

Aalto University  
School of Science  
Master's Programme in Security and Cloud Computing

Juan Pablo Valencia Gómez

# Leveraging spot instances for resource provisioning in serverless computing

Master's Thesis  
Espoo, July 31, 2023

Supervisors: Professor Mario Di Francesco, Aalto University  
Professor Raja Appuswamy, EURECOM

<b>Author:</b>	Juan Pablo Valencia Gómez		
<b>Title:</b>	Leveraging spot instances for resource provisioning in serverless computing		
<b>Date:</b>	July 31, 2023	<b>Pages:</b>	55
<b>Major:</b>	Security and Cloud Computing	<b>Code:</b>	SCI3113
<b>Supervisors:</b>	Professor Mario Di Francesco Professor Raja Appuswamy		
<p>Cloud computing has become a dominating paradigm across the IT industry. However, keeping cloud costs under control is a major challenge for organizations. One option to save costs is using spot instances: virtual machines that have highly discounted prices at the expense of lower reliability and availability.</p> <p>Serverless computing is a paradigm that allows developers to build and deploy applications in the cloud without provisioning or managing backend infrastructure. Function as a Service (FaaS) is the prevalent delivery model of this paradigm, which allows developers to execute functions in the cloud as a response to a request or an event. The developer focuses only on the code, and the cloud provider handles the execution and scaling of the functions. This is convenient for developers, but comes with some limitations and can become very expensive at scale.</p> <p>This thesis investigates leveraging spot instances for running serverless functions, and potentially achieve both higher flexibility and cost reduction compared to commercial FaaS solutions. For this purpose, we present a system design, suitable for applications that tolerate some execution latency, and implement it in Google Cloud Platform. Our implementation is compared against Google Cloud Run, a service that offers a similar functionality.</p> <p>Our system achieves significant cost savings: assuming a function execution time of two minutes, our system has the same price as the Cloud Run solution at around 8,000 requests per month, and at, for example, 20,000 requests per month, the cost is less than half of Cloud Run. However, one important design decision is that a spot instance is provisioned on the fly for every request. While this introduces latency, it also allows the system to achieve no significant reduction in reliability, as was confirmed in our evaluation.</p>			
<b>Keywords:</b>	cloud computing, spot instances, serverless computing		
<b>Language:</b>	English		

<b>Auteur:</b>	Juan Pablo Valencia Gómez		
<b>Titre:</b>	Leveraging spot instances for resource provisioning in serverless computing		
<b>Date:</b>	Juillet 31, 2023	<b>Pages:</b>	55
<b>Diplôme:</b>	Security and Cloud Computing	<b>Promo:</b>	2023
<b>Superviseurs:</b>	Professeur Mario Di Francesco Professeur Raja Appuswamy		
<p>L'informatique en nuage (ou cloud computing) est devenue un paradigme dominant dans l'industrie informatique. Cependant, la maîtrise des coûts du cloud est un défi majeur pour les organisations. Une option pour réduire les coûts est d'utiliser des instances spot : des machines virtuelles dont les prix sont fortement réduits au détriment d'une fiabilité et d'une disponibilité moindres.</p> <p>L'informatique sans serveur (ou serverless computing) est un paradigme qui permet aux développeurs de créer et de déployer des applications dans le cloud sans provisionner ni gérer l'infrastructure backend. La fonction en tant que service (ou Function as a Service – FaaS) est le modèle de livraison prédominant de ce paradigme, qui permet aux développeurs d'exécuter des fonctions dans le cloud en réponse à une demande ou à un événement. Le développeur se concentre uniquement sur le code, et le fournisseur de cloud gère l'exécution et la mise à l'échelle des fonctions. Ceci est pratique pour les développeurs, mais présente certaines limitations et peut devenir très coûteux à grande échelle.</p> <p>Cette thèse étudie l'utilisation d'instances spot pour exécuter des fonctions sans serveur (serverless functions), et potentiellement atteindre à la fois une plus grande flexibilité et une réduction des coûts par rapport aux solutions FaaS commerciales. À cette fin, nous présentons une conception de système, adaptée aux applications qui tolèrent une certaine latence d'exécution, et l'implémentons dans Google Cloud Platform. Notre implémentation est comparée à Google Cloud Run, un service qui offre une fonctionnalité similaire.</p> <p>Notre système permet de réaliser des économies de coûts significatives : en supposant un temps d'exécution de la fonction de deux minutes, notre système présente un coût équivalent à celui de la solution Cloud Run à environ 8,000 requêtes par mois, et à, par exemple, 20,000 requêtes par mois, le coût est inférieur à la moitié de Cloud Run. Cependant, une décision de conception importante est qu'une instance spot est provisionnée à la volée pour chaque demande. Bien que cela introduise une latence, cela permet également au système de ne pas réduire significativement la fiabilité, comme confirmé dans notre évaluation.</p>			
<b>Mots-clés:</b>	cloud computing, spot instances, serverless computing		
<b>Langue:</b>	Anglais		

# Acknowledgements

First of all, I would like to thank my parents Luis and Olga for all their love and support. This achievement would not have been possible without them.

