as a docker container exposing it's gRPC service. Following steps need to be performed to implement a node as docker container exposing it's gRPC service.

1. **Define the service:** Each node performs a specific function in a pipeline. This function can be a data broker service, machine learning, deep learning, or symbolic AI model. The service can be defined in any programming language chosen by the model provider.

2. **Define the messages and services in a proto file:** Each model or a node in a pipeline has to have a protobuf interface which defines the input message, output message and the gRPC service.

3. **Generate the gRPC classes for a programming language chosen to implement the service:** we need to install the required libraries and then call the protoc compiler to generate the needed stubs and skeletons.

4. **Creating a gRPC server:** The server will import the generated files from the protoc compiler and the function created in previous step. Then we will define a class that will take a request from the client or the orchestrator and uses the service function to return a response. After that, we will use `add_Servicer_to_server`

from `model_pb2_grpc` before adding the class Servicer to the server. Once we have implemented all the methods, the next step is to start up a gRPC server so that clients or the orchestrator can call the service. The gRPC server is expected to run on port 8061, and an optional HTTP-Server for a Web-UI for human interaction is expected to run on port 8062.

5. **Include the license file for the model**

6. **Prepare the Docker file** Include all the dependencies and a build a docker image. This docker image will then be used by the Kubernetes client. Kubernetes client takes care of scaling and container port mapping.

4.1 Simple Pipelines

This section explains the experiment done for a simple AI pipeline. Simple AI pipeline in this scenario consists of two nodes, i.e., a data broker node and a simple AI model node that performs the required task. The data broker node is our input node which reads a record from the database and forwards it to the next node. A database can be a simple CSV file or an actual database. In AI4EU experiments, different types of data brokers will be developed for different applications. The data broker and the other model implementations are excepted to be provided by the user. However, this data broker or the model can be published in a public catalog called Marketplace, where it will be accessible to the general public.

4.1.1 House price prediction pipeline

The house price prediction pipeline consists of two nodes, i.e., CSV data broker and a machine learning regression model, which predicts a house price based on the already trained model trained on training data. The CSV data broker reads a record from the test data and the generic orchestrator dispatches the records sequentially to get a prediction for each record. The data broker and the model is implemented by the steps described above. The models are then on boarded to Acumos platform by using dockerized URI method and attaching the appropriate license and protobuf file. The on boarded models are then connected together in AcuCompose to form a pipeline as shown in figure 4.1.

Figure 4.2 shows the message dispatching in the house price prediction pipeline. The generic orchestrator gets blueprint and dockerinfo files as input. It parses both the files to get individual properties and the IP address of the nodes. The orchestrator's IP

