Cost-Effective Machine Learning Inference with AWS Lambda

Evaluating Serverless Resource Configurations

Rick Timmer

Supervisors:
V. (Vasilios) Andrikopoulos, Prof
J. (Justus) Bogner, Prof

Faculty of Science and Engineering
University of Groningen
July, 2024

Cost-Effective Machine Learning Inference with AWS Lambda
Evaluating Serverless Resource Configurations

Student:
Rick Timmer (S4567846)
r.timmer.9@student.rug.nl
ricktimmer98@gmail.com

Supervisors:
Vasilios Andrikopoulos
v.andrikopoulos@rug.nl
Justus Bogner
j.bogner@vu.nl

Department:
Faculty of Science and Engineering
University of Groningen

July, 2024

Acknowledgements:
I would like to express my sincere gratitude to my supervisors, Prof. V. (Vasilios) Andrikopoulos and Prof. J. (Justus) Bogner, for their invaluable guidance, support, and expertise throughout this research project. Their insights and encouragement were instrumental in shaping this work. I am also grateful to the Faculty of Science and Engineering at the University of Groningen for providing the resources and environment conducive to this research.

Abstract

In cloud computing, serverless offerings like AWS Lambda offer notable benefits in scalability and resource management. In theory, the flexibility and auto-scaling features of serverless platforms align well with machine learning (ML) demands, allowing computational resources to be dynamically allocated based on the real-time requirements of ML models. In order to optimize the efficiency and affordability of machine learning inference tasks, it is crucial to have an understanding of the various resource configurations and their implications. This thesis explores the performance and cost of different AWS Lambda resource setups for running ML inference workloads. The study examines how memory allocation and concurrency settings affect computational efficiency and costs by conducting systematic experiments with various ML algorithms—unsupervised, supervised, and large language models. The results reveal significant differences in cost and performance across various configurations. For instance, allocating 1024 MB of memory often provides a good balance between cost and performance for unsupervised and supervised algorithms. In contrast, large language models are not able to run efficiently on AWS Lambda due to significant latency and high costs, making them unsuitable for real-time applications. In terms of implications, this research provides insights into the trade-offs between computational resources and execution costs, helping stakeholders make informed decisions that balance efficiency and budget in a serverless environment. The findings contribute to developing best practices for serverless ML deployments.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Serverless computing is a paradigm offering remarkable scalability and cost-efficiency while being fully managed [1]. In this model, developers can build and deploy applications without having to manage the underlying infrastructure. Instead, cloud providers dynamically allocate resources as needed, charging only for the actual usage, which eliminates the need for over-provisioning and reduces operational costs. Functions as a Service (FaaS), a core component of serverless computing, allows developers to execute code in response to events while the cloud platform handles resource provisioning and management [1]. This makes it ideal for applications with variable workloads.

Machine learning (ML) is a subset of artificial intelligence where algorithms are designed to learn from and make predictions or decisions based on data. ML algorithms can range from simple linear regressions to complex large-language models (LLMs). These algorithms often require significant computational resources for training and inference, making efficient resource management crucial. Serverless computing's inherent ability to dynamically allocate resources and manage workloads could prove useful for unpredictable and bursty machine learning tasks [2]. For instance, during peak times, an ML model might need to handle numerous requests simultaneously, a scenario where traditional fixed-capacity infrastructure might struggle.

The focus of this project is to produce a general overview of the performance of machine learning inference on cloud computing, specifically on AWS Lambda. Different classes of ML algorithms will be used as benchmarks to see if there are significant differences between these groups. The evaluation will consider various resource configurations, such as memory allocation and concurrency limits, to understand their impact on both performance and cost-efficiency. By systematically analyzing these factors, the project aims to provide a comprehensive understanding of how different ML algorithms perform in a serverless context.

The following research questions and objectives guide this study:

RQ1. What state-of-the-art implementations of machine learning algorithms using serverless computing currently exist?

RQ2. How do different serverless computing resource configurations impact the performance of machine learning inference?

RQ3. How do serverless resource configurations impact the cost of machine learning inference?

RQ4. What is the relation between cost and performance of the different serverless resource configurations?

This thesis could be used by anyone interested in running machine learning tasks on functions as a service, specifically AWS Lambda. It contains some guidelines and useful information that could prove helpful when optimizing cost and response times of ML deployments. For instance, understanding the trade-offs between different memory configurations and concurrency settings can help in making informed decisions that balance performance and cost. Additionally, the findings of this research could aid in the development of best practices for deploying ML models in serverless environments, thereby enhancing the scalability and efficiency of these applications.

This thesis is structured as follows: Chapter 1 introduces the concept of serverless computing and Function-as-a-Service (FaaS), highlighting its relevance to machine learning. Chapter 2 reviews the existing literature, setting the stage for the subsequent analysis. Chapter 3 outlines the methodology, detailing the experimental setup, the selection of machine learning algorithms, and the configurations of AWS Lambda resources used. Chapter 4 describes the experiments were designed to evaluate the performance and cost implications of different serverless configurations. Chapter 5 presents the results, analyzing the data obtained from these experiments to understand the impact on performance and cost, as well as discusses the results. Finally, Chapter 6 concludes the thesis by summarizing the findings.

Chapter 2

Literature

This chapter explores how serverless computing and machine learning are coming together to improve both performance and cost-effectiveness. It looks at important contributions in the field and sets the stage for a detailed discussion on using AWS Lambda for machine learning tasks.

2.1 Related Work

Serverless computing and machine learning has gained significant attention in recent years. A comprehensive overview of this field is provided by Barrak et al. [1] in their systematic mapping study. Their work synthesizes existing research on serverless machine learning, highlighting key trends, challenges, and opportunities in this domain.

Several studies have explored specific aspects of serverless ML. Carreira et al. [3] introduced Cirrus, a serverless framework for end-to-end ML workflows. Yu et al. [4] proposed Gillis for serving large neural networks in serverless functions, while Ali et al. [5] developed Batch, a system for ML inference serving with adaptive batching.

Cost efficiency, a crucial aspect of serverless deployments, was addressed by Sarroca and Sánchez-Artigas [6] with their MLLESS framework. Performance benchmarking efforts, such as those by Elordi et al. [7] using MLPerf, have provided insights into the capabilities of serverless platforms for ML tasks.

Our work extends these efforts by focusing specifically on AWS Lambda and analyzing how different resource configurations affect both the performance and cost of ML inference workloads. Unlike previous studies, which primarily address frameworks and broad performance benchmarks, this thesis provides a detailed investigation, including a set of automated benchmark tooling, into the trade-offs between computational efficiency and cost across various ML algorithms, including unsupervised, supervised, and large language models. This targeted analysis addresses a gap in the literature by offering practical insights for optimizing serverless ML deployments on AWS Lambda, helping stakeholders make informed decisions that balance efficiency and budget constraints.

2.2 Review Extension

One of the initial inspirations of this project has been the systematic mapping study written by Barrak et al. [1] This mapping study was published in 2022, and given that the study provides us with the (Scopus) queries used for collecting the papers, we can collect documents on the same topic released after the mapping study. We will then combine these with the original references and enrich these references to find the most relevant studies for us. An illustration of this process can be found in Figure 2.1.

Before we can rerun the queries used in the mapping study by Barrak et al. [1]. we have to make some minor changes. The query was executed in June 2022, so we looked for studies published afterward. We used two queries since Scopus only allows filtering after a given year, not a month. One of these gives us the complete collection after 2023, and the other is for the published studies in 2022, which were then filtered by month.

TITLE-ABS-KEY(("serverless" OR "lambda architecture" OR "function as a service") AND ("machine learning" OR "deep learning")) AND PUBYEAR > 2022

TITLE-ABS-KEY(("serverless" OR "lambda architecture" OR "function as a service") AND ("machine learning" OR "deep learning")) AND PUBYEAR > 2021 AND PUBYEAR < 2023

After combining these sets we further extended it with studies citing the mapping study, however, these all ended up being duplicates also found in the queries. This resulted in an initial set of 132 new papers. Combining these with the original 53 analyzed papers in the mapping study gives us a set of 185 papers to work with. In the case of this project, we are specifically curious about papers that release their source code to the public, since we want to use their implementations in our benchmarks. To sift out these papers we start with a static keyword search for one of the following: "source code", "GitHub", or "GitLab". This gives us 100 papers that potentially contained open-source implementations. This set was manually sifted through to obtain a list with actual open-source implementations, giving us 36 papers to consider further Table 2.1.