I would like to thank Professor Mario Di Francesco at Aalto University for selecting me for this project and for his guidance and constructive feedback throughout its development. I would also like to thank Professor Raja Appuswamy at EURECOM for his advice in the initial stages of this thesis.

Last but not least, I would like to thank all the friends I have made over the past two years. They have made this journey the experience of a lifetime.

Espoo, July 31, 2023

Juan Pablo Valencia Gómez

# Abbreviations and Acronyms

API	Application Programming Interface
AWS	Amazon Web Services
AZ	Availability Zone
CD	Continuous Deployment
CI	Continuous Integration
FaaS	Function as a Service
GCP	Google Cloud Platform
HTTP	Hypertext Transport Protocol
IaaS	Infrastructure as a Service
JSON	JavaScript Object Notation
PaaS	Platform as a Service
REST	Representational State Transfer
RPC	Remote Procedure Call
SaaS	Software as a Service
SLA	Service Level Agreement
SLO	Service Level Objective
VM	Virtual Machine
VPC	Virtual Private Cloud

# Contents

<b>Abbreviations and Acronyms</b>	<b>5</b>
<b>1 Introduction</b>	<b>8</b>
1.1 Contributions . . . . .	9
1.2 Structure of the thesis . . . . .	10
<b>2 Background</b>	<b>11</b>
2.1 Cloud computing . . . . .	11
2.2 Computing resources in the cloud . . . . .	15
2.3 Spot instances . . . . .	16
2.4 Serverless computing . . . . .	24
2.5 Related work . . . . .	27
<b>3 Design</b>	<b>30</b>
3.1 Motivation . . . . .	30
3.2 Requirements . . . . .	31
3.3 Preliminary considerations . . . . .	31
3.4 System design . . . . .	32
<b>4 Implementation</b>	<b>37</b>
4.1 Code structure . . . . .	37
4.2 Deployment considerations . . . . .	37
4.3 Controller . . . . .	38
4.4 Function container . . . . .	39
4.5 Worker . . . . .	39
<b>5 Evaluation</b>	<b>41</b>
5.1 Cost analysis . . . . .	41
5.2 Performance . . . . .	46
5.3 Benefits . . . . .	47
5.4 Drawbacks . . . . .	48



# Chapter 1

## Introduction

Cloud computing is a model for enabling on-demand access to computing resources over the internet [32]. Users can quickly acquire and release resources as needed, and pay only for the time of usage. As a consequence, users do not need to plan ahead for the required capacity. These features have made cloud computing extremely popular over the last decade.

The cloud allows engineers to innovate quickly, however, cloud spending in an organization can easily become unmanageable without adequate visibility and control. In a recent survey [13], 37% of the respondents said they have been surprised by their cloud bills, 49% stated that getting their cloud costs under control is a difficult process, and 89% identified that optimizing cloud spending and reducing the costs was a top priority.

One of the main options to save costs in the cloud is leveraging spot instances [31]. Spot instances are virtual machines allocated from spare computational resources of the cloud providers. Users obtain access to these instances at highly discounted prices (typically 30% to 90%), but with some trade-offs: a request for obtaining a spot instance may fail if the cloud provider does not have enough capacity at the time of requesting, and a running spot instance can be claimed back (interrupted) by the cloud provider at any time with only a short notice (30 seconds to 2 minutes, depending on the cloud provider).

Due to their unreliable nature, spot instances are best suited for stateless and fault-tolerant workloads [4]. Some examples include big data analytics, batch processing, image or media rendering and transcoding, high performance computing, CI/CD pipelines and training machine learning models. Some researchers have designed solutions to leverage spot instances for running latency-sensitive web services [21], scientific computing workloads [28, 39] and seismic image processing [33] as well. One direction that has been explored marginally, if at all, is serverless computing on top of spot

instances.

