SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

AI-Assisted Scheduling for ADAS Traffic Simulations

Thomas Beranek

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

AI-Assisted Scheduling for ADAS Traffic Simulations

Author: Thomas Beranek
Examiner: Prof. Hans-Joachim Bungartz
Supervisors: Santiago Narvaez Rivas (TUM), Katharina Donauer (BMW Group)
Submission Date: 12.05.2025

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 12.05.2025 Thomas Beranek

Acknowledgments

The topic of this thesis was the result of a joint effort between my colleagues Vasco, Katharina, and myself. Thanks to both, I was able to stay at my dream team at BMW Group, Team Dijkstra, while working on this thesis and continue contributing to its state-of-the-art simulation tools.

I am grateful for the support I received from my academic advisor, Santiago, who, despite his tight schedule, agreed to be my supervisor from the university side. Furthermore, I would like to thank Professor Bungartz for allowing me to do my thesis at his chair and being my examiner.

A great thanks goes to the Thesis Award team at BMW Group, which granted me the status of an internal applicant in relation to my work on this thesis. Thanks to them, plus my colleagues Arun and Jens, I was able to find a suitable permanent position regardless of the challenges facing the German automotive industry at the time of writing. I thank all my colleagues at BMW Group who have supported me on this journey.

Last but not least, I sincerely thank my former colleague, Dianna, for motivating me to pursue an advanced degree.

Abstract

In this master's thesis, a cloud-based architecture including an artificial intelligence (AI) assisted scheduling for cost-efficiently running advanced driver assistance system (ADAS) traffic simulations with the open-source openPASS simulator is designed, implemented, and evaluated. The development of ADAS and other automated driving features requires thoroughly testing the safety of the systems, which cannot be purely achieved by on-the-road testing, and ideally, issues should be discovered as early as possible in the development process. Large-scale traffic simulations are used to test features in virtual models of the cars for millions to billions of kilometers in a fraction of the time and cost of on-the-road testing. The amount of compute resources required is unknown for new simulations, unevenly distributed, and peaks unexpectedly based on demand. Therefore, cloud computing in combination with batch processing is increasingly preferred for running these simulations. Resource requirements for batch jobs are specified at submission, and charges occur based on these provided requirements rather than the actual usage. We introduce a resource usage prediction approach that applies a combination of linear regression models and a vector database with the Amazon Titan Text Embeddings V2 model to accurately predict the memory usage and execution time of new simulations with openPASS. These AI-assisted predictions are used by our scheduling approach to combine simulations to batch jobs of optimal size and selecting the right compute environment before submitting them to AWS Batch. By only using the 64 example configurations supplied with openPASS and a 50% split, our predictions using the combined approach achieved a mean absolute percentage error (MAPE) of 7.4% for memory usage and 74.5% simulation duration. Further analysis revealed a correlation between the matching score of the vector database results, which increases with the filling of our database. Starting at a matching score of 0.85, which applies to the majority of the test dataset, we received a MAPE of 1.8% for memory usage and 21.7% for simulation duration. When integrated into our architecture, our evaluation showed the simulation time of complete batch jobs went beyond the target in the worst case by only 13.3% and the absolute percentage error including incomplete jobs never exceeded 17.4%. Due to the minimum memory requirements of AWS Batch compute environments, the memory usage predictions were not applicable for our implementation.

Contents

1 Introduction

In the early 20th century, the concept of self-driving cars started as an idea in science fiction without any practical implementations. Then in 1925, the first demonstration of a remotely controlled car occurred, and by the 1950s, there were experiments with vehicles guided by wires in the road. The first practical autonomous vehicle was developed by Japan's Tsukuba Mechanical Engineering Laboratory in the 1970s, using a camera system to follow road markings at 20 mph. A decade later, in the 1980s, Mercedes-Benz, in partnership with Ernst Dickmanns and his team from the Bundeswehr University Munich, demonstrated a vehicle that could autonomously drive on an empty highway at almost 100 kph. By 1989, the use of neural networks for controlling autonomous vehicles was pioneered by the Navlab of Carnegie Mellon University. Nine years later, in 1998, Toyota introduced the first Adaptive Cruise Control (ACC) system on a production vehicle. [1]

In the last two decades, more and more Advanced Driver Assistance Systems (ADAS) were introduced to production vehicles, and today there are the first public testings of driverless cars and taxis.

The state-of-the-art production implementation of automated driving at the time of writing is the L3 Personal Pilot in the BMW 7 series. It allows the driver to be distracted from the traffic on highways if there's a leading vehicle and the speed does not exceed 60 kph. Furthermore, it is the only system on the market with seamless transition from the less advanced Level 2 ADAS to Level 3 ADAS. While using the L3 Personal Pilot, the vehicle manufacturer is responsible for any traffic violations, and the driver must only be ready to take over within 10 seconds. It currently is only available in Germany. [2]

In order to ensure the safety and functionality of ADAS before shipping it to customer vehicles, it needs to be thoroughly tested. With increasing autonomy levels, the requirements for these tests increase as well. Ideally, you want to perform tests before even deploying the software to prototype or test vehicles. This can be achieved using ADAS traffic simulations, which are computer simulations allowing you to expose digital twins of the vehicle to predefined or generated traffic scenarios. One example of an ADAS traffic simulator is openPASS, which is jointly developed by different automobile manufacturers and suppliers.

Historically, these simulations were, and in some cases still are, run on on-premise

compute infrastructure, such as own data centers. However, under these conditions, hardware is often not fully utilized or becomes a bottleneck as technology continues to evolve and workloads increase over time. For that reason, also the scientific computing community has started transitioning to partially or completely using cloud computing for High Performance Computing (HPC).

When using conventional software, which is unaware of the cloud environment, the right instance or virtual machine needs to be chosen with the right amount of resources. The resource requirements for new simulations, such as new traffic scenarios, are usually not known beforehand. Thus, simulation engineers often make an educated guess about resource usage when triggering the first run of a new simulation. Both picking too much and too little wastes resources and results in unnecessary costs. The former underutilizes resources, while the latter one causes the simulator to fail, often without an intermediate result.

In this thesis, we introduce and implement a cloud-based architecture with AI-assisted scheduling for cost-efficiently running ADAS traffic simulations using openPASS. First, we explain the technical background behind autonomous driving and ADAS simulations. Second, we present related works about optimizing and predicting resource usage in the cloud and for simulations. Third, we describe the complete architecture of our cloud-based openPASS simulation service. Fourth, we analyze the resource usage of the simulator, its correlation to the input configurations, the overhead of our implementation, and the minimal achievable costs. Fifth, based on the data analysis, we propose different machine learning model architectures and benchmark them against each other to choose our final prediction approach. Afterwards, we evaluate our implementation and scheduling by comparing its performance to the optimal solution determined during data analysis. Finally, we summarize the results and discuss future work.

2 Technical Background

Our work requires a basic understanding of automated driving systems, the standards for their simulation and the openPASS simulator. In this chapter, we introduce all the necessary principles and terminology.

2.1 Automated and Autonomous Driving

Automated Driving refers to the use of Driver Assistance Systems (DAS), which partly or fully take control of some or all the vehicle's driving and parking features. There are different levels of Automated Driving with Autonomous Driving being the highest level. We introduce these levels below as a necessary prerequisite for understanding the DAS subcategory of Advanced Driver Assistance Systems (ADAS) and their complexity.

2.1.1 SAE Levels of Driving Automation

The SAE, a global association of engineers and technical experts in the aerospace, automotive, and commercial-vehicles industries, defines six levels of driving automation. Features in Levels 0 to 2 are only considered driver support features while Level 3 and higher are automated driving features. [3]

In the following, we explain each level in principle and the properties DAS that belong to them need to fulfill.

Level 0: No driving automation

Assistance features limited to warnings or momentary assistance such as the acoustic parking assistant and Automatic Emergency Braking (AEB).

Level 1: Assisted Driving

DAS providing either steering or braking/acceleration support to the driver like the Lane Keeping Assistant (LKA) or Adaptive Cruise Control (ACC).

Level 2: Partly Automated Driving

Extension of Level 1 where multiple features can be used in parallel instead of exclusively. These features are considered Advanced Driver Assistance Systems (ADAS). When ADAS are active the car may already entirely maneuvers on its own, but the driver must remain focused, be ready to take over anytime and is legally responsible.

Level 3: Highly Automated Driving

Starting with this level no person is driving the vehicle when the feature is active. However, when the feature requests, the person in driver's seat must take over within a predefined time frame. While the feature is engaged, the vehicle manufacturer takes responsibility of the vehicle's actions. Systems in this category can only be activated under limited conditions and will not operate unless all required conditions are met.

Level 4: Fully Automated Driving

Extension of Level 3 where no occupant of the car is driving at any point in time or needs to take over. The system is still restricted to certain conditions and for instance may refuses to leave a certain area.

Level 5: Autonomous Drivng

The feature drives under all conditions.

2.2 ADAS Traffic Simulations

Advanced Driver Assistance Systems (ADAS) traffic simulations are virtual simulations of interactions between traffic participants for assessing ADAS and their effects. The simulations are usually scenario-based, contain one or multiple ego vehicles and an arbitrary amount of surrounding traffic participants. Scenarios can be synthetic, generated or based on recordings of real traffic scenarios such as accidents or test drives. Maps define the road network and terrain used by the scenarios. The former and latter are both defined using standardized formats to ensure interoperability between different simulation software.

2.2.1 ASAM OpenX Standards

The Association for Standardization of Automation and Measuring Systems (ASAM) defines standards for automotive development test and validation. Their standards

for simulation are called the ASAM OpenX standards. We focus on the standards for defining traffic scenarios and road networks as these are used by the openPASS simulator utilized in this work.

ASAM OpenSCENARIO XML

OpenSCENARIO XML defines a file format based on XML for describing traffic scenarios. The file extension is xosc. [4]

ASAM OpenDRIVE

OpenDRIVE defines a file format based on XML for describing road networks, also known as the map. The file extension is xodr. [5]

2.2.2 Functional Mock-up Interface

In the context of traffic simulations, the Functional Mock-up Interface (FMI) defines a standardized format for providing traffic participants, such as vehicles, in a container for loading during simulation runtime. These containers are called Functional Mock-up Unit (FMU). For example, when testing a new ADAS, it is usually be provided as a so called virtual car FMU. By being able to load it dynamically, no changes are required to the simulator. [6]

FMUs from the perspective of the simulator or user are a black box with own, additional resource requirements.

2.3 openPASS Simulator

The open Platform for Assessment of Safety Systems (openPASS) is an open-source ADAS traffic simulator, which is jointly developed by different automobile manufacturers including BMW. In the following, we explain the input configurations, its modularity, the concept of agents and ego entities, plus the basic principles of an openPASS simulation.

2.3.1 Configs

The input configurations of openPASS consist of multiple files, which all need to be placed in the configs folder before execution. The required and most important files are listed in table 2.1 below. Any paths used in config files are interpreted relative to SimulationConfig. Thus, it is recommended to also place in the configs folder any additional files used by configurations, such as FMUs.

openPASS Input Configuration Files		
Name	Format	Description
SimulationConfig	XML	Defines the high level parameters of a simulation invocation, such as the environment, random seed, number of invocations, which spawners to use and the file paths of the ProfilesCatalog and Scenario files.
Scenario	OpenSCENARIO (XML)	Defines which entities are doing what, when and where in the Scenery referenced by file path.
Scenery	OpenDRIVE (XML)	Defines the map or world where the scenario is taking place.
ProfilesCatalog	XML	Defines profiles, spawners, traffic groups and traffic rules for the scenario.
SystemConfigBluePrint	XML	Defines dynamic agents to be used in the ProfilesCatalog, the modules they consist of and their parameters.

Table 2.1: Input configuration files of the openPASS Simulator.

2.3.2 Modules, Agents and Ego Entities

OpenPASS is modular, custom modules are provided as shared libraries in its libs folder. Agents are the mechanisms/systems that control entities and consist of multiple modules connected by signals and modules can be composed of submodules. One special agent is the ego agent, which controls the ego vehicle or entity. The ego entity is the system under test, for instance the virtual model of an autonomous driving car which is to be evaluated in the simulation.

2.3.3 Simulation Basics

OpenPASS in principle loads input configuration files, performs the simulation accordingly, writes the results and terminates. Afterwards, the simulation run can be visualized using a separate tool called opVisualizer. The simulation can contain any number of ego vehicles. Simulations are always executed completely, partial executions are not possible.

For more information check out the official documentation of openPASS. [7]

3 Related work

Our research topic of optimizing the scheduling and resource allocation in the cloud for a single computer traffic simulator is a niche topic. To the best of our knowledge, there are no directly comparable scientific publications at the time of writing. Nevertheless, there is plenty of research about efficient resource allocation for cloud workloads in general based on time-series predictions. Furthermore, related work with regard to computational fluid dynamics (CFD) simulations and the partitioning of vehicle traffic simulations exists. In the following, we give an overview of related publications in the aforementioned areas.

3.1 Time-series Resource Usage Prediction

Time-series predictions forecast the values of one or multiple variables in a time-range or at a certain point in time based on historical data. When applied to resource usage, it helps commissioning and decommissioning the right amount of resources on a macro-level at the right time to prevent bottlenecks and reduce costs. It does, however, not assist in determining the micro-level resource requirements of a specific unknown workload as we want to achieve. Still, it can be used to achieve further cost optimization on top of our approach.

