Vrije Universiteit Amsterdam

BSc. Thesis, Report

A Reference Architecture for Serverless Machine Learning Systems

Author: Tiberiu Iancu (2659445)

1st supervisor:	Prof. dr. ir. Alexandru Iosup
daily supervisor:	M.Sc. ir. Tijl van Vliet (Hadrian)
2nd reader:	Dr. ir. Animesh Trivedi

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

May 30, 2023

Contents

Abstract

In recent years, serverless has emerged as a promising cloud computing paradigm, allowing users to deploy granularly-billed applications without the need to manage the underlying infrastructure. Simultaneously, machine learning (ML) has seen wide-spread adoption across many industries: healthcare, finance, entertainment, etc. The success of ML applications has attracted much research into improving the performance and the operation of ML models.

Due to the relative novelty of both serverless computing and machine learning, there is currently a knowledge gap in the architectural patterns of serverless ML systems. To address this issue, in this thesis we propose a reference architecture for serverless ML. We validate the reference architecture against 19 open- and closed-source serverless ML platforms, and find a good fit. In the process, we document emerging patterns in the operations of the serverless ML systems.

To further understand the performance characteristics of serverless ML systems, we make ML model deployments in a serverless framework in Kubernetes. Upon analysing the results, we have found that vertical CPU scaling of serverless ML models improves performance per core by 50%.

Data analysis notebooks and experiment code can be found at `https://github.com/tiberiuiancu/thesis-experiments-public`.

Chapter 1

Introduction

1.1 Context on Machine Learning and Serverless Computing

Artificial intelligence (AI), and especially machine learning (ML), is developing at a rapid rate. In recent years AI has excelled, and often outperformed humans, at a variety of tasks: computational photography [1] (reaching every smartphone user), image and video segmentation and classification [2, 3, 4], and Natural Language Processing (NLP) [5, 6], among others. Perhaps most notably, ChatGPT [7], a chatbot service launched by OpenAI, is the fastest ever online platform to reach 100 million users [8]. The recently emerging practice of pre-training large models and releasing the parameters to the public for free on easily accessible platforms such as Hugging Face [9], enables developers to easily incorporate ML in their products, accelerating the adoption of AI. The impact and future growth of AI is undeniable.

Figure 1.1 depicts a high level overview of the main stages in the life-cycle of a machine learning model. We distinguish two main stages: training and inference. During training, the model is optimized using samples from a dataset. The model is then deployed. In the inference stage, the model is used to make predictions on new data samples that it has not seen before. During this stage, the model could, for example, run in the back-end of a web service.

Serverless computing is an emerging cloud computing paradigm that hides the operational logic from the developer, and provides a granular billing model, with on-demand provisioning [10]. Serverless computing enables developers to easily scale applications as user demand increases, and only pay for the resources used. In other words, it aims to be a more efficient and scalable cloud computing model. Thanks to these two characteristics, serverless has seen wide-spread adoption. Since the introduction of AWS Lambda in 2014 [11], major cloud platform serverless architectures are often used in conjunction with events [10]. Events are messages that components of the system (e.g., applications, server-

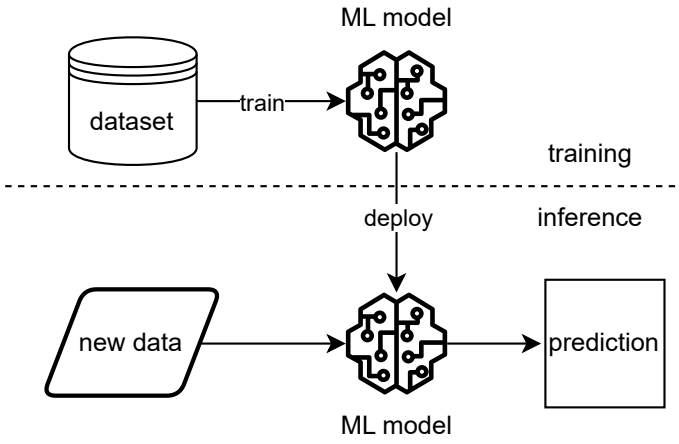


Figure 1.1: A high level illustration of the stages in machine learning model life-cycles.

less functions, services, etc.) may produce or consume, to communicate with each other. The main advantage of such architecture is that it enables a great level of modularity and independent scalability of each component.

1.2 Problem Statement

Due to the success of AWS Lambda, researchers have gained interest in applying serverless computing to machine learning, for both training and inference [12]. Furthermore, cloud platforms such as AWS SageMaker [13] already offer serverless machine learning options [14]. Given the fast development pace and large societal outreach of both serverless computing and machine learning, it is important to understand the performance characteristics and architectural patterns of systems combining the two technologies.

By designing a reference architecture, we gain a better understanding into the components and tools that are currently available. In this work we propose a reference architecture for serverless machine learning systems. We document the characteristics and architectural patterns associated with serverless machine learning workloads, and provide a performance analysis on real-world deployments. This bachelor thesis was conducted as part of an internship at Hadrian [15]. The models studied in this research are part of Hadrian's system.

Index terms: serverless, machine learning, MLOps, reference architecture

1.3 Research Questions

In this work, we aim to answer three research questions:

RQ1 What is the reference architecture of serverless machine learning systems?

To the best of our knowledge, there is currently no reference architecture for serverless ML systems. Due to the novelty and widespread adoption of serverless and ML technologies, it is important to have an understanding of the different components, common practices, and the range of tools.

RQ2 What can we learn about the operation of serverless machine learning systems through data collection and data-driven analysis of real world deployments?

We answer this research question by performing a use-case study on real-world ML deployments. By conducting this analysis, we will understand the nature of the workload. This will give valuable insight into the interactions between components, and the general computational characteristics of each component.

RQ3 How to analyse the performance of serverless machine learning systems?

Having analysed specific use-cases, we perform a sensitivity study on several variables in our deployments. Answering this research question will give us understanding into how the performance of the system changes under a range of conditions. This gives us valuable information on how to properly set up serverless ML systems, and how to prevent and recover from failures.

1.4 Research Methodology

In this research we employ three broad research methodologies: design, use-case study, and experimental research:

M1 Design. We answer **RQ1** by providing a design for the reference architecture of serverless ML. First, we review MLOps and serverless literature to provide an early version of the reference architecture. We then test the design against open and closed source platforms by mapping their components to the reference architecture (matching components in the reference architecture with their counterparts in the platform). We employ an iterative process: if a platform does not map to the reference architecture, we adjust the reference architecture and revise the mapping.

M2 Use-case study. We answer **RQ2** by collecting data on serverless ML model deployments and performing a data-driven analysis.

M3 Experiments. Using the insights gained by answering **RQ2**, we run further experiments to test the sensitivity of the system to several variables. We answer **RQ3** by designing and running experiments, and deriving insights from the collected data.

1.5 Thesis Contributions

In this work, we propose a reference architecture for serverless ML systems, and subsequently examine the performance characteristics of this class of systems. The reference architecture is based upon the SPEC-RG FaaS reference architecture [16], which we extend with layers and components particular to ML and MLOps. The data-driven analysis is based on already existing ML model deployments in Hadrian's system. The contributions in this thesis are presented below:

1. **Reference architecture** – conceptual. We provide a high-level reference architecture of serverless ML systems that we derive from the SPEC-RG FaaS reference architecture (**RQ1**).
2. **Sensitivity study** – experimental. We design and run experiments to gain an understanding into the performance characteristics of serverless ML systems (**RQ3**).
3. **Data & Analysis Notebooks** – artifact. We release the collected data, as well as the data analysis notebooks, for open access.
4. **Experiment script** – artifact. We release the Python script used during experimentation open-source.

1.6 Plagiarism Declaration

I confirm that this thesis work is my own work, is not copied from any other source (person, Internet, or machine), and has not been submitted elsewhere for assessment.

1.7 Thesis Structure

This thesis is structured as follows: chapter 2 introduces the concepts and terminology necessary to understand this work. Chapter 3 introduces the main contribution of this work: the reference architecture. Here we give an overview of the model, explain the novel contributions, and test the model against popular serverless ML platforms. Chapter 4 presents a data-driven analysis on real-world machine learning model deployments, followed by presenting the results of three experiments. Chapter 5 presents the related serverless computing and machine learning (operation) literature. Finally, chapter 6 summarises the work, and presents possible directions for future research, as well as weak points and threats to validity of the thesis.

Chapter 2

Background

2.1 Serverless Computing and Microservices

Serverless computing is a cloud computing paradigm that allows users to run granularly billed applications without having to provision the underlying infrastructure. [10]

Serverless applications often use an event-driven approach. Events are messages that applications can either produce, consume or observe. In such architecture, the complex state of a system can be broken down into many small events, and the code-base can be split into small applications (microservices) that each consume certain event types. This reduces complexity, and offers higher levels of scalability, as each component may scale independently to accommodate demand. Furthermore, such loosely-coupled architecture enables easy module replacement (for example, when performing updates).

The **microservice** architecture [17] refers to the practice of splitting an application into multiple components (each called a microservice), each fulfilling a small and precise role. Microservice architectures often leverage container technology (e.g., Docker [18]) for easy scalability and packaging.

Function-as-a-service (FaaS) is perhaps the most popular instance of serverless computing. Using FaaS, users can deploy short-lived, stateless functions, by only providing the function code. Subsequently, the user can invoke the function over the network. The underlying infrastructure is provisioned by the cloud platform to meet the scaling demands. Most cloud providers offer FaaS solutions (e.g., AWS Lambda [19], Google Cloud Functions [20]). The terms *FaaS* and *serverless* are sometimes used interchangeably. In this research we study serverless as a whole.

2.2 Kubernetes

The massive success of containerization seen with **Docker** [18] (2013) induced the need for a management solution for containerized applications. **Kubernetes** [21] (K8s) is an open-source resource manager for containers; it handles deployment, scalability, and running state. In Kubernetes, the developer defines the desired state of the system, and the framework brings the system to the desired state by creating, deleting or restarting **pods**.

> **Pods** are the smallest deployable unit in Kubernetes. A pod is made up of one or more containers that together provide a single purpose.

Users may specify, for each pod, "requests" and "limits" regarding the resource consumption (i.e., memory and CPU utilization). Requests represent the minimum amount of the respective resources that the pod needs to function. If the requested amount is not available in the cluster, the pod remains in the *pending* status until the resources become available. Limits represent the maximum amount of the given resource that the pod may consume. In the case of memory, K8s kills the pod if the memory limit is exceeded. For CPU utilization, K8s throttles the pod (schedules the pod to run for less than the limit amount) if the limit is surpassed for longer periods of time.

Although Kubernetes does not offer a full serverless stack out-of-the-box, a serverless environment can be achieved thanks to Kubernetes' vast plugin ecosystem. Knative [22] is one of the most popular examples of such serverless plugins. It offers three components: Knative Eventing (events framework), Knative Serving (serverless framework), and Knative Functions (FaaS framework). In this research we use Knative Serving and Knative Eventing, as these two components are already deployed in Hadrian's cluster.

In Knative Serving, each application is called a Knative service (or Kservice). For each Kservice, Knative manages the scale, i.e., the number of active pods, where each pod is running the same application. When demand is high, Knative may create more pods. Similarly, when demand is low, Knative may destroy pods to save resources. Knative can scale between 0 (scale-to-zero) and as many as the K8s cluster allows (although the minimum and maximum may also be manually configured by the user). Note that Knative Serving does not operate in the resource management plane, and cannot scale the amount of resources (i.e., CPU and memory amounts).

2.3 Machine Learning Operations

This section presents the common machine learning and machine learning operations (MLOps) terminology.