Serverless computing is a paradigm that emerged in the last few years, which allows developers to build and deploy applications in the cloud without provisioning or managing backend infrastructure [19]. In fact, the cloud provider fully manages the scaling of the applications by adapting to the increase and decrease of demand. Function as a Service (FaaS) is the prevalent delivery model of serverless computing. In this model, a code function is executed as response to a request or an event trigger.

While the main selling point of serverless computing is its ease of use, it is necessary to understand well its features and limitations, and carefully consider if it is an adequate solution for the intended use case. Understanding its economic aspect is particularly important. In fact, for low work volumes serverless can be very cost-effective, whereas it can become much more expensive than traditional cloud solutions such as Infrastructure as a Service (IaaS) or Platform as a Service (PaaS) for large volumes [25]. Other drawbacks of FaaS are vendor lock-in, bounded execution time, limited selection of programming languages, and constraints on the computational resources a function can use.

In this context, there is an opportunity to leverage spot instances to build a system for running serverless functions. This could lead to cost reductions, given the low price of spot instances, and also to higher flexibility, since the system would not be subject to the restrictions that cloud providers impose on FaaS solutions. However, the design needs to take into account the unreliable nature of spot instances. This thesis explores this possibility by designing a system and evaluating its effectiveness along different dimensions.

## 1.1 Contributions

The contributions of this thesis are as follows:

- A thorough examination of the characteristics of spot instances as available at different cloud providers.
- A literature review on spot instances, covering research that characterizes their performance and availability, as well as applications built on top of them.
- The design of a system to run serverless functions on spot instances, along with a proof-of-concept implementation.

- An evaluation of the proposed solution in terms of both qualitative (benefits and drawbacks) and quantitative (performance, costs) aspects.

## 1.2 Structure of the thesis

The rest of this thesis is structured as follows. Chapter 2 overviews cloud computing, followed by a comprehensive examination on spot instances. It also introduces serverless computing as well as the Function as a Service delivery model, and the related work. Chapter 3 presents the requirements, scope and design for a system that leverages spot instances to run serverless functions. Chapter 4 describes a proof-of-concept implementation of the conceived solution. Chapter 5 evaluates the proposed system. Finally, Chapter 6 provides concluding remarks and presents suggestions for future work.

## Chapter 2

# Background

This chapter overviews the core concepts of cloud computing, followed by a comprehensive discussion on spot instances. It then introduces serverless computing, and the related work.

## 2.1 Cloud computing

### 2.1.1 General characteristics

Cloud computing is a model for enabling on-demand access to computing resources over the internet [32]. These resources can include, for instance, servers, networks, storage or applications.

There are a few key characteristics that differentiate cloud computing from the “traditional” model of computing [22, 32]. First of all, cloud computing offers the user the impression of infinite computing resources being available. These resources can be acquired at any time, and so quickly that users do not need to plan their capacity in advance. Moreover, cloud computing eliminates up-front commitments by the users, favoring a “pay as you go” model, wherein users can acquire and release resources as needed and pay only for the time of usage.

The cloud is a cost-effective way for companies to provision computing resources, since it offsets the costs associated with hosting infrastructure on-premises [18]. This, in addition to the “pay as you go” model and the ability to rapidly respond to changes in demand (known as *elasticity*), has made cloud computing extremely popular over the last decade.

### 2.1.2 Use cases

Cloud computing is especially suitable for use cases that present fluctuations in the required computing resources [22], for instance, a service with periodic spikes in traffic (e.g., high utilization during the day and low during the night) or with increased traffic due to particular events (e.g., being featured in the news, or the Black Friday sales for e-commerce sites). In this situation, provisioning a data center to handle the peak load would be a high investment and will lead to under-utilization for most of the time. Instead, using the cloud and taking advantage of its elasticity to accommodate to the demand can entail significant cost savings.

Another case where cloud computing stands out is when the demand is unknown. For instance, it can be hard for a start-up company to accurately estimate the amount of users they will have. Overestimating would result in wasted resources, but underestimating could lead to a low quality of service, which is a critical factor since it can turn away potential users [22]. Again, the elasticity of the cloud can solve this problem.

Cloud computing is also employed to speed up big data analytics, since it costs the same to use 1,000 machines for one hour than one machine for 1,000 hours [22]. Similarly, the cloud is leveraged to speed up scientific computing workloads and the training of machine learning models.

Cloud providers have data centers in multiple locations around the world. This eases the implementation of disaster recovery and business continuity strategies [18], since it allows customers to set up redundancy in a cost-effective way, and distribute the resources at different physical locations to mitigate the impact of local phenomena such as power outages or natural disasters.