Agomuo et al. [8] use a combination of machine learning techniques and optimization approaches for finding the optimal resource allocation in cloud computing environments. They utilize Long Short-Term Memory (LSTM) networks for forecasting resource demands, Particle Swarm Optimization (PSO) for initial resource allocation, Q-learning for dynamic resource adjustment, and Linear Regression (LR) for predicting energy consumption. Their LSTM reaches an overall accuracy of 83%, both PSO and Q-learning show efficiency improvements over 100 iterations. The PSO effectiveness of resource allocation increased steadily, while Q-learning showed some fluctuations.

Al-Asay et al. [9] propose using Diffusion Convolutional Recurrent Neural Networks (DCRNNs) for forecasting the CPU usage for Virtual Machines (VMs) in cloud data centers. They observe that their approach outperforms other deep learning approaches existing in 2021, namely Deep Belief Networks (DBNs), in both Mean Absolute Percentage Error (MAPE) and Root-Mean-Square Error (RMSE) for the PlanetLab dataset.

Bi et al. [10] present a hybrid prediction model named VAMBiG that integrates Variational mode decomposition, an Adaptive Savitzky-Golay (SG) filter, a Multi-head attention mechanism, Bidirectional and Grid versions of LSTM networks. The model predicts the number of workloads, plus CPU and RAM usage for cloud data centers (CDCs).

Bi et al. [11] introduce a new forecasting method for multidimensional time series resource usage data of CDCs called FISFA for short. FISFA stands for Forecasting method based on the Integration of a SG filter, a FEDformer model, and a FECAM. In addition, they use genetic simulated annealing-based particle swarm optimization (GSPSO) for optimizing the hyperparameters, which is based on an integration of PSO, simulated annealing (SA), and genetic algorithm (GA). With their approach, they achieve improvements of the prediction accuracy by 32.14% against LSTM, 25.49% against transformer, and 27.71% informer methods.

Dogani et al. [12] employ a hybrid deep learning model for multi-step workload prediction. Their model combines a convolutional neural network (CNN) and gated recurrent unit (GRU), where the latter one is optimized using an attention mechanism. They show that their model achieves better accuracy compared to baseline methods in previous research at the time, with improvements ranging from 2% to 28% for Google cluster data.

In their 2017 article [13], Gupta and Dinesh were among the first to propose using multivariate and bidirectional LSTMs for predicting the resource usage of cloud workloads. They demonstrate the superiority of stateful over stateless, multivariate over univariate, and bidirectional over unidirectional LSTMs based on the Google cluster trace and RMSE of their predictions.

Follow-up publication [14] introduces sparse bidirectional LSTMs for cloud resource usage prediction with a new method for building this kind of model. Their new method allows building sparse bidirectional LSTMs such that they support fast online adaptation while maintaining comparable or achieving better prediction abilities. The gradient descent (GD) and Levenberg-Marquardt (LM) adaptation algorithms are used for performing the online adaptation.

Li et al. [15] experiment with a combination of bidirectional LSTM and GRU for predicting the CPU usage of cloud instances. They take advantage of the high accuracy of bidirectional LSTMs and the short prediction time of GRU. For the Google trace dataset, the combined model achieves lower RMSE and MAPE than GRU, unidirectional and bidirectional LSTM as well as a combination of auto-regressive and moving average model (ARIMA) and LSTM. However, it only outperforms ARIMA-LSTM in prediction time.

Nawrocki et al. [16] introduce the Self-Adapting Data-Driven Prediction System (SA-DDPS). Instead of using a single machine learning (ML) model, SA-DDPS first

selects the most suitable from multiple models and then uses it to perform the prediction. This preselection process could also be applied to traffic scenarios as scenarios with certain characteristics, like using virtual car FMUs, may benefit from a specialized model.

Nawrocki et al. [17] present an approach for forecasting long-term cloud resource usage in HPC. While other work focuses on dynamic allocation of resources, they predict the optimal amount of reserved compute capacity. Cloud computing providers offer rebates for reserving compute instances, thus long-term forecasting enables further cost optimization.

3.2 Simulation Resource Usage Prediction

Ladd et al. [18] optimize the cloud resource usage of hemodynamic simulation by predicting the performance of the input on different compute instances. They estimate the simulation throughput of each instance in millions of fluid point updates (MFLUPS) and use this metric to determine runtime and costs. Based on runtime and instance costs, the most efficient instance is chosen. While their solution is not directly applicable to our traffic simulations, the approach of using an objective performance metric, such as simulation throughput, for estimating the runtime of new simulations on different instance types without executing them is worth considering.

3.3 Traffic Simulations

Siguenza-Torres et al. [19] introduce a multilevel heterogeneous performance-aware re-partitioning algorithm for microscopic vehicle traffic simulations called ENHANCE. They use the parallel agent-based microscopic traffic simulator CityMoS, which, unlike openPASS, supports multi-threading and distributed simulation execution. The distribution is achieved by partitioning the road network by cutting roads and moving traffic between partitions using inter-process communication with MPI.

4 Architecture

Our architecture in principle is derived from our two main use cases of running openPASS locally and on a remote datacenter with batch processing. In the first use case, which is the more common one, the user directly executes the simulator exclusively on their local machine via the command line for one input configuration at a time. On the contrary, for the second use case, the user utilizing a script submits multiple configs of their choosing as a batch job to a remote batch processing service, periodically checks the status, and downloads the simulation outputs on completion.

We combine the user experience of using openPASS directly on their system with cost-optimized batch processing by introducing a client with a similar command-line interface to the original openPASS simulator and an Application Programming Interface (API) abstracting the remote batch processing logic.

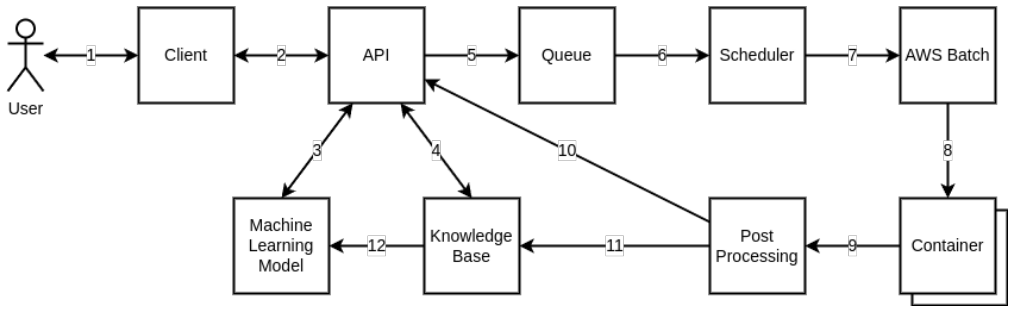

Figure 4.1: Overview of the high-level architecture and service usage.

Figure 4.1 depicts the resulting use case combined with the high-level architecture. The User in the bidirectional relation 1 executes the Client via command line and specifies the path to a folder containing a single or multiple input configurations. In return, they receive the status of the simulation execution(s) and simulation output on completion. In bidirectional relation 2, the Client submits the configs to the API, periodically checks the status of their execution, and eventually receives the simulation output on successful completion. The API requests and receives resource usage estimates for new, unknown simulation configs via 3 and 4, using a Machine Learning Model and Knowledge Base, respectively. Through unidirectional relation 5, the configurations are submitted as separate simulation jobs to a Queue with a final resource usage

estimation each. Our Scheduler periodically takes simulation jobs from the Queue via 6, combines them to optimal batch jobs, and submits these in 7 to AWS Batch, a batch processing service. AWS Batch temporarily commissions new compute resources based on requirements set by the Scheduler and uses them to run the batch jobs in an individual container each through 8. On completion, each Container triggers the Post Processing via 9, which updates the simulation execution status and makes the simulation output available to the API through 10. In unidirectional relation 11, the actual recorded resource usage for each simulation is submitted to the Knowledge Base. Finally, via 12, the Machine Learning Model is periodically updated using the Knowledge Base, improving its performance.

In the following sections, we elaborate on the implementation of the individual components introduced in Figure 4.1. All cloud resources are defined and deployed to a single AWS account using Terraform, which is a Infrastructure as a Code (IaaC) tool. If not explicitly specified, the programming language used is Go.

4.1 Client

Our client is a single executable program, which uses a command line interface similar to the opSimulation executable provided by openPASS with extensions for selecting multiple simulation input configurations at once. We use the AWS SDK to communicate with our API, the AWS Go SDK v2 to be exact, as we implement the client in the Go programming language. The AWS SDK allows programmatic use of all AWS services, while taking care of authentication via its Identity and Access Management (IAM) service. Data transfers are reduced by not uploading configs every time and instead submitting the SHA256 hashes of the configs to the API. Configs are only uploaded if requested to do so by the API. AWS Simple Storage Service (S3) is the primary cloud storage solution of AWS and used for uploading and storing the configuration files. Its instances are called buckets and files considered as objects. The client periodically checks for the status of all still uncompleted simulations and notifies the user about status updates and where to find the output of completed simulations.

4.2 API

The API component uses the AWS API Gateway service in combination with an S3 bucket and several AWS Lambda functions. API Gateway provides the API endpoint and internally redirects requests based on their path and other criteria to other AWS services. The endpoint is protected using the default AWS IAM authorizer. In our case, these other services are different Lambda functions, which are minimal pre-warmed

containers running a single executable on demand. The executable contains a function that must follow a pre-specified interface. Our API processes three different types of requests: Config Upload, Simulation Submission, and Status Check. These are introduced in the following subsections.

4.2.1 Config Upload

As stated in section 4.1, configuration files are directly uploaded to an S3 bucket. The bucket has a trigger attached, which executes a Lambda function every time a file is uploaded. This function automatically extracts quantitative features from the files and stores them in different DynamoDB tables based on the file type for later use by the Machine Learning Model. Amazon DynamoDB is a serverless NoSQL database requiring only a partition key, which optionally can be combined with a sort key, sometimes referred to as a range key. In our case, the partition key is the SHA256 hash of the file the features belong to. Having no fixed schema, we can introduce new or omit columns at any time.

The feature extraction process is described in more detail in chapter 5.

4.2.2 Simulation Submission

Simulations are submitted using their config's SHA256 hashes, referred to in the following as config identifiers or ConfigIDs. These requests are processed using a AWS Lambda function. If a ConfigID is unknown, the caller is requested to upload its files to the previously introduced S3 bucket and submit it again. For already available configs, the Machine Learning Model is invoked using the extracted quantitative features providing a resource usage estimate. In addition, the Knowledge Base is queried for each config providing a second estimate. The exact process is determined and introduced in chapter 6. All available configs with their respective final resource usage prediction are submitted to the simulation job queue. The Lambda function returns the unique identifiers for the simulation jobs submitted allowing later querying for their status.

4.2.3 Status Check

Expects a list of simulation job identifiers and returns their status plus where to find their output files if available. Like the simulation submission request, the status check logic is implemented using a Lambda function. The status and file locations are obtained through querying the DynamoDB tables filled by the Container and Post Processing components, which are introduced in sections 4.8 and 4.9.

4.3 Machine Learning Model

We initially build and train the model locally in a Python Jupyter notebook with scikit-learn. Based on the notebook, we create a Python script that can run unsupervised and provide it as a Docker image. Using a EventBridge rule, we periodically create a SageMaker training job that checks for updates in the Knowledge Base and retrains the model. SageMaker is an AWS service providing various cloud resources for training and hosting machine learning models. The models created by the training job are stored in a S3 bucket and added to SageMaker. We deploy them via a SageMaker Endpoint, which like an API can be invoked with the input parameters and returns the predicted output parameters, in our case the resource usage predictions. The model selection process is described in detail in chapter 6.

4.4 Knowledge Base

The knowledge base component consists of a vector database and a text embedding model. A vector database is a database specialized for efficiently querying its entries based on a vector column, in particular finding the entries with the vectors closest to a specified one. Using a text embedding model, any text-based input can be converted to a vector that relates to its meaning, also known as a semantic vector. By converting the input configurations to a single semantic vector, including it for storing recent resource usage measurements from previous simulation runs, the resource usage of the most similar known config can be derived for a new yet unknown one as a prediction. We discuss this in more detail in chapter 6.

Our implementation uses Amazon OpenSearch Serverless as a vector database, which is a modified, more cost-efficient variant of OpenSearch especially made for AWS.

4.5 Queue

We define a single Amazon Simple Queue Service (SQS) FIFO queue for collecting simulation jobs before bundling them into batch jobs. In production use, this queue uses content-based deduplication, meaning any unique configuration or ConfigID only occurs once in the queue. For testing purposes and due to our limited number of testing configurations, deduplication is disabled in the following.

Each configuration is stored as a config id with its estimated resource usage and duration, to speed up the later scheduling process.

4.6 Scheduler

Our scheduler consists of a single Lambda function consuming the queue introduced before. It is triggered by an EventBridge rule. Amazon EventBridge is an AWS service that allows defining rules which can be activated by predefined events and result in predefined actions being taken, such as invoking a Lambda function. We configure a time-based rule, which triggers our scheduling function every 15 minutes.