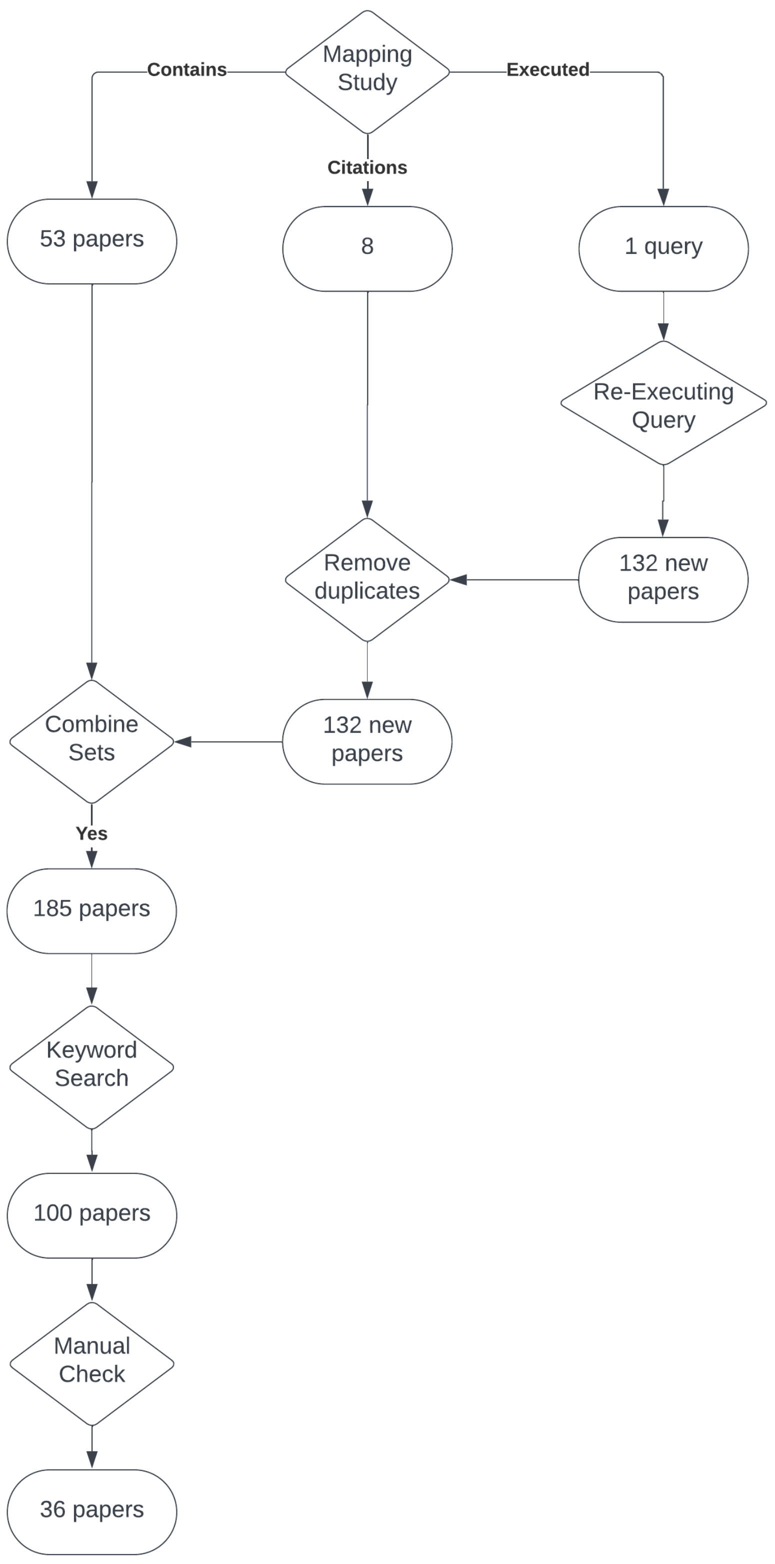

Figure 2.1: Paper selection workflow.

Table 2.1: Selected papers

#	Ref	Title
P1	[8]	Holistic cold-start management in serverless computing cloud with deep learning for time series
P2	[9]	GeoPM-DMEIRL: A deep inverse reinforcement learning security trajectory generation framework with serverless computing
P3*	[6]	MLLESS: Achieving cost efficiency in serverless machine learning training
P4	[10]	Function-as-a-Service performance evaluation: A multivocal literature review
P5	[11]	Generation of a dataset for DoW attack detection in serverless architectures
P6	[12]	Development and deployment of a big data pipeline for field-based high-throughput cotton phenotyping data
P7*	[3]	Cirrus: A Serverless Framework for End-To-end ML Workflows
P8*	[13]	Refactoring of Neural Network Models for Hyperparameter Optimization in Serverless Cloud
P9	[14]	Prebaking functions to warm the serverless cold start
P10	[15]	FaasCache: Keeping serverless computing alive with greedy-dual caching
P11*	[16]	Distributed double machine learning with a serverless architecture
P12	[17]	OFC: An opportunistic caching system for FaaS platforms
P13*	[18]	Towards Demystifying Serverless Machine Learning Training
P14*	[19]	You Do Not Need a bigger boat: Recommendations at Reasonable Scale in a (Mostly) serverless and open stack
P15	[20]	An empirical study on challenges of application development in serverless computing
P16*	[2]	INFless: A native serverless system for low-latency, high-Throughput inference
P17	[21]	WISEFUSE: Workload Characterization and DAG Transformation for Serverless Workflows
P18	[22]	FuncPipe: A Pipelined Serverless Framework for Fast and Cost-Efficient Training of Deep Learning Models
P19	[23]	ElasticFlow: An Elastic Serverless Training Platform for Distributed Deep Learning
P20	[24]	FMI: Fast and Cheap Message Passing for Serverless Functions
P21	[25]	OSCAR-P and aMLLibrary: Performance Profiling and Prediction of Computing Continua Applications
P22	[26]	AsyFunc: A High-Performance and Resource-Efficient Serverless Inference System via Asymmetric Functions
P23	[27]	Tournament-Based Pretraining to Accelerate Federated Learning
P24	[28]	Leveraging Intra-Function Parallelism in Serverless Machine Learning
P25	[29]	Serverless Prediction of Peptide Properties with Recurrent Neural Networks

P26*	[5]	Batch: Machine learning inference serving on serverless platforms with adaptive batching
P27*	[7]	Benchmarking Deep Neural Network Inference Performance on Serverless Environments with MLPerf
P28	[30]	Cloud Services Enable Efficient AI-Guided Simulation Workflows across Heterogeneous Resources
P29	[31]	COUNSEL: Cloud Resource Configuration Management using Deep Reinforcement Learning
P30*	[32]	λdNN: Achieving Predictable Distributed DNN Training with Serverless Architectures
P31	[33]	Exploring the Impact of Serverless Computing on Peer To Peer Training Machine Learning
P32	[34]	SCP4ssd: A Serverless Platform for Nucleotide Sequence Synthesis Difficulty Prediction Using an AutoML Model
P33*	[4]	Gillis: Serving large neural networks in serverless functions with automatic model partitioning
P34	[35]	iPaaS: Intelligent Paging as a Service
P35	[36]	Contention-aware container placement strategy for docker swarm with machine learning based clustering algorithms
P36	[37]	SPIRT: A Fault-Tolerant and Reliable Peer-to-Peer Serverless ML Training Architecture

** Peer-viewed papers included in A. Barrak el al[1].*

The set of papers seen in Table 2.1 is further analysed and labeled to try and find papers that were related to the goals of this work. First we check whether the paper concerns machine learning on functions as a service. If it does we then extract the platform and the machine learning stage which it concerns. We added notes to each of the papers not concerning machine learning on functions as as service that give a general idea of what the paper is about instead. This could perhaps be helpful for people coming across this work that are interested in a very specific topic which could be covered by one of these works. These labeled papers can be found in Table 2.2.

Our initial goal is to leverage existing implementations found in the literature. However, we encountered significant challenges: many of these implementations were difficult to work with and relied on various external frameworks to function. In some cases documentation was not optimal, or even completely missing. Moreover, the number of available implementations is limited. These limitations made us consider a native approach. As a result, we decided to develop our own implementations to evaluate a broader range of machine learning algorithms directly on AWS Lambda, how this was done can be found in Section 4.3. This decision allowed us to avoid the complications associated with external frameworks and provided a consistent environment for our experiments.

CHAPTER 2. LITERATURE

Table 2.2: Summary of Research Papers on Serverless Computing and Machine Learning

#	ML on FaaS	Platform	Stage	Note
P1	False			Proposes a cold-start management policy.
P2	True	AWS Lambda	Training	
P3	True	IBM Functions	Training	
P4	False			Performance review
P5	False			Simulation of function invocations
P6	False			Serverless, but not FaaS
P7	False			End-to-end framework
P8	False			Hyperparameter optimization
P9	False			Cold-start problem
P10	False			Cold-start problem
P11	False			Double machine learning
P12	False			Caching
P13	False			FaaS vs IaaS performance review
P14	False			Template data stack
P15	False			Development challenges
P16	True	AWS Lambda	Inference	
P17	False			Automated execution planning
P18	False	AWS Lambda, Alibaba Functions	Training	Training framework
P19	False			Training platform
P20	False			Communication in FaaS
P21	False			Auto-profiling tool

P22	False			Shadow Functions
P23	False			Federated training framework
P24	True	AWS Lambda	Inference	
P25	False			Serverless, but not FaaS
P26	True	AWS Lambda	Inference	
P27	False			Benchmarking FaaS Inference
P28	True	FuncX	Training, Inference	
P29	False			Deep reinforcement framework
P30	False			Resource provisioning framework
P31	False			Serverless networking
P32	False			ML Automation
P33	False			Automatic partitioning of models
P34	False			Serverless, but not FaaS
P35	False			Containerization
P36	False			Peer-to-peer serverless connections

CHAPTER 2. LITERATURE

Chapter 3

Methodology

In this section, we cover the experiment methodology used to assess the performance of machine learning algorithms on AWS Lambda. It is designed to give us a comprehensive analysis of different types of algorithms, such as unsupervised algorithms, supervised algorithms, and large language models. By focusing on detailed aspects such as execution time, initialization time, and overall billed duration, this project aims to offer insight into the scalability and economic viability of using AWS Lambda in machine learning applications.

3.1 Objective

This study aimed to benchmark the cost efficiency of AWS Lambda using various machine learning algorithms, considering execution time as a key component of the overall cost. We do this by designing an experiment set that allows us to answer the research questions. These experiments will allow us to benchmark both performance (RQ2) and calculate the cost of running workloads (RQ3), which we then use to determine any relations (RQ4).

3.2 Experiment Variables

Understanding the distinction between dependent and independent variables is crucial for designing experiments, analyzing data, and drawing conclusions in experimental research. Independent variables are the factors that researchers manipulate to observe how they affect other variables. In contrast, dependent variables are the results or responses measured in the experiment that are presumed to be influenced by independent variables.