> **MLOps** refers to the practice of deploying and maintaining machine learning models and pipelines in a production environment. [23]

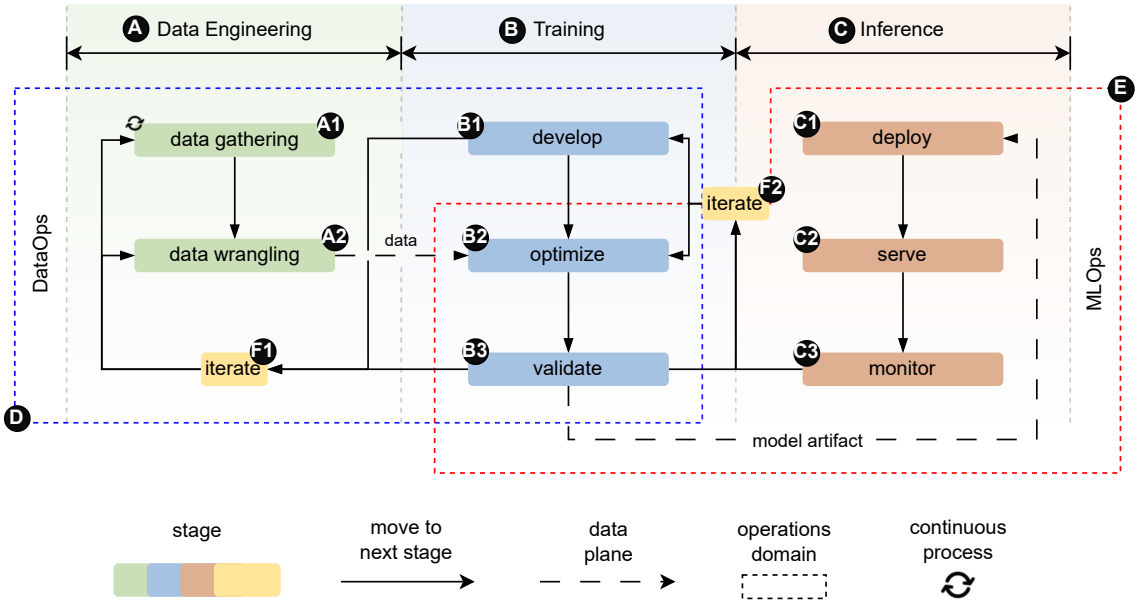


Figure 2.1: The three main stages in the machine learning model life-cycle: data engineering (A), training (B), and inference (C). The life-cycle starts with data gathering, which is an iterative process.

Figure 2.1 shows the three main stages and the substages of the ML model life-cycle. The first step is data gathering (A1). A few substages can be identified: data collection, cleaning, extraction, labelling, exploration, statistical characterization. The data is then wrangled (A2) – integrated in the system, formatted accordingly, transformed to fit the task (and possibly augmented).

After the Data Engineering stage, an ML researcher or data scientist develops a model based on a novel, or pre-existing algorithm (B1) [23]. Development can be a vast process and may involve multiple people with business, data science, and engineering expertise [23]. Within development, we can identify several substages: specifying the modelling goals, selecting a modelling approach, building the model in code, and (unit) testing. The model is then optimized using the collected data (B2). The training process often requires engineers to pick specific hardware and distributed training algorithms for the use-case and specific model [24, 25, 26]. The optimization step results in a **model artifact**, which is tested for validity and performance (B3). During development (B1) or model validation (B3), the dataset may be iterated upon (F1), either by collecting more data, or by applying different data wrangling methods. Similarly, the modelling approach itself may not be sufficiently performant. In this case, the project returns to the development phase (F2), or the model is re-optimized (e.g., hyperparameter optimization [27]).

The practice concerned with data pipelines, data transformation, data availability, and

data storage is DataOps (D). DataOps is outside the scope of this research.

Finally, in the inference (C) stage, the model is used to make predictions on new data, in a production environment.

Here we can distinguish three substages. First, the model artifact produced during training is deployed (C1). The deployment process varies greatly across ML platforms and cloud providers. The application that exposes the model for prediction (usually through an API endpoint) is called a **model server** (C2). The model is then monitored (C3). We can differentiate two main types of monitoring: application level metrics (APM), and prediction monitoring: APM provides information about how the model runs (e.g., resource utilization, request duration, etc.), while the latter is a qualitative indication of the model output [28].

> **Model server** refers to a program that manages access to a machine learning model, by receiving network requests, and running the underlying model. A model server may serve multiple models.

MLOps (E) encapsulates the latter inference stage of the ML life-cycle, as well as part of the training stage. In particular, as new data is gathered, the model needs to be re-trained (but the architecture or algorithm does not need to be changed). The practice to automatically re-train and re-deploy ML models belongs to the practice of MLOps. Workflows represent the means by which the automatic processes can take place.

> **Workflow**: series of steps and processes involved in developing, deploying, and managing machine learning models.

The main purpose of workflows is to automate and integrate processes spanning multiple stages of the ML model life-cycle.

2.4 Hadrian

This research was conducted as part of an internship at Hadrian [15]. Hadrian is a cyber-security company that specializes in real time exposure management. Hadrian's platform leverages an event-driven architecture in Kubernetes to continuously discover assets (web services, open ports, IP addresses, DNS records, etc.) and find risks and vulnerabilities related to those assets. Hadrian leverages machine learning to enhance asset discovery, labelling, and finding vulnerabilities.

In this work, we make Kubernetes deployments in Hadrian's event-driven architecture, which consume and produce events within the platform in a production-like environment. Due to intellectual property law, the deployment specification, the machine learning model code and artifacts, and the raw data are not made public. However, our deployment

methodology and parameters are described in detail in Chapter 4.1. We release the processed, anonymized data, as well as the R and Python code used during data analysis, openly (see Appendix A).

Chapter 3

Reference Architecture for Serverless Machine Learning

In this chapter, we introduce the main contribution of this work, reference architecture of serverless machine learning. We first explain the design methodology we used, followed by an overview of the layers, components, and design choices. We then present an analysis of the novel elements in the architecture. Third, we evaluate the reference architecture by mapping different cloud services and tools to it. Finally, we present one common architectural pattern we have identified in the observed platforms.

3.1 Design Methodology

We start the design from the SPEC-RG FaaS reference architecture [16] (Figure 3.1), and adapt it to fit the serverless ML field in two stages: first, we add two new layers, corresponding to machine learning and eventing, respectively. Second, we modify the existing layers to better represent machine learning applications, by adding or modifying components within each layer.

For the ML layer, we start by comparing the features of the three biggest end-to-end ML solutions in the industry: AWS SageMaker, GCP Vertex AI, and Azure Machine Learning. We identify common features between the platforms, which we split into groups, where each group represents a component in the reference architecture. We start with 6 components, corresponding to each of the 6 stages in the ML model life-cycle training and inference phases (Figure 2.1). For example, the following subset of features is grouped into a component corresponding to the *develop* stage: Integrated Development Environment (IDE), machine learning frameworks, and pretrained models library. Features that we are only able to identify in one platform are disregarded as implementation particularities.

We employ the same methodology for the eventing layer, but using the eventing solutions

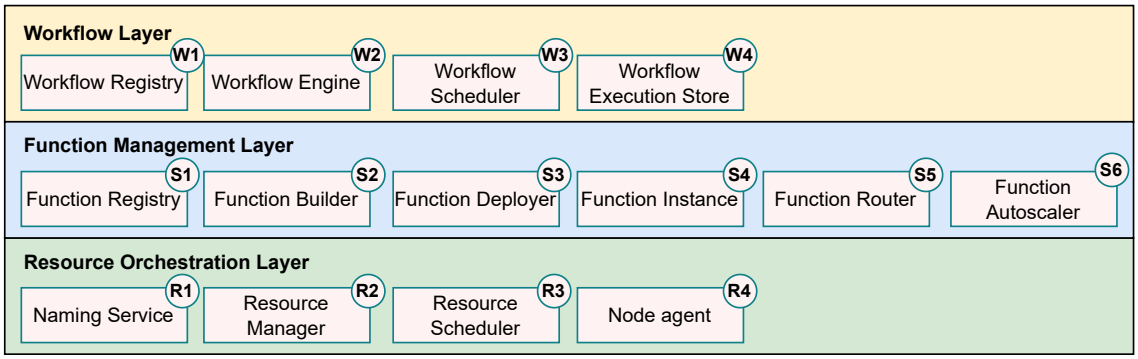


Figure 3.1: SPEC-RG reference architecture for FaaS, introduced by van Eyk et al. [16]. This constitutes the starting point of our design.

from the same cloud providers, namely: AWS EventBridge, GCP Eventarc, Azure Event Grid. In this layer, each feature becomes its own component.

We progressively add platforms to the study, and iterate on the reference architecture if more components have been found, or if we have identified a better grouping of functionality into components. Each iteration, we may choose to divide components if we deem the functionality they fulfil to be too broad. We explain this decision on a component basis.

We then adapt the other three layers (Resource Management, Function Management, and Workflow) for serverless ML. More precisely, we add components within each layer for functionality that we found to be specific or crucial in the operation of serverless machine learning during the study of the platforms.

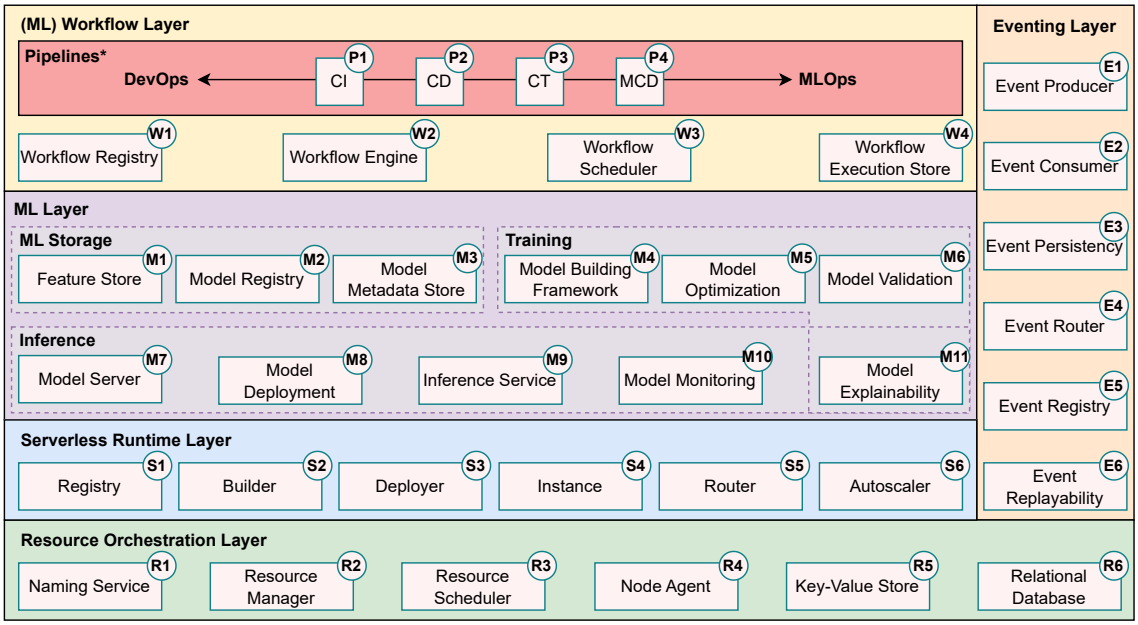
We present an overview of the result in section 3.2, and provide the validation of the reference architecture against the studied platforms in 3.4.

3.2 Reference Architecture Overview

Figure 3.2 presents our reference architecture for serverless machine learning. The model consists of 5 layers, and a sub-layer within the ML Workflow Layer. The vertical positioning of the layers indicates interlayer dependencies: a layer higher up in the diagram makes use of components in the layers underneath. For example, the Serverless Runtime Layer relies on the Resource Orchestration Layer to provision and manage resources. Note that the ML Layer is laid out vertically in two rows. This has the purpose to improve visibility, and does not have the explained connotation above.

The eventing layer is placed orthogonally to the serverless, ML and workflow layers, and on top of the resource layer. The intuition behind this choice is that components in the three orthogonal layers may make use of eventing to communicate with each other.

Below we explain the components and roles that each layer plays, and the design choices



*** CI** = Continuous Integration, **CD** = Continuous Delivery
CT = Continuous Training, **MCD** = Model Continuous Delivery

Figure 3.2: Reference architecture for serverless machine learning systems.

and possible alternatives.

1. **Resource Orchestration Layer**: This layer connects the workloads from the all other layers with physical resources.

   (R1) **Naming Service**: keeps track of the resource names across the whole system.

   (R2) **Resource Manager**: monitors the resource usage of the whole system by communicating with the Node Agents.

   (R3) **Resource Scheduler**: takes actions to bring the current state of the system to the desired state.

   (R4) **Node Agent**: oversees local (node) resource usage and executes instructions received from the Resource Manager.

   (R5) **Key-Value (KV) Store**: serves as the main storage engine for some of the machine learning components in the layers above. In particular, it is a good storage medium for unstructured data, such as model artifacts, or experiment charts/graphs.

   (R6) **Relational Database**: used in conjunction with the KV Store as a means to provide structure to experiments and model deployments.