The function, when invoked, continuously reads and removes items from the queue for a certain amount of time, by default one minute. This is due to the restriction that only 10 messages can be read from the queue at a time. Using the metadata provided about the duration of each config, the amount of batch jobs is determined based on the desired batch job duration. For simplicity and with focus on performance, we do not attempt a perfect bin packing strategy; instead, we fill a batch job until it reaches the desired batch job duration. The final batch job is filled with all remaining simulation jobs. Resource requirements for batch jobs are set based on the maximum resource requirements of the included simulation jobs. Depending on the total amount of batch jobs, they are either submitted to the EC2 or Fargate batch job queue in AWS Batch. The exact scheduling strategy is determined in chapter 5.

4.7 AWS Batch

For the actual batch processing of the jobs created by our scheduler, we use the AWS Batch service. The service primarily consists of the following three types of resources:

Job Definition Defines a template for a batch job with its default parameters. Contains information about the required compute environment, required resources, container image, permissions, environment variables and other information based on the use case.

Job Queue Defines a queue to submit batch jobs to. Each has a priority and at least one target compute environment.

Compute Environment Defines compute resources for executing batch jobs. Can be either managed or unmanaged Elastic Cloud Compute (EC2), Elastic Container Service (ECS) and Elastic Kubernetes Service (EKS) resources.

It assigns the batch jobs in a job queue to an available compute resource based on the job definitions. In case of a managed compute environment, AWS Batch also takes care of the provisioning and decommissioning of resources based on demand. Charges only occur for the compute resources.

We created a job definition and job queue each for our compute environments, which are managed EC2 and Fargate (a ECS variant) with on demand pricing. As previously stated in section 4.6, the scheduler takes care of choosing the right compute environment by assigning the job accordingly.

4.7.1 Elastic Cloud Compute (EC2)

The Elastic Cloud Compute (EC2) service provides VMs on-demand with different resource configurations and underlying hardware. These VMs are called EC2 instances and charged based on the type of instance, the resource configuration, the time running and network traffic. There are also so called metal instances that occupy a whole physical server with limited virtualization. Instances can be charged at lower costs on-premise, if long time use of a specific instance or instance type is known beforehand. Furthermore, for interruption-proof workloads there are the spot instances, which offer lower costs than on-demand instances but might terminate at any time. The minimum time charged for on-demand pricing is one minute with any additional time rounded up to the nearest second.

4.7.2 Elastic Container Service (ECS) and Fargate

Amazon Elastic Container Service (ECS) is a fully managed container orchestration service, which can be used in combination with AWS Fargate to run containers serverless. Fargate offers a secure isolation between containers and charges based on the runtime of the container and the resource requirements specified. Like on-demand EC2, the minimum time charged is one minute with any additional time rounded up to the nearest second.

4.8 Container

Workloads for batch jobs in AWS Batch are provided as Docker container images. We use a single container image with an entry point, which is a script or program executed at container start. Our base image used is Ubuntu 22.04 and it comes preinstalled with openPASS 1.1.0. The entry point receives a list of config identifiers via the batch job metadata and performs the following steps for each config:

1. Obtain config files from S3 buckets.
2. Start the openPASS simulator and record simulation duration.
3. Read resource usage via Rusage syscall and record it to DynamoDB table.

4. Copy output to new folder in designated S3 bucket.

5. Clear working directory.

The container image is provided to AWS Batch and its compute environments through the Amazon Elastic Container Registry (ECR).

4.9 Post Processing

Our post processing consists of multiple Lambda functions, a EventBridge rule each and has its own S3 bucket plus DynamoDB tables. The rules react to any state changes in the compute resources managed by AWS Batch and their Lambda functions record the timestamp of their occurence to the DynamoDB tables. Periodically or on request, a Lambda function updates the knowledge base with new simulation runs, their config's semantic vector and the measured resource usage. For evaluation purposes we also have a Lambda function that based on the recorded state changes calculates the compute cost of historic runs and generates CSV files, which are stored to the S3 bucket.

Through the updates about state changes inserted to the DynamoDB, the API can determine when a simulation job is completed and where to find its output.

5 Data Analysis

In this chapter, we first present the features extracted by our implementation from the openPASS input configurations. Second, we run openPASS configs under different resource requirements with each of the batch service compute environments introduced in section 4.7. The 64 example configurations delivered with openPASS are used in this process. Third, we separately analyze the results for both compute environments. Fourth, we compare both results and draw conclusions for our AI-assisted scheduling.

5.1 Setup

5.1.1 Feature Extraction

We restrict our extraction process to the Scenario and ProfilesCatalog files as the other config files do not differ significantly between the example configurations. The features are primarily quantitative with few binary ones indicating whether certain features are used or not.

Scenario

The following features are extracted from the Scenario:

NumEntities The amount of entities defined for the whole scenario.

Init.NumActors The amount of actors defined in the initialization story.

Init.Actions.* Contains the amount of actions in the initialization story, plus information about how many actions of which type (i.e., LongitudinalAction) and subtype (i.e., SpeedAction) are defined.

Stories.Num The amount of stories defined in the story board.

Stories.Acts.* Contains the total amounts of acts, maneuver groups, actors, maneuvers, events, and executions in the complete story board.

Stories.Acts.AvgAct.* Contains the average amount of maneuver groups, actors, maneuvers, events, and executions in all acts.

Stories.Acts.Actions.* Contains the amount of actions in all stories, plus information about how many actions of which type (i.e., LongitudinalAction) and subtype (i.e., SpeedAction) are defined.

StopTrigger.Has Indicated whether the scenario has a stop trigger with SimulationTimeCondition.

StopTrigger.SimTime The time given by the SimulationTimeCondition.

ProfilesCatalog

The following features are extracted from the ProfilesCatalog:

Spawner.PreRun.SpawnZones.* Contains the amount of PreRun spawn zones and details about the size of the first spawn group.

Spawner.PreRun.TrafficGroups.* Contains the amount of PreRun and the average values of the parameters controlling their velocity and distribution.

Spawner.Runtime.SpawnZones.* Contains the amount of Runtime spawn zones and details about the size of the first spawn group.

Spawner.Runtime.TrafficGroups.* Contains the amount of Runtime and the average values of the parameters controlling their velocity and distribution.

5.1.2 Data Collection

We acquire data for our analysis by executing all 64 example configurations delivered with openPASS with different resource restrictions. As stated in chapter 4, our architecture enables automatic recording of any resource usage metrics and dataset creation on demand. Therefore, except for submitting the configs to the simulation job queue no additional setup is required. We do so using a simple Lambda function that submits all example configurations with the resource requirements and force the usage of a specific compute environment. The compute environments of AWS Batch have different restrictions for resource requirements. Thus, we have to handle the execution of the example configurations separately for each compute environment.

AWS Fargate

Fargate allows using fractional vCPUs of either 0.25 or 0.5, however there are specific values the memory must take based on the amount of vCPUs [20]. At this point we do not know the exact resource usage of the example simulations. Thus, we attempt

using the minimum memory requirement of 512 MiB per 0.25 vCPUs. This results in the following resource configurations:

0.25 vCPUs 512 MiB

0.5 vCPUs 1024 MiB

1 vCPUs 2048 MiB

2 vCPUs 4096 MiB

4 vCPUs 8192 MiB

AWS EC2

The AWS EC2 compute environment allows a more flexible selection of the memory requirement; however, only full vCPUs can be assigned. Furthermore, the underlying EC2 instance types have at least 2 GiBs of memory per vCPU core [21], resulting in the same minimum requirements as for Fargate. We use the following resource configurations:

1 vCPUs 2048 MiB

2 vCPUs 4096 MiB

4 vCPUs 8192 MiB

We restrict AWS Batch to only use instances of type C5 for our experiment. The service automatically picks the right size of C5 instance based on the submitted batch jobs.

5.2 Feature Correlation

We analyze the correlation between memory usage and extracted features as well as duration and extracted features. The results for EC2 and Fargate are equal with regard to these correlations. All correlations for single feature are less than 0.5 except for the amount of lane changes and lateral actions to the memory usage, which is 0.56. Thus, no single feature is sufficient for accurately predicting memory usage or execution time. The correlation between memory and duration is 0.43 for Fargate and 0.56 for EC2, however this difference of 0.13 is due to hardware difference between Fargate and our C5 type instance used for the EC2 environment.

5.3 Multi-Threading / VCPUs

We observe the accumulated simulation duration in relation to the amount of vCPUs provided for the Fargate and EC2 compute environment. As depicted in figure 5.1, there's no performance benefit for having more than one vCPU. This is due to the lack of multi-threading support by openPASS at the time of writing. Therefore, we do not need to predict the ideal amount of vCPU based on the input configurations.

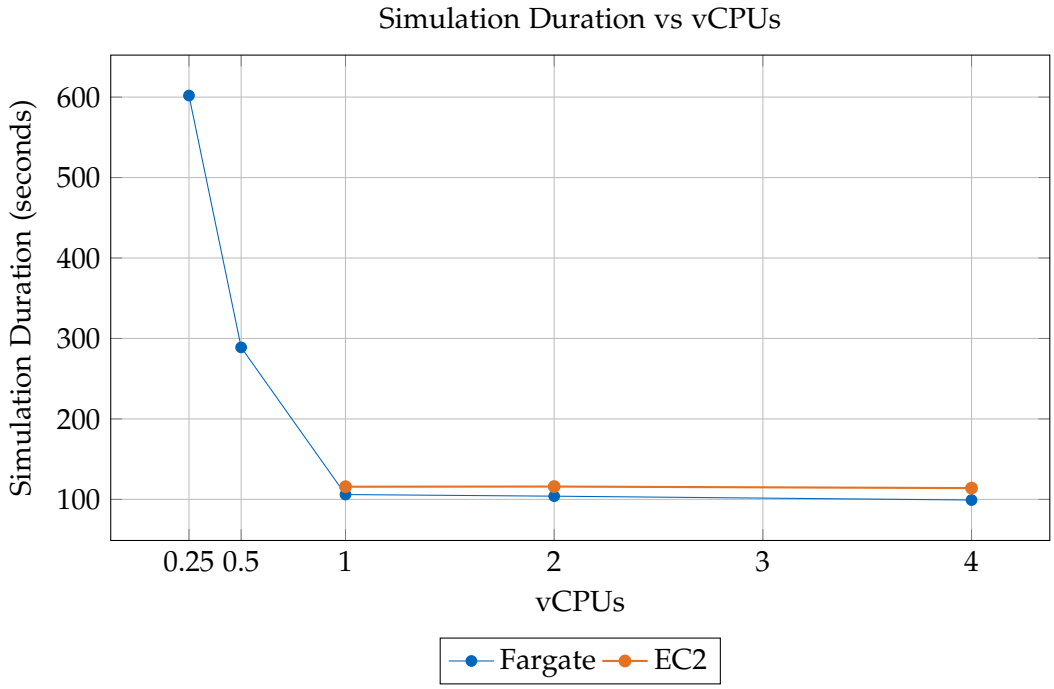

Figure 5.1: Accumulated simulation duration for all openPASS example configurations with different amounts of vCPUs and different compute environments.

Fractional vCPU assignments increase the time until simulation completion as on the corresponding share of CPU time is assigned to the instance. Longer execution time can be beneficial, if the container does not reach the minimum charged duration of one minute. In our case, it is contra productive because our scheduling bundles all simulations in a single batch job. Figure 5.2 shows the most cost effective vCPUs requirement is one vCPU for Fargate, however for EC2 it is two vCPUs according to the graph. The later observation is due to the smallest EC2 instance of type C5 having 2 vCPUs [22]. When running two of our batch jobs at the same time, the cost per batch job is halved making one vCPU the most cost efficient assignment.

The time a batch job spends waiting can be influenced by resource requirements

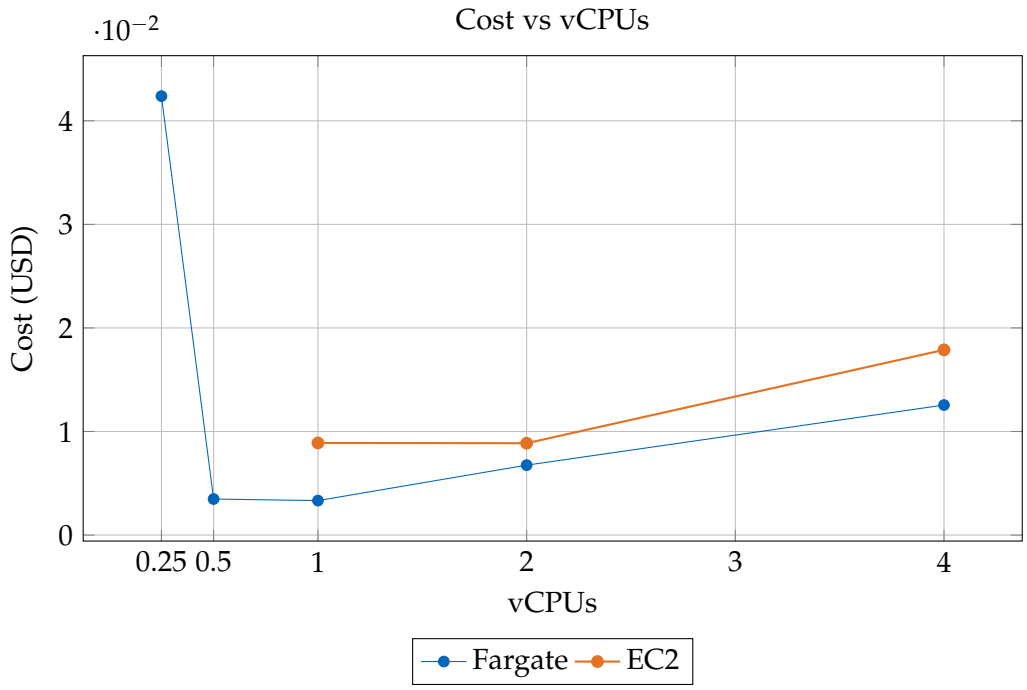

Figure 5.2: Cost for running a batch job with all openPASS example configurations with different amounts of vCPUs and different compute environments.

for the EC2 compute environment, because requests for larger instance are prioritized over smaller ones. Nevertheless, the wait time is more influenced by demand and not relevant for our use case as it does not result in any charges.