The global reach of the cloud also allows companies to deploy applications near to where their end users are located, thereby reducing latency and improving the quality of service. Furthermore, it facilitates compliance with regulations that may limit where data can be stored and processed.

### 2.1.3 Service models

Traditionally, cloud computing has been divided into three service models [32]:

- Software as a Service (SaaS): an application software that is hosted on the cloud, and that user accesses over the internet, typically through a web browser, a mobile application, or an API [18]. The user does not manage or control the application nor its underlying infrastructure.

- Platform as a Service (PaaS): the user deploys an application onto the cloud infrastructure of the provider. The provider takes care of managing the underlying infrastructure, while the user focuses only on developing the application.
- Infrastructure as a Service (IaaS): the user provisions low-level computing resources offered by the provider, such as servers, networks and storage. The user is responsible for configuring and managing said resources, as well as the applications deployed on them.

### 2.1.4 Deployment models

Cloud computing is typically divided into deployment models as well [32]:

- Public cloud: a provider makes the computing resources available to the general public. The cloud provider’s infrastructure is shared by all the customers [18].
- Private cloud: the computing resources are dedicated to the use of a single organization. The cloud infrastructure may be owned and managed by the organization itself or by a third-party provider, and may reside on or off premises [32].
- Hybrid cloud: when an organization makes use of both a private and a public cloud, along with technology that allows to share data and applications between them. One subset of this model is *cloud bursting*, where the public cloud is used to offload tasks than cannot be handled in the private cloud in times of high demand [22].

Amazon Web Services (AWS) is the largest public cloud provider, with a market share of 32% in the first quarter of 2023 [36]. It is followed by Microsoft Azure with 23% and Google Cloud Platform (GCP) with 10%. These providers have also started expanding to the private cloud market (therefore enabling hybrid possibilities too). Other vendors include Cisco, Dell, HPE, IBM, Oracle and VMware [17].

In recent years there has been a rising trend to adopt a multi-cloud approach, that is, use multiple cloud providers. One motivation behind this is to avoid “vendor lock-in”, where a company is dependent on a single provider and faces great difficulties migrating to a different one. In addition, multi-cloud offers extra redundancy to mitigate the impact of failures or outages experienced by a particular provider.

On a recent survey [1], 87% of the participants reported they are using multi-cloud: 72% with a hybrid approach, 13% with multiple public providers and 2% with multiple private providers. The rest of the respondents are using a single provider: 11% a public one and 2% a private one.

### 2.1.5 Regions and zones

Cloud providers have data centers in multiple countries all around the world. These locations are commonly referred to as *regions*. Each region is divided into *availability zones* (AZs), also just called *zones*. The exact definition of a zone varies across providers, but in general a zone can be defined as an independent and isolated set of data centers.

As an example, the AWS region *eu-north-1* is located in Stockholm (Sweden), and consists of three AZs (*eu-north-1a*, *eu-north-1b*, *eu-north-1c*). There can be multiple regions located in the same country, for example, there are two AWS regions in Japan (*ap-northeast-1* in Tokyo and *ap-northeast-3* in Osaka), each with their corresponding AZs.

The cost of a cloud service typically varies across regions. Also, some services may be available only on some regions.

### 2.1.6 Cost management and optimization

On-demand access to computational resources and lack of upfront investments allow engineers to innovate quickly [40]. However, since engineers are not typically used to think about costs and finance, cloud spending in an organization can quickly become unmanageable without adequate visibility and control. This phenomenon is known as *cloud sprawl* [37].

In a recent survey [13], 37% of the respondents said that they have been surprised by their cloud bills. Moreover, 49% stated that getting their cloud costs under control is a difficult process, and 89% identified that optimizing cloud spending and reducing the costs was a top priority. Among the challenges to controlling cloud spending, participants mentioned lack of true visibility into cloud usage and costs (53%), complex pricing models (50%), and complex multi-cloud environments (49%).

There are a few strategies that businesses may use to optimize their cloud costs, including appropriate selection of instance types for the workloads, identifying and deleting unused resources, and using regions with lower cost (keeping in mind that it could lead to increased latency). Additionally, cloud providers offer discounts for long-term commitments. They also often offer spare resources at discounted prices (e.g., spot instances), with the caveat that they may be re-claimed at any time.

Monitoring, controlling and optimizing cloud costs is an ongoing process. The FinOps practice, a collaboration between engineering, finance and business teams, has emerged to help companies in this task [40].