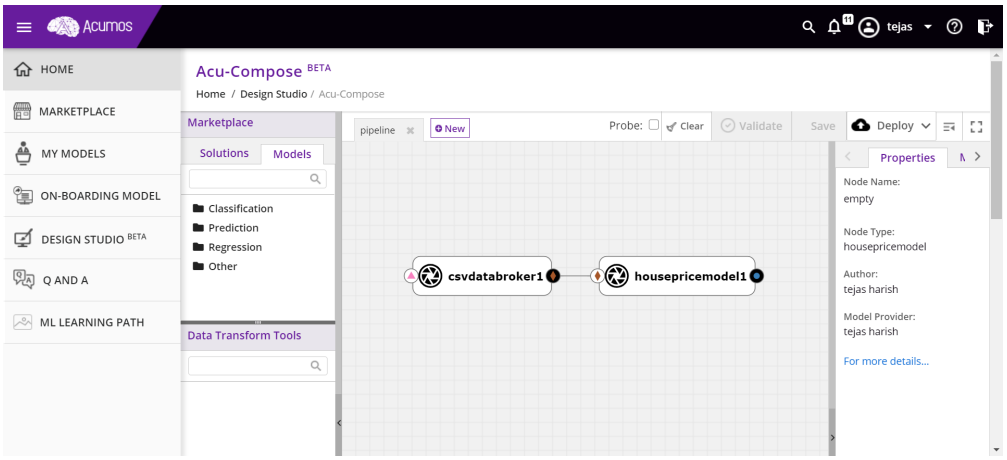

FIGURE 4.1: House price pipeline

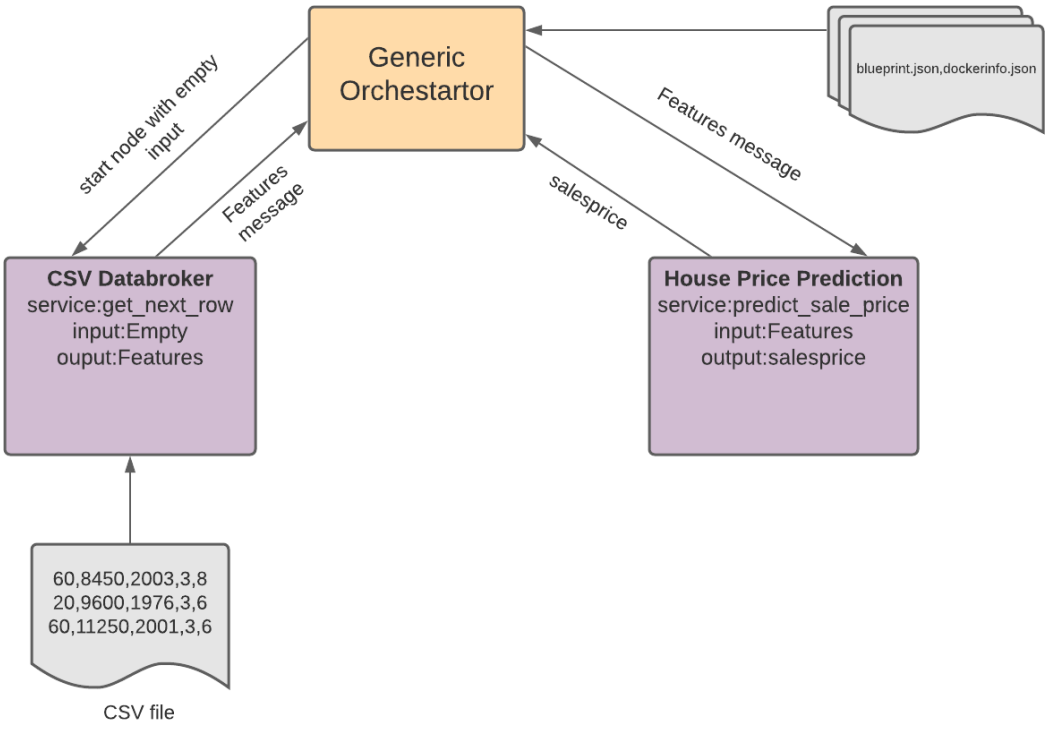

FIGURE 4.2: Message dispatching in House price pipeline

address to reach the nodes and other properties give node names, input message, output message, and services implemented. The node's protobuf files are merged into a single consolidated protobuf file. This consolidated file contains all the messages and services without duplication. The stubs are generated on the fly and are used to call the relevant services of the node. The orchestrator calls the first node, the CSV data broker, with an empty input request. The service `get_next_row` returns the first Features record from the CSV file. The orchestrator dispatches the Features message from the first node to the next node, the house price prediction node. The orchestrator now uses the

prediction node's stub to call the service predict_sale_price, which gives a sales price of the house as a response. The generic orchestrator always dispatches the output of the previous node as an input to the next node. This process is repeated until the CSV data broker is exhausted and has no more records to return. The orchestrator server returns a "success" response to the orchestrator client if the message dispatching is successful or returns the appropriate exception if something goes wrong during the process.