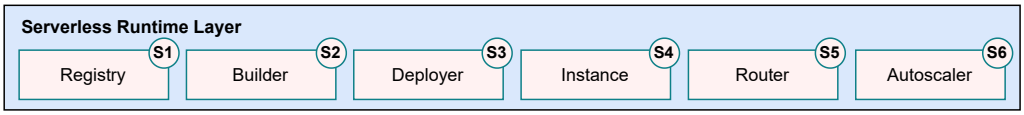


Figure 3.3: Reference architecture: isolated serverless runtime layer.

2. **Serverless Runtime Layer**: This layer corresponds to the "Function Management Layer" in the FaaS reference architecture. Here we remove the function specific components and generalize to a universal serverless layer that does not make assumptions about the underlying technology, be it containers (e.g., Docker), or virtual machines (e.g., Firecracker [29]).

   (S1) **Registry**: stores and manages ML application runtime images, such as containers or VMs.

   (S2) **Builder**: packages the necessary code to run ML models (sometimes including the model artifact) into a deployable image. The build output is stored in the Registry.

   (S3) **Deployer**: (usually) takes as input a set of parameters, and makes a certain runtime image available to be instantiated as an instance (next component).

   (S4) **Instance**: worker running a previously deployed image.

   (S5) **Router**: routes incoming (inference) request to one of the available instances.

   (S6) **Autoscaler**: increases or decreases the number of available instances based on demand and its own configuration.

3. **Machine Learning Layer**: The ML layer sits on top of the serverless runtime layer. Corresponding to the three main stages of ML lifecycle (Data Engineering, Training, Inference, see Chapter 2, Figure 2.1), we identify three categories of ML components: Storage, Training and Inference.

   ML Storage

   (M1) **Feature Store**: repository where data is transformed into features and stored. The feature store bridges the gap between collected data and model input, by providing real-time data transforms (for inference), as well as store and provision training and testing data. The feature store provides a toolkit to help data scientists transform raw data into features (feature engineering). Recently, the feature store has emerged as a central component in machine learning operations [30, 23].

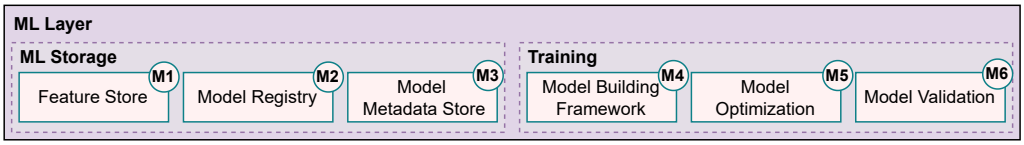


Figure 3.4: Reference architecture: isolated ML layer.

(M2) **Model Registry**: stores model artifacts that can be served. Here, models are versioned and managed, either automatically (in pipelines) or manually.

(M3) **Model Metadata Store**: used during the training phase to store experiment results, logs and model hyperparameters. It helps data scientists guide model research and development. The model registry and the model metadata store are closely related, and can be often perceived as the same component (e.g., in AWS SageMaker). In this work, we examine these two separately. The intuition behind this choice is that the model registry is tailored for use during inference (it stores and versions models). On the other hand, the model metadata store is used during development and optimisation. It may contain experiments, graphs, logs, etc.

Training

(M4) **Model Building Framework**: set of tools used during the development phase to build and test models.

(M5) **Model Optimisation**: the process that optimises an ML model using a dataset. The Model Optimisation component handles the distributed training logic, as well as hyperparameter optimisation, and presents the data scientists and developers with a dashboard displaying metrics during the optimisation process. It may be triggered either manually, or automatically through a pipeline (see (P3)).

(M6) **Model Validation**: the job/component that follows model optimisation, and tests the performance and validity of the model. Some common functions include checking if the model is under- or over-fit, running the model on a test set to check accuracy, analysing the output distribution of the model, checking for bias, etc.

Inference

(M7) **Model Server**: instance that makes ML models available for predictions on new data, most often through a REST or gRPC API. The model server trans-

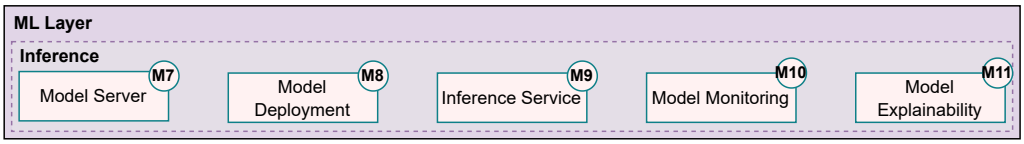


Figure 3.5: Reference architecture: isolated ML layer.

forms the input it receives (for example by using the Feature Store), runs the ML model, and performs a transformation (interpretation) on the model output, then returns the result.

- (M8) **Model Deployment**: the mechanism by which a trained model becomes available to be served in a production environment by the Model Server (see **M8**). The deployment of a model may be triggered manually, or by a pipeline ((P4)).
- (M9) **Inference Service**: abstraction of the model server that may make requests to multiple model servers, to combine the results of multiple models. The inference service also enables traffic splitting between multiple models for A/B testing.
- (M10) **Model Monitoring**: records model predictions, analyses them (e.g., outlier detection, drift detection [31]), and potentially includes the newly seen data in the training dataset, or flags the prediction for human review. The model monitor may also include a dashboard where different application and model level metrics are displayed. In some architectures (e.g., AWS SageMaker [13], see explanation in section 3.4, Model Monitoring is split into two subcomponents: prediction/metrics recording and analysis.
- (M11) **Model Explainability**: provides insights into the predictions of the model. More concretely, it can identify the features that have the highest weight in a certain prediction, reverse engineer input data to trick the model, and detect model bias. The model explainability component may be used both during training and during inference. During inference, the model monitor ((M10)) may make use of the model explainability component to decide if the predictions of the model are accurate. This process is called drift detection and is described in detail in section 3.5

4. **ML Workflow Layer**: Above the ML layer, there is the ML workflow layer. The workflow layer pieces components of the ML layer together to automate and schedule tasks (e.g., model training).

   - (W1) **Workflow Registry**: stores the workflow configurations and components.
   - (W2) **Workflow Engine**: oversees the execution of the different components in the workflow, restarts failed jobs, and cancels long-running jobs.

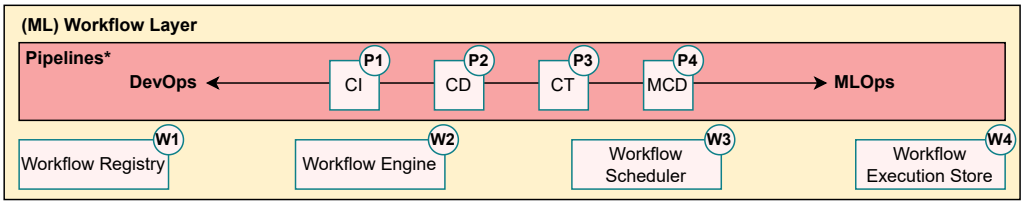


Figure 3.6: Reference architecture: isolated ML workflow layer.

(W3) **Workflow Scheduler**: decides which jobs get executed and when.

(W4) **Workflow Execution Store**: stores the data of job runs: status, logs, start time and duration, etc.

5. **Pipelines**

   Within the workflow layer, we identify the pipelines sub-layer, a central component to the practice of MLOps [32, 33]. We define pipelines as specialized ML(Ops) workflows. We identify here four pipeline types, two of which are shared with DevOps.

(P1), (P2) **Continuous Integration** and **Continuous Delivery** are the DevOps practices of automatically building (CI) and deploying (CD) the latest code version.

(P3) **Continuous Training**: pipeline that re-trains and subsequently re-evaluates an ML model when certain events are produced, on a schedule, or on a manual trigger.

(P4) **Model Continuous Delivery**: ensures automatic and continuous deployment of the best performing ML trained ML model to stating and production environments.

6. **Eventing Layer**: The eventing layer is situated on top of the resource orchestration layer and along-side serverless runtime, ML and workflow layers, because components in all these layers can make use of eventing. Here, we identify four components:

(E1) **Event Producer**: any program (microservice) that creates events.

(E2) **Event Consumer**: any program that receives events originating from one or more producers. Consumers and producers are not mutually exclusive, i.e., a module may be both consumer and producer.

(E3) **Event Persistence**: ensures event are stored on persistent storage (e.g., SSD). The component acts as a backup mechanism to help recover from system failures, and to record the state of the system in time.

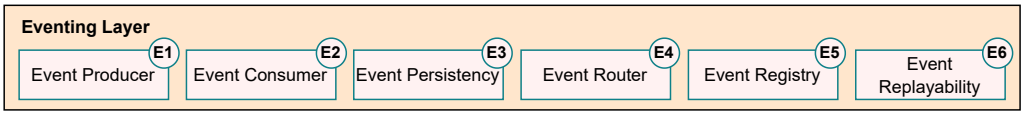


Figure 3.7: Reference architecture: isolated eventing layer.

- (E4) **Event Router**: responsible for routing events from producers to the consumers. The routing model, logic, and mechanisms may differ by platform (e.g., pub-sub, event bus). We do not make assumptions on the underlying structure of event routing.
- (E5) **Event Registry**: acts as a catalogue of the supported event types.
- (E6) **Event Replayability**: component that can replay past events in the system, e.g., to restore the state from a failure.

3.3 Novelty Analysis

In this section, we discuss the novelty of the components we introduce in 3.2. We discuss each layer (from bottom to top in Figure 3.2):

1. **Resource Orchestration Layer**: We derive this layer from the SPEC-RG reference architecture for FaaS [16]. Components **R1-4** remain unchanged. In addition, we introduce two novel components in this layer: the Key-Value Store ((R5)), and the Relational Database ((R6)), specific to serverless ML workloads.
2. **Serverless Runtime Layer**: In this layer, we reuse the Function Management Layer from the FaaS reference architecture, and adapt the components to the generic serverless use-case.
3. **ML Layer**: This is a novel addition in this work, and a central component of the reference architecture of serverless ML.
4. **(ML) Workflow Layer**. We derive this layer from the Workflow Layer in the FaaS RA. Components **W1-4** remain unchanged. Within the workflow layer and on top of **W1-4** we add the Pipelines sub-layer.
5. **Pipelines**. The pipelines in this layer refer to the common use case ML workflows, and are a novel element of this work.
6. **Eventing Layer**. This is a novel component we introduce to capture the emerging event-driven paradigm.

3.4 Mapping Real World Platforms to the Reference Architecture

In this section, we map platforms and services to the reference architecture. Mapping refers to identifying which components from the reference architecture are present in each of the selected platforms.

Methodology. We consider a total of 19 platforms, 12 of which are open-source. We select the platforms as follows: first, we include the end-to-end ML solutions of AWS, GCP, and Azure, as well as the respective eventing solutions from each cloud provider. Next, we add 7 open-source ML platforms that together fulfil the whole spectrum of ML functionality. We add Kubernetes and all three Knative components (i.e., Serving, Functions, and Eventing), because they represent a popular choice of open-source platforms for the resource and serverless layers. Finally, we add one popular KV store and one relational database platforms: AWS S3 [34], and PostgreSQL [35].

For each platform we consult documentation, diagrams, papers, presentations, and code (where available) to determine which components from the reference architecture are present. We determine a component is present if we identify a component of the examined software that matches the functions of the respective component introduced in the reference architecture. Specifically, for the pipelines sub-layer, we determine a certain pipeline component (**P1-4**) maps to the reference architecture if the given platform offers a workflow template that matches the functions of the respective pipeline.

We summarize the results in Table 3.1, and expand on the ML, Workflow and Pipeline layers in Table 3.2 for those platforms whose main functions occur in these layers.

For each platform and component, we either record “Yes" if we are certain the component is present in the platform, “No" if we are certain the component is not present, “Delegated" if the functionality is delegated to another platform, and “Unknown" if not enough information could be found to perform the mapping.

Kubernetes is an open-source container resource orchestrator.

AWS S3 is a popular Key-Value store choice. It integrates with AWS EventBridge (see below) to provide eventing functionality.

PostgresSQL is one of the most widely adopted open-source relational databases [36].

Knative Serving is a Kubernetes serverless computing add-on. It does not provide its own container builder or registry, but otherwise maps to the entire Serverless layer.