However, using the cloud can be very expensive for an organization, even with adequate management and thorough optimization. Wang and Casado [41] recently estimated that the cloud can be two to three times more expensive than running on-premises for companies operating at scale. The cloud is the most cost-effective way to get started for a company in its early stages, but as the company grows, there is a point where it starts being weighed down by the cloud costs [41]. For this reason, there has been a *repatriation* trend, that is, companies migrating some of their workloads from the cloud into on-premises infrastructure.

## 2.2 Computing resources in the cloud

One of the main services offered by cloud providers is computing in the form of virtual machines (VMs), also called *instances*. The service model in this case corresponds to IaaS (Infrastructure as a Service). Examples include AWS EC2<sup>1</sup> (Elastic Compute Cloud) and GCP Compute Engine<sup>2</sup>. The cloud provider allows the user to choose the operating system (such as Windows or different Linux distributions), as well as the resources that the VM will have (e.g., the amount of RAM, the number of cores, whether it has a GPU, and so on).

Typically there are multiple instance families and types from which the user can choose. For instance, GCP has families of general purpose, compute optimized, memory optimized and accelerator optimized instances [5]. Each of these families has different available sizes. To illustrate, E2 is a general-purpose instance family. An *e2-standard-2* VM has 2 vCPUs and 8 GB of RAM, while an *e2-standard-8* has 8 vCPUs and 32 GB of RAM.

The cloud provider commits to maintain a level of service – usually of 99.99% availability – which means an instance will not be offline for more than 4.32 minutes per month. This is known as a Service Level Agreement (SLA). The provider pays a penalty to the customer if the agreement is breached. As will be discussed later, some services are not covered by SLAs, such as spot instances.

There are three purchase models for computing resources in the cloud:

- On-demand: the user creates and deletes instances whenever they

---

<sup>1</sup><https://aws.amazon.com/ec2/>

<sup>2</sup><https://cloud.google.com/compute/>

choose. There are no long-term commitments. The user is charged per second for the period of time the instance was running. This is the most expensive model.

- Long-term commitment: the user commits in advance for a certain amount of usage for a number of years (usually one or three). The provider applies a discount accordingly.
- Spot: in exchange for lower instance availability and reliability, this model offers discounts that generally exceed those of long-term commitment, but without any commitments. The following section explores in detail how this purchase model works.

## 2.3 Spot instances

### 2.3.1 General characteristics

Cloud providers need to provision a high amount of computing resources in their data-centers to accommodate unpredictable user demands, giving the impression of “infinite” resource availability. However they have a low resource utilization on average (around 20 to 25% according to estimates [21, 31]). This incentivizes providers to sell the spare resources.

Spot instances are virtual machines (VMs) allocated from unused capacity of on-demand and reserved instances. Users obtain access to these VMs at highly discounted prices, but with some trade-offs:

- A spot instance may not be always obtainable. A request for obtaining a spot instance may be delayed or fail if the cloud provider does not have capacity for it at the moment.
- A running spot instance can be claimed back by the cloud provider at any time.

In this sense, spot instances are beneficial both for cloud providers (extra revenue and increase in resource utilization) and for users (lower prices). However, because of their unreliable nature, spot instances are not subject to any SLAs or uptime guarantees [4, 12, 14].

AWS was the first major cloud provider to introduce *Spot Instances* back in 2009. Nowadays, both Azure and GCP offer them as well, under the name of *Spot VMs*. It is also worth mentioning that GCP had a previous version called *Preemptible VMs*, where the instances had a maximum running time of 24 hours [7]. In the literature the terms *transient* and *volatile* are also

used to refer to this kind of instances. Throughout this work we will use the terms *spot instances* and *spot VMs* interchangeably.

As mentioned, spot instances can be reclaimed at any time by the cloud provider. AWS calls these events *interruptions* [20], Azure *evictions* [14], and GCP *preemptions* [12]. The literature also uses the term *revocations*. Throughout this work we will use the term *interruptions* for consistency. The warning time for an interruption is 2 minutes in AWS and 30 seconds in Azure and GCP.

AWS and Azure make no guarantees about the discounts that spot instances can have over on-demand VMs. They only mention that the discount can be “up to 90%” [3, 10]. GCP, on the other hand, guarantees that the discount will be at least 60% and at most 91% [12]. In all three providers the discounts vary between regions, availability zones and instance types.

Spot instances are best suited for stateless and fault-tolerant workloads [4]. Some examples include big data analytics, batch processing, image or media rendering and transcoding, high performance computing, CI/CD pipelines, and training machine learning models.