4.2 Advanced Pipeline: Audio Mining Pipeline

This section explains the experiment done for advanced pipelines. An advanced pipeline in this scenario has more than two nodes. We have considered an audio mining example where the whole operation is modeled as four reusable building blocks with clean protobuf interfaces according to AI4EU container specification. The idea behind this audio mining pipeline is to demonstrate that a real-time application such as audio mining can be fragmented into reusable building blocks, connected together in Acumos Visual Studio, i.e., AcuCompose, and to show that the generic orchestrator designed in this master thesis is capable of handling the dispatching of messages according to the topology of nodes. Since the master thesis concentrates on the design of a generic orchestrator, we make use of pre-trained audio models for audio segmentation and converting segmented audio files to text. pykaldi [16] is used for segmentation and automatic speech to text functionality. We explain the protobuf interfaces and how each of the four nodes is implemented according to the gRPC protocol used in AI4EU experiments. The four nodes in the audio mining pipeline are described in the following sub-sections.

4.2.1 Audio Data Broker

The protobuf interface for the audio data broker node is show below.

```
syntax = "proto3";
import "google/protobuf/empty.proto";

message AudioFileJob {
  string job_uuid = 1;
  int64 priority = 2;
  string file_name = 3;
  string work_dir = 4;
  int64 length = 5;
}
```

```
service AudioFileBroker {
  rpc getAudioFileJob(google.protobuf.Empty) returns(AudioFileJob);
}
```

The audio data broker has an input message google.protobuf.empty and the output message AudiofileJob. Since a real-time audio application can require multiple audio files to be processed and it will occupy a large memory space, we are not looking to send these audio files as binary data over the gRPC channel. Instead, we send the file names of audio files along with the working directory where audio files are stored. The field `job_uuid` will contain a unique UUID for each audio file job. The service AudioFileBroker has a single RPC method, getAudioFileBroker, which takes in Empty message and gives AudiofileJob message as the response. The RPC getAudioFileJob will take into account multiple audio files present in the working directory and passes the files sequentially to the orchestrator.

4.2.2 Audio Segmentation

The protobuf interface for the audio segmentation node is as follows.

```
syntax = "proto3";

message AudioFileJob {
  string job_uuid = 1;
  int64 priority = 2;
  string file_name = 3;
  string work_dir = 4;
  int64 length = 5;
}

message AudioSegment {
  string job_uuid = 1;
  string segment_uuid = 2;
  string file_name = 3;
  string segment_file = 4;
  string work_dir = 5;
  int64 index = 6;
  int64 length = 7;
  float start_time = 8;
```

```
  float end_time = 9;
}

service AudioSegmentation {
  rpc getNextAudioSegment(AudioFileJob) returns(AudioSegment);
}
```

The audio segmentation node has the service getNextAudioSegment, which takes input AudioFileJob from the previous node, audio data broker, and responds with AudioSegment message. AudioSegment message has the following fields, and their purpose is listed below:

1. `job_uuid`: Indicates which audiofilejob the current segment belongs
2. `file_name` refers to the original audio wave file.
3. `segment_file` refers to the current segmented file.
4. `work_dir` refers to the working directory
5. `length` field gives the total segments the original wave file is segmented to.
6. `start_time and end_time` indicates the start and end time of the segment

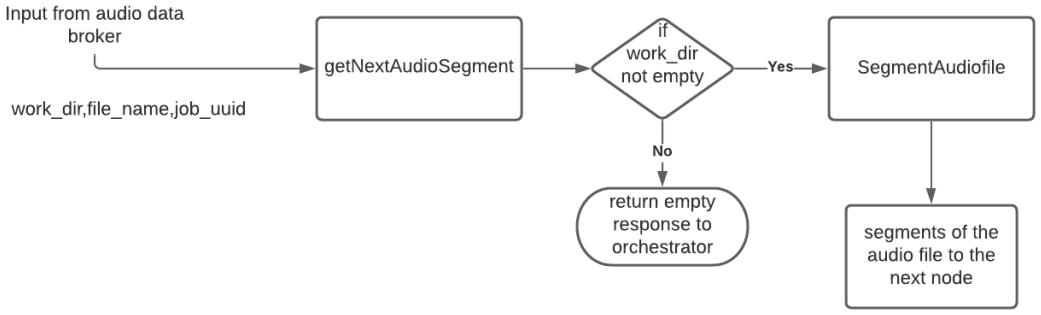

FIGURE 4.3: Audio Segmentation

Figure 4.3 shows the implementation of service AudioSegmentation. The audio segmentation node receives the input AudiofileJob with fields `work_dir`,`file_name` and the job_uuid. The remote procedure call getNextAudiosegment performs the the actual segmentation only if there is a response from audio data broker node. If the there is no response from the audio data broker node, an empty response is returned to the orchestrator and the orchestrator terminates when it receives a empty response from any one if the nodes. The segmentation of the audiofile is done by using pre-trained models from

pykaldi [16] which gives the segments for a audio file by indicating the start time and end time of each segment. The start time and end time is made use to make segmented audio files by making use of pydub python library.