Knative Functions is a Knative component that aims to bridge the gap between Knative Serving and FaaS. It provides a set of tools to manage code as functions (packaged as

Table 3.1: Platform mapping on the Resource, Serverless, and Eventing layers. Colours indicate the layers which fit the functions of each platform best.

● Yes ○ No ◑ Delegated – Unknown ↓ See Table 3.2

Platform	Open source?	Resource (R) 1	2	3	4	5	6	Serverless (S) 1	2	3	4	5	6	Eventing (E) 1	2	3	4	5	6	M *	W *	P *
Kubernetes	●	●	●	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
AWS S3	○	–	–	–	–	●	–	–	–	–	–	–	–	●	○	◑	◑	◑	◑	○	○	○
PostgreSQL	●	◑	◑	◑	◑	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Knative Serving	●	◑	◑	◑	◑	○	○	◑	◑	●	●	●	●	○	○	○	○	○	○	○	○	○
Knative Functions	●	◑	◑	◑	◑	○	○	◑	●	●	◑	◑	◑	○	○	○	○	○	○	○	○	○
Knative Eventing	●	◑	◑	◑	◑	○	○	○	○	○	○	○	○	●	●	●	●	●	○	○	○	○
AWS EventBridge	○	–	–	–	–	–	–	–	–	–	–	–	–	●	●	●	●	●	●	○	○	○
GCP Eventarc	○	–	–	–	–	–	–	–	–	–	–	–	–	●	●	●	●	–	●	○	○	○
Azure Event Grid	○	–	–	–	–	–	–	–	–	–	–	–	–	●	●	○	●	○	○	○	○	○
MLflow	●	◑	◑	◑	◑	◑	◑	○	○	○	○	○	○	○	○	○	○	○	○	↓	○	○
Feast	●	◑	◑	◑	◑	◑	◑	○	○	○	○	○	○	○	○	○	○	○	○	↓	○	○
Evidently	●	◑	◑	◑	◑	○	○	○	○	○	○	○	○	○	○	○	○	○	○	↓	○	○
SeldonIO Alibi	●	◑	◑	◑	◑	○	○	○	○	○	○	○	○	○	○	○	○	○	○	↓	○	○
KServe	●	◑	◑	◑	◑	○	○	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	○	↓	○	○
Kubeflow	●	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	◑	○	↓	●	○
Azure ML	○	–	–	–	–	◑	–	–	–	–	–	–	–	●	○	○	◑	○	○	↓	–	○
AWS SageMaker	○	–	–	–	–	◑	–	–	–	–	–	–	–	●	○	◑	◑	◑	◑	↓	–	↓
GCP Vertex AI	○	–	–	–	–	◑	–	–	–	–	–	–	–	●	○	◑	◑	–	◑	↓	–	↓
Apache Airflow	●	◑	◑	◑	◑	◑	◑	○	○	●	●	○	◑	○	○	○	○	○	○	○	●	○

containers), and delegates the most of the runtime (**S4-6**) functionality to Knative Serving. In addition to Knative Serving, it provides a feature to build containers from the function code.

Knative Eventing fully maps to the proposed eventing layers, except the replayability component (**S6**).

AWS EventBridge, GCP Eventarc, Azure Event Grid are eventing solutions offered by the three biggest cloud service providers. Each of these provides eventing functionality for the end-to-end ML solution offered by the respective cloud service provider.

MLflow is an open-source solution for ML model management. It offers a model registry, a model metadata store, and a Python library for logging, experimenting, and model packaging.

Table 3.2: Platform mapping on the ML, Workflow, and Pipelines layers. Colours indicate the layers which fit the functions of each platform best.

● Yes ○ No ◑ Delegated – Unknown ↑ See Table 3.1

Platform	Open source?	R	S	E	ML (M)											Workflow (W)				Pipelines (P)			
		*	*	*	1	2	3	4	5	6	7	8	9	10	11	1	2	3	4	1	2	3	4
MLflow	●	◑	○	○	○	●	●	●	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○
Feast	●	◑	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
Evidently	●	↑	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○
SeldonIO Alibi	●	↑	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○
KServe	●	↑	◑	↑	○	○	○	○	○	○	●	●	●	○	◑	○	○	○	○	○	○	○	○
Kubeflow	●	◑	◑	↑	◑	○	●	●	●	○	◑	◑	◑	○	◑	●	●	●	●	○	○	○	○
Azure ML	○	↑	–	↑	○	●	●	●	●	●	●	●	●	◑	●	–	–	–	–	○	○	○	○
AWS SageMaker	○	↑	–	↑	●	●	●	●	●	●	●	●	●	●	●	–	–	–	–	◑	◑	●	●
GCP Vertex AI	○	↑	–	↑	●	●	●	●	●	●	●	●	●	●	●	–	–	–	–	○	○	●	○
Apache Airflow	●	◑	↑	○	○	○	○	○	○	○	○	○	○	○	○	●	●	●	●	○	○	○	○

Feast is an open-source standalone feature store that integrates with Kubeflow. It relies on external KV stores (e.g., AWS S3) to store data structures.

Evidently is the only open-source model monitoring solution that we are aware of.

SeldonIO Alibi is a model explainability application that integrates with KServe.

KServe, originally part of the Kubeflow project, is a Kubernetes machine learning inference plugin. It provides high level interfaces for ML model deployment, and delegates the Serverless Runtime Layer to Knative Serving.

Kubeflow is an open-source machine learning life-cycle management solution for Kubernetes. It offers limited model development tools (in the form of Python notebooks), and delegates the inference functionality to KServe (see below). Kubeflow provides full workflow functionality through Kubeflow Pipelines. Despite being a framework specialized in ML, it leaves the responsibility of creating CI/CD or CT/CMD (Pipelines layer) components to the user.

Azure Machine Learning is the ML as a service platform offering from Azure. The platform maps to the entire ML layer, with two exceptions: the feature store (it offers no such solution) and the model monitor (only provides application level monitoring, and the dashboard is delegated to Azure Monitor).

AWS SageMaker is the end-to-end machine learning solution from AWS. SageMaker

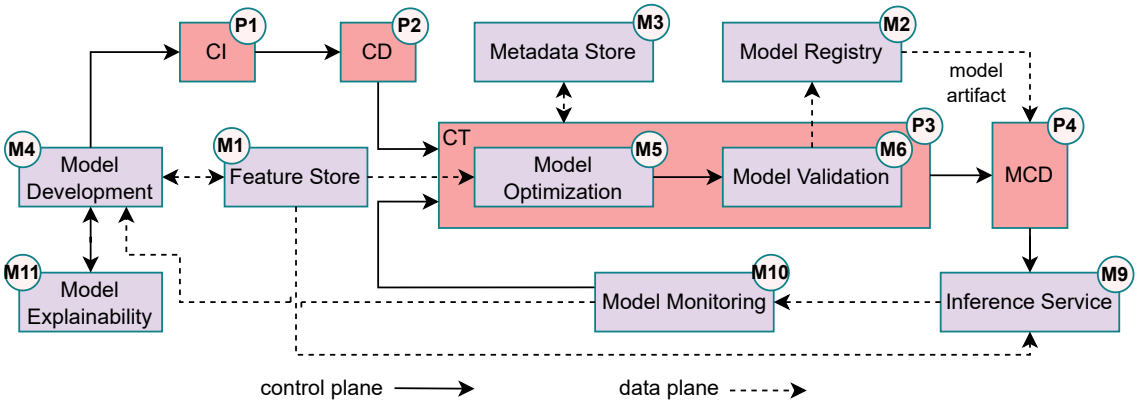


Figure 3.8: Google Cloud Platform MLOps architecture [37].

maps to the entire ML layer, by providing components that fulfil all ML functions: SageMaker Feature Store (**M1**), Model registry (**M2**, **M3**), Studio & Canvas (**M4**), Model training (**M5**), Clarify (**M6**, **M11**), Model deployment (**M7**), Inference toolkit (**M8**), Endpoint (**M9**), Model monitor (**M10**, dashboard feature delegated to AWS CloudWatch).

SageMaker Pipelines provides full functionality for ML workflows. The closed source nature of AWS limits our understanding of the mapping in the Workflow layer. SageMaker delegates code CI/CD to AWS CodeCommit and CodeBuild, respectively. AWS does not provide sufficient information to map the Resource and Serverless layers.

GCP Vertex AI Google Cloud Platform's Vertex AI maps to the whole of the Machine Learning Layer. Vertex AI provides workflow functionality through Vertex AI Pipelines. Despite being compatible with the Kubeflow pipeline format, we are unable to find evidence that the two platforms also share their runtime. Vertex AI Pipelines offers CT pipeline templates that can be further customized.

Apache Airflow is a general purpose workflow orchestration tool. It provides some serverless functionality, and maps to all components in the workflow layer. Although not specifically designed for machine learning, we include Airflow in the list of platforms as a popular orchestrator choice.

The mapping in Tables 3.2 highlights a major issue with open-source ML software: currently there is no end-to-end solution for the ML model lifecycle. Developers have to deploy and integrate to approach the functionality offered by the platforms of the cloud service providers. Each of these open-source tools is in different stages of maturity, and may require additional infrastructure deployments.

To further test the reference architecture, we map Google's recommended process for automated MLOps to components in the reference architecture. The result is presented in

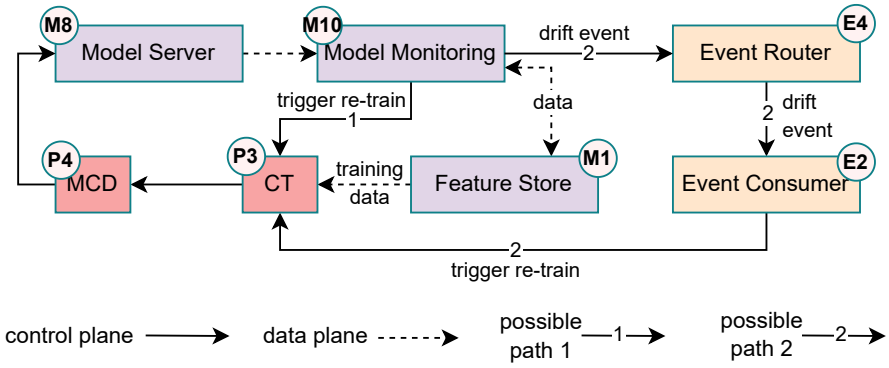


Figure 3.9: Model drift detection operational pattern.

Figure 3.8.

3.5 Operational Pattern: Automatic Drift Detection

While documenting existing practices around serverless ML, we have identified an emerging MLOps pattern: automatic drift detection. Automatic drift detection emerged recently as a result of the maturity of MLOps, and encompasses components in the ML layer, as well as the ML Workflow layer. In this section we explain the concept in more detail.

Over time, the input data distribution (data drift), or the relation between input and output (concept drift) may change. Over time, this can result in a slow decay of ML models' accuracy. The model monitor (M10) is the component responsible for intercepting these trends and taking appropriate action. Drift detection ensures that model predictions remain accurate over time.

Figure 3.9 depicts a high-level overview of how drift detection works in AWS SageMaker, GCP Vertex AI, and Azure Machine Learning. The model server (M8) sends every input it receives, along with the prediction, to the model monitor (M10). The monitor analyses the data it receives from the model in a longer time frame against the training data it obtains from the feature store (M1). The model monitor may also add the newly observed sample to the dataset.

Once drift is detected by the model monitor, two patterns emerge. First, the model monitor may trigger the continuous training pipeline (P3) directly. Second, the model monitor may produce an event that drift was detected. In the latter case, the event is received by the router (E4) and sent to a consumer (E2). In the platforms in this study, the consumer is usually a function-as-a-service instance. Finally, the consumer triggers the continuous training pipeline. The advantage of using events for this purpose is the ability to have multiple consumers (e.g., a consumer may send an email alert to an ML engineer). We omit the sub-steps within the CT pipeline (optimizing, validating, etc.) for simplicity.

The model is then re-trained with fresh data, after which the MCD pipeline (Ⓟ4) deploys the new version of the model.

Chapter 4

Data Analysis on Real World Deployments

In this chapter we conduct a data-driven analysis on three real world serverless Machine Learning model deployments, and run experiments to further understand the serverless ML workload.

The chapter is split in two main sections. In section 4.1 we monitor three machine learning models already deployed in Hadrian's Kubernetes cluster, and analyse the collected data. In section 4.2 we conduct experiments by deploying additional ML model workloads.