5.4 Memory Usage

We measure the memory usage of each of the 64 openPASS example configurations. The entrypoint of the simulation container, introduced in 4.8, includes a mechanism for measuring the memory usage which is used here. It uses the Rusage syscall on the simulator process after completion and reads the Maxrss field as the maximum memory usage.

As depicted in figure 5.3, the memory usage is about equal for both compute environments with minor differences at 80 and 90 MiBs. Almost 97% (62 of 64) of example configs fall between 80 and 130 MiBs of maximum memory usage, with two outliners at 210 and 230 MiBs. These later two configs are both using FMUs, which coincides with our experience for more complex traffic scenarios. Especially for more

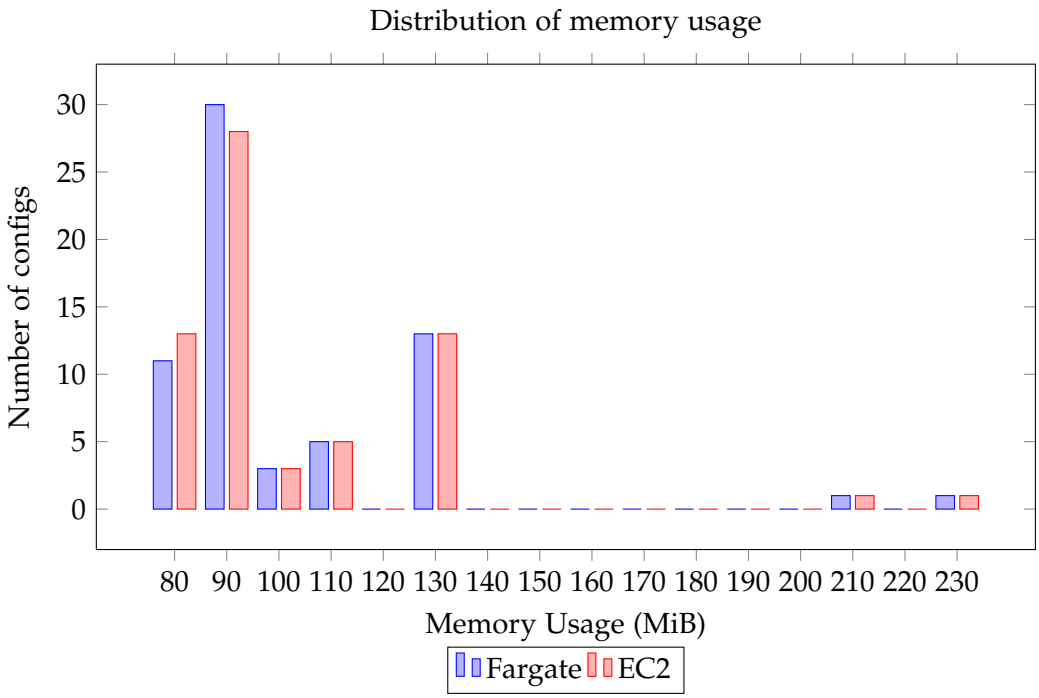

Figure 5.3: Distribution of maximum memory usage for the 64 example configurations delivered with openPASS on different compute environments.

sophisticated driver agents, such as level 3 autonomous driving functions, the resource usage of the FMUs can significantly exceed the openPASS simulator.

As the highest memory usage measured is below our minimum available memory of 2 GiBs per vCPU core, we determine that the benefits of minimizing the memory requirement are negligible. In practice, it is still beneficial as we have proprietary scenarios with FMUs that exceed 2 GiBs of memory usage though to ensure reproducibility and publicibility these can not be used here. The observation that FMUs exceed the resource usage of the simulator means the numeric features extracted are not sufficient for estimating total resource usage for all scenarios. This is mitigated by the knowledge base and further discussed in the following chapter about model selection.

Despite the negligibility, we test the amount of successful simulation completions for different memory assignments with the EC2 environment. The results are depicted in figure 5.4. We perform our tests with increments of 8 MiBs and see in contrast to our memory usage measurements the simulations succeed starting at 24 MiBs and all complete at 160 MiBs of assigned memory. This observation has three reasons, first by default an additional 32 MiBs of memory are reserved for the system [23], second

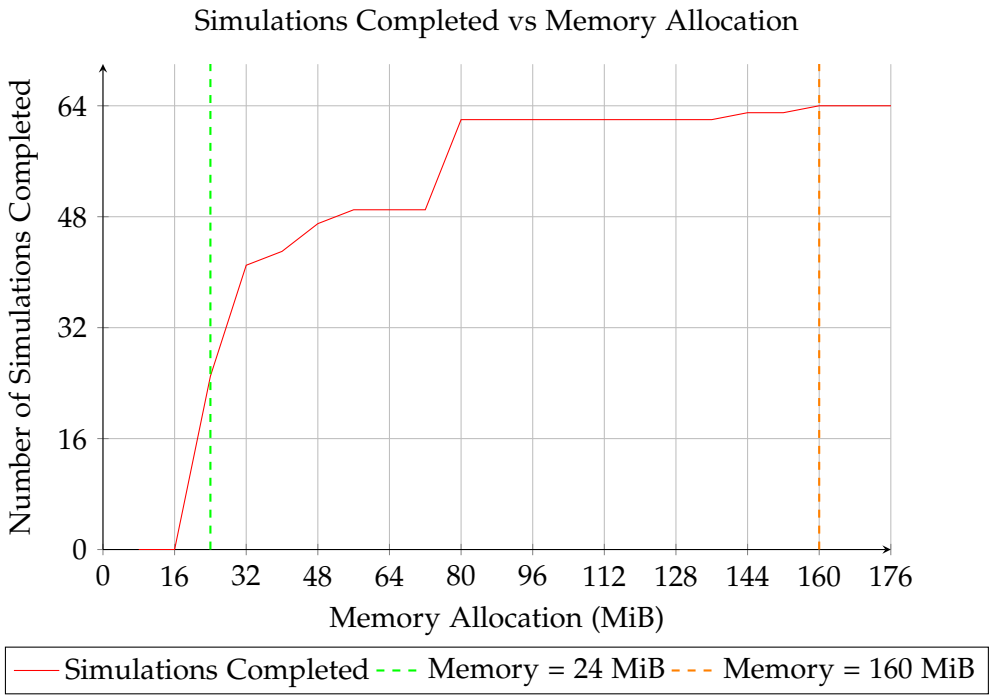

Figure 5.4: Simulations completed for different memory requirements/allocations and 1 vCPU with EC2 compute environment.

memory swapping and third the memory limit can be temporarily exceeded if there is sufficient memory in the VM. Nevertheless, for successful runs the memory limit is never exceeded by more than 64 MiBs and there is no guarantee this works every time. Therefore, the benefit of under provisioning memory are also negligible especially as in case of EC2 we are paying for the resources anyway.

5.5 Time

In this section, we analyze the simulation duration of our input configurations and the time overhead of the batch job for both compute environments. In addition, for the EC2 environment we determine the overhead of the instance runtime.

5.5.1 Simulation Duration

We measure the simulation duration of each of the 64 openPASS example configurations on all compute environments. Like the memory usage, the duration is recorded by the

entrypoint of the simulation container, introduced in 4.8. With simulation duration we refer to the execution time of the openPASS simulator when run with a specific input configuration.

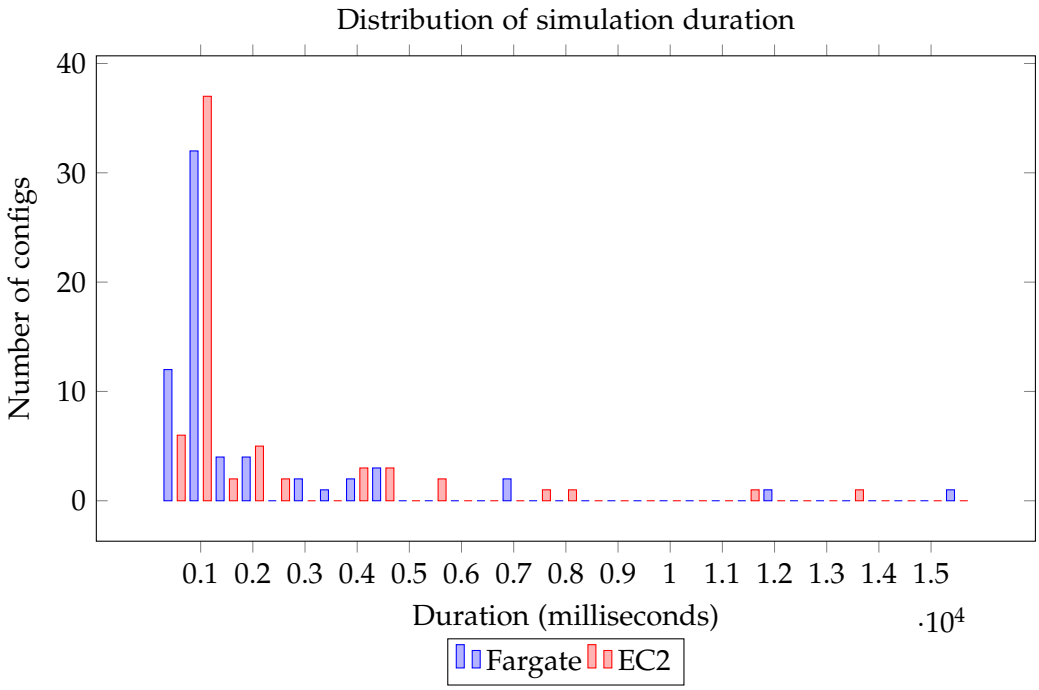

Figure 5.5: Distribution of simulation duration for the 64 example configurations delivered with openPASS on different compute environments.

Figure 5.5 shows the distribution of simulation duration for the example configs. When comparing it to figure 5.3 a relation between memory usage and duration can be seen. However, the correlation is not strong as we know from our analysis in section 5.2. The minor difference between compute environments is due to the different underlying hardware.

According to AWS the ideal runtime of a batch job is between 3 and 5 minutes [24], while the maximum simulation duration of a single config in the example config set is about 15 seconds. Therefore, using the AWS Batch service is only feasible if we bundle input configurations as presented for our scheduling approach in section 4.6. We have proprietary input configuration that have execution times of 30 minutes of more, for those the ideal job runtime cannot be reached. OpenPASS does not support partially running scenarios, thus splitting is not an option. Besides simplicity, the later restriction is why we do not attempt perfect bin packing for bundling batch jobs.

5.5.2 Batch Job Overhead

We determine the time overhead of our batch jobs by subtracting the total time spent simulating from the recorded job runtime. In sections 5.3 and 5.4, we determine that the optimal resource requirements are 1 vCPU and 2 GiBs of memory. Therefore, we only need to analyze the overhead with these requirements for both compute environments.

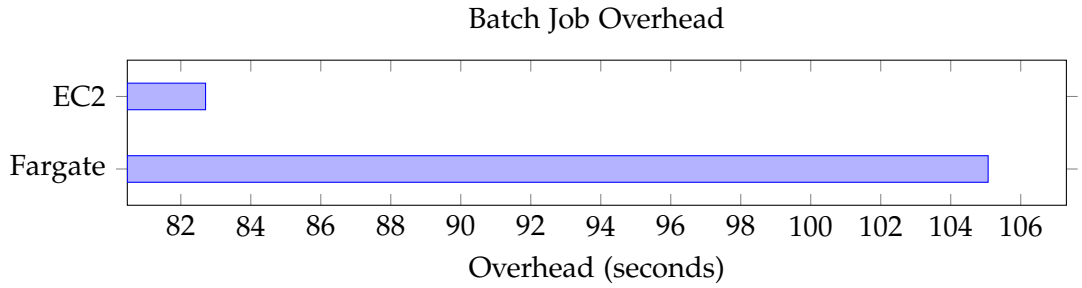

Figure 5.6: Overhead of runtime for a batch job with all openPASS example configurations on different compute environments.

As depicted in figure 5.6 the batch job overhead is significantly larger with around 23 seconds of additional overhead. This is due to the higher level of isolation for Fargate containers and the included warm up time. For EC2 the required instance preparation is separate as instances are independent from individual batch jobs and might be used for multiple ones. The time for preparing the container itself is lower with EC2 due to the reduced level of isolation.

Our observations show that EC2 can be more cost effective starting depending on the time overhead of the EC2 instances and the amount of batch jobs. Thus, in our scheduling approach, presented in section 4.6, the decision whether to run new batch jobs on Fargate or EC2 is made based on the amount of incoming jobs. The overhead of a single batch job can further be reduced, if more simulations are run in a single batch as more time is spent simulating. However, there is also an overhead per configuration due to the reconstruction, data transfers and performance measurements which we evaluate later in this work. This overhead ideally should be determined and utilized in the batch job bundling process.