4.2.3 Audio to text

The protobuf interface for the audio to text node is as follows.

```
syntax = "proto3";

message AudioSegment {
  string job_uuid = 1;
  string segment_uuid = 2;
  string file_name = 3;
  string segment_file = 4;
  string work_dir = 5;
  int64 index = 6;
  int64 length = 7;
  float start_time = 8;
  float end_time = 9;
}

message SegmentText {
  AudioSegment segment = 1;
  string text = 2;
  string language = 3;
}

service AudioToText {
  rpc audiototext(AudioSegment) returns(SegmentText);
}
```

The service audiototext takes the input message AudioSegment from the previous audiosegmentation node and responds with SegmentText message. SegmentText message has the following fields, and their purpose is listed below:

1. segment : This field holds the entire message AudioSegment from the audio segmentation node.

2. text : This field contains the converted text information from a segmented audio file.

3. language : This field indicates the language of the audio files.

The remote procedure call audiototext performs the conversion by using pre-trained pykaldi [16] model.

4.2.4 Audio Dialog Creator

The protobuf interface for the audio dialog creator node is as follows.

```
syntax = "proto3";

message AudioSegment {
  string job_uuid = 1;
  string segment_uuid = 2;
  string segment_file = 3;
  string work_dir = 4;
  int64 index = 5;
  int64 length = 6;
  int64 start_time = 7;
  int64 end_time = 8;
}

message SegmentText {
  AudioSegment segment = 1;
  string text = 2;
  string language = 3;
}

message DialogResponse {
int64 status = 1;
}

service AudioDialogCreator {
  rpc addSegment(SegmentText) returns(DialogResponse);
}
```

The service AudioDialogCreator has a single remote procedure call addSegment which takes in SegmentText message from the previous node and responds with a DialogResponse. DialogResponse message has a single field, "status," which will be set to "1" if the RPC call was successful and set to "0" if unsuccessful. The RPC method addSegment creates a text file and writes all the converted text messages received from the audiototext node. This text file will have a consolidated converted text from all the audiofile jobs processed. The role of the audio dialog creator is to arrange the converted text from different segments in the correct order.In this audio mining pipeline, the audio dialog creator node acts as a data sink.