Note: throughout this chapter, we define **inference request duration** to be the time it takes the model server to process a request. We define **inference request latency** to be the total time it takes one inference request to complete, as recorded on the client that initiated the request. In other words, the inference request latency includes the inference request duration, as well as network and autoscaling latency.

4.1 Data Analysis

In this section we monitor three machine learning models, and analyse the collected data. **Important note**: the three models studied are already deployed in a production environment in Hadrian's Kubernetes cluster. The cluster is not equipped with GPU accelerators, and training of the models is done separately, outside the cluster. Hence, this section focuses on the inference phase of the ML model life-cycle. For data collection, we do not make any modifications to the deployments. Section 4.2 further explores the performance characteristics of the serverless ML inference workloads through a series of experiments.

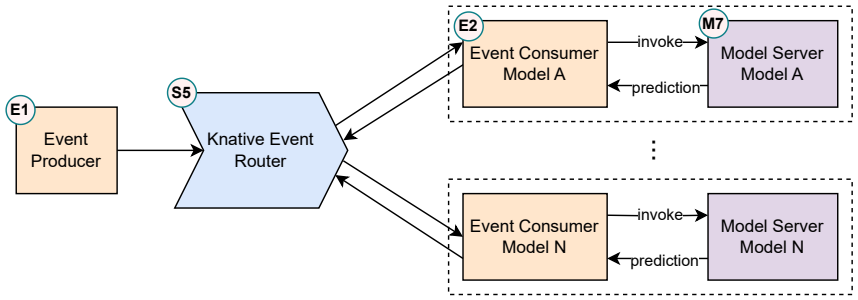


Figure 4.1: Inference setup overview.

4.1.1 Setup

We monitor three machine learning model deployments in Hadrian's [15] Kubernetes cluster. The cluster amounts to a total of around 1 TB of memory and 200 CPU cores, and sits at less than 50% utilization at all times during our experimentation.

Note that we do not utilize machine learning accelerators (GPUs, TPUs) for these deployments. This yields a setup similar to AWS SageMaker Serverless Inference [14].

The models servers (M10) are packaged as Docker containers. Each model runs as a Knative service (M9) with minimum scale 0, i.e., when there are no more incoming requests, the number of pods gets decreased to 0 to save compute resources.

The system uses Knative Eventing as an eventing layer. Whenever an event is produced, Knative routes it to the appropriate event consumer. In our setup, the event consumers are simple programs that perform pre-processing on the event payload and perform a prediction by invoking a model server with the request. The model server runs the model and returns a prediction. A new event may be produced using the prediction. A high-level overview of the process is depicted in Figure 4.1.

Before each model server is started, its respective model artifact is downloaded locally from the model registry (M2). All model servers are set up such that they may only consume one request at a time. For each model, we set the maximum CPU and memory usage, as well as the maximum number of pods of a model that may be running concurrently; all models are scale-to-zero. The configuration is summarized in Table 4.1.

Model A is a Convolutional Neural Network [38] (CNN) that performs classification on input images. The model is implemented in PyTorch.

Model B is a K-Nearest Neighbors (KNN) search algorithm, where the dataset is generated using Word2Vec [39] from a text corpus. The model receives as input a sequence of words that are converted into vectors through a database lookup. Each word is associated

[1]Convolutional Neural Network
[2]K-Nearest Neighbors
[3]Multi Layer Perceptron

Table 4.1: Configuration of deployed ML models. Note: CPU usage represents the number of CPU cores allocated for the given pod, per second (e.g., a value of 0.25 means that the pod is scheduled to compute for 0.25 seconds on one CPU core, every second). *The resource amounts represent the requested and the limit amounts, respectively (see section 2.2).

Model	Architecture	Parameters [Millions]	CPUs*	Memory* [GB]	Max. scale	Container size [GB]
A	CNN[1]	8	0.25 – 1.0	2 – 10	1	2.12
B	KNN[2]	–	0.25 – 0.5	1 – 2	1	0.42
C	MLP[3]	1.3	0.25 – 1.0	2 – 10	1	0.67

with a unique vector. Next, we compute the average vector by combining all the individual word vectors. This averaged vector is then utilized to search for the k nearest neighbours (by cosine similarity) by performing another database lookup. Our dataset consists of around 800,000 word embeddings, and each KNN search performs the cosine similarity function against all of these to determine the maximum. The database is deployed in a Virtual Machine with a fixed amount of resources, hosted by a third party, and is only used by this model. The model server is implemented in Python, and the database is an Elasticsearch [40] cluster.

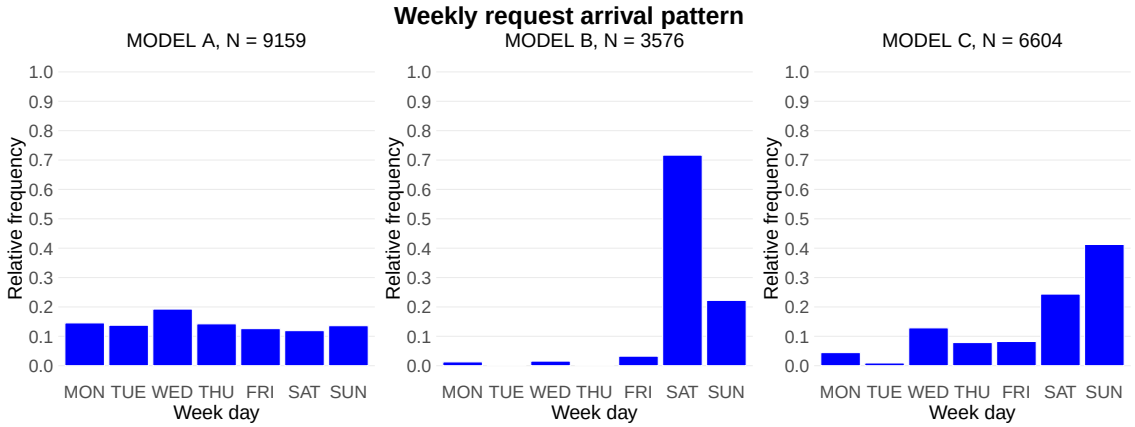
Model C is a simple Multi Layer Perceptron (MLP [41]), with a tokenizer that preprocesses text inputs. We describe in detail the architecture and performance of this model in a separate publication [42]. The model is implemented in PyTorch.

For each model, we collect data for the model server and its corresponding event consumer module. We collect the following metrics: CPU utilization, memory usage, autoscaler level metrics (number of pods, number of concurrent requests), time of request arrival, and request duration.

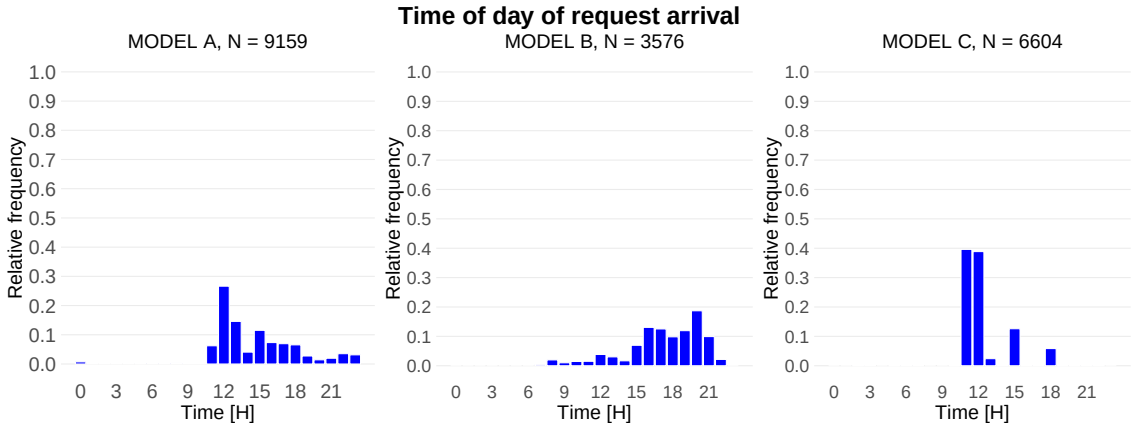
4.1.2 Analysis

First, we analyse the request patterns for each model. Figure 4.2 displays the relative frequency of each weekday (4.2a) and time of day (4.2b) for each model. We notice that Model A has a fairly uniform weekly request distribution, while B and C predominantly receive requests during the weekends (92% and 63% respectively). Within each day, Model A receives a large number of requests around noon. Similarly, Model C receives the majority of request (80%) at 11 or 12 hours.

Next, we look at the inference request duration. The results are presented in Figure 4.3. We notice that models A and C exhibit very high outliers compared to their median. This is especially odd, since the plotted data only represents the amount of time it takes to run



(a) Day of the week of request arrival, per model.



(b) Time of day of request arrival, per model.

Figure 4.2: Weekly and daily request arrival patterns, per model. The number of samples is presented on top of each plot.

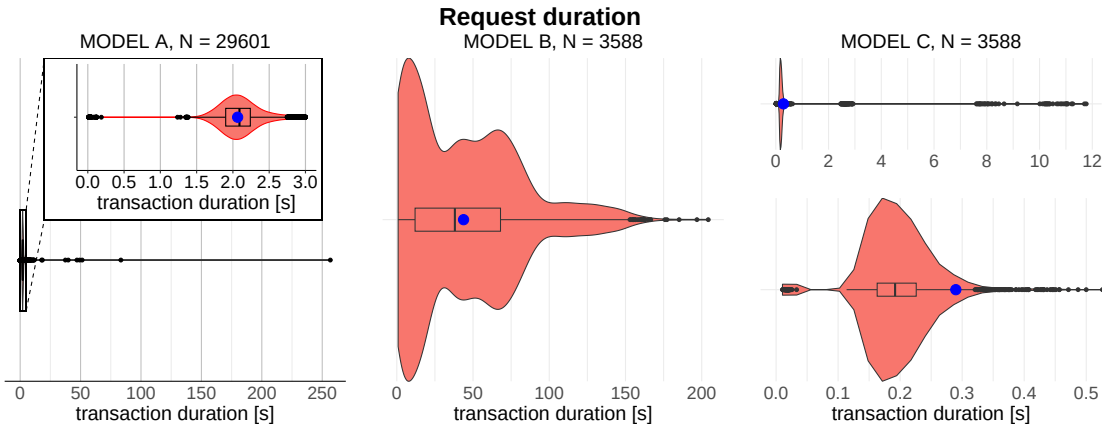


Figure 4.3: Request duration distribution by model. The mean is marked with a blue dot. The number of samples is presented on top of each plot. **Important note**: the request duration is considered the time it takes for the model server to run the model on the input, and does not include any time the request might spend in the network, or in Knative queueing mechanisms.

a model on a single sample. We conjecture that the high performance variability is due to Kubernetes CPU throttling the pods when running on certain (busier) nodes. From here, we derive our first **E**xperiment **Q**uestion:

EQ1. Does Kubernetes cause high performance variability in serverless ML workloads?

Model B does not exhibit the same high outlier behaviour as A and C, but rather we notice a multimodal distribution. The database used for this model is deployed outside the cluster with a fixed amount of available resources. In practice, this should translate to consistent request times. The observed distribution is given by the fact that the model server C does not perform the computation itself, rather it makes two queries to a database. Each database search is linear in time complexity with the input size, which can vary between 1 and 100 words.

Figure 4.4 displays the distribution of memory and CPU usage in the three models. We first note that the memory usage has little variability for all three deployments. Furthermore, we see that the models consume around a fifth of the allocated memory.

We observe that the CPU usage of model A has very high variability, and that its outliers are approaching the set limit of 1 CPU. Model B's CPU variability is very low, due to its computation being essentially offloaded to the database. Model C uses a lower amount of CPU, due to its smaller size and simpler architecture. From these observations, we derive Experiment Question 2:

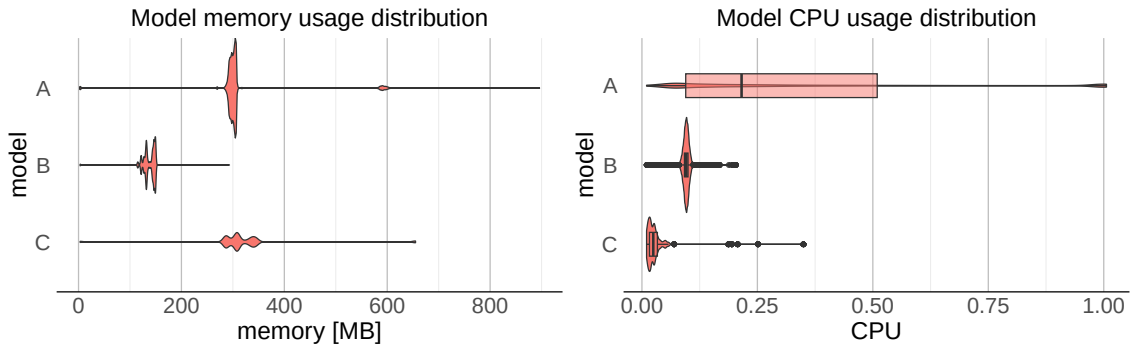


Figure 4.4: Comparison of resource usage distribution between the three models. The distributions only contain data points from non-idle pods (CPU > 0.01).