### 2.3.2 Pricing models

AWS initially used a dynamic pricing scheme for spot instances. Potential customers would place a bid, that is, specify the maximum price they were willing to pay for an instance, and they would obtain it as long as the spot price was less than or equal to the bid. When the spot price exceeded the bid, the instance was interrupted.

The exact algorithm used by AWS to establish the spot price was never disclosed, however, it has been argued that it leveraged an auction model [27]. This means that AWS would set the price to be the  $n$ -th highest bid for  $n$  available spot instances, resulting in the  $n$  highest bidders obtaining them. If a new bid arrived, AWS would update the price to be the new  $n$ -th bid, and the previous holder will be interrupted. Similarly, if capacity decreased and then there were only  $n - 1$  available instances, AWS would update the price and interrupt the  $n$ -th bidder. This behavior is illustrated in Figure 2.1.

It can be seen that in this pricing model the availability of spot instances totally depended on the user bids and the spot prices. Moreover, the nature of the model resulted in highly volatile prices, often rising above the prices of on-demand instances [23]. For these reasons, research on spot instances mainly focused on predicting prices and finding optimal bidding strategies.

From the user perspective, the complexity and volatility of the pricing model was undesirable. Perhaps for this reason, the dynamic pricing mecha-

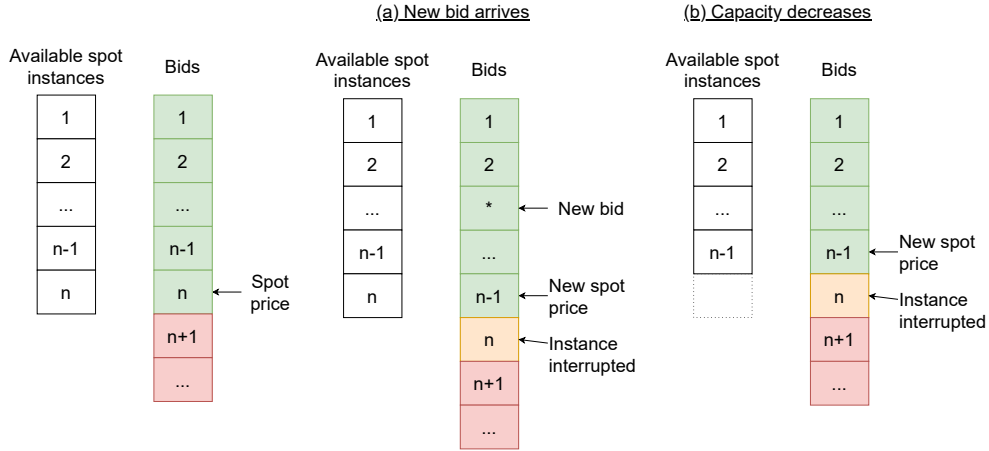


Figure 2.1: Behavior of the dynamic pricing model believed to have been used by AWS.

nism of AWS was replaced in late 2017 by a much simpler one [6], in which the spot prices fluctuate according to long-term trends on supply and demand, and are guaranteed to never exceed the on-demand prices. While the new pricing mechanism is still not transparent to users, it has much more stable prices. This makes it more understandable, potentially attracting new users and also making it easier for companies to estimate project budgets [31].

Under this new model, the interruptions of the spot instances are no longer driven by the bids and the spot prices. Now, interruptions can happen whenever AWS needs the capacity back. This renders most of the previous research obsolete [30].

Baughman et al. [23] performed an empirical analysis comparing the old and new AWS pricing models. They took into account nine months of pricing history, encompassing the time before, during and after the model change, and considered multiple instance types and availability zones. They found that price changes indeed happen more infrequently after the change, and the variance of the prices greatly decreased. This confirms that the new pricing model is more stable.

More recently, Lee et al. [30] performed an empirical study where they collected and analyzed 6 months of data from AWS, consisting of spot prices and two spot availability metrics. Among other results, they found that the spot price does not show any correlation with the other metrics, thus confirming that after the 2017 changes the price no longer reflects the availability of spot instances.

Azure and GCP have similar pricing models as the current one of AWS.

One implication of this model is that, since the price does not reflect availability, users have difficulties figuring out the availability of their resources. Instance interruptions become unpredictable since users do not have visibility over provider capacity. Cloud providers offer some statistics and historical data to help users make decisions though, as will be discussed next.