4.2.5 Message dispatching between the audio mining pipeline nodes by the generic orchestrator

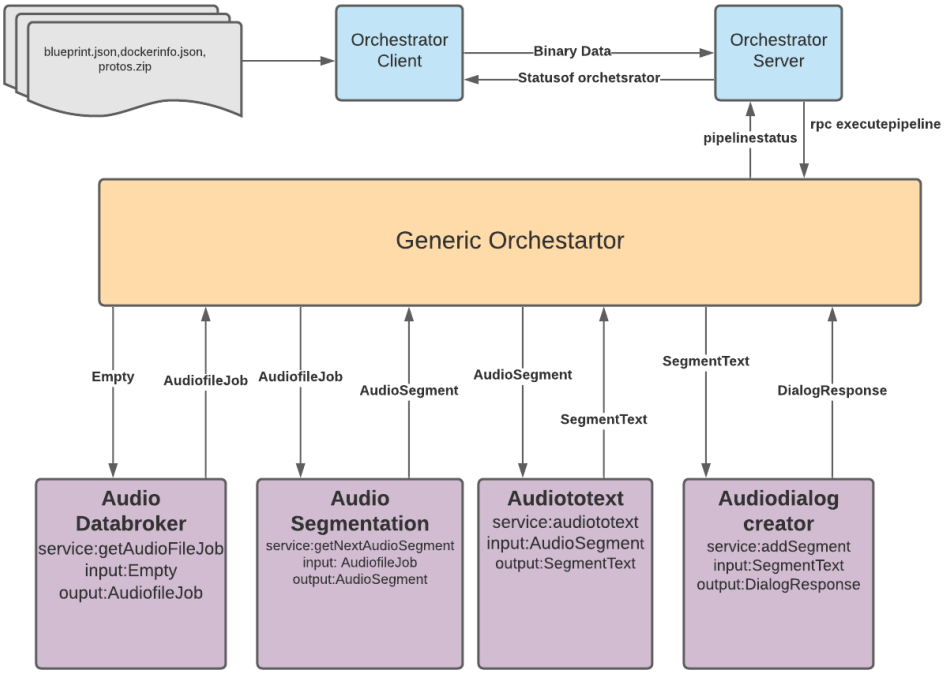

FIGURE 4.4: Message dispatching between the nodes for advanced audio mining pipeline

After writing the protobuf interfaces for each node, implementing the gRPC servers for the node's services, and creating docker images for the nodes, the pipeline nodes are onboarded to Acumos. The pipeline is composed in the visual studio of Acumos, and the solution package for the Kubernetes environment is downloaded. The solution.zip has the deployments for all the nodes, blueprint.json, dockerinfo.json, the Kubernetes client script, and the orchestrator client script. The user needs to execute the Kubernetes client script with a namespace as an input parameter for deploying all the docker images

in a Kubernetes cluster. This target Kubernetes environment can be an actual local Kubernetes cluster, or a Kubernetes cluster created using minikube on the user's laptop for development purposes. After successful deployment, the user needs to execute the orchestrator client script with the orchestrator server's IP address and the port. This IP address and port will be given as output of the Kubernetes client script. The orchestrator client will pass the blueprint, dockerinfo, and the zip file of protobuf files as serialized data over the gRPC call to the orchestrator server. The orchestrator server will, in turn, perform pre-processing steps like deserialization, merging of protobuf files, and automatic generation of stubs and then passes the control to the generic orchestrator module. Figure 4.4 shows the dispatching of messages between the nodes by the generic orchestrator. The orchestrator successfully identifies the starting node or the input node of the pipeline, creates a graph data structure for the nodes, and calls the BFS traversal to get the order in which the message has to be dispatched. The orchestrator then dispatches the output of the previous node as input of the next node. The orchestrator terminates the execution when it receives an empty response from any node or when an exception occurs. Upon successful execution, the orchestrator server sends the status code and the status message to the orchestrator client. The user can view the message dispatching output by logging into the orchestrator's Kubernetes pod. The text file containing the converted audio to text can also be found in the orchestrator's Kubernetes pod.

4.3 Important Challenges faced and solved

In this section we explain briefly the main challenges faced during the gRPC implementation of the nodes and during the design process of a generic orchestrator.

4.3.1 The problem of restarting the pipeline containers

The audio mining pipeline's docker containers running the gRPC servers terminate when the node service returns a response. This termination is not ideal as we need the docker containers always running, and the containers for each node should be able to process multiple requests.

For audio data broker, this problem was solved by scanning the working directory for multiple wave files and storing them in an array. A counter is set to control the array's index, and when all wave files are processed, the data structures used for storing the wave files, the counter variable, and other variables are reset to their default values.

For the segmentation node, the getNextAudioSegment was changed to keep it lightweight. A new function segmentaudiofile was implemented to perform the segmentation for an audio file. This is shown in figure 4.3. The function segmentaudiofile is called only once to get all the audio segments for an audio file. These segments are stored in an array, and the getNextAudioSegment call for the next run will use the segments stored in the array to give the next segment to the orchestrator. The data structures of the node are reset to default after processing an audio file job. The gRPC server is also active after processing all the audio file jobs.

For the audiototext node, new wave files needed to be created by using the start time and end time of the segments. An open-source python library pydub was used to create new segmented wave files by taking the segment's start time and end time.