3.2.1 Independent Variables

Changing the memory size of functions as a service can have unforeseen consequences. Platforms like AWS assign CPUs to the function depending on the memory size [38]. All of these hidden variables affect the performance that we want to measure. We limit the number of concurrent executions in our Lambda function. This strategy is primarily to reduce the number of cold starts-instances where a new execution environment is initialized [39], which can add latency to the initial request. We are mainly interested in hot runs, where the function executes in an already active environment, but we will also experiment with cold starts to see if limiting these is more cost-effective. By default, we use 10 for each benchmark, but the default in Lambda is to have a maximum of 1000 per account. Another variable we want to change is the batch size. In our case, the batch size would be the number of items processed per function per request. So when a function is triggered with a batch size of 100, we want it to process 100 items before returning. We need to change these variables because some of the algorithms perform too well to be able to distinguish real performance changes between the different memory allocations. This is something that will become apparent in the results of the experiment. Finally, as mentioned, we want to benchmark different machine learning algorithms. The different values that we ended up using for these variables are:

- **Memory size (In MBs)**: 256, 512, 1024, 2048, 4096, 8192, 10240
- **Max concurrent executions (In number of instances)**: 10, 20, 50, 1000
- **Batch size (In number of items)**: 100, 1000
- **Categories of machine learning algorithms**: Unsupervised, supervised, and LLMs

Machine learning algorithms

The algorithms tested were divided into three categories. Unsupervised algorithms, supervised algorithms, and large language models. Particularly in the case of large language models, there are some restrictions given that AWS Lambda has restricted memory and storage.

- **Large Language Models (LLMs)**:
  - TinyLlama-1.1B-Chat-v1.0 (1.1B parameters) [40]
  - TinyLLama-v0 (4.62M parameters) [41]
  - bert-base-uncased (110M parameters) [42]
- **Supervised Learning Algorithms**: Logistic Regression, Support Vector Machine (SVM), and Random Forest.
- **Unsupervised Learning Algorithms**: K-Means, Gaussian Mixture Model, and Principal Component Analysis (PCA).

3.2.2 Dependent variables

In this experiment, we assess machine learning algorithms' efficiency and operational cost within a serverless architecture by measuring specific aspects of function execution. These measures are crucial for understanding the performance and economic impact of the varied configurations tested. The dependent variables include:

- **Initialization Time**: The duration it takes for the model to initialize before processing begins. This time is significant as it impacts the latency and user experience, but only relevant for cold starts. Initialization time will be tracked through logs generated during the execution of the functions.

- **Processing Time**: The actual time spent processing the data. This metric directly reflects the computational efficiency of the function under different memory allocations and algorithmic approaches. Processing time will also be tracked through logs generated during the function execution.
- **Billed Duration**: The total time charged by the provider encompasses initialization and processing times. This is crucial for evaluating the cost-effectiveness of different configurations. Billed duration is provided by AWS Lambda as part of the platform's logs.
- **Total Cost**: Calculated based on the billable time and the resource utilization during the execution. This helps assess the financial implications of deploying each configuration in a real-world scenario. Total cost is determined by multiplying the billed duration by the cost per millisecond, as specified by AWS Lambda pricing.

These metrics will be tracked to analyze how different memory sizes, batch sizes, concurrency levels, and types of machine learning algorithms influence performance and costs in a cloud-based function-as-a-service platform. This analysis aims to identify the most cost-effective and performance-optimized configurations to implement machine learning functions on AWS Lambda.

3.3 Datasets

The experiments use two datasets, one for the large language models and one for the other algorithms. These datasets are used to create batches for the system to process. To ensure that enough batches of a given size can be created the dataset wraps around when it runs out of items, ensuring that we are able to create large enough batches to put some strain on the algorithms. Since the LLMs are pre-trained, we only use the dataset for testing as no training was required. We use the ARC AI2 Reasoning Challenge dataset [43] to test these algorithms, which can be used to judge the reasoning levels of different models. In our case, we are not so interested in the level of reasoning, but the dataset provides us with a consistent and varied number of prompts, which is enough for our benchmarks. For the other algorithms, we use an emotion dataset used by a natural language project for judging the sentiments of different texts [44]—this dataset we use for both training the models and benchmarking them. The entire dataset is used for training purposes without any additional preprocessing. For the unsupervised models, we drop the labels from the dataset, while for the supervised models, the labels are of course used. This approach ensures that both supervised and unsupervised models can be effectively evaluated using the same data source.

3.4 Measurement Metrics

The primary metric for benchmarking was the billed duration, encompassing total execution time, initialization time (time required for model loading), and processing time (duration of item processing by the model). Using the billed duration we are able to determine the approximate runtime cost.

Chapter 4

Experiments

In this chapter, we cover how the experiments were executed, including how the experiment was produced in a way that we consider the 3-Rs—repeatability, reproducibility, and replicability [45]. Using a diverse suite of tools, the setup allows us to automate most of the process and extend the experiments, allowing us to examine the behaviors we observed with follow-up experiments further. The experimental setups are available on GitHub which the reader is encouraged to check out. They contain documentation on how to get started using the experiment suite.

4.1 Experiment Design

Each algorithm was subjected to 100 experimental runs on AWS Lambda, with 10 concurrent executions. This setup provided a mix of cold starts (10) and hot runs (90), ensuring a comprehensive performance analysis under different invocation conditions. For a single instance, we ran the different memory configurations with a different number of maximum concurrent executions to see what effect that would have on the data. Most experiments were carried out using a batch size of 100, which was increased for cases where the number of items was not enough to strain the algorithm enough.

4.2 Environment setup

To ensure consistency and reproducibility in our experimental environment, we utilized Terraform by HashiCorp, an infrastructure as code (IaC) tool, for setting up and managing the cloud resources required for our experiments. Terraform allows us to define our infrastructure using a high-level configuration syntax, which it then uses to create, update, and manage all underlying resources with predictable and repeatable deployment steps. This approach guarantees that the same configurations and resources are available for each experiment, reducing discrepancies that could arise from manual setups. In our infrastructure as code, we define several AWS services, an overview of which can be found in Figure 4.1, these being:

- **AWS Elastic Container Service:** Amazon's fully managed container registration service, which uses the AWS Fargate compute engine to provision resources for tasks automatically. These tasks are responsible for publishing messages on a message queue, which are then picked up by the different Lambda instances.

- **AWS Elastic Container Registry:** Amazon's container registration where all the container images are stored.
- **AWS Simple Storage Service:** Amazon's cloud object storage service where we store our datasets for the experiments. these are read by the experiment task, which then publishes the messages containing the data.
- **AWS Simple Notification Service:** Amazon's managed Pub/Sub service, which we use to trigger our AWS Lambda functions.
- **AWS Lambda:** Amazon's FaaS compute service where code responds to events and automatically manages the required computational resources.
- **AWS CloudWatch:** Amazon's cloud monitoring tool where all of our logs are collected. We can then extract them using simple queries, and by applying some simple preprocessing, we can get our hands on a usable dataset.

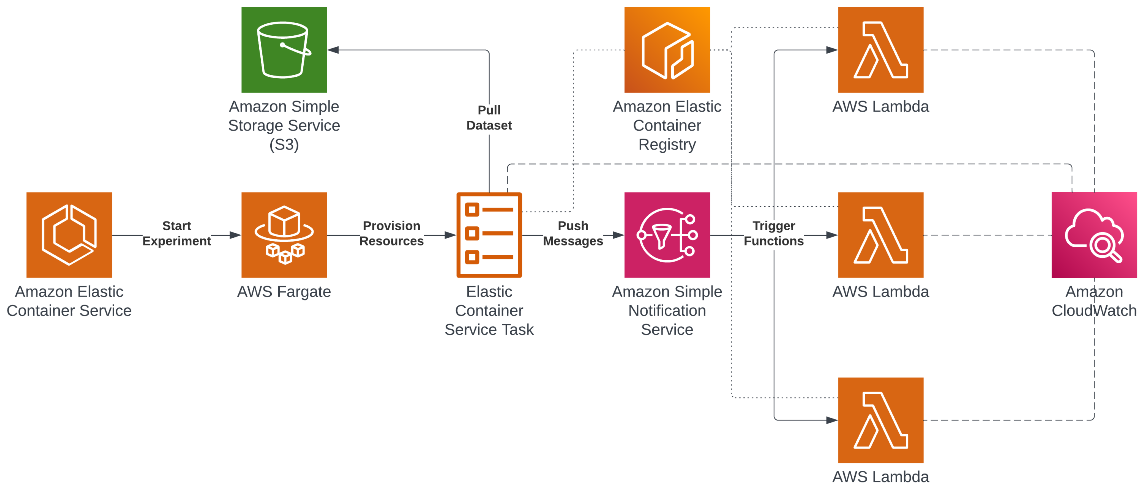
Amazon Simple Storage Service (S3)
Pull Dataset
Amazon Elastic Container Registry
AWS Lambda
Amazon Elastic Container Service
Start Experiment
AWS Fargate
Provision Resources
Elastic Container Service Task
Push Messages
Amazon Simple Notification Service
Trigger Functions
AWS Lambda
Amazon CloudWatch
AWS Lambda

Figure 4.1: An overview of the AWS infrastructure used for the experiments

4.3 Algorithm Implementation

For the different machine learning algorithms we apply pre-built algorithms in Lambda using Python with libraries like PyTorch and scikit-learn, which is then containerized using Docker so it can be deployed on Lambda. The LLMs were pulled from HuggingFace whilst the other algorithms were trained using the scikit-learn python implementations.

To make the project extendable, the algorithms were all implemented using a wrapper. This wrapper file also contains all necessary logging for the eventual analysis of the data. This means that to extend the project, one copies one of the existing implementations, changes around the algorithm, and that is it. All the logging is automatically handled in the wrapper. This should allow future researches to apply the project for their own benchmarking needs, so they can test the viability of other algorithms on AWS Lambda.

The implementation of the wrapper is relatively simple, see Algorithm 1. The user simply passes through the event, a callback for the processing, an initializer in case of a cold start, and the request identifier from Lambda. The wrapper then handles the initialization when required, does some logging, and starts processing the items.