EQ2. Does vertical scaling of the model server decrease inference request duration and variability?

In this section we only analyse the inference request duration, and do not take into account the overhead of the Knative mechanism, network latency, or container initialization delay. Furthermore, we only examine deployments that scale between 0 and 1 pods. Hence, we propose the third experiment question:

EQ3. Does horizontal scaling of the model server decrease inference request latency?

4.2 Experiments

In this section we answer the three Experiment Questions proposed in 4.1, by designing and running three experiments, each corresponding to one of the questions:

EQ1 Does Kubernetes cause high performance variability in serverless ML workloads?

EQ2 Does vertical scaling of the model server decrease inference request duration and variability?

EQ3 Does horizontal scaling of the model server decrease inference request latency?

By running the experiments we derive three **M**ain **F**indings, which we summarise below:

MF1 Constraining serverless scheduling of ML inference workloads to dedicated nodes reduces the 99th percentile inference request duration by 99%.

MF2 Vertical scaling greatly decreases the inference duration, and can offer more than 50% percent improvement in the relative performance per core.

Table 4.2: Experiment configuration overview. Node refers to whether we deploy the model on a dedicated node, or allow K8s to schedule the pods on all nodes in the cluster.

Section	Focus	Node	CPU	Max. Scale
4.2.2	Kubernetes scheduling	any/dedicated	1	1
4.2.3	Vertical scaling	dedicated	0.5/1/2/4/8	1
4.2.4	Horizontal scaling	dedicated	1	1/2/4/8

MF3 Horizontal scaling increases the total inference request latency by up to 190%, and should not constitute the preferred scaling method, due to the increased computational cost.

4.2.1 Experimental Setup

Model In all three experiments, we only deploy and observe model A. We believe that the performance of model A is representative of the average ML model, by size and computational load. Furthermore, Model A is the only one of the three models deployed for analysis that is able to utilize more than 1 CPU core for computation, as shown in the previous section.

Load In these experiments, we bypass the eventing system, and create a script to send requests directly to the model server. The script mimics the request arrival patterns that we noticed in model A. More precisely, we record the request arrival times of model A in one day, and replay each registered event burst to the model server. We consider two consecutive requests to be part of the same burst if there is less than one minute in-between them, at the time that they were observed. Between each request burst, there is a 3-minute break.

There are three main reasons for why we chose to bypass the eventing system. First, it allows us to decrease the time it takes to run an experiment. If the time gap between two consecutive requests is greater than 2 minutes, Knative scales the model server down to zero pods. Hence, in our setup, we wait a maximum of three minutes in-between request bursts. This allows us to replay a day worth of events in under one hour. Second, it allows us to easily (without interfering with the already existing deployments in Hadrian's system) record the total time it takes for a request to complete (including pod starting time, network latency, etc.), which is an important step towards answering **EQ3**. Finally, this experimental setup allows us to measure the different configurations in similar conditions (same number of requests in the same period of time), which should result in more consistent results.

We implement the script in Python, and release it for open use (see A).

Model Input The implemented script may send one of three sample images, at random, as input for the model. All images have the same resolution (224 × 224 pixels).

Experiment Parameters To answer the three Experiment Questions, we vary one parameter at a time (Node, CPU, and Scale), and record the resulting changes in performance. The experiment variables are summarized in Table 4.2.

The **Node** parameter refers to whether we allow the scheduling of the pods on a dedicated node (dedicated), or on any cluster node (any). **CPU** represents the limit set for CPU usage (e.g., 2 CPU means the pod may use a maximum of 2 CPU cores every second). Note that we set the request to 0.25 CPU in all experiments. **Scale** refers to the maximum allowed Knative scale. Our experimental setup yields a total of 8 deployment configurations (note that all three experiments share the configuration with CPU 1 and scale 1).

Cluster Setup We deploy the model configurations in Hadrian's cluster. In addition to the cluster setup in Section 4.1, we add one more node with 8 CPU cores, and 32 GB memory, dedicated to this experiment, to test the performance impact of running the models alongside other applications.

Note that each configuration is tested separately, to make sure that there are enough resources left on the dedicated node to schedule all pods.

4.2.2 Experiment 1: Kubernetes Scheduling Performance

In this experiment we test the performance and performance variability of the deployed models, when the models run on a dedicated node, compared to when K8s may schedule them on any node. We hypothesise that, when running on the dedicated node, the performance variability and the inference request duration (time spent to run the model) decrease due to K8s allocating resources more consistently to the model server. We also expect that scheduling the pod on a dedicated node should eliminate the high outliers observed in 4.1.

Figure 4.5 displays the comparison between the two deployed configurations. The result for the model running on any node is consistent with the high outlier pattern noticed in the previous section. In line with our expectations, we note that the model server running on the dedicated node has significantly reduced performance variability. More precisely, we see an inference duration decrease of 98% for the 95th percentile, and 98.6% for the 99th percentile data points. In the 90% percentile inference times we see a decrease of around 20% in the dedicated node deployment. The distribution of the inliers remains

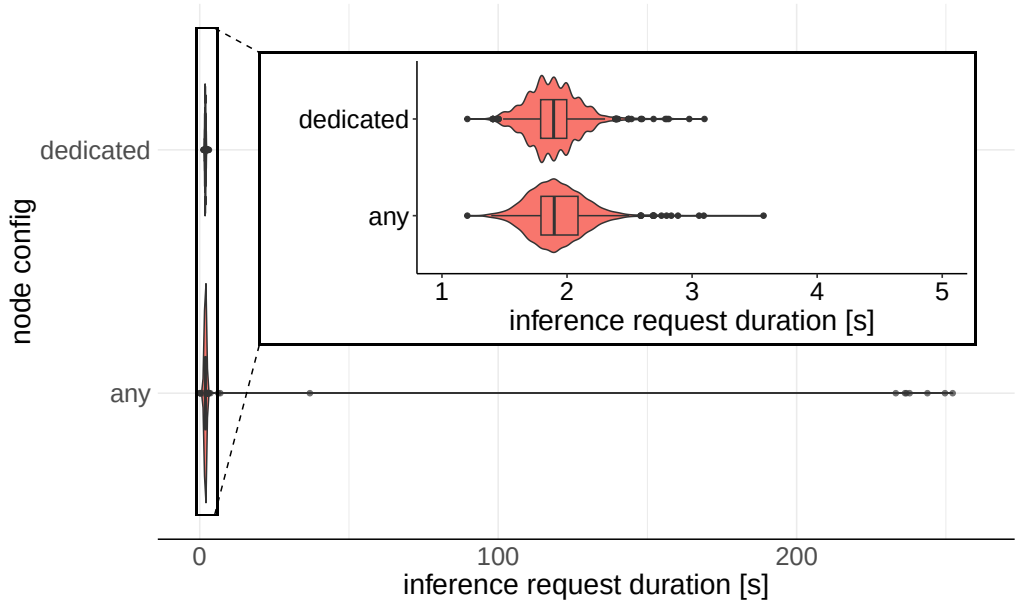


Figure 4.5: Inference request duration comparison between two configurations: ML model scheduled on a dedicated node vs model scheduled on any node in the cluster. The inlet plot presents a zoomed-in view of the box- and violin- plots, for improved visibility.

approximately the same for the two deployments, with a negligible difference in means which we attribute to system noise. From here we draw our first **M**ain **F**inding:

> **MF1**. Constraining serverless scheduling of ML inference workloads to dedicated nodes reduces the 99th percentile inference request duration by 99%.

This finding is especially surprising since all nodes in the K8s cluster are sitting at less than 50% utilisation at all times, and most of them are equipped with 32-core CPUs, while the dedicated node has a mere 8 cores. We would therefore expect the performance (variability) of the two deployments to be similar, since both model servers are configured to only consume 1CPU core at a time. Furthermore, we observe the same pattern of high inference duration in the outliers on all 9 (generic) nodes.

One possible explanation for this finding is that the cluster is not, in fact, sitting at less than 50% utilisation at all times, but rather certain nodes experience spontaneous bursts. In this research we are limited to sampling system metrics every 30 seconds (due to cluster setup), thus small bursts in CPU utilisation may not show during analysis.

4.2.3 Experiment 2: Vertical Scaling

This experiment is aimed at testing the vertical scalability (i.e., the performance when allocating more resources to each pod) of the deployment. We test 5 configurations, and

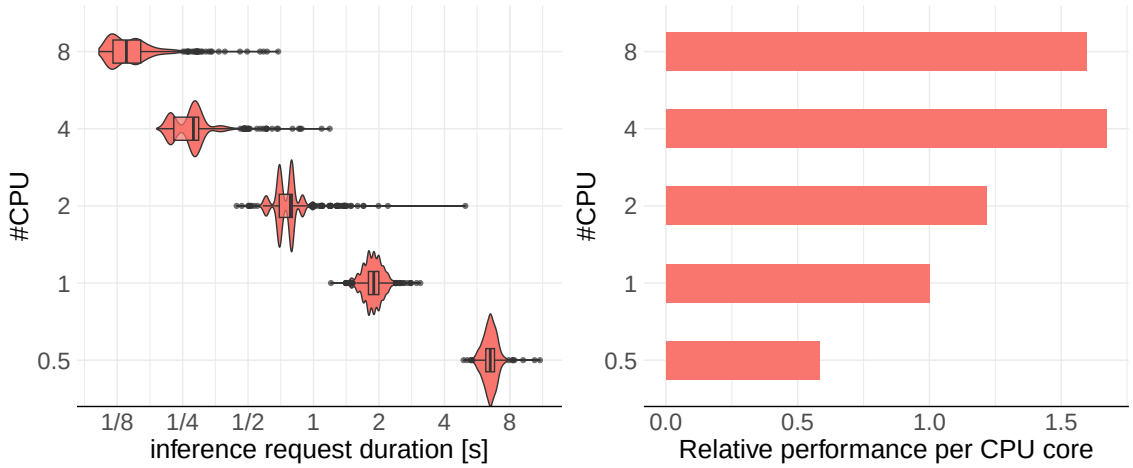


Figure 4.6: Inference request duration comparison with increasingly more CPU time allocated. On the left plot, we show the inference request duration distribution. **Log scale used to improve visibility**. On the right, we show the performance per core, relative to that of the deployment with 1 CPU.

in each we double the maximum CPU that the pod may use (0.5, 1, 2, 4, and 8). The result of this experiment is displayed in Figure 4.6

In the left plot we show the inference request duration across the 4 configurations. We see that allocating more CPU to the deployments decreases the inference time significantly. On the right side of Figure 4.6 we plot the relative performance per core. We calculate this by multiplying the mean inference duration of each configuration by the number of CPU cores, then dividing the mean inference duration of the 1 CPU deployment by all the other values. The plot shows that the best *value* configuration is the one with 4 cores: quadrupling the number of cores yields a 6 times increase in performance. From here we derive main finding 2:

MF2. Vertical scaling greatly decreases the inference duration, and can offer more than 50% percent improvement in the relative performance per core.

In theory, we would expect to see diminishing marginal returns (and hence a decrease in relative performance per core) as we add more CPU cores, due to the overhead that comes with parallel computing. We believe that the reason for this finding is related to the pod misreading the amount of CPU cores it has available. More precisely, we have noticed that the model server reports that it has available all CPU cores of the node it is running on (e.g., the deployment with 1 CPU core running on the dedicated 8 core node reads that it has 8 cores available). ML libraries (such as PyTorch) can take advantage of multicore computing to accelerate workloads.