4.3.2 Search for data broker node in blueprint.json

In the early version of the generic orchestrator, an assumption was made that the first node in the pipeline was always the data broker node. However, this assumption had to be changed to allow cyclic typologies in future. The latest version of the generic orchestrator searches for the data broker node which has the input message with "Empty" or "google.protobuf.Empty" and passes this node as the source node to the Breadth first search algorithm(BFS). The BFS algorithm then gives the traversal path for the orchestrator accordingly.

Chapter 5

Related Work

5.1 Acumos and Design Studio

Acumos is an Open Source Platform that supports the training, integration, and deployment of AI models. Acumos allows data scientists to publish AI models while guarding them against developing fully integrated solutions. A model in Acumos is interoperable with any other microservices of Acumos, regardless of the toolkit (such as TensorFlow, SciKit Learn, H20)used to build it. This interoperability is achieved as the model is packaged into an independent, containerized microservice. Models developed with Acumos supported toolkit and language including Java, Python, and R can be onboarded, packaged, and cataloged. The microservices of Acumos are easy to integrate into real-time applications, for any software developer, without specialized knowledge of data science and AI.[17]

Acumos provides a marketplace for easy collaboration and sharing of artificial intelligence solutions. Development and business teams across one company or across multiple domains can share the solutions securely. The marketplace also allows ratings, descriptions, tagging, licensing options for each model on-boarded on the platform. The platform supports communication between AI experts and software developers that automate the process for model selection while automating error reporting and software updates for models that have been acquired and deployed through the marketplace. AI models can be downloaded from the marketplace, trained, graded on their ability to analyze datasets, and integrated into completed solutions or used to compose pipelines in AcuCompose. The development teams can access encapsulated AI models without knowing the inner details of the models and make connections with various data sources, using a range of data adaptation brokers to build complex applications through a simple chaining process. Unlike other similar platforms, Acumos is not tied to any one runtime

infrastructure or a cloud service. Acumos provides an extensible mechanism for packaging, sharing, licensing, and deploying AI models .They can be published in a secure catalog that is easily distributed between peer systems.

Acumos has a graphical interface called Design Studio for combining multiple models, data brokers into a full end-to-end solution that can be deployed into any runtime environment, such as a cloud services such as Azure, and AWS, a private data center, or a specialized hardware environment designed to accelerate AI applications. Acumos only requires a containerization tool, like Docker, to deploy and execute portable general purpose applications.

AI4EU experiments use the Acumos platform and its features for onboarding and sharing the marketplace models. However, AI4EU experiments go beyond the capabilities of Acumos by supporting not only AI or machine learning models but any model or software built following the container specification as explained in section 2.5. Acumos supports onboarding mainly by using model bundles, and protobuf's are generated automatically. Nevertheless, this automatic generation results in many constraints in the usage of protobufs. Acumos has different model runners or orchestrators for various programming languages supported by it to execute the pipelines. These different model runners have different paths where REST services are exposed and are not standardized. AI4EU experiments, therefore, decided to come up with its own generic orchestrator, which can execute any pipeline composed in the visual studio of Acumos. The generic orchestrator designed in the scope of this master thesis is programming language agonistic and would not have a problem with dependencies. The model provider has all the flexibilities to build the container as needed, including all dependencies, and provide a gRPC service.

5.2 Kubeflow

Kubeflow is a machine learning toolkit for Kubernetes. Kubeflow Pipelines [18] is a platform for building and deploying portable, scalable machine learning (ML) workflows based on Docker containers. A pipeline in Kubeflow is a description of an ML workflow. This pipeline includes all components in the workflow and how they combine in the form of a graph. The pipeline also contains the definition of the inputs required to run the pipeline and the inputs and outputs of each component. A pipeline component is a self-contained Docker image that performs one step in the pipeline.

At an abstract level, the execution of a pipeline in kubeflow proceeds as follows:

1. Python SDK: The SDK is used to create components or build a pipeline using Kubeflow domain-specific language(DSL)

2. DSL compiler: The compiler transforms the pipeline written in Python code into a static configuration (YAML).

3. Pipeline Service: call the service to create a pipeline run from the static configuration.

4. Kubernetes resources: Kubernetes API server is called by the pipeline service to create the necessary Kubernetes resources to execute the pipeline.