### 2.3.3 Spot instances in different cloud providers

#### AWS

AWS started offering spot instances in 2009. As already mentioned, they used a highly volatile pricing model until late 2017, when it was replaced by a more stable one.

Creating a spot instance in AWS requires a spot instance request [20], which includes the desired number of instances, the instance type and the availability zone. The request is fulfilled immediately if enough capacity is available. Otherwise, the request remains in an *open* state until the resources become available. Once the request is fulfilled, the spot instance is launched.

There are two kinds of spot requests: one-time and persistent. The difference is that a persistent request is automatically resubmitted when the spot instance is interrupted.

AWS has three different interruption behaviors, that is, what happens when the instance is interrupted:

- Terminate (delete) the instance. Only available for one-time spot requests.
- Stop the instance. Only available for persistent spot requests. The instance is restarted when capacity becomes available.
- Hibernate the instance. Only available for persistent spot requests. The instance is resumed when capacity becomes available.

The root volume (disk) is preserved on stopping and hibernating, and on the latter case the RAM is preserved as well (i.e. it is saved to a file in the root volume) [20]. The user is not charged for a stopped or hibernated instance, but continues to pay for the root volume storage.

Even after the change in the pricing model, AWS still allows the user to specify the maximum price they are willing to pay for a spot instance. However, they discourage users to do so, since this could lead to higher interruption frequencies. If the user does not set a maximum price, the instance is never interrupted due to pricing changes, but it can still be interrupted

due to capacity shortages [20]. The spot price is guaranteed to never exceed the on-demand price.

AWS publishes multiple metrics related to spot instances. Users can leverage this data to pick the instance types and locations according to their needs and constraints. They are:

- Pricing information: the current prices as well as the pricing history over the past 90 days [10]. It can be filtered by instance type, operating system and region.
- Frequency of interruption: the rate at which instances have been interrupted in the previous month, filtered by instance type, operating system and region [9]. The frequency is given in ranges: <5%, 5-10%, 10-15%, 15-20% and >20%.
- Spot placement score: indicates how likely is a spot instance request to succeed [11]. It ranges from 1 to 10, with 10 meaning highly likely. The user specifies the requirements of the spot instances, and AWS returns the ten highest scored regions or availability zones.

The pricing history can be viewed on the AWS portal as well as obtained programatically via the AWS CLI [10]. The same goes for the spot placement score, with some restrictions [11]. The frequency of interruption is only officially available on the website [9], but it can be programatically obtained with the SpotInfo<sup>3</sup> open-source CLI tool.

Lee et al. [30] performed an empirical study to determine the usefulness of these different metrics. They issued 503 spot requests and tracked their interruptions over 24 hours. They found that taking into account both the spot placement score and interruption frequency resulted in better interruption predictions than using only the interruption frequency.

## Azure

Azure started offering spot instances in 2017 under the name “Low-priority VMs”. In 2019 they released “Spot VMs” and started phasing out Low-priority VMs.

Creating a spot instance in Azure follows the same process as that for an on-demand instance. The user only needs to specify the *priority* parameter to be *spot*, optionally specifying the eviction policy (analogous to the interruption behavior in AWS) and maximum price [14].

The available eviction policies are:

---

<sup>3</sup><https://github.com/alexei-led/spotinfo>

- Deallocate (stop) the instance. The disk is preserved and the user is still charged storage costs while the instance is stopped. An important difference with AWS is that Azure does not automatically restart a stopped spot instance – it can be restarted by the user provided that the capacity is available [14].
- Delete the instance.

Just like AWS, Azure discourages users to specify a maximum price since it can lead to higher interruption frequencies. Setting the *maximum price* parameter to “-1” will cause the user to be charged at the current price (which never exceeds the on-demand price) and never be interrupted based on pricing changes [14].

Users can view the pricing history in the Azure portal or obtain it through Azure Resource Graph [14]. The current prices can also be seen on the website [3] or obtained programatically via an API [2].

Azure also offers the eviction rates (interruption frequencies) for the trailing month. They can be seen in the portal or obtained through Azure Resource Graph [14]. The ranges are: 0-5%, 5-10%, 10-15%, 15-20% and 20+%.

## GCP

GCP started offering spot instances under the name “Preemptible VMs” in 2015. These instances had a maximum running time of 24 hours. In 2021 they introduced “Spot VMs”, a new version of preemptible instances without the 24 hour limit [7].

Creating a spot instance in GCP is essentially the same as creating an on-demand instance. The user only needs to specify the *provisioning model* parameter to be *spot*, and optionally specify the termination action (analogous to the interruption behavior in AWS). Unlike AWS and Azure, GCP does not support specifying a maximum price for the instance [12].