Algorithm 1 Event Handler Wrapper Function

```
1: function WRAP(event, callback, initializer, requestId)
2:     // 1: Extract data from event.
3:     // 2: Run the initializer function if the trigger was a cold start.
4:     // 3: Run the callback function, which runs a machine learning algorithm.
5:     // 4: Return the results.
6: end function
```

4.4 Extraction and processing

After running the experiments we need to first extract the logs from AWS, and after we have pulled them locally we need to do some preprocessing before we can start analyzing it. For this we have a number of scripts, also available on the GitHub page linked prior. Firstly, we use a script that calls the AWS CLI to extract all the logs relevant for that particular experiment from Cloudwatch. This is done using a timestamp which logs the start time of the experiment. After we have pulled the logs, we have to put them in a format that allows us to analyze it. For this we first identify the relevant logs, and then using regex we conditionally extract the values that we want to keep in our dataset. For example, the billed duration of a certain call. Little preprocessing is required further, just some grouping of the data. All these are then grouped per execution and stored in a CSV format so we can plot figures using the data.

To analyze the performance and cost of various AWS Lambda resource configurations for machine learning inference, we tracked key metrics such as initialization time and processing time through AWS Lambda logs. Billed duration, which encompasses both initialization and processing times, was also extracted from these logs. The total cost was calculated by multiplying the billed duration by the AWS Lambda cost per millisecond. We examined how different memory sizes, batch sizes, concurrency levels, and types of machine learning algorithms influenced these metrics. Data visualization techniques were employed to interpret the results. This analysis aimed to identify the most cost-effective and performance-optimized configurations for deploying machine learning functions on AWS Lambda. By interpreting these metrics through visual representations, we addressed the research questions and provided insights into optimizing serverless ML deployments.

Chapter 5

Results

In this chapter, we analyze the results from our experimental runs, focusing on evaluating the cost-effectiveness and performance efficiency of various configurations. We will first present the gathered data through figures and then discuss these results in the context of our three main research questions. The data on which these results are based can be found on Kaggle. In this chapter we will first focus on the experimental results in Section 5.1, after which we discuss their implications with regards to the research questions in Section 5.2.

5.1 Overview of Experimental Results

This section introduces the figures and tables derived from the experiments, which will be discussed in detail in the following sections.

Before we can plot the cost-effectiveness of our plots, we need to know the actual costs. Amazon provides a table showing the cost per millisecond runtime per amount of memory which can be seen in Table 5.1. However, this table initially did not have a 256 MB memory allocation price. When we plot the price in a figure, see Figure 5.1, we can tell that the price is linearly increased. Based on this, we estimate the missing memory allocation, which we then use in our cost calculation in our plot. As a nice to know, we also extracted the number of CPUs assigned to each configuration. We extracted this information using the *os.cp_count*() found in Python. The result of this can be seen in Figure 5.2.

For the unsupervised algorithms, we ran Gaussian mixture, k-means, and pca every 100 times per memory allocation with a maximum number of concurrent instances of 10. This ensured we had 10 cold runs and 90 hot runs for each experiment. The initial results of this run can be found in Figure 5.3. The top three plots show the average runtime for the 90 hot runs, and at the bottom, we can see the cost per function. Each was run with a batch size of 100 items processed at a time.

Since both the k-means and pca algorithms showed plots that seemed limited by the current configurations, we decided to run both of these one more time with some other parameters. The number of requests and max instances stays the same, but we reduced the memory size and, in the case of k-means, increased the batch size to 1000 items. These can be seen in Figures 5.4 and 5.5. After increasing the batch size and decreasing the memory account, we can see that both the figures much more closely resemble the

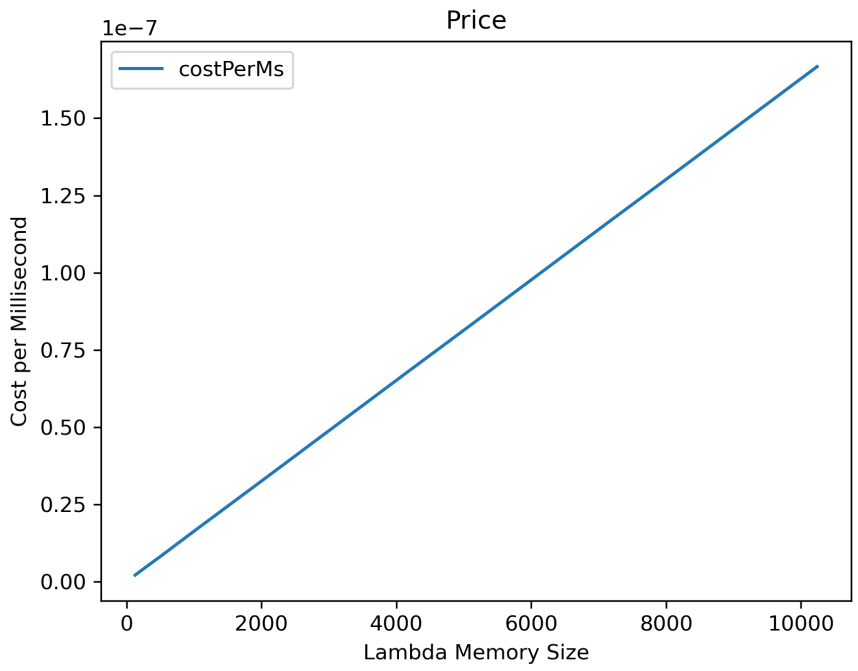

Figure 5.1: Pricing on AWS Lambda per Memory Size (MB).

Memory (MB)	**Price per 1ms**
128	$0.0000000021
256	*$0.0000000042
512	$0.0000000083
1024	$0.0000000167
1536	$0.0000000250
2048	$0.0000000333
3072	$0.0000000500
4096	$0.0000000667
5120	$0.0000000833
6144	$0.0000001000
7168	$0.0000001167
8192	$0.0000001333
9216	$0.0000001500
10240	$0.0000001667

Table 5.1: AWS Lambda pricing per memory allocation [46] (* *Estimation*)
.

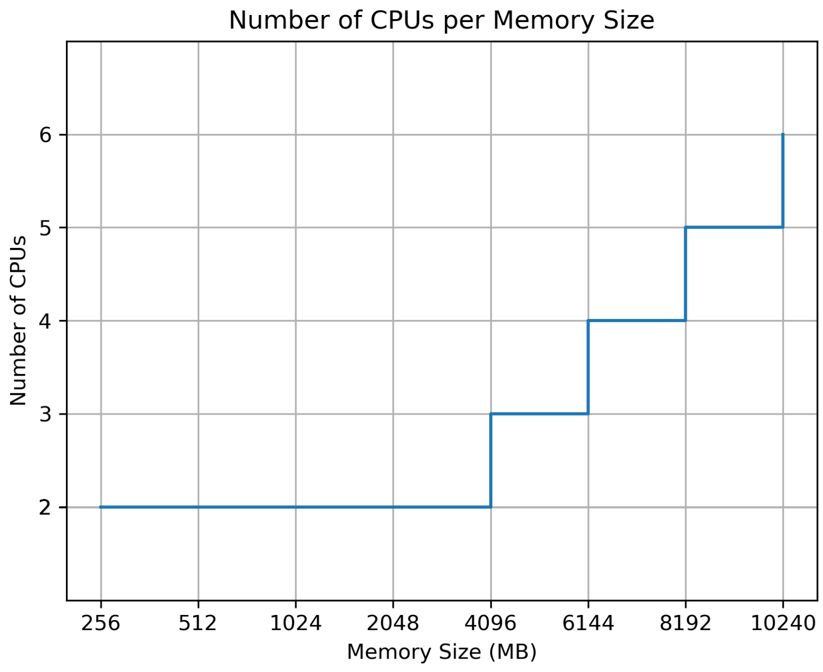

Figure 5.2: Number of CPUs per memory allocation.

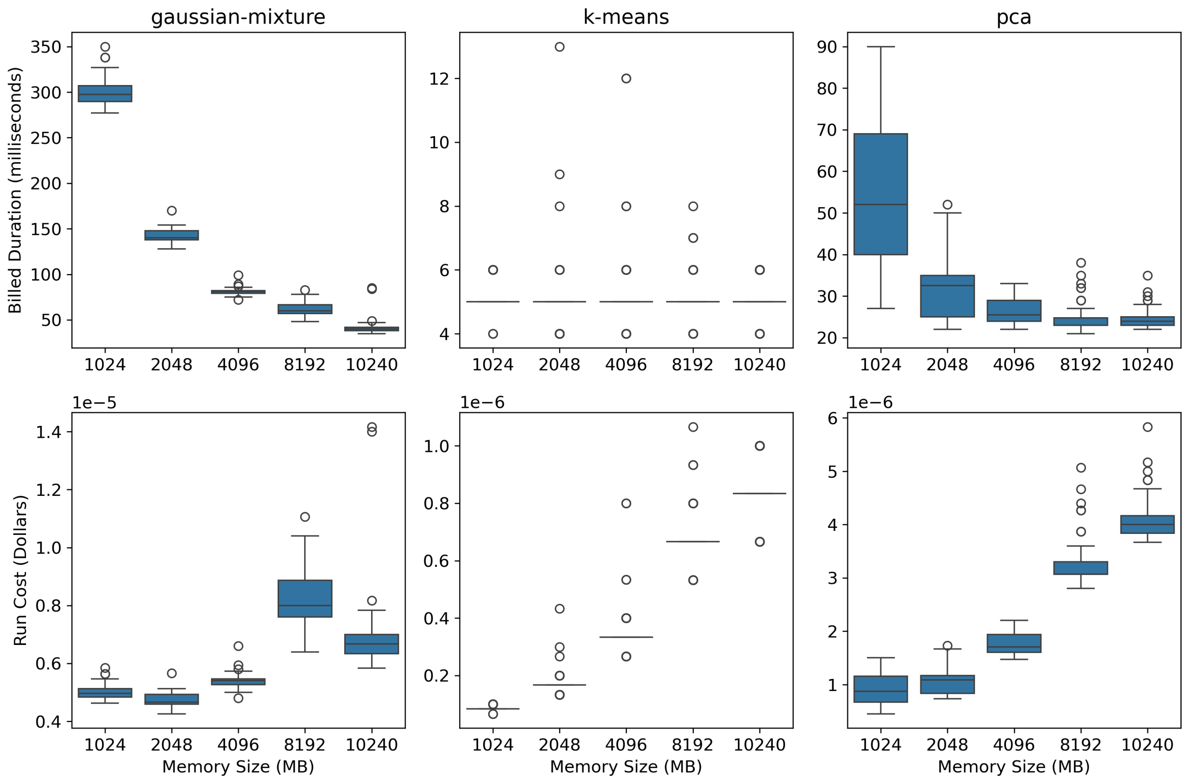

Figure 5.3: Performance and cost of unsupervised algorithms.