5. Orchestration controllers: execution of containers in a pipeline is done by a set of orchestration controllers. Argo Workflow controller is one such orchestrator controller.

6. Artifact storage: The Pods store metadata and artifacts.

7. Persistence agent: The pipeline service creates the Kubernetes resources and is monitored by a persistent agent. It also records the set of containers executed along with their input and output.

8. Pipeline web server: The pipeline web server is used to collect data from different services to display important information such as debugging information, execution status of pipelines.

The Kubeflow Pipelines SDK provides a set of Python packages that users can use to specify and run their machine learning (ML) workflows.

5.2.1 Building pipeline components

Components in a Kubeflow pipeline are containerized applications that perform a step in the ML workflow. Pipeline components can be defined in two ways. [19]

1. A containerized application can be used as a pipeline component. A component specification needs to be created to define a container image as a pipeline component.

2. A python function can be made as a pipeline component. The Kubernetes pipelines SDK enables the conversion of a python function into a pipeline component.

5.2.2 Understanding how data is passed in the Kubeflow pipeline

When Kubeflow Pipelines executes a component, a container image is started in a Kubernetes Pod, and inputs are passed as command-line arguments. The outputs are returned as files. The definition of inputs, outputs, and how they are passed to the program as command-line arguments are present in the specification of the component. Typically, small inputs are passed to the components as values, and significant inputs are passed as file paths. Python function-based components handle the complexity of passing inputs into the component and passing the function's outputs back to the pipeline. [19]

5.2.3 Comparison of Kubeflow with AI4EU experiments

In Kubeflow, the users need to use the Kubeflow Pipeline SDK to write the pipeline components, compile the component using a DSL compiler to get the static configuration, and finally run the pipeline using the Kubeflow Pipeline SDK. In comparison, the generic orchestrator designed for AI4EU experiments in this master thesis automatically generates the communication artifacts on the fly and can execute any generic pipeline which it has not seen before.

Chapter 6

Conclusion and Future work

In this thesis, we have designed a generic orchestrator capable of executing an AI4EU pipeline. The design of the generic solution combines multiple open-source technologies like gRPC communication, protocol buffers, Docker, and Kubernetes. The generic orchestrator creates a valid request, gets the response for each node in the pipeline, and is responsible for dispatching messages between the nodes following the pipeline topology by dynamically linking the nodes. The problem of triggering the generic orchestrator is also solved by using gRPC communication. The files generated by the AI4EU experiments platform after a pipeline is composed in its visual editor are sent over as binary to the gRPC orchestrator server by the gRPC orchestrator client, which will start the generic orchestrator module. The gRPC orchestrator server returns a status code and a status message to the client after executing the pipeline. With the help of gRPC, the generic orchestrator designed is capable of executing any pipeline about which it has no prior knowledge. The usage of Docker as the containerization tool enables nodes/models in a pipeline to be implemented in any programming language and using any libraries and tools. Docker gives flexibility to the creator of the node, and all his needs are nicely encapsulated in a Docker container. The use of Kubernetes as deployment environments enables automatic scaling of the nodes and the orchestrator. The generic orchestrator is not limited to machine learning or deep learning models, but it executes any pipeline composed following the AI4EU container specification. For Experiments and evaluation, we have implemented a simple and advanced pipeline. A simple pipeline has two nodes: a CSV data broker and a machine learning model that predicts the house price. An advanced pipeline is an audio mining pipeline. For this experiment, we have four nodes: an audio data broker, audio segmentation, audio to text, and an audio dialog creator. The generic orchestrator successfully handled the message dispatching and dynamic linking of the nodes deployed in a Kubernetes cluster.

Our work promises several future research directions. As part of Experiments, we have implemented evaluation pipelines, also called production pipelines. In the future, experiments for training pipelines are planned. There is also the scope of having a web-UI for the orchestrator for user interaction. There will be support for the execution of cyclic pipeline topologies as well. The generic orchestrator designed in the scope of this master thesis follows a serial execution pattern. The AI4EU project plans to have several orchestrators for parallel execution, handling streaming and batch processing data. There is scope for supporting the following utility nodes or the infrastructure nodes in the AI4EU pipelines.