5.5.3 EC2 Instance Overhead

In our test, the total runtime of a EC2 instance spawned for a batch job which runs all openPASS example configurations was always around 360 seconds or 6 minutes independent from the instance size. With a batch job runtime of around 200 seconds

this resulted in an overhead of about 160 seconds. In our evaluation, we determine the exact overhead and the optimal threshold for using EC2 over Fargate. Though, we can already conclude that for a single batch job of less than 5 minutes of runtime Fargate shall be preferred.

5.6 Cost

In figure 5.2, we show for a single batch job running all openPASS example configurations the Fargate environment is always more cost effective than EC2. The same figure also tells us, when we fully utilize the vCPUs by running a batch job per core we can minimize the cost difference. Furthermore, we know from the time overhead analysis that Fargate has a higher overhead for time charged per batch job while EC2 has an additional charged overhead for every instance spawned. In addition, according to the AWS pricing for the eu-central-1 region at the time of writing the minimum charging period for both EC2 and Fargate is one minute, plus for 2 GiBs of memory per vCPU the hourly costs per vCPU for Fargate are USD 0.05678 (for memory 0.00511 USD/GiB/hour and for vCPU 0.04656 USD/vCPU/hour [25]), and for EC2 C5 instances USD 0.0435 (for c5.large with 2 vCPUs 0.087 USD/vCPU/hour [26]). As the hourly costs are lower for EC2 than for Fargate, a reduction of the overhead per batch job by better utilization of the EC2 instance makes the EC2 environment more efficient under certain conditions.

We assume in the following that our batch jobs all have the optimal duration of 5 minutes, including the overhead for the container entrypoint (EC2 batch job overhead), as this is the ultimate goal of our batch job bundling. From our recorded batch job and EC2 instance state change events, we determine the charged time for the instance where it can not be used for processing is around 60 seconds. We conclude based on our data and the information provided by AWS that for a variable amount of 5 minute EC2 batch jobs AWS Batch starts EC2 instances of the ideal size, which run and are charged for 6 minutes each.

We plot the cost curves for both compute environments in figure 5.7. Fargate has linearly increasing costs because we are charged for each batch job individually, while the cost curve for EC2 moves in steps because instances are only offered with at least two vCPUs and thus any odd number of vCPUs is rounded up by one. Under the pricing at the time of testing the EC2 environment is more cost effective for any full utilization of vCPUs and starting at 7 optimal batch jobs it outperforms Fargate also with an idle vCPU. Thus, under these conditions, Fargate should only be used for the last job if the total number is below 6 and odd. However, the pricing for AWS changes frequently and therefore in productive use there ideally is a mechanism that

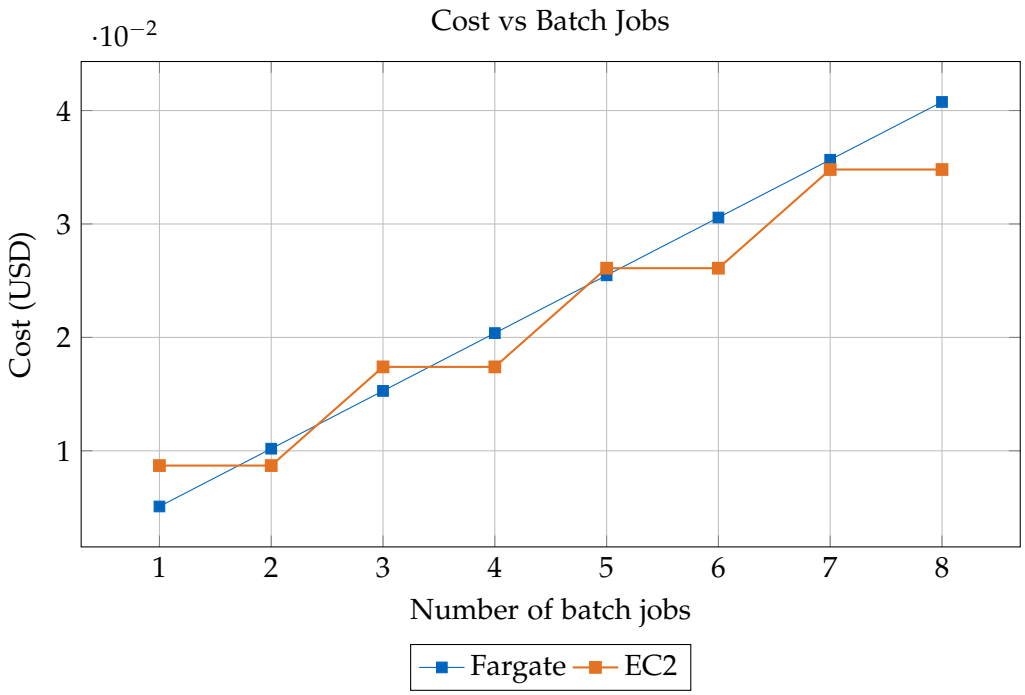

Figure 5.7: Costs for executing different amounts of optimal batch jobs with different compute environments.

determines the most cost effective instance type and when to use it based on current pricing. Furthermore, prices can differ based on your contract with AWS.

Besides the compute costs there are other costs, i.e. for data transfer and storage, these are not influenced by the scheduling and thus not covered here. In addition, the use of spot pricing in AWS offers even more competitive prices for compute resource, but comes with the catch that the resource might terminate at any time. OpenPASS does not save intermediate simulation progress and has no failure recovery, therefore spot pricing is not applicable here.

6 Model Selection

In this chapter, we first compare three different linear regression models based on their performance in predicting the memory usage and duration of openPASS input configurations. Later, as discussed in chapter 5, we select a text embedding model to improve our predictions by finding similar input configurations and their historic runs. Originally, we intended to test neural networks as well, however due to the small size of our dataset we do not expect any viable results.

6.1 Linear Regression

We train, optimize and compare a ridge regression, decision tree regressor and gradient boost model. All models are trained and tested using the same train test split, where 20% of the 64 openPASS example configs are used only for testing and the remainder for training. The input variables are the features extracted from Scenario and ProfilesCatalog, while the output features are memory usage and simulation duration. We use the following performance and error metrics for our models:

1. Mean absolute percentage error (MAPE)
2. Median absolute error (MedAE)
3. Root mean square error (RMSE)

6.1.1 Memory Usage Prediction

The results of our model evaluation are shown in table 6.1. We see that the decision tree regressor model outperforms the others in two metrics, MAPE and RMSE. Gradient boost excels in MedAE, however due to the low samples size this metric is less significant than the others. Memory usage is measured and predicted in KiBs, while AWS Batch only allows specifying it in MiBs. Therefore, for our dataset and use case the model is highly accurate. The reasons for the high accuracy include that the majority of openPASS example configurations have very close memory requirements and logically memory usage is connected to quantitative features like the amount of traffic participants or actions, which are all covered by our extracted features. Nevertheless,

as discussed in chapter 5, the FMUs can have memory requirements exceeding the simulator and are not represented by the extracted features. Thus, under the latter use case all models would certainly perform worse.

Memory Usage Prediction Performance			
Model \ Metric	MAPE	MedAE	RMSE
Ridge Regression	0.10996683	11439.464	10087.310
Decision Tree	0.03957907	1305.6000	6017.4670
Gradient Boost	0.05586668	1216.9644	8755.5218

Table 6.1: Performance of different linear regression models for memory usage prediction.

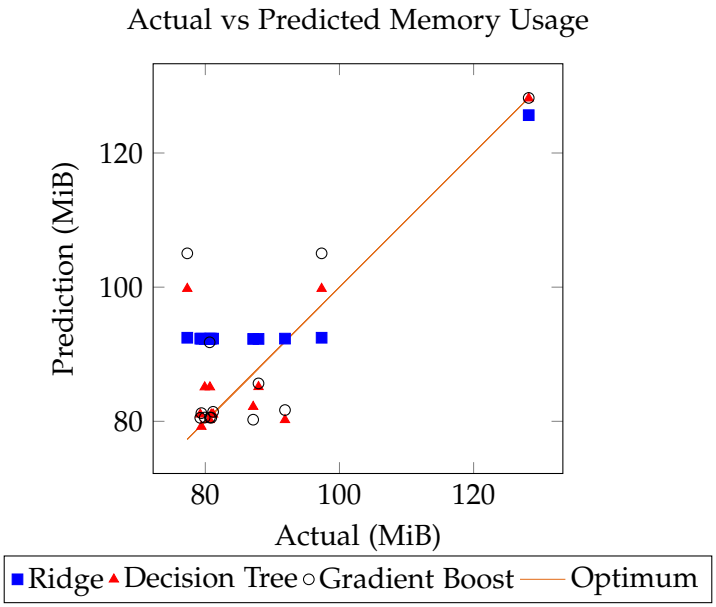

Figure 6.1: Predicted memory usage by different linear regression models and optimal solution.

We take a closer look at the distribution of the memory usage predictions, which are depicted in the scatter plot figure 6.1. The orange line represents the optimum, meaning perfect predictions would lie on this curve. Over estimates are above the curve and under estimates below it. We see that the ridge model always predicts about 92 MiBs, with only one exception for the outliner being under estimated at around 126 MiBs. Slight under estimates are generally worse than slight over estimates, because too

little memory assigned causes the container to be terminated and requires resubmitting the complete input configuration. The predictions of the other models are closer to the optimum, thus resulting in better evaluation results. Gradient boost overestimates more compared to decision tree regressor, thus performs worse on the MAPE and RMSE metrics. In our case, the underestimates are negligible as any tested input configuration requires less than the minimum memory limit. As can be seen in the scatter plot, the two outliners seen at the right most part of memory usage distribution figure 5.3 are not included in the test dataset. These input configs use FMUs where the memory used by the FMU is close to the one of the simulator. We know from other evaluations where at least one of them was in the test dataset that all models significantly underestimate the memory usage (absolute error of around 100 MiBs).

Based on the evaluation results our decision tree regressor model is to be used for memory usage predictions, however an alternative for edge cases such as simulations including complex FMUs is desirable.

6.1.2 Simulation Duration Prediction

The results of our model evaluation for simulation duration are shown in table 6.2. Simulations duration is generally measured in nanoseconds. All models have poor forecasting performance with the MAPE metric exceeding 1 or 100%. Gradient boost achieves the best MAPE and MedAE with a value of 445 milliseconds. The ridge regression model performs best with regard to RMSE with a value of around 1.8 seconds.

Simulation Duration Prediction Performance			
Model \ Metric	MAPE	MedAE	RMSE
Ridge Regression	1.7315652	1164557736	1846358009
Decision Tree	1.3188959	877300354	2271403873
Gradient Boost	1.3000544	445991747	2215502454

Table 6.2: Performance of different linear regression models for simulation duration prediction.

A scatter plot visualizing the predictions of the models for the test dataset can be seen in figure 6.2. The orange line is the optimum, meaning a perfect prediction model would only produce estimates on the line. Overestimates are above the line and underestimates are underneath it. We see all models tend to predict exceeding simulation durations for the shortest input configurations, while with growing actual simulations duration the models tend to increasingly underestimate the execution time.

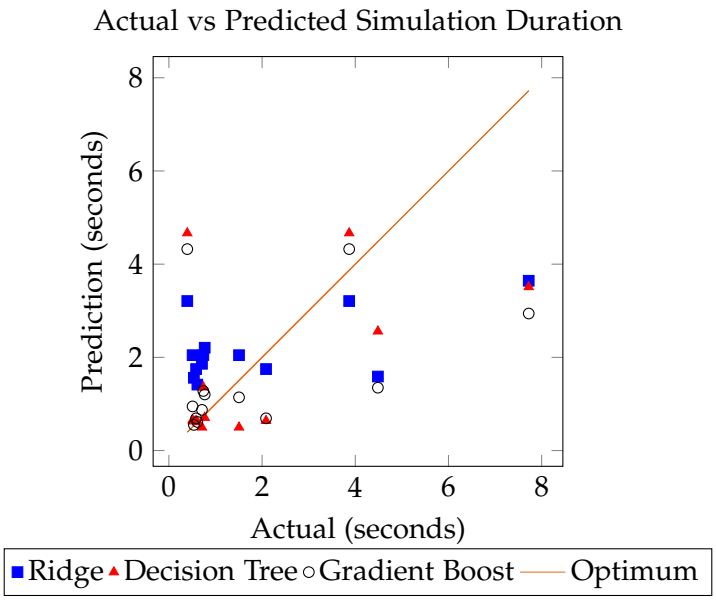

Figure 6.2: Predicted simulation duration by different linear regression models and optimal solution.

None of the models can be recommended for predicting the simulation duration of openPASS input configurations. The underwhelming results are related to the previously determined low correlation between extracted quantitative features in section 5.2. Therefore, we require an alternative method for reliably estimating the execution time of input configurations.

6.2 Text Embeddings

We use a text embeddings model to convert our input configurations to semantic vectors and find similar configs. As we are using AWS and do not want to concern ourselves with hosting such kinds of models we are restricted to using models provided in their Bedrock service. At the time of writing there are 5 models available [27], which are listed with their properties in table 6.3. We want to directly provide the model with a combination of the openPASS Scenario, Scenery and ProfilesCatalog, thus only two of the Titan models are applicable as we require a large vector and maximum token size. Despite Titan Embeddings G1 offering a larger vector size, we choose Titan Text Embeddings V2 as it is newer and generally better at processing long XML documents according to AWS. A token size of 8000 allows us to generate a vector for any text with

at most 20000 characters. For this reason, any combination of Scenario, Scenery and ProfilesCatalog is cut at the 20000th character.