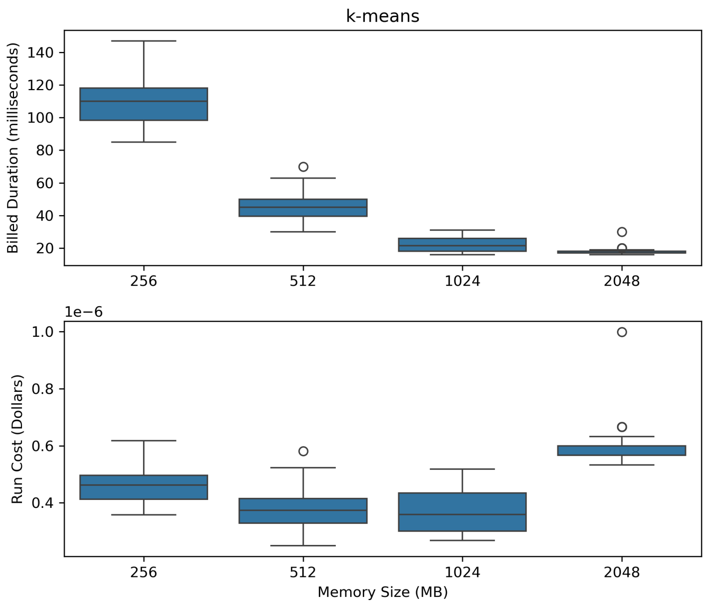

Figure 5.4: Performance and cost of k-means for lowered memory and batch size 1000.

almost exponential decrease we saw in the results for the Gaussian mixture.

Similarly to the unsupervised algorithms, we have one figure containing the overall results of the first experiment for all three algorithms; these can be seen in Figure 5.6. These are based on 90 hot runs with a batch size of 100. Similar to some of the algorithms we saw before, we can see that for both the logistic regression and the svm are not strained enough to get any meaningful information out of them.

Because both logistic regression and svm were not strained enough we ran both of these experiments again with slightly different configurations. For both, we lowered the memory allocations, and in the case of logistic regression, we increased the batch size to 1000. These results are found in Figures 5.7 and 5.8. In both cases, we can once again spot a somewhat exponential decrease in the processing time which we saw before at the other algorithms. The cost per request stays almost similar, only seeing a significant increase at the 2048 MB allocation.

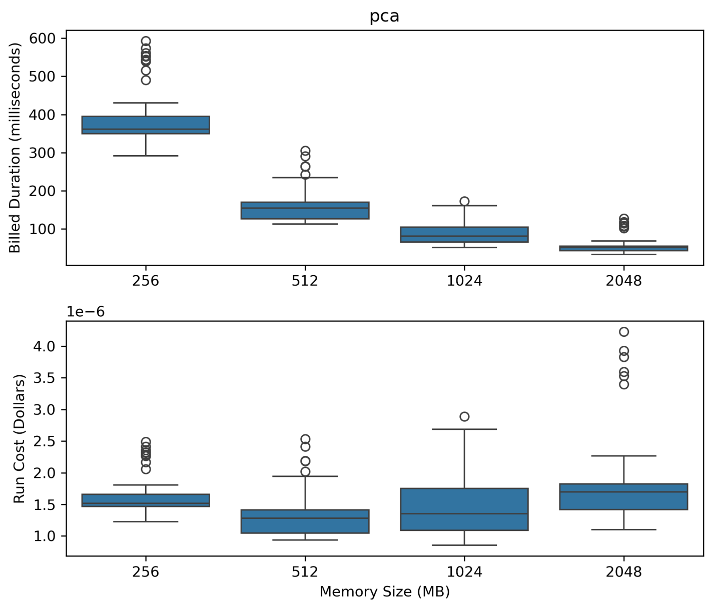

Figure 5.5: Performance and cost of pca for lowered memory and batch size 100.

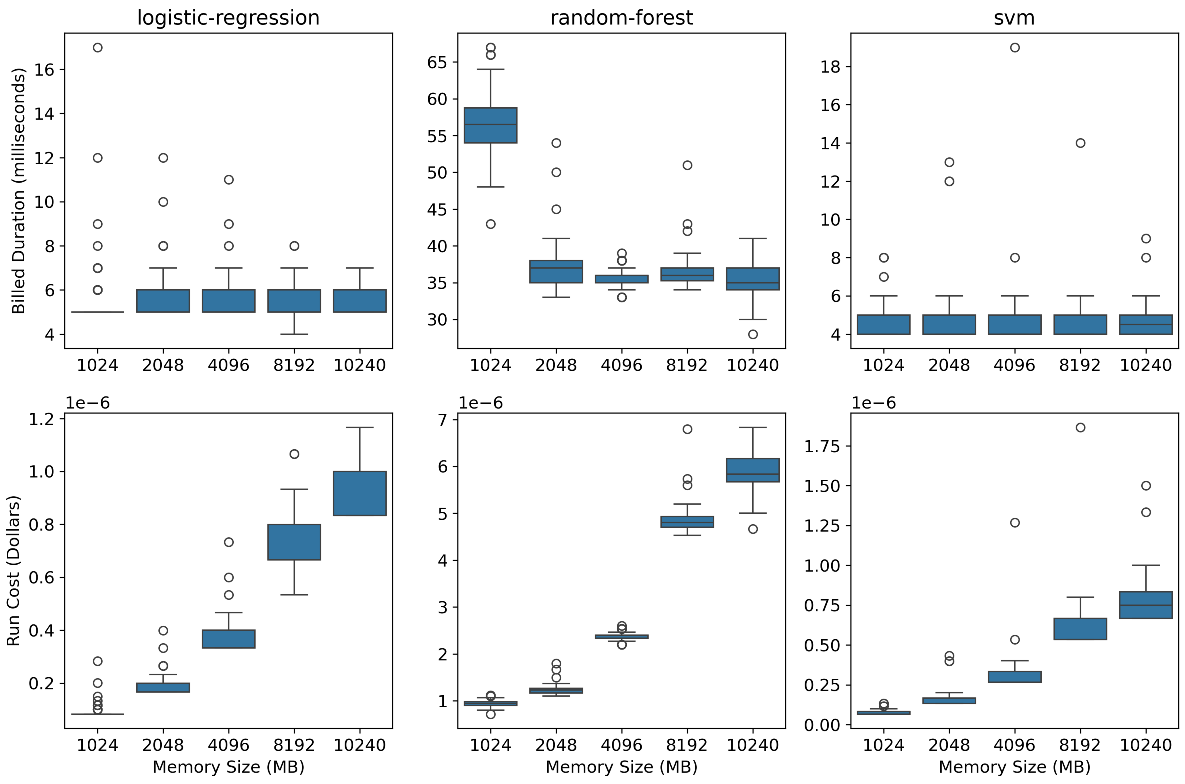

Figure 5.6: Performance and cost of supervised algorithms.

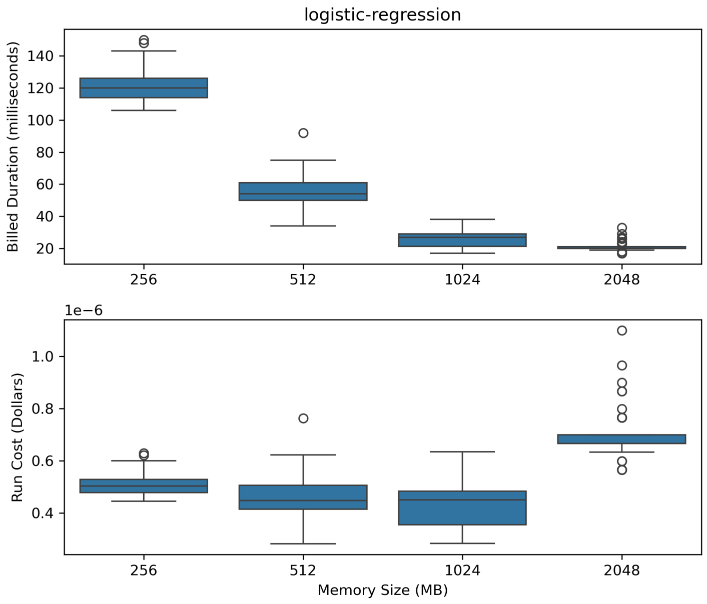

Figure 5.7: Performance and cost of logistic-regression for lowered memory and batch size 1000.