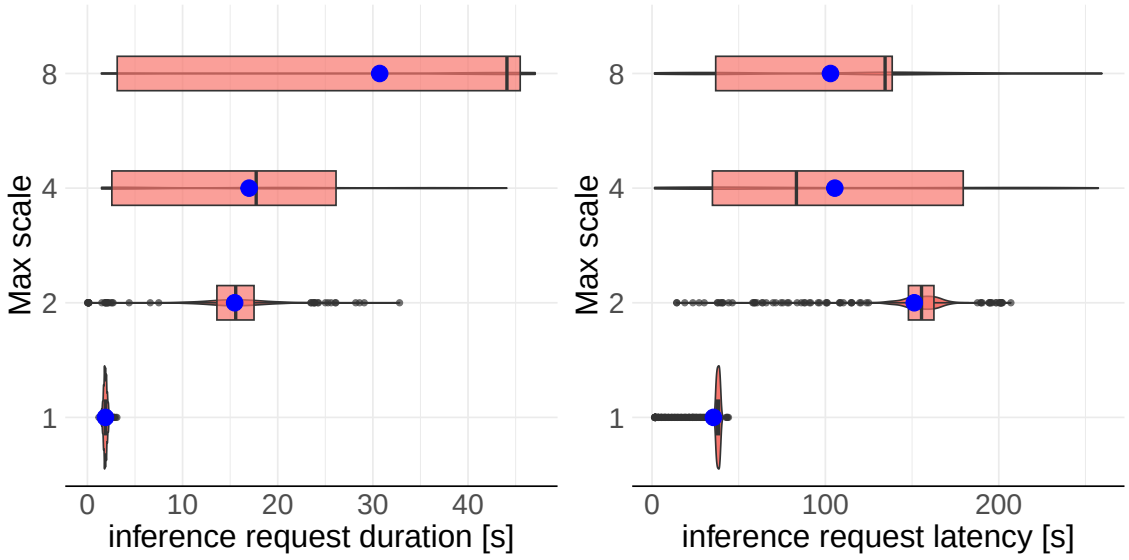


Figure 4.7: Inference request duration and latency comparison with 4 horizontal scaling configurations. The plot on the left displays the time it takes to run the ML model. The plot on the right also takes into account additional latency, such as Knative scale, pod startup time, and network time.

We believe that in our deployments, the computation was optimized for running on 8 cores, even when the pod has less resources available. The deployments with 1 CPU core does not get the benefit of multicore computing, while incurring the parallelism overhead. As we move up in the number of cores, we see that the overhead of incorrectly estimating the number of available cores gets diminished, and hence the performance per core increases. When moving from 4 to 8 cores, the performance per core drops. The drop is minimal and could be attributed either to system noise, or to the fact that the deployment may not get the full 8 CPU cores to compute, due to operating system and K8s consuming small amounts of CPU on the node.

4.2.4 Experiment 3: Horizontal Scaling

This experiment tests the horizontal scalability (i.e., increasing the number of pods) of the deployment. The sensitive variable is the maximum scale that Knative is allowed to scale a pod up to. We pick 4 scale values: 1, 2, 4, and 8. In this experiment we analyse both the inference request duration, and the overall inference latency (inference duration + pod initialisation time, network latency, etc.), to determine whether the latency or variability improve.

We show the results in Figure 4.7. First, we analyse the left plot: the inference request duration. This represents the time it takes one model server to process a single request.

In theory, since we do not vary the amount of CPU used for the different configurations, we would expect to see similar distributions across all deployments. However, we observe that the deployments with higher scalability factors perform significantly worse. This can only be attributed to K8s CPU scheduling. More precisely, as more pods get created, K8s throttles the already existing pods, presumably to save resources. This leads to our third main findings:

MF3. Horizontal scaling increases the total inference request latency by up to 190%, and should not constitute the preferred scaling method, due to the increased computational cost.

This behaviour also reflects in the inference request latency, which is displayed in the plot on the right in Figure 4.7. First, we notice that the deployment with maximum scale 1 has the best performance, and the least performance variability. We also note the large discrepancy between the inference request duration and latency. This can be explained by relatively slow pod startup times, combined with the bursty nature of the requests: when the request burst arrives, all requests get queued (in the Knative activator) while the pod is starting. Once the pod is ready, the requests from the queue get processed one by one, resulting in the high latency we observe.

Despite the higher inference duration of the models with maximum scale 4 and 8, the inference latency of these two configurations is overall lower than that with scale 2. The intuition behind this result is that the deployments with higher number of pods can handle multiple requests in parallel, hence the overall inference latency gets divided by their maximum scale.

Chapter 5

Related Work

MLOps The rise in popularity of Machine Learning sparked interest in ML hardware and systems research. Verbraeken et al. [26] presents a comprehensive study of distributed ML. The term MLOps appeared around 2020 (see Figure 5.1). Since, there has been an influx of research into the practices [23, 44], components [32], and tools in MLOps [33, 45].

Serverless Computing Serverless computing emerged in the 2010s [10], and matured during the decade, simultaneously with ML. In 2014, AWS launched Lambda [11], which is currently the most popular function-as-a-service platform [46]. Van Eyk et al. [16] introduce a reference architecture for FaaS, which this research is closely related to, and builds upon.

Serverless Machine Learning Advances in serverless computing and machine learning have sparked interest in serverless ML workloads, in both training [47, 48] and inference [49, 50, 51]. Studies on the performance of serverless ML on inference workloads are closely related to our experiments (chapter 4). Barrak et al. [12] perform a literature review of the serverless machine learning literature. This relates to our research in two ways: first, it gives a holistic view of the serverless ML landscape; second, it provides us with sources for the reference architecture components.

This work aligns with the AtLarge vision on distributed systems research [52], by extending serverless research to the area of machine learning, and characterizing the performance of serverless ML systems.

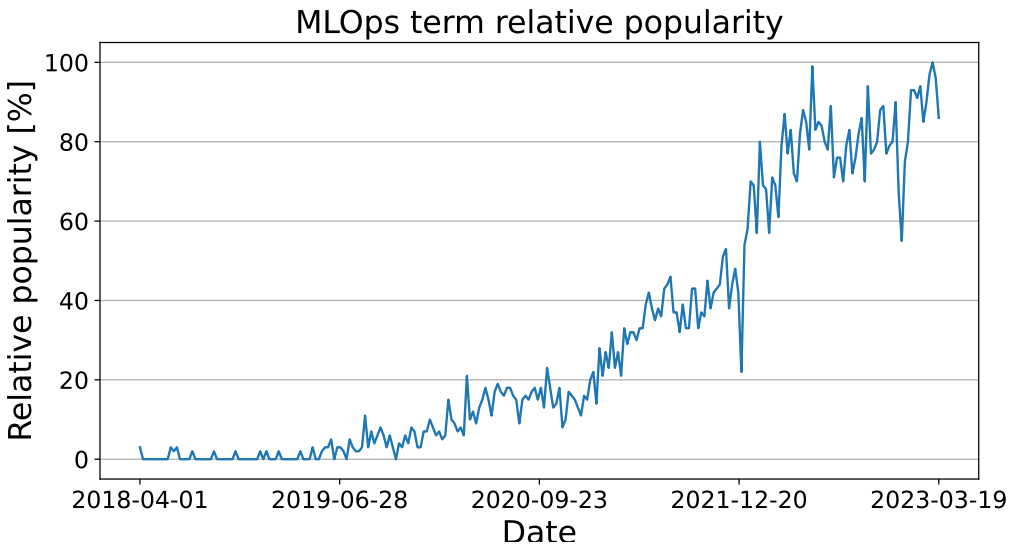


Figure 5.1: “MLOps" search term relative popularity. Source: Google Trends [43]. Time interval: 01-04-2018 to 01-04-2023. MLOps begins to gain popularity around late 2019.

Chapter 6

Conclusion

Machine learning is impacting society in a wide variety of fields (medicine, education, entertainment, etc.), and is becoming a central part of our lives. At the same time, serverless computing is emerging as a promising cloud computing model, that has seen wide-spread adoption in the industry.

In this research we investigate the emerging practices and design patterns of serverless machine learning systems. The main contribution of this work is a reference architecture for serverless machine learning, which we validate against a set of 19 platforms, and find a good fit. We then run experiments to understand the performance and scalability characteristics of the serverless ML inference workloads.

6.1 Answering The Research Questions

RQ1 What is the reference architecture of serverless machine learning systems?

In Chapter 3 we introduce the reference architecture for serverless machine learning. We derive three of the layers (Resource Orchestration, Serverless Runtime, and Workflow Layer) from the SPEC-RG reference architecture for FaaS [16], and introduce three novel layers: Machine Learning, Pipelines, and Eventing.

To evaluate the reference architecture, we select a set of 19 representative platforms, and for each we identify which components from the reference architecture are present. We find that the reference architecture accurately captures the components of the tested platforms. Here we identify an issue with open-source software, namely the lack of end-to-end solutions, and the poor integration between different platforms.

Finally, we present and explain a commonly identified pattern (using the reference architecture), namely model drift detection.

RQ2 What can we learn about the operation of serverless machine learning systems through data collection and data-driven analysis of real world deployments?

Section 4.1 presents a data analysis on real world ML deployments. We monitor three machine learning model deployments, two of which are neural networks (CNN and MLP), and one is a KNN search algorithm. For each deployment, we collect metrics (CPU and memory utilization, autoscaler metrics, etc.) and traces.

From the data analysis, we derive experiment questions to help answer unexpected results, and better understand the performance of serverless ML systems.

RQ3 How to analyse the performance of serverless machine learning systems?

Having proposed three experiment questions, we provide answers through running three experiments. Each experiment is a sensitivity analysis on one variable of the ML model deployment, namely node scheduling (dedicated vs any node), horizontal scaling factor, CPU usage.

From the analysis, we derive three main findings (MF):

MF1 Constraining serverless scheduling of ML inference workloads to dedicated nodes reduces the 99th percentile inference request duration by 99%.

MF2 Vertical scaling greatly decreases the inference duration, and can offer more than 50% percent improvement in the relative performance per core.

MF3 Horizontal scaling increases the total inference request latency by up to 190%, and should not constitute the preferred scaling method, due to the increased computational cost.

6.2 Limitations and Future Work

6.2.1 Reference Architecture

Component Delimitation The reference architecture captures a high-level overview of the layers that we consider most relevant to serverless ML. Due to the vastness of the ML field, we have identified a wide variety of components, and functions that these components fulfil, within the ML layer. The exact delimitation of each component differs very much per platform. For instance, the Model Registry (**M2**) and the Model Metadata Store (**M3**) could be seen as the same component. Furthermore, end-to-end solutions such as AWS SageMaker or Azure Machine Learning may integrate a wide range of functions in one solution, hence not offering proper delimitation. The unclear lines between the different components, as well as the difference between the concepts that different platforms use, may constitute a threat to the validity of the proposed model.

Reference Architecture Validation In this work, we test the reference architecture using a set of 19 platforms. Due to the large number of layers and components, this may not be enough to properly validate the model.

For future studies, we recommend more platforms be investigated to better validate the reference architecture. The additional knowledge gained while mapping more platforms could also result in better delimitation of the components, or the identification of more layers.

DataOps Machine Learning is inherently dependent on data. The scope of this research does not include components/layers around the practice of DataOps. We believe an important direction for future research to be adding DataOps to the reference architecture.

6.2.2 Data Analysis and Experiments

ML Model Lifecycle Phase The deployments in this work focus on the inference phase of the ML model lifecycle. This is mainly due to the fact that the Kubernetes cluster where the models are deployed does not have ML accelerators. Training of ML models is usually compute intensive, and heavily optimized for the GPU. CPU training would not constitute a realistic workload, hence we leave the training stage for future research to tackle.

Machine Learning Accelerators All of our deployments perform inference on the CPU, and do not utilize any specialized devices, such as GPUs or TPUs [53]. The main reason for this choice is cost. We would like to see future research analyse the use of accelerators in serverless inference settings: analyse the cost, and the overhead of sharing hardware between multiple deployments.

Model Variety Our three ML models are realistic use cases, but they do not offer a holistic view of the ML landscape. Specifically, the neural network models are relatively small compared to the state-of-the-art models (e.g., GPT-4) used in the industry. Although this research focuses on inference using the CPU, it is important to understand the limitations of the serverless ML workloads, i.e., if larger models can run well in such system. We leave this question for future research to explore.

Batch Computation One potential weak point of this work is the lack of input batching. ML frameworks are optimized for parallel computation on multiple input samples (batching). For simplicity, we process each request independently. However, as shown in this research, vertical scaling is to be preferred (where possible) over horizontal scaling. As such, batching could potentially significantly reduce inference request duration during bursts.