Embedding Models on AWS Bedrock		
Model	Vector sizes	Max tokens
Titan Text Embeddings V2	1024, 512, 256	8000
Titan Multimodal Embeddings G1	1024, 384, 256	128
Titan Embeddings G1 - Text	1536	8000
Cohere Embed English	1024	512
Cohere Embed Multilingual	1024	512

Table 6.3: Comparison of different embedding models available on AWS Bedrock.

In contrast to focusing on quantitative features like our linear regression models, the semantic vector represents the actual meaning of configuration files. Ideally the model is trained on the ASAM standards such as OpenDRIVE and OpenSCENARIO allowing it to understand the input configurations the same way an experienced simulation engineer reading them would understand them.

In order to efficiently compare semantic vector and find similar configs, we use a vector database. Originally, we tested using the Bedrock Knowledge Base service. It uses OpenSearch as its vector database and manages the configurations, filling and updating of the database. However, it refuses the handling of XML documents even when submitting them as text. Thus, we ultimately directly use the OpenSearch Serverless as our vector database and set it up ourselves.

For analysis purposes, we fill the database in steps with one simulation run result at a time and after every step try to query all input configurations. The results are plotted in figures 6.3. We see that there is a delay between the confirmation of the insertion and the runs being returned for queries in 6.3a. A perfect score refers to when the vectors are equal, a score of 1 or 100%, while a perfect match is when the ConfigID of the result and the request vector are equal. Despite thorough testing we were unable to find a practical time after which a single entry is available in OpenSearch Serverless, thereby continued with the best benchmark run we achieved. The perfect score results do not match the perfect matches. This is due to the hard limit of 20000 characters. The config mismatches remain constant at 7 configs, because there are 8 configs that use the same scenario with more than 20000 characters. By chance the first simulation run inserted belongs to one of those configs and is returned any time their vector is queried. The only way to fix this is by shrinking the input configuration enough so the text used for generating the vector is below the character limit. This can be achieved via text summarization with a Large Language Model (LLM), but goes beyond the scope of this

work.

In figure 6.3b the average and minimum match scores for different fill factors including perfect score results are provided. On the other hand, figure 6.3c shows the same metrics plus maximum score excluding perfect score results. We see, even excluding perfect scores, starting at a theoretical fill factor of 24 entries the average match score is 87%. This number exceeds an important feasibility threshold, which we explain in more detail in the following.

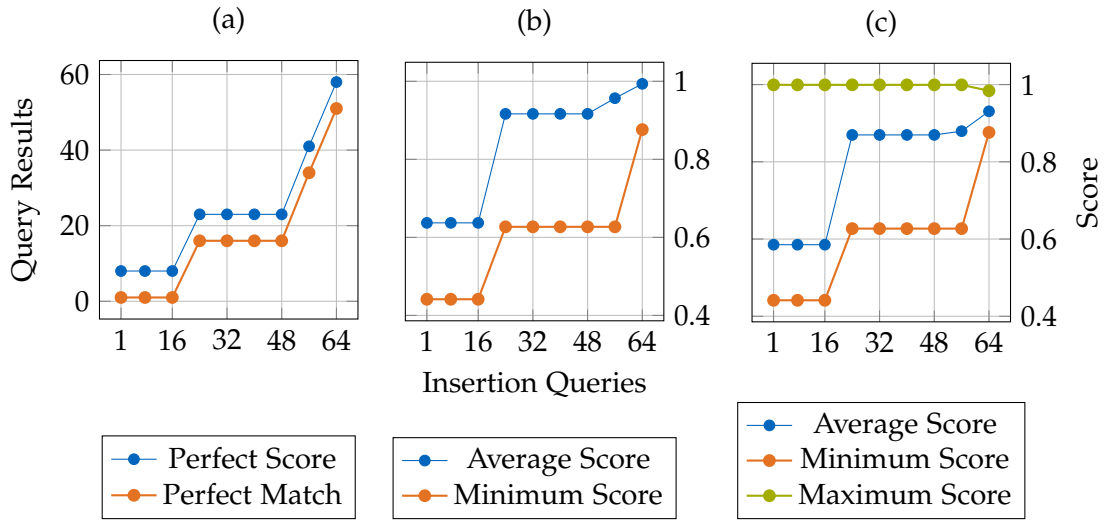

Figure 6.3: Visualization of (a) query results for perfect scores and perfect matches, as well as scores (b) including and (c) excluding perfect score results with different amounts of insertion queries to the knowledge base.

We compare the performance of our text embeddings model and vector database approach to our linear regression models for different fill factors in terms of theoretical database entries. To do so, we consider the closest match as the prediction of this solution and use the same metrics, MAPE, MedAE, and RMSE, for evaluating it.

The error metrics for memory usage prediction are plotted in figures 6.4. Our approach outperforms ridge regression for MAPE and MedAE starting at a theoretical fill factor of 24 entries, while performing worse than decision tree and gradient boost regressor. Only with a theoretical fill factor of 64, a complete fill, the approach outperforms all models. We underperform in RMSE for all fill factors below 56 entries. Based on figures 6.3, we can conclude there is a relation between the match score and the accuracy of our memory usage predictions.

In contrast to the memory usage predictions, estimating simulation duration performs

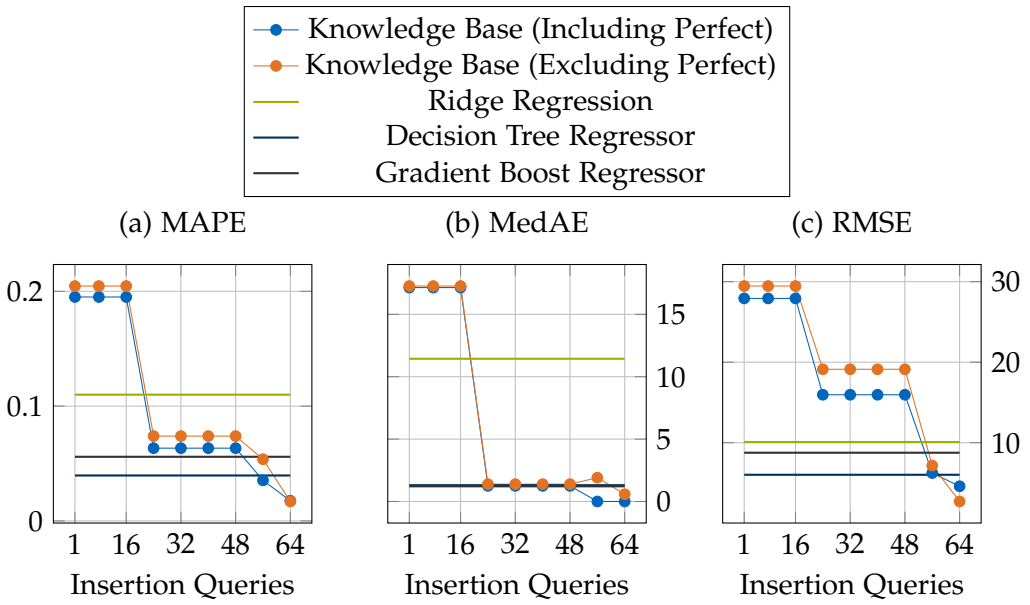

Figure 6.4: Visualization of (a) MAPE, (b) MedAE, and (c) RMSE error metrics for memory usage prediction with knowledge base including and excluding perfect vector matches.

significantly better than the linear regression models starting at 24 entries theoretical fill factor. Our results for different fill factors are depicted in figure 6.5. Only for RMSE ridge regression performs better until the knowledge base reaches 56 entries. At 24 entries our approach achieves a MAPE of 0.8 or 80%, which still is poor forecasting accuracy. However, at 56 the MAPE including perfect scores reaches 64% and at 64 reasonable 35%. It is worth noting, that due to the inconsistency the snapshot knowledge base is not completely but nearly full. In longterm use, the vector database is very likely to be what we consider as nearly full here as the proportion of unknown, new scenarios shrinks.

Our performance analysis shows a correlation between the accuracy and match score of the simulation run. We plot the error metrics for a fill factor of 32 entries and different scores in figures 6.6. For our example dataset, the knowledge base approach outperforms all models in all metrics starting at a score of 0.85 or 85%. With achieve record low MAPE values with 0.2% for memory prediction at a score of about 0.9 and 4.4% for simulation duration with a score of around 0.975. The performance decreases at a score of or close to 1, which is related to mismatches resulting from the character limit of the embeddings models. Putting the mismatches aside, our knowledge base

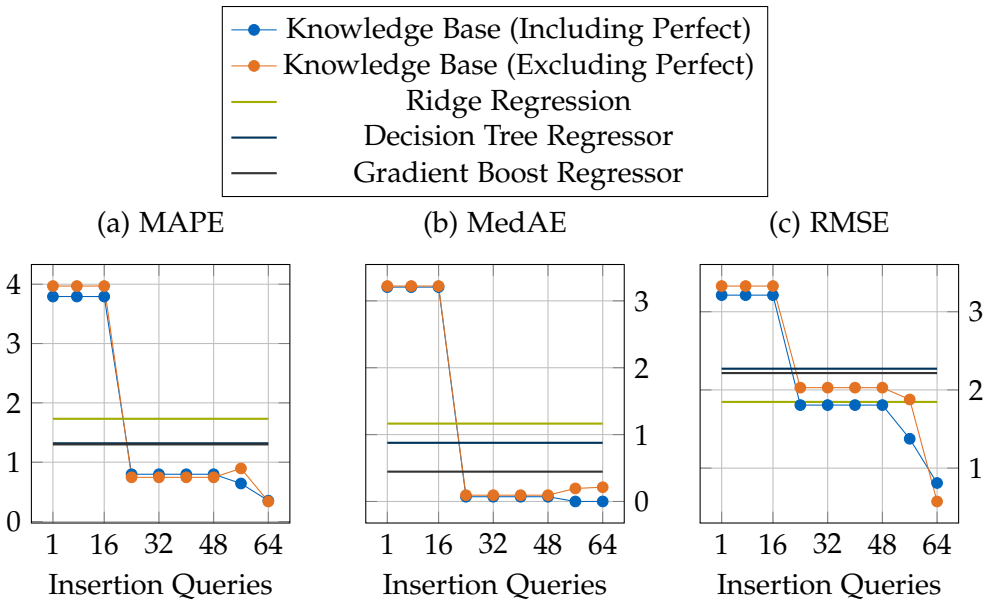

Figure 6.5: Visualization of (a) MAPE, (b) MedAE, and (c) RMSE error metrics for simulation duration prediction with knowledge base including and excluding perfect vector matches.

approach performs better than linear regression if the query result has a match score of 0.85 or higher.

6.3 Combination

We determined that using text embeddings together with a vector database is a superior approach to linear regression, if the match score is 0.85 or more and the result is no mismatch. Thus, we use the linear regression result in case the best vector query result has a lower score. Furthermore, before invoking the embeddings model we check in the DynamoDB if there exists a run for the ConfigID to prevent unnecessary model inference. The latter process is also more cost efficient, because invoking models, especially text based, and vector search is more resource intensive than searching for a primary key. In the following evaluation chapter, we analyze the compute costs and performance of the combined final solution.

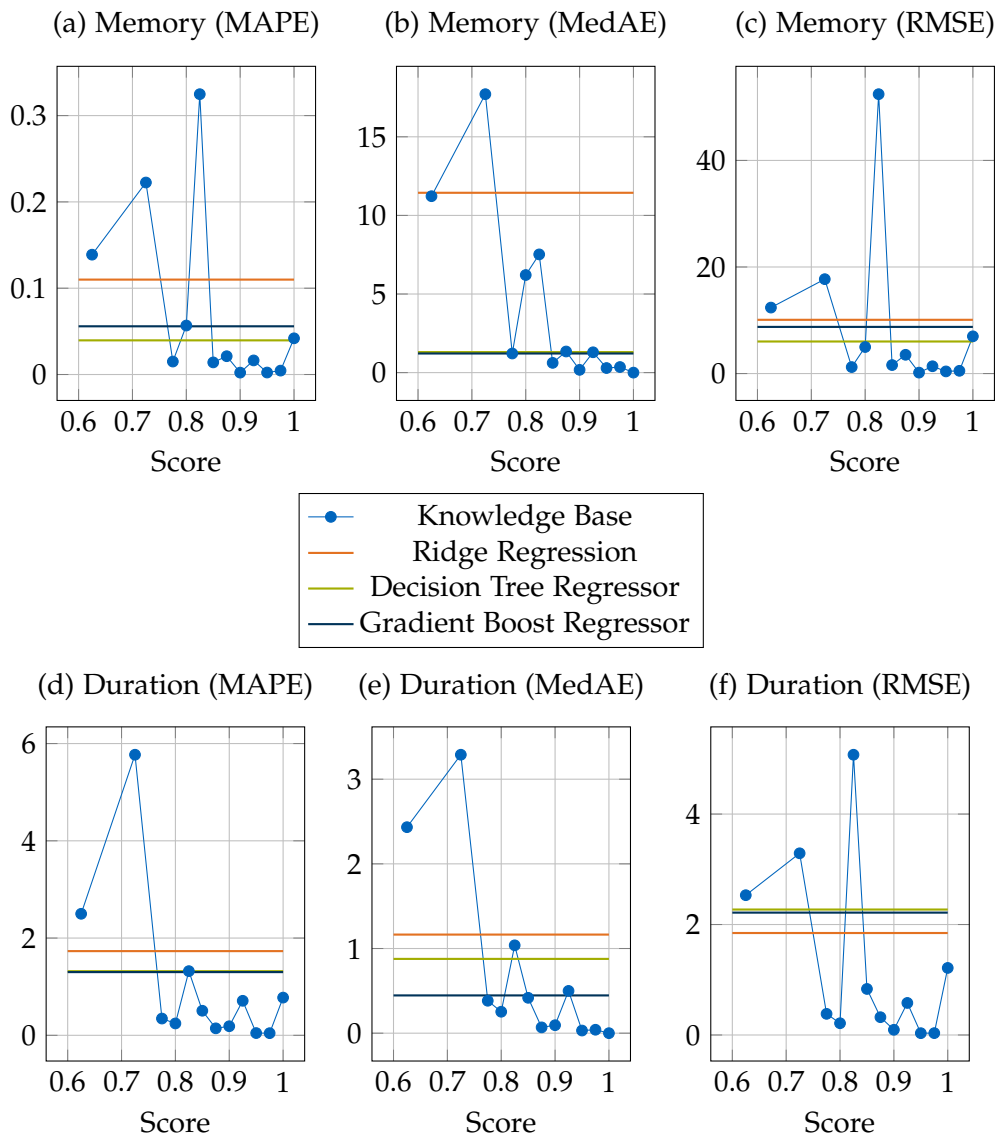

Figure 6.6: Visualization of (a) + (d) MAPE, (b) + (e) MedAE, and (c) + (f) RMSE error metrics for memory usage (a to c) and simulation duration (d to f) prediction with knowledge base and linear regression models for comparison. Memory usage is measured in MiBs, while simulation duration is recorded in seconds.