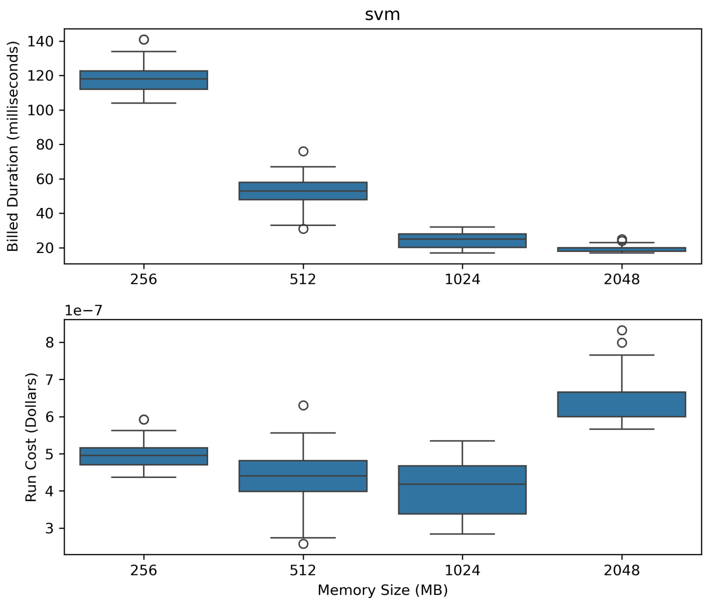

Figure 5.8: Performance and cost of svm for lowered memory and batch size 100.

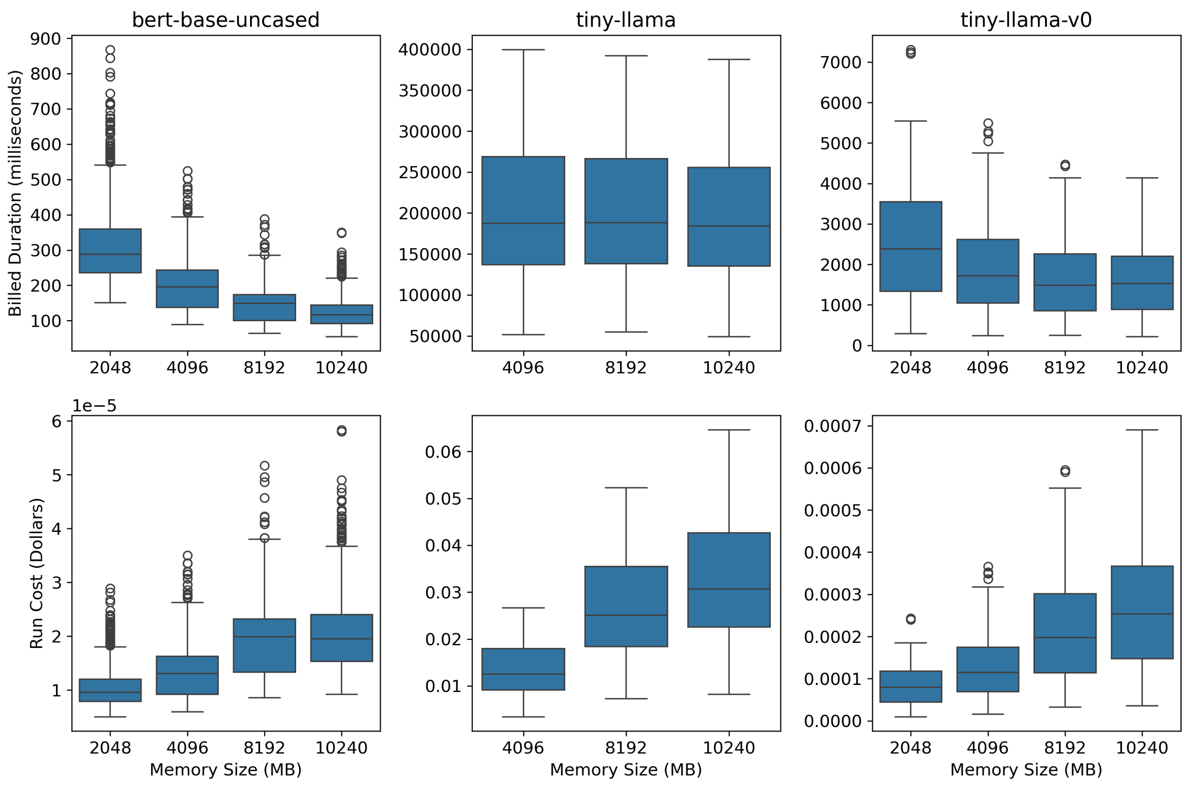

Figure 5.9: Performance and cost of large language models.

In the case of large language models, we only used the memory allocations that could load in the model. This means that we only have one model for the largest of the ones we tested. We also ran all of these tests with a batch size of only one item since one item already takes quite a long time. The results of these tests can be found in Figure 5.9. Based on these results, we can find interesting findings for people trying to run LLMs in FaaS. Something that should be noted is that for these models, the default ephemeral storage size needs to be increased, which also comes with a hidden cost. This is something to consider when trying to execute a similar experiment.

As we already mentioned, most of the experiments were run using a maximum parallelization of 10 instances at a time. This was mainly done to ensure we obtained a dataset with enough hot runs. By running the SVM algorithm with multiple memory sizes and concurrent parallel instances, we plot the execution time of each request, which directly translates to the cost. The results of these experiments are plotted in Figures 5.10 and 5.11. At the top of these plots, we can see how much time it takes to handle 100 hot and cold requests. This is separated in the initialization and processing of the request. At the bottom we find the unique instances used to handle the 100 requests.

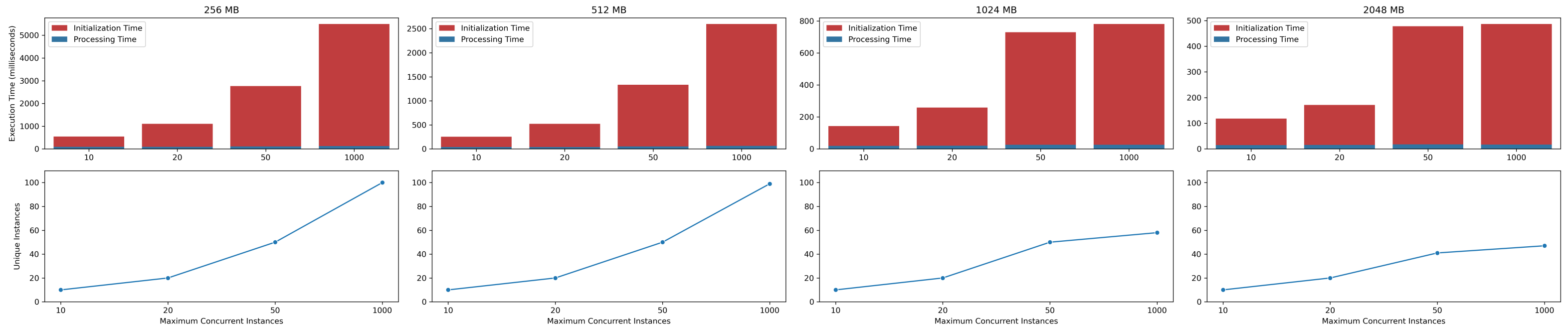

Figure 5.10: Execution time of svm for different maximum concurrent instances grouped by memory.

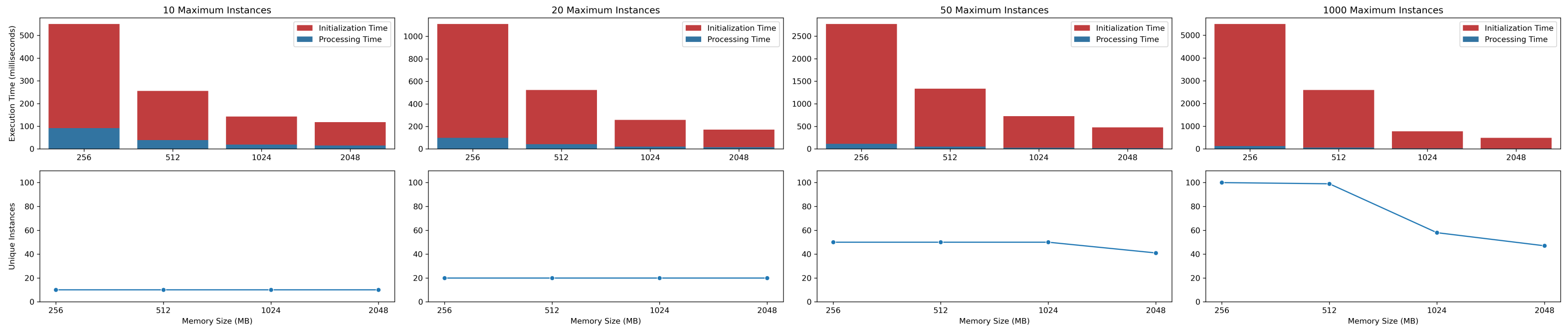

Figure 5.11: Execution time of svm for different maximum concurrent instances grouped by number of instances.

5.2 Answering the Research Questions

In this section we will discuss the introduced figures and reason about how they can answer research question 2, 3, and 4. For research question 1 the reader can refer to Chapter 2.

RQ2: How do different serverless computing resource configurations impact the performance of machine learning inference?

In evaluating the impact of serverless computing resource configurations on the performance of machine learning inference, the experimental results demonstrate notable variations across different memory allocations. Increasing the memory allocation generally leads to a decrease in runtime, as seen in the supervised and unsupervised algorithm experiments. In quite a couple of our figures we can observe an almost exponential decrease in billable duration for each run, this can be seen for example in Figures 5.4, 5.5, 5.7 and 5.8. However, clearly there is a limit to this. At the 2048MB configuration mark the runtimes for some of these algorithms are around the 20 milliseconds, at which point there are little reductions to be made. When someone would run into such an observations when running function we would advise against increasing the memory size.

For the large-language models the plotted results are very different. First of all, when we look at Figure 5.9 we can see that the billed durations are way higher than any that we had seen before. Especially in the cases of tiny-llama and tiny-llama-v0 where they are basically unusable for any use-case. For the bert-base-uncased algorithm perhaps a use-case could be found where this performance is acceptable. We can also see that for this algorithm the performance is still increasing even at the maximum memory amount of 10240MB. Unfortunately no more memory can be allocated and therefore we cannot tell if this trend would continue.

RQ3: How do serverless resource configurations impact the cost of machine learning inference?