1. Splitter Node: The splitter splits a protobuf message to multiple models.
2. Data Mapper Node: Maps multiple protobuf messages coming from multiple models.
3. Diagnostic Node: Connect diagnostic tools likes Tensorboard.
4. Model Initializer: Some models require initialization before it is ready for pipeline execution.
5. Persistent Volumes: Some models/nodes in a pipeline need to share a common file system.

For the models or nodes in the pipeline to connect to the infrastructure/utility nodes, those connections needs to be reflected in the protobuf interface of the models. The orchestrator designed in this master thesis supports the Kubernetes deployment environment. There is scope for designing orchestrators for various other deployment environments as well. A separate orchestrator can be designed for Robot operating system(ROS) and for each cloud provider, such as Amazon Web Services(AWS), Microsoft Azure, Google cloud, GAIA-X, and several others.

List of Figures

List of Tables

Bibliography

[1] Arthur Ryman. Simple object access protocol (soap) and web services. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, page 689, USA, 2001. IEEE Computer Society. ISBN 0769510507.

[2] Authors oracle. Simple object access protocol overview - oracle. https://docs.oracle.com/cd/A97335_02/integrate.102/a90297/overview.htm.

[3] Leonard Richardson, Mike Amundsen, and Sam Ruby. *RESTful Web APIs*. O'Reilly Media, Inc., 2013. ISBN 1449358063.

[4] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. *Doctoral dissertation, University of California, Irvine*, 2000.

[5] Authors gRPC. grpc docs. https://grpc.io/docs/, 02 2019.

[6] Authors docker. Docker overview. https://docs.docker.com/get-started/overview/.

[7] Shuai Zhao, Manoop Talasila, Guy Jacobson, Cristian Borcea, Syed Anwar Aftab, and John F Murray. Packaging and sharing machine learning models via the acumos ai open platform. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 841–846, 2018. doi: 10.1109/ICMLA.2018.00135.

[8] David Kriesel. *A Brief Introduction to Neural Networks*. 2007. URL availableathttp://www.dkriesel.com.

[9] Maysam Abbod, James Catto, Derek Linkens, and Freddie Hamdy. Application of artificial intelligence to the management of urological cancer. *The Journal of urology*, 178:1150–6, 11 2007. doi: 10.1016/j.juro.2007.05.122.

[10] Developers Google. Protocol buffers. https://developers.google.com/protocol-buffers, 07 2008.

[11] Developers Google. Intro to grpc and protocol buffers — by trevor kendrick. https://medium.com/well-red/intro-to-grpc-and-protocol-buffers-c21054ef579c.

[12] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[13] developers kubernetes. Production-grade container orchestration: Automated container deployment, scaling, and management. https://kubernetes.io/.

[14] Martin Welß. Ai4eu container specifications. https://github.com/ai4eu/tutorials/blob/master/Container_Specification/ai4eu_container_specification.pdf, 04 2020.

[15] Authors gRPC. Generated code reference python grpc. https://grpc.io/docs/languages/python/generated-code.

[16] Dogan Can, Victor R. Martinez, Pavlos Papadopoulos, and Shrikanth S. Narayanan. Pykaldi: A python wrapper for kaldi. In *Acoustics, Speech and Signal Processing (ICASSP), 2018 IEEE International Conference on*. IEEE, 2018.

[17] The Linux Foundation ATT. Acumos an open source ai machine learning platform. https://www.acumos.org/wp-content/uploads/sites/61/2018/03/acumos_open_source_ai_platform_032518.pdf.

[18] The Kubeflow Authors. Introduction to the pipelines sdk — kubeflow. https://www.kubeflow.org/docs/components/pipelines/sdk/sdk-overview/, 2018-2021.

[19] The Kubeflow Authors. Building pipeline with sdk — kubeflow. https://www.kubeflow.org/docs/components/pipelines/sdk/sdk-overview/, 2018-2021.