System Under Test Although the proposed reference architecture introduces 11 novel components in the ML layer, the experiments mainly focus on the model server performance. For future research we recommend a holistic approach, or more components to be included in the study.

K8s CPU Allocation One observation we have made during the experiments is that the model server container reports that it has available all cores of the node it is running on (e.g., on a node with 32 cores, a container with a limit of 1 CPU core reports that it has 32 cores available). This could constitute a performance issue when running ML models. Specifically, ML frameworks, such as PyTorch, make efficient use of CPU parallelism. The framework may try to make use of parallel computing, resulting in a potential overhead during the computation. This issue is a potential source of performance variability. We strongly recommend future research to explore this direction.

Knative Tuning Knative offers many parameters that can be adjusted to optimize each deployment. In this work we only explore one parameter, namely the maximum scale. Here we list more parameters that we believe are important to be studied, to understand the performance of serverless ML workloads: minimum scale, request concurrency/requests per second target, scale down delay.

QoS Classes Kubernetes classifies pods into three "Quality of Service" classes [54]: guaranteed, burstable, and best effort, depending on the resource requests and limits. The K8s scheduler uses these classes (along-side many other variables) to make decisions on resource scheduling. All of our deployments fit into the burstable QoS class. We believe that the QoS class can have a big impact on performance variability, hence we believe this is an important topic for future research.

Bibliography

[1] Mauricio Delbracio et al. *Mobile Computational Photography: A Tour*. 2021. DOI: 10.48550/ARXIV.2102.09000. URL: https://arxiv.org/abs/2102.09000.

[2] Jiahui Yu et al. *CoCa: Contrastive Captioners are Image-Text Foundation Models*. 2022. DOI: 10.48550/ARXIV.2205.01917. URL: https://arxiv.org/abs/2205.01917.

[3] Yujian Mo et al. "Review the state-of-the-art technologies of semantic segmentation based on deep learning". In: *Neurocomputing* 493 (2022), pp. 626–646. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2022.01.005. URL: https://www.sciencedirect.com/science/article/pii/S0925231222000054.

[4] Wenguan Wang et al. "A Survey on Deep Learning Technique for Video Segmentation". In: (July 2021).

[5] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: 2303.08774 [cs.CL].

[6] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2018. DOI: 10.48550/ARXIV.1810.04805. URL: https://arxiv.org/abs/1810.04805.

[7] *Introducing chatgpt*. URL: https://openai.com/blog/chatgpt.

[8] Ubs. *Let's chat about chatgpt*. Sept. 2019. URL: https://www.ubs.com/global/en/wealth-management/our-approach/marketnews/article.1585717.html.

[9] *Hugging face*. URL: https://huggingface.co.

[10] Erwin van Eyk et al. "Serverless is More: From PaaS to Present Cloud Computing". In: *IEEE Internet Computing* 22.5 (Sept. 2018), pp. 8–17. DOI: 10.1109/mic.2018.053681358. URL: https://doi.org/10.1109/mic.2018.053681358.

[11] *Introducing AWS Lambda*. Accessed: 2023-03-27. URL: https://aws.amazon.com/about-aws/whats-new/2014/11/13/introducing-aws-lambda/.

[12] 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. URL: https://doi.org/10.1109/access.2022.3206366.

[13] *AWS SageMaker*. Accessed: 2023-05-26. URL: https://www.google.com/search?client=safari&rls=en&q=aws+sagemaker&ie=UTF-8&oe=UTF-8.

[14] *AWS SageMaker Serverless Inference*. Accessed: 2023-05-29. URL: https://docs.aws.amazon.com/sagemaker/latest/dg/serverless-endpoints.html.

[15] *Hadrian*. Accessed: 2023-05-08. URL: https://hadrian.io.

[16] Erwin van Eyk et al. "The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms". In: *IEEE Internet Computing* 23.6 (2019), pp. 7–18. DOI: 10.1109/MIC.2019.2952061.

[17] Nuha Alshuqayran, Nour Ali, and Roger Evans. "A systematic mapping study in microservice architecture". In: *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE. 2016, pp. 44–51.

[18] *Docker*. Accessed: 2023-03-30. URL: https://www.docker.com.

[19] *AWS Lambda*. Accessed: 2023-05-29. URL: https://aws.amazon.com/lambda/.

[20] *Google Cloud Functions*. Accessed: 2023-05-29. URL: https://cloud.google.com/functions.

[21] *Kubernetes*. Accessed: 2023-03-30. URL: https://kubernetes.io/.

[22] *Knative*. Accessed: 2023-03-30. URL: https://knative.dev/.

[23] Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. *Machine Learning Operations (MLOps): Overview, Definition, and Architecture*. 2022. eprint: arXiv:2205.02302.

[24] Deepak Narayanan et al. "Efficient Large-Scale Language Model Training on GPU Clusters". In: *CoRR* abs/2104.04473 (2021). arXiv: 2104.04473. URL: https://arxiv.org/abs/2104.04473.

[25] Jordan Hoffmann et al. *Training Compute-Optimal Large Language Models*. 2022. arXiv: 2203.15556 [cs.CL].

[26] Joost Verbraeken et al. "A Survey on Distributed Machine Learning". In: *ACM Comput. Surv.* 53.2 (Mar. 2020). ISSN: 0360-0300. DOI: 10.1145/3377454. URL: https://doi.org/10.1145/3377454.

[27] Tong Yu and Hong Zhu. *Hyper-Parameter Optimization: A Review of Algorithms and Applications*. 2020. arXiv: 2003.05689 [cs.LG].

[28] Tong Yu and Hong Zhu. "Hyper-Parameter Optimization: A Review of Algorithms and Applications". In: *CoRR* abs/2003.05689 (2020). arXiv: 2003.05689. URL: https://arxiv.org/abs/2003.05689.

[29] Alexandru Agache et al. "Firecracker: Lightweight Virtualization for Serverless Applications". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. ISBN: 978-1-939133-13-7. URL: https://www.usenix.org/conference/nsdi20/presentation/agache.

[30] *MLOps: Continuous delivery and automation pipelines in machine learning*. Accessed: 2023-03-23. URL: https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning#characteristics_3.

[31] Rosana Noronha Gemaque et al. "An overview of unsupervised drift detection methods". In: *WIREs Data Mining and Knowledge Discovery* 10.6 (July 2020). DOI: 10.1002/widm.1381. URL: https://doi.org/10.1002/widm.1381.

[32] Yue Zhou, Yue Yu, and Bo Ding. "Towards MLOps: A Case Study of ML Pipeline Platform". In: *2020 International Conference on Artificial Intelligence and Computer Engineering (ICAICE)*. 2020, pp. 494–500. DOI: 10.1109/ICAICE51518.2020.00102.

[33] Nipuni Hewage and Dulani Meedeniya. "Machine Learning Operations: A Survey on MLOps Tool Support". In: (2022). DOI: 10.48550/ARXIV.2202.10169. URL: https://arxiv.org/abs/2202.10169.

[34] *AWS S3*. Accessed: 2023-05-30. URL: https://aws.amazon.com/s3/.

[35] *PostgreSQL*. Accessed: 2023-05-30. URL: https://www.postgresql.org.

[36] *Ranking of the most popular relational database management systems worldwide, as of January 2022*. Accessed: 2023-05-30. URL: https://www.statista.com/statistics/1131568/worldwide-popularity-ranking-relational-database-management-systems/.

[37] *MLOps: Continuous delivery and automation pipelines in machine learning*. Accessed: 2023-04-06. URL: https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning#mlops_level_2_cicd_pipeline_automation.

[38] Keiron O'Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks*. 2015. arXiv: 1511.08458 [cs.NE].

[39] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL].

[40] *Elastic*. Accessed: 2023-05-22. URL: https://www.elastic.co.

[41] Hassan Ramchoun et al. "Multilayer perceptron: Architecture optimization and training". In: (2016).

[42] Klaas Meinke et al. "A Bag of Tokens Neural Network to Predict Webpage Age". In: *Proceedings of the Conference on Scientific Computing and Machine Learning (CSCML)*. 2023.

[43] *MLOps term popularity in Google Search*. Accessed: 2023-04-01. URL: https://trends.google.com/trends/explore?date=2018-01-04%5C%202023-01-04&q=%5C%2Fg%5C%2F11h1vbjpbg&hl=en.

[44] Matteo Testi et al. "MLOps: A Taxonomy and a Methodology". In: *IEEE Access* 10 (June 2022), pp. 63606–63618. DOI: 10.1109/ACCESS.2022.3181730.

[45] G. Symeonidis et al. *MLOps – Definitions, Tools and Challenges*. 2022. arXiv: 2201.00162 [cs.LG].

[46] Simon Eismann et al. *A Review of Serverless Use Cases and their Characteristics*. 2021. arXiv: 2008.11110 [cs.SE].

[47] Joao Carreira. "A Case for Serverless Machine Learning". In: 2018.

[48] Hao Wang, Di Niu, and Baochun Li. "Distributed Machine Learning with a Serverless Architecture". In: *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. IEEE, Apr. 2019. DOI: 10.1109/infocom.2019.8737391. URL: https://doi.org/10.1109/infocom.2019.8737391.

[49] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. *Serving deep learning models in a serverless platform*. 2017. DOI: 10.48550/ARXIV.1710.08460. URL: https://arxiv.org/abs/1710.08460.

[50] Ahsan Ali et al. "BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching". In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Nov. 2020. DOI: 10.1109/sc41405.2020.00073. URL: https://doi.org/10.1109/sc41405.2020.00073.

[51] Simon Shillaker and Peter Pietzuch. "FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing". In: USENIX Association, 2020. ISBN: 978-1-939133-14-4.

[52] Alexandru Iosup et al. *The AtLarge Vision on the Design of Distributed Systems and Ecosystems*. 2019. DOI: 10.48550/ARXIV.1902.05416. URL: https://arxiv.org/abs/1902.05416.

[53] Albert Reuther et al. “AI and ML Accelerator Survey and Trends”. In: *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2022, pp. 1–10.

[54] *Kubernetes Quality of Service classes*. Accessed: 2023-05-24. URL: `https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/`.

Appendix A

Reproducibility

A.1 Abstract

This appendix presents information on how to access the open-source artifacts produced during this research, namely the processed data used during analysis, as well as the data used during experiments, and the respective R analysis notebooks. Additionally, we include the

A.2 Artifact check-list (meta-information)

Obligatory. Use just a few informal keywords in all fields applicable to your artifacts and remove the rest. This information is needed to find appropriate reviewers and gradually unify artifact meta information in Digital Libraries.

- **Program:** data analysis on serverless ML deployments
- **Data set:** three CSV files containing metrics and traces of the deployments
- **Run-time environment:** R, Python, Jupyter Notebook
- **Metrics:** CPU & memory utilization, autoscaler metrics
- **Output:** plots
- **How much disk space required (approximately)?:** 4 GB
- **Publicly available?:** yes
- **Code licences (if publicly available)?:** GPL
- **Data licences (if publicly available)?:** CC-BY 4.0

A.3 Description

A.3.1 How to access

Clone the GitHub repository:

The code is available on GitHub, at:

`https://github.com/tiberiuiancu/thesis-experiments-public`.

Due to the dataset size, we upload the data in Microsoft OneDrive. The code and data can be downloaded from:

`https://1drv.ms/f/s!AgN5TjceGVdJiyo6Qv3gR8dxBGKM?e=iQOr9g`

A.3.2 Software dependencies

The data analysis notebooks rely on Jupyter Notebook to be installed, as well as R and the R kernel for Jupyter.

A.3.3 Data sets

The data sets are available for download together with the code at:

`https://1drv.ms/f/s!AgN5TjceGVdJiyo6Qv3gR8dxBGKM?e=iQOr9g`

A.4 Installation

Assuming you have a valid Python3 installation, you can install Jupyter notebook by running the following in the terminal:

```
pip install jupyter notebook
```

To install the R kernel for Jupyter (assuming you have R already installed):

```
R
install.packages('IRkernel')
IRkernel::installspec()
```

A.5 Evaluation and expected results

To run the analysis notebooks, first start a Jupyter notebook in the folder that contains the code and data:

```
jupyter notebook
```

In your browser, go to *localhost:8888* and navigate to the desired analysis file (ending in *.ipynb)*. Without running any cell, the notebook already contains the plots displayed in the report. Re-running the notebook should always produce the same results.

A.6 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html