In terms of cost we can see that in a number of our figures the cost remain relatively stable across the memory allocations, this is the case in for example Figure 5.5. However, in others we do see an increase at the higher allocations of memory, in Figures 5.4, 5.7 and 5.8 we see that when we reach the 2048MB configuration the cost per run increases quite significantly again. Considering that we double the memory for this last step we still can see that the cost does not scale linearly like the pricing from the platform which we saw in Figure 5.1, since the cost is not doubled. This could possibly be a worthwhile tradeoff for a use-case where real-time data processing is crucial.

For the large-language models in Figure 5.9 we see that the costs almost rise linearly for most of them. Only in the case bert-base-uncased do we see that at the highest memory allocation the rise in cost is not as high as the ones before. However, this could very well also be explained by the last incremental step being not a doubling of the memory since it is only an increase of 1.25 times the previous value. Unfortunately this is the maximum memory allocation that the platform allows. Comparing the costs per requests for the LLMs to the ones of the supervised and unsupervised algorithms we can see that they are significantly higher. And taking into account that these only run for a single item, the cost per item processed is even more significant.

It also seems that the number of maximum concurrent instances can have a big impact on the maximum cost as was seen in Figures 5.10 and 5.11. Consider this approach particularly when immediate item processing isn't necessary. In scenarios with highly variable workloads, numerous function instances may be created at one point, all with a very costly cold-start, just to shut down again after processing a single request. Reducing this number of concurrent instances can significantly decrease cold starts, though it means items may experience a slight delay before processing resumes. This trade-off between cost and processing speed should be evaluated based on specific requirements.

RQ4: What is the relation between cost and performance of the different serverless resource configurations?

In Figure 5.1 we've seen that the pricing for the different memory configurations scales linearly. This means that when we double our memory, as long as the runtime reduces by half our cost stays even. Starting off with the supervised and unsupervised algorithms we can even see this in a couple of our figures, in Figure 5.4 for example we see that out cost from 256MB to 1024MB stays almost even. This corresponds to the runtimes as well, where we see a approximate halfing when we double the memory. Only when we start increasing to the 2048MB configuration do we see that the pricing starts picking up again due to the reduction in runtime being too little. A very similar result was seen in both Figures 5.7 and 5.8 where they also end up performing best at the 1024MB configuration in terms of performance and cost.

However, when we look at the Figure 5.5 we see that the performance cost is still pretty similar at the 2048MB configuration. This could be because the general runtime is already way longer than the other algorithms, still taking around 100 milliseconds at the 1024MB configuration as opposed to approximately 20ms for the k-means algorithm. This might mean that there is more performance headroom in this particular algorithm to make use of these computationally higher configurations. Therefore, whilst for most of these algorithms tested the 1024MB configuration ended up being most optimal it does not always have to be the case. Our recommendation therefore would be to start with a configuration of 1024MB, and when it is observed that an algorithm still takes more than 100ms, one could try to increase the memory allocation to see if there is still room for improvement.

For the large-language models in Figure 5.9, generally both the performance and cost is both dreadful. Since the performance onmly increased slightly we can really see the linear increase of the memory pricing take place. However, when we look at the bert-based-uncased results we do see that the pricing seems to stabilizes off a bit at the 10240MB memory configuration. This perhaps indicates the tipping point at which this algorithm starts to be able to run slightly more efficiently. Unfortunately, this is the most memory that AWS Lambda functions can be allocated, and therefore we cannot see if further increases have this effect as well. The different behavior observed in large language models compared to more traditional ML algorithms underscores the complexity of deploying such models in a serverless environment. The scaling of costs and benefits, particularly evident in large language models, demands a more nuanced approach to resource allocation and optimization, which is for example available on traditional hardware but is absent in these particular serverless functions with the restrictions at hand.

5.3 Discussion

5.3.1 Summary

Our analysis revealed that memory allocation significantly impacts the performance and cost of machine learning algorithms executed in a serverless environment. As memory increases, runtime generally decreases, particularly for supervised and unsupervised machine learning models, with diminishing returns observed beyond 2048MB. However, LLMs exhibited substantial latency and cost, which may limit their practical deployment in serverless setups like AWS Lambda.

5.3.2 Interpretation

The observed decrease in runtime with increased memory allocation suggests that serverless platforms can effectively accelerate certain types of machine learning inference by optimizing resource configurations. Yet, the performance benefit stabilizes at higher memory levels, indicating a threshold beyond which additional resources do not translate into proportional gains. LLMs' poor performance underscores their extensive resource demands, which exceed what current serverless infrastructures can cost-effectively provide.

5.3.3 Implication

These results are crucial for stakeholders considering serverless computing for machine learning tasks. They highlight the importance of tailored resource allocation to balance performance and cost. For LLMs, the findings suggest that serverless platforms might currently be unsuitable for deployment without significant optimizations or a rethinking of resource management policies.

5.3.4 Threats to Validity

There are a number of factors that could threaten the validity of these findings.

Firstly, these results may not be generalizable to other serverless platforms. Different platforms could for example handle parallelism differently resulting in different performance [47]. These types of differences between platform should be kept in mind.

Furthermore, the metrics captured may not cover all dimensions that are relevant for a specific use case. Perhaps a more useful metric in some cases would be the time it took for all batches to be completed from start to finish, or another use-case may have less bursty workloads but instead a more consistent stream.

Also, worth noting is that these results were all captured inside a single region, it could entirely be possible that the resource allocation done by the platform is entirely region dependent. If this is the case, this would mean that the performance of some algorithms would vary for each configuration depending on the region. On a similar note, these results might be less relevant in due time due to the time-bound nature of the provider. Cloud providers frequently update their services, pricing models, and underlying hardware. The results may become less relevant over time as the platform evolves.

Finally, these results do not compare the findings to a traditional deployment on the same cloud platform and without a direct comparison to non-serverless deployments of the same algorithms, it's difficult to contextualize the relative benefits or drawbacks of the serverless approach.

5.3.5 Future Works

Future research could build upon this work in a variety of ways. The enhancements and extensions of the current study could for example involve exploring additional algorithms or examining other aspects of the machine learning lifecycle.

Firstly, with the provided tooling, future research could focus on a more in-depth look into the algorithms shown to be most viable. More tests could for example be run for the Bert model to get a more complete picture of whether large language models could be viably run on FaaS.

Secondly, a cross-platform comparison could be interesting. Expanding the study to include other major serverless platforms like Google Cloud Functions, Azure Functions, and IBM Cloud Functions to provide a comprehensive comparison across providers would give an interesting overview as to what platform is most suited for running these types of workloads.

Furthermore, automated resource configuration could be researched, Development and evaluation if algorithms or tools that can automatically determine the optimal configuration (memory, concurrency, etc.) for a given ML workload could be integrated with the tooling to see if an optimum can be reached automatically.

Additionally, the tooling could also be expanded to test other stages in the machine learning lifecycle. It would be interesting to see whether distributing a training workload over several function instances would provide reasonable performance.

Lastly it would be valuable to compare the current findings, as well as new findings, to a traditional approach on the same cloud provider. For example an approach where instead of using serverless functions a virtual private server is being used instead. Especially when comparing the costs of both this could prove very interesting.

Chapter 6

Conclusion

This thesis aimed to evaluate the feasibility of deploying various machine learning algorithms on AWS Lambda, focusing on four key research questions. The findings for each question are summarized in the following:

The review of current literature and experiments indicated there are implementation proposals of machine learning algorithms on AWS Lambda. However, a lot of these proposals do not provide open-source code, and the ones that do require significant effort to get up and running with. Therefore, we opted into writing the implementations of the functions ourselves. By combining a number of tools we were able to build a toolkit that allows us to automatically benchmark our functions on Lambda.

Memory allocation and batch size adjustments were found to significantly influence the performance of deployed models up until a certain point. Memory settings above 2048 MB generally provided diminishing returns in terms of computational speed, except where specific workloads explicitly benefited from higher allocations.

The cost efficiency of machine learning inference varied significantly across different configurations. Lower memory allocations (such as 1024 MB) were often sufficient for maintaining a balance between performance and cost, especially for unsupervised and supervised algorithms. Conversely, the cost associated with LLMs was much higher, reflecting the greater resource demands.

A critical trade-off was identified between cost and performance, particularly evident in the more demanding models. While higher memory allocations generally improved performance, this most of the time not cost-effective. The study demonstrated that careful optimization of memory and concurrency settings could yield substantial cost savings while maintaining high performance.

Overall, these results show that while AWS Lambda presents as a viable option for ML workloads, careful consideration of specific algorithm requirements and resource configurations is essential to optimize both performance and cost efficiency.

Bibliography

[1] Amine Barrak, Fabio Petrillo, and Fehmi Jaafar. "Serverless on Machine Learning: A Systematic Mapping Study". In: *IEEE Access* 10 (2022), pp. 99337–99352. DOI: 10.1109/ACCESS.2022.3206366.

[2] Y. Yang et al. "INFless: A native serverless system for low-latency, high-Throughput inference". In: cited By 26. 2022, pp. 768–781. DOI: 10.1145/3503222.3507709.

[3] J. Carreira et al. "Cirrus: A Serverless Framework for End-To-end ML Workflows". In: cited By 107. 2019, pp. 13–24. DOI: 10.1145/3357223.3362711.

[4] M. Yu et al. "Gillis: Serving large neural networks in serverless functions with automatic model partitioning". In: vol. 2021-July. cited By 25. 2021, pp. 138–148. DOI: 10.1109/ICDCS51616.2021.00022.

[5] A. Ali et al. "Batch: Machine learning inference serving on serverless platforms with adaptive batching". In: vol. 2020-November. cited By 67. 2020. DOI: 10.1109/SC41405.2020.00073.

[6] Pablo Gimeno Sarroca and Marc Sánchez-Artigas. "MLLESS: Achieving cost efficiency in serverless machine learning training". In: *Journal of Parallel and Distributed Computing* 183 (2024). Cited by: 1. DOI: 10.1016/j.jpdc.2023.104764.