7 Evaluation

In this chapter, we evaluate our final solution utilizing the combined resource usage estimation for scheduling purposes. First, we introduce our evaluation setup, second we run the evaluation, third we analyze the measured results.

7.1 Setup

Unlike in our model selection, we pre-train the model and pre-fill the knowledge base both with only 50% of the dataset we created during data analysis. The other 50% are used for the evaluation, where we fill the SQS queue in a way that a perfect scheduler would create a specific number of batch jobs. Thus, based on the actual simulation duration we extend the queue until a certain total execution time, a multiple of 5 minutes, is reached.

We do not include the container overhead in the scheduling process as we focus on predicting the simulation duration rather than the container runtime. Training and evaluation dataset are separated by picking the lexically first 32 configs for training and the others for evaluation based on their SHA256 hash, also known as the ConfigID.

The already introduced existing functionality for collecting performance metrics about batch jobs and creating datasets is reused. In contrast, the linear regression models are trained local and uploaded to a S3 bucket, while we fill the knowledge base and queue with two new lambda functions:

CreateEvaluationKB Cleans and fills knowledge base with simulation runs belonging to training dataset.

FillEvaluationQueue Fills SQS queue with simulation jobs consisting of configs belonging to the test dataset. It uses actual resource usage metrics to fill the queue in a a way that a perfect scheduler would create a specific number of jobs. However, the submitted resource requirements are estimated by our combined approach excluding directly accessing historic runs. For convenience and efficiency, we provide the linear regression predictions with the function as the SageMaker endpoint has not been implemented at this point due to time constraints.

For testing purposes, we do not attempt querying historic simulation runs with the ConfigID and directly use the knowledge base with vector search. Furthermore, the model and knowledge base are not updated during testing. Thereby, we can submit the 32 test configs multiple times without effect on the estimations. In addition, evaluation of the memory usage is skipped as we already know it is negligible for our input configurations.

7.2 Results

We look at the actual number of batch jobs created, which is depicted in figure 7.1. Our approach creates an additional incomplete batch job for the submitted simulation jobs, which tells us that we overestimate the simulation duration. The actual amount of batch jobs remains consistently one above the desired amount.

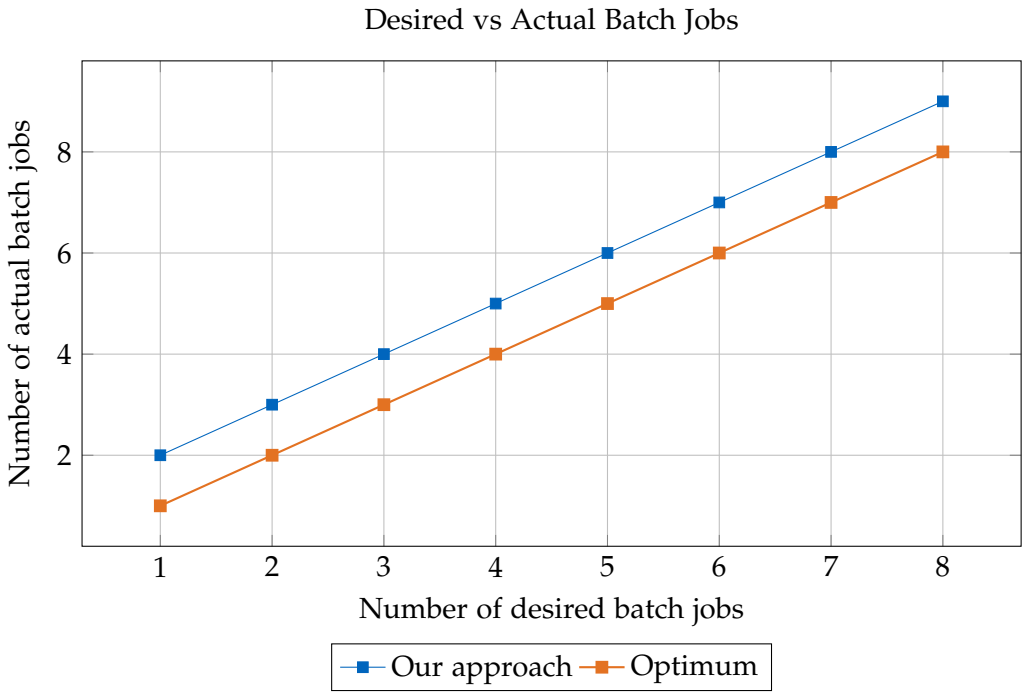

Figure 7.1: Number of batch jobs created by our approach in relation to the anticipated number for an optimal solution.

Figure 7.2 displays the average time spent simulating for the created batch jobs with separate curves for complete and incomplete jobs. The average simulation time for all jobs significantly rises due to the amount of incomplete jobs staying constant at

one, thus their proportion decreasing. We see that the duration of both types of jobs increases with the amount of batch jobs, which is due to more batch jobs sharing machine resources such as the filesystem. At the beginning cause of overestimation the duration stays below the target of 300s, while later the slow down by shared IO etc. makes the jobs exceed the target by up to 40 seconds. There is a slight decrease of the average simulation time for complete jobs at 7 desired jobs.

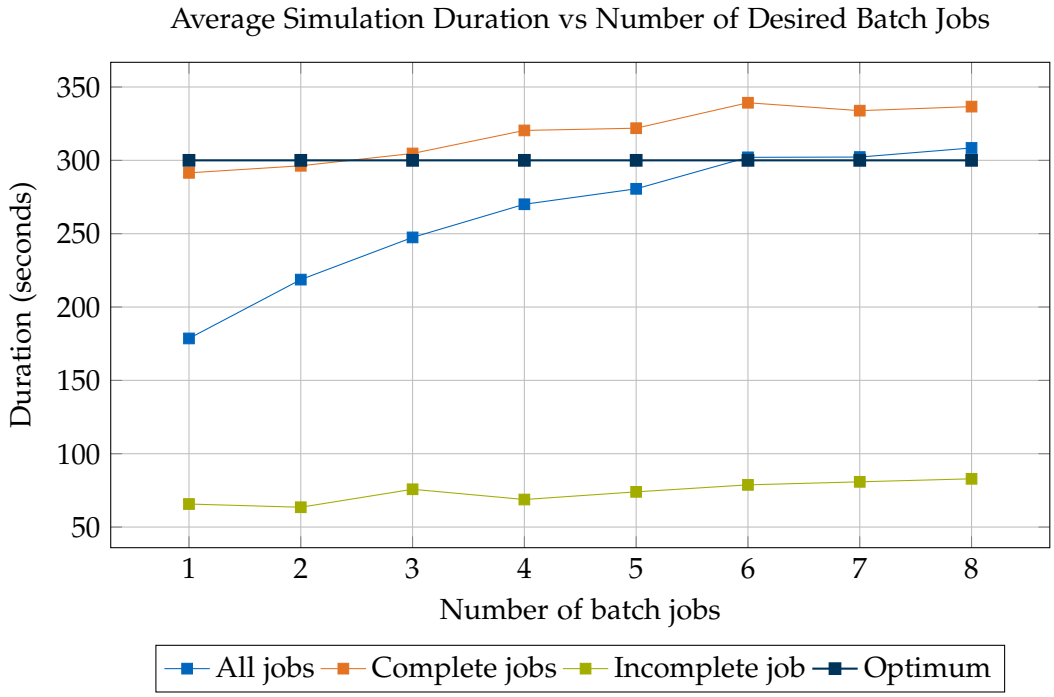

Figure 7.2: Average simulation duration of a batch job in relation to the number of desired batch jobs.

The absolute and optimal cost curves for running the simulations are shown in figure 7.3. We see that the actual costs for our approach is always more than double the closest optimum and the distance grows over time. Unlike our assumption, the AWS Batch EC2 environment does not choose the best combination of instance types based on the queue. Instead, it always makes sure there is at least one free vCPU core. Furthermore, for small amounts of batch jobs such as our case the compute capacity provisioned is doubled in every growth stage. This means that for a desired amount of 7 batch jobs, which effectively results in 8 jobs, a total of 16 vCPUs is provisioned. For comparison we also measured the costs for just using the Fargate environment. It can be seen that we also exceed the optimum and the costs grow at about double the rate than our

optimal solution. Therefore, the threshold for switching to the EC2 environment is much higher than previously thought and outside of our testing range.

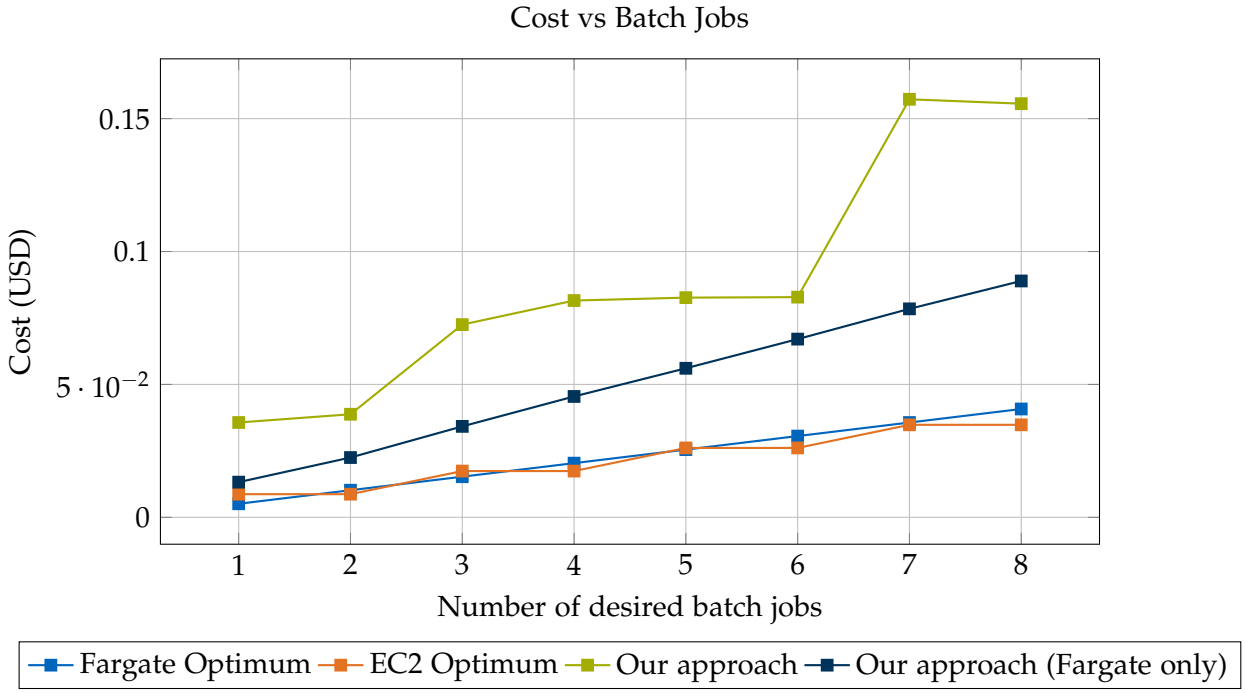

Figure 7.3: Costs for executing number of desired batch jobs with our approach and optimal solutions.

For better understanding of the higher than anticipated costs, we analyze the overhead of our batch jobs compared to the time spent simulating. It is depicted in figure 7.4. The overhead, when separated for complete and incomplete jobs, decreases slightly over time. In addition, the overhead is almost equal to the time spent simulating, thereby doubles the batch job execution time. The latter observation explains the about double actual costs compared to the optimum, which assumes no overhead.