[7] U. Elordi et al. "Benchmarking Deep Neural Network Inference Performance on Serverless Environments with MLPerf". In: *IEEE Software* 38.1 (2021). cited By 5, pp. 81–87. DOI: 10.1109/MS.2020.3030199.

[8] Tam n. Nguyen. "Holistic cold-start management in serverless computing cloud with deep learning for time series". In: *Future Generation Computer Systems* 153 (2024). Cited by: 0; All Open Access, Green Open Access, 312 – 325. DOI: 10.1016/j.future.2023.12.011.

[9] Yi-rui Huang et al. "GeoPM-DMEIRL: A deep inverse reinforcement learning security trajectory generation framework with serverless computing". In: *Future Generation Computer Systems* 154 (2024). Cited by: 0, 123 – 139. DOI: 10.1016/j.future.2024.01.001.

[10] J. Scheuner and P. Leitner. "Function-as-a-Service performance evaluation: A multivocal literature review". In: *Journal of Systems and Software* 170 (2020). cited By 49. DOI: 10.1016/j.jss.2020.110708.

[11] José Manuel Ortega Candel, Francisco José Mora Gimeno, and Higinio Mora Mora. "Generation of a dataset for DoW attack detection in serverless architectures". In: *Data in Brief* 52 (2024). Cited by: 0; All Open Access, Gold Open Access, Green Open Access. DOI: 10.1016/j.dib.2023.109921.

[12] Amanda Issac et al. “Development and deployment of a big data pipeline for field-based high-throughput cotton phenotyping data”. In: *Smart Agricultural Technology* 5 (2023). Cited by: 2; All Open Access, Gold Open Access. DOI: `10.1016/j.atech.2023.100265`.

[13] A. Kaplunovich and Y. Yesha. “Refactoring of Neural Network Models for Hyperparameter Optimization in Serverless Cloud”. In: cited By 1. 2020, pp. 311–314. DOI: `10.1145/3387940.3392268`.

[14] P. Silva, D. Fireman, and T.E. Pereira. “Prebaking functions to warm the serverless cold start”. In: cited By 45. 2020, pp. 1–13. DOI: `10.1145/3423211.3425682`.

[15] A. Fuerst and P. Sharma. “FaasCache: Keeping serverless computing alive with greedy-dual caching”. In: cited By 73. 2021, pp. 386–400. DOI: `10.1145/3445814.3446757`.

[16] M.S. Kurz. “Distributed double machine learning with a serverless architecture”. In: cited By 11. 2021, pp. 27–33. DOI: `10.1145/3447545.3451181`.

[17] D. Mvondo et al. “OFC: An opportunistic caching system for FaaS platforms”. In: cited By 41. 2021, pp. 228–244. DOI: `10.1145/3447786.3456239`.

[18] J. Jiang et al. “Towards Demystifying Serverless Machine Learning Training”. In: cited By 52. 2021, pp. 857–871. DOI: `10.1145/3448016.3459240`.

[19] J. Tagliabue. “You Do Not Need a bigger boat: Recommendations at Reasonable Scale in a (Mostly) serverless and open stack”. In: cited By 8. 2021, pp. 598–600. DOI: `10.1145/3460231.3474604`.

[20] J. Wen et al. “An empirical study on challenges of application development in serverless computing”. In: cited By 26. 2021, pp. 416–428. DOI: `10.1145/3468264.3468558`.

[21] Ashraf Mahgoub et al. “WISEFUSE: Workload Characterization and DAG Transformation for Serverless Workflows”. In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6.2 (2022). Cited by: 14; All Open Access, Bronze Open Access. DOI: `10.1145/3530892`.

[22] Yunzhuo Liu et al. “FuncPipe: A Pipelined Serverless Framework for Fast and Cost-Efficient Training of Deep Learning Models”. In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6.3 (2022). Cited by: 2; All Open Access, Green Open Access. DOI: `10.1145/3570607`.

[23] Diandian Gu et al. “ElasticFlow: An Elastic Serverless Training Platform for Distributed Deep Learning”. In: vol. 2. Cited by: 1. 2023, 266 – 280. DOI: `10.1145/3575693.3575721`.

[24] Marcin Copik et al. “FMI: Fast and Cheap Message Passing for Serverless Functions”. In: Cited by: 3; All Open Access, Green Open Access. 2023, 373 – 385. DOI: `10.1145/3577193.3593718`.

[25] Enrico Galimberti et al. “OSCAR-P and aMLLibrary: Performance Profiling and Prediction of Computing Continua Applications”. In: Cited by: 1; All Open Access, Bronze Open Access. 2023, 139 – 146. DOI: `10.1145/3578245.3584941`.

[26] Qiangyu Pei et al. “AsyFunc: A High-Performance and Resource-Efficient Serverless Inference System via Asymmetric Functions”. In: Cited by: 1. 2023, 324 – 340. DOI: 10.1145/3620678.3624664.

[27] Matt Baughman et al. “Tournament-Based Pretraining to Accelerate Federated Learning”. In: Cited by: 0. 2023, 109 – 115. DOI: 10.1145/3624062.3626089.

[28] Ionut Predoaia and Pedro García-López. “Leveraging Intra-Function Parallelism in Serverless Machine Learning”. In: Cited by: 0. 2023, 36 – 41. DOI: 10.1145/3631295.3631399.

[29] Mehrad Ansari and Andrew D. White. “Serverless Prediction of Peptide Properties with Recurrent Neural Networks”. In: *Journal of Chemical Information and Modeling* 63.8 (2023). Cited by: 6; All Open Access, Green Open Access, Hybrid Gold Open Access, 2546 – 2553. DOI: 10.1021/acs.jcim.2c01317.

[30] Logan Ward et al. “Cloud Services Enable Efficient AI-Guided Simulation Workflows across Heterogeneous Resources”. In: Cited by: 0; All Open Access, Green Open Access. 2023, 32 – 41. DOI: 10.1109/IPDPSW59300.2023.00018.

[31] Adithya Hegde, Sameer G. Kulkarni, and Abhinandan S. Prasad. “COUNSEL: Cloud Resource Configuration Management using Deep Reinforcement Learning”. In: Cited by: 0. 2023, 286 – 298. DOI: 10.1109/CCGrid57682.2023.00035.

[32] F. Xu et al. “λdNN: Achieving Predictable Distributed DNN Training with Serverless Architectures”. In: *IEEE Transactions on Computers* 71.2 (2022). cited By 21, pp. 450–463. DOI: 10.1109/TC.2021.3054656.

[33] Amine Barrak et al. “Exploring the Impact of Serverless Computing on Peer To Peer Training Machine Learning”. In: Cited by: 1; All Open Access, Green Open Access. 2023, 141 – 152. DOI: 10.1109/IC2E59103.2023.00024.

[34] Jianqi Zhang et al. “SCP4ssd: A Serverless Platform for Nucleotide Sequence Synthesis Difficulty Prediction Using an AutoML Model”. In: *Genes* 14.3 (2023). Cited by: 0; All Open Access, Gold Open Access, Green Open Access. DOI: 10.3390/genes14030605.

[35] Bokkeun Kim et al. “iPaaS: Intelligent Paging as a Service”. In: *IEEE Network* 37.2 (2023). Cited by: 0, 238 – 245. DOI: 10.1109/MNET.123.2100764.

[36] Ron C. Chiang. “Contention-aware container placement strategy for docker swarm with machine learning based clustering algorithms”. In: *Cluster Computing* 26.1 (2023). Cited by: 2, 13 – 23. DOI: 10.1007/s10586-020-03210-2.

[37] Amine Barrak et al. “SPIRT: A Fault-Tolerant and Reliable Peer-to-Peer Serverless ML Training Architecture”. In: Cited by: 0; All Open Access, Green Open Access. 2023, 650 – 661. DOI: 10.1109/QRS60937.2023.00069.

[38] *AWS Lambda Operator Guide: Computing Power.* Accessed: 07-18-2024. Amazon Web Services. URL: https://docs.aws.amazon.com/lambda/latest/operatorguide/computing-power.html.

[39] *AWS Lambda Operator Guide: Execution Environments.* Accessed: 07-18-2024. Amazon Web Services. URL: https://docs.aws.amazon.com/lambda/latest/operatorguide/execution-environments.html.

[40] Hugging Face. *Tiny Llama 1.1B Chat v1.0.* https://huggingface.co/TinyLlama/TinyLlama-1.1B-Chat-v1.0. Accessed: June 2024. 2023.

[41] Hugging Face. *Tiny LLama v0.* https://huggingface.co/Maykeye/TinyLLama-v0. Accessed: June 2024. 2023.

[42] Google and Hugging Face. *BERT-Base-Uncased.* https://huggingface.co/google-bert/bert-base-uncased. Accessed: June 2024. 2023.

[43] Jerome Blanchet. *ARC AI2 Reasoning Challenge.* https://www.kaggle.com/datasets/jeromeblanchet/arc-ai2-reasoning-challenge. Accessed: June 2024. 2019.

[44] Jesse Charis. *Emotion Dataset for NLP.* https://github.com/Jcharis/end2end-nlp-project/blob/main/notebooks/data/emotion_dataset_raw.csv. Accessed: June 2024. 2020.

[45] *ACM Policies for Artifact Review and Badging.* Accessed: 07-18-2024. Association for Computing Machinery. URL: https://www.acm.org/publications/policies/artifact-review-badging.

[46] *AWS Lambda Pricing.* https://aws.amazon.com/lambda/pricing/. Online; accessed: 2024-05-31.

[47] Daniel Barcelona-Pons and Pedro García-López. "Benchmarking parallelism in FaaS platforms". In: *Future Generation Computer Systems* 124 (2021), pp. 268–284. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2021.06.005. URL: https://www.sciencedirect.com/science/article/pii/S0167739X21001990.