Overall our scheduling approach achieves its goal of creating batch jobs with a certain simulation duration, but ignores other factors that contribute to the batch job execution time. With our test setup and state of implementation, we are unable to show how our implementation reacts to changes in the actual runtime. Our batch jobs have a significant overhead close to 100%, though this is unrelated to our scheduling approach. In practice, optimizing the batch job runtime is more important than just the simulation duration.

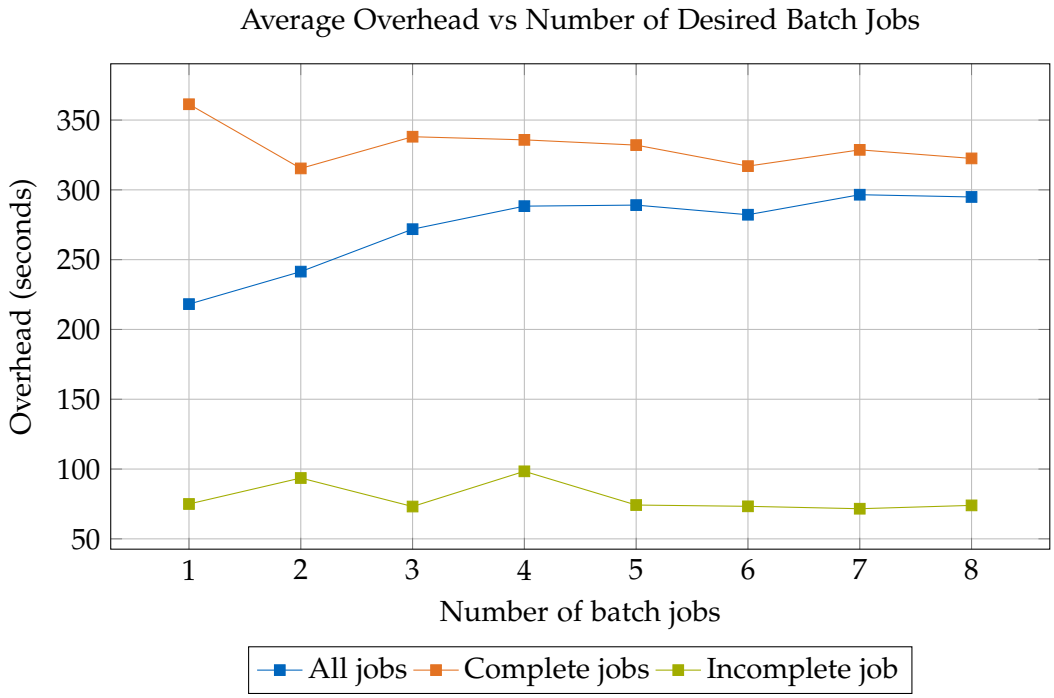

Figure 7.4: Average duration overhead of a batch job in relation to the number of desired batch jobs.

8 Conclusion

We designed a self-improving scheduling for openPASS simulation jobs with AWS Batch. We showed that the openPASS simulator does not benefit from multi-threading, and under optimal conditions, EC2 instances are more cost-effective than AWS Fargate. During our model selection process, we proved that the memory usage of ADAS simulations can be predicted using quantitative features of their input configurations. However, we also discussed how the presence of sophisticated FMUs and other external traffic participants makes estimations based on numeric features unreliable. In addition, we presented evidence that quantitative features are insufficient for forecasting the execution time of simulations using linear regression. Decision tree regression performed best for memory usage, and gradient boost regression best for simulation duration. With our alternative approach utilizing the Amazon Titan Text Embeddings V2 model in combination with a vector database, we achieved superior results for the majority of example configurations using only a 50% split compared to the linear regression models with an 80% dataset split. We displayed that the embeddings model has a decent understanding of the simulation based on the input files, and the provided matching score by the vector database is a feasible indicator for the quality of both the memory usage and simulation duration prediction. For our final approach, we determined a matching score of 0.85 or 85% as the optimal threshold for using the knowledge base over the best linear regression model. Finally, when integrated in our architecture, we show that our combined approach generally slightly overestimates the simulation duration, resulting in an additional non-optimal batch job being created. Nevertheless, for less than 4 batch jobs, the time spent on simulation is close to the optimum. For larger numbers of jobs, the simulation duration increases, exceeding the optimum of 300 seconds by up to 40 seconds or 13%. The later observation was connected to the shared resource usage between batch jobs running on the same machine and would be compensated by the continuous re-training in productive use. Furthermore, we discovered that AWS Batch uses a different, less efficient algorithm for provisioning EC2 instances than we anticipated for our optimal solution. Based on these observations, it only becomes feasible to use the EC2 environment over Fargate far outside of our testing range of up to 8 batch jobs. When only using the Fargate environment, we saw the cost be double compared to the optimum. We traced the cost overhead of our implementation's batch job containers, which have an average runtime overhead of almost 100%, being higher

than in our initial analysis.

Future work should focus on combining our simulation duration predictions with estimations for the batch job overhead, plus using the resulting forecasted batch job runtime for creating the optimal batch jobs. Additionally, the overhead of the containers shall be minimized to increase the cost efficiency of the implementation. Further evaluation with more input configurations, ideally created using some generation approach such as generative artificial intelligence, as well as sophisticated FMUs is recommended to show the general applicability of our approach. In addition, the resource provisioning algorithm of the EC2 compute environment should be more thoroughly analyzed to determine the optimal threshold for choosing EC2 over Fargate. We recommend looking into using AWS Lambda for executing short simulations like the examples provided with openPASS as the warming up time is shorter than with Fargate and lower resource requirements can be requested. Furthermore, we did not have time to test the AWS EKS environment or using a Kubernetes cluster without AWS Batch, which is worth considering. Besides batch processing, other concepts for large-scale execution of simulations are also viable, such as having continuously running micro services processing simulations.

Last but not least, it is worth noting that the next version of openPASS, 2.0, uses a completely new simulation core called GT-Gen-Core. This version has not been released at the time of writing, and performance observations stated here for the current version will no longer apply. Notable upcoming changes include multi-threading support and input configurations that more closely adhere to the ASAM standards. Multi-threading support will make determining the optimal vCPUs requirement per simulation feasible and reduce the simulation duration for scenarios with multiple agents. The closer adherence to standards is likely to improve the accuracy of the semantic vectors created by the embeddings model and thus improve predictions using our knowledge base approach.

Abbreviations

ACC	Adaptive Cruise Control
ADAS	Advanced Driver Assistance Systems
AEB	Automatic Emergency Braking
API	Application Programming Interface
ARIMA	auto-regressive and moving average model
ASAM	Association for Standardization of Automation and Measuring Systems
BMW	Bayerische Motoren Werke
CDC	cloud data center
CFD	computational fluid dynamics
CNN	convolutional neural network
DAS	Driver Assistance Systems
DBN	Deep Belief Network
DCRNN	Diffusion Convolutional Recurrent Neural Network
EC2	Elastic Cloud Compute
EKS	Elastic Kubernetes Service
ECR	Elastic Container Registry
ECS	Elastic Container Service
FMI	Functional Mock-up Interface
FMU	Functional Mock-up Unit
GA	genetic algorithm
GD	gradient descent
GRU	gated recurrent unit
GSPSO	genetic simulated annealing-based particle swarm optimization
HPC	High Performance Computing
IaaC	Infrastructure as a Code
IAM	Identity and Access Management
LKA	Lane Keeping Assistant
LLM	Large Language Model
LM	Levenberg-Marquardt
LR	Linear Regression
LSTM	Long Short-Term Memory
MAPE	Mean Absolute Percentage Error
MFLUPS	millions of fluid point updates
ML	machine learning
PSO	Particle Swarm Optimization
RMSE	Root-Mean-Square Error
S3	Simple Storage Service
SA	simulated annealing
SA-DDPS	Self-Adapting Data-Driven Prediction System
SQS	Simple Queue Service
VM	Virtual Machine

List of Figures

List of Tables

Bibliography

[1] F. Kröger, *From Automated to Autonomous Driving*. Springer, 2024.

[2] BMW Group, *Bmw combines levels 2 and 3 in automated driving*, https://www.bmwgroup.com/en/news/general/2024/automated-driving.html, Last accessed 29 December 2024.

[3] SAE, *Sae levels of driving automation*, https://www.sae.org/blog/sae-j3016-update, Last accessed 23 March 2025.

[4] ASAM e. V., *Asam openscenario xml*, https://www.asam.net/standards/detail/openscenario-xml/, Last accessed 23 March 2025.

[5] ASAM e. V., *Asam opendrive*, https://www.asam.net/standards/detail/opendrive/, Last accessed 23 March 2025.

[6] Modelica Association, *Functional mock-up interface specification*, https://fmi-standard.org/docs/3.0.2/, Last accessed 23 March 2025.

[7] openPASS Working Group, *Openpass documentation*, https://eclipse.dev/openpass/content/html/index.html, Last accessed 3 January 2025.

[8] O. C. Agomuo, O. W. B. Jnr, and J. H. Muzamal, "Energy-aware ai-based optimal cloud infra allocation for provisioning of resources," in *2024 IEEE/ACIS 27th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, IEEE, 2024, pp. 269–274.

[9] M. S. Al-Asaly, M. A. Bencherif, A. Alsanad, and M. M. Hassan, "A deep learning-based resource usage prediction model for resource provisioning in an autonomic cloud computing environment," *Neural Computing and Applications*, vol. 34, no. 13, pp. 10 211–10 228, 2022.

[10] J. Bi, H. Ma, H. Yuan, and J. Zhang, "Accurate prediction of workloads and resources with multi-head attention and hybrid lstm for cloud data centers," *IEEE Transactions on Sustainable Computing*, vol. 8, no. 3, pp. 375–384, 2023.

[11] J. Bi, H. Ma, H. Yuan, R. Buyya, J. Yang, J. Zhang, and M. Zhou, "Multivariate resource usage prediction with frequency-enhanced and attention-assisted transformer in cloud computing systems," *IEEE Internet of Things Journal*, 2024.

[12] J. Dogani, F. Khunjush, M. R. Mahmoudi, and M. Seydali, "Multivariate workload and resource prediction in cloud computing using cnn and gru by attention mechanism," *The Journal of Supercomputing*, vol. 79, no. 3, pp. 3437–3470, 2023.

[13] S. Gupta and D. A. Dinesh, "Resource usage prediction of cloud workloads using deep bidirectional long short term memory networks," in *2017 IEEE international conference on advanced networks and telecommunications systems (ANTS)*, IEEE, 2017, pp. 1–6.

[14] S. Gupta, A. D. Dileep, and T. A. Gonsalves, "Online sparse blstm models for resource usage prediction in cloud datacentres," *IEEE Transactions on Network and Service Management*, vol. 17, no. 4, pp. 2335–2349, 2020.

[15] X. Li, H. Wang, P. Xiu, X. Zhou, and F. Meng, "Resource usage prediction based on bilstm-gru combination model," in *2022 IEEE International Conference on Joint Cloud Computing (JCC)*, IEEE, 2022, pp. 9–16.

[16] P. Nawrocki, P. Osypanka, and B. Posluszny, "Data-driven adaptive prediction of cloud resource usage," *Journal of Grid Computing*, vol. 21, no. 1, p. 6, 2023.

[17] P. Nawrocki and M. Smendowski, "Long-term prediction of cloud resource usage in high-performance computing," in *International Conference on Computational Science*, Springer, 2023, pp. 532–546.

[18] W. Ladd, C. Jensen, M. Vardhan, J. Ames, J. R. Hammond, E. W. Draeger, and A. Randles, "Optimizing cloud computing resource usage for hemodynamic simulation," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2023, pp. 568–578.

[19] A. Siguenza-Torres, A. Wieder, Z. Meng, S. Narvaez Rivas, M. Gao, M. Grossi, X. Du, S. Bortoli, W. Cai, and A. Knoll, "Enhance: Multilevel heterogeneous performance-aware re-partitioning algorithm for microscopic vehicle traffic simulation," *ACM Transactions on Modeling and Computer Simulation*, 2024.

[20] Amazon Web Services, *Reference: Job definition parameters for containerproperties*, `https://docs.aws.amazon.com/batch/latest/userguide/job_definition_parameters-copy.html`, Last accessed 9 May 2025.

[21] Amazon Web Services, *Amazon ec2 instance types*, `https://aws.amazon.com/ec2/instance-types/`, Last accessed 9 May 2025.

[22] Amazon Web Services, *Amazon ec2 c5 instances*, `https://aws.amazon.com/ec2/instance-types/c5/`, Last accessed 9 May 2025.

[23] Amazon Web Services, *Reserve system memory*, `https://docs.aws.amazon.com/batch/latest/userguide/ecs-reserved-memory.html`, Last accessed 9 May 2025.

[24] Amazon Web Services, *When to use aws batch*, `https://docs.aws.amazon.com/batch/latest/userguide/bestpractice1.html`, Last accessed 22 April 2025.

[25] Amazon Web Services, *Aws fargate pricing*, `https://aws.amazon.com/fargate/pricing/`, Last accessed 19 December 2024.

[26] Amazon Web Services, *Amazon ec2 on-demand pricing*, `https://aws.amazon.com/ec2/pricing/on-demand/`, Last accessed 19 December 2024.

[27] Amazon Web Services, *Supported foundation models in amazon bedrock*, `https://docs.aws.amazon.com/bedrock/latest/userguide/models-supported.html`, Last accessed 9 May 2025.