The available termination actions are:

- Stop the instance. The disk is preserved and the user is still charged storage costs while the instance is stopped. An important difference with AWS is that GCP does not automatically restart a stopped spot instance [12]. It can be restarted by the user provided that the capacity is available.
- Delete the instance.

GCP states that the spot prices change up to once every 30 days [12]. The current prices can be seen on the website [16], but there is no historical data available. Interruption frequency is not provided by GCP either.

Kadupitiya et al. [28] empirically analyzed the interruption frequencies in GCP. However, the study was conducted on Preemptible VMs with the 24 hour running time limit. To the best of our knowledge, no study has been made on the interruption frequencies of GCP Spot VMs.

### 2.3.4 Spot instance groups

Spot instances can operate as standalone VMs or as part of an instance group. An example of such a group is an Azure Scale Set<sup>4</sup>, a managed group of VMs that allows autoscaling, that is, automatically adding VMs to the group in response to load increases, and removing them when the load decreases. It also supports using multiple zones to increase resiliency. It is possible to create a Scale Set of spot VMs. It is also possible to use the “Spot Priority Mix” feature, which allows to combine on-demand and spot instances in the same group.

A Managed Instance Group (MIG)<sup>5</sup> in GCP is similar to an Azure Scale Set. It is possible to create a MIG of Spot VMs, but it is not possible to use a combination of on-demand and spot instances. MIGs can be in a single zone or multiple zones within a region.

When using spot instances in this kind of groups, there is no guarantee that capacity is going to be available. This means that creating a new spot instance (either to replace an interrupted VM or to add a new one due to a load increase) could potentially take a long time. Using multiple zones can reduce the probability of mass interruptions and also increase the chances of replacing interrupted instances quickly.

While Azure and GCP allow to use spot instances within instance groups, AWS goes a step further with a dedicated product called Spot Fleet<sup>6</sup>, specifically designed to manage spot instance groups. In addition to the aforementioned features of autoscaling, using multiple zones, and the ability to combine on-demand and spot instances, Spot Fleet offers the advantage of using multiple instance types in the same group. It also offers enhanced control over instance selection with different allocation strategies. This allows the users to optimize for factors like cost (Spot Fleet will choose the instance types and AZs with lowest prices) or capacity (those with highest available capacity).

---

<sup>4</sup><https://azure.microsoft.com/en-us/products/virtual-machine-scale-sets/>

<sup>5</sup><https://cloud.google.com/compute/docs/instance-groups/>

<sup>6</sup><https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-fleet.html>

It is also worth mentioning that all three cloud providers support using spot instances as cluster nodes in their managed Kubernetes solutions (EKS, AKS and GKE, in AWS, Azure and GCP respectively).

### 2.3.5 Fault-tolerance techniques for spot instances

There are generally three strategies that can be used to deal with spot interruptions [31]:

- Migration: the application deployed on spot instances is migrated to other instances. This can be done proactively (by predicting when an interruption will happen and migrating before that) or reactively (migrating when the interruption notification is received).
- Checkpointing: the application state is periodically saved in persistent storage. After an interruption, the work is resumed from the last saved state instead of from the beginning.
- Replication: the application is deployed in multiple spot instances at the same time, and possibly some non-spot instances too. When a spot instance is interrupted, the application continues running on the remaining instances. This can be combined with a load balancer to distribute the traffic among the instances that are actually running.

Migration and checkpointing are suited for fault-tolerant and flexible applications, such as big data analytics, batch processing, image or media rendering, high performance computing, CI/CD pipelines and training machine learning models. These kinds of workloads usually tolerate delays, and the impact of a spot interruption might not be very high. This is what cloud providers generally recommend spot instances for [4].

Replication is suitable for applications that must always be available, such as web services for example. Nevertheless, using spot instances for these workloads is generally not recommended, since spot interruptions can lead to service downtime. Built-in features like Spot Fleet (AWS) and Spot Priority Mix (Azure) allow to easily combine spot and on-demand instances, such that the service is not completely disrupted upon spot interruptions. However, the quality of service may still be degraded since the surviving instances might not be able to handle all incoming traffic.

## 2.4 Serverless computing

### 2.4.1 Overview

Traditionally, cloud computing has been divided into three service models: IaaS, PaaS and SaaS. More recently, the *serverless computing* paradigm has emerged. Serverless allows developers to build and deploy applications without provisioning or managing backend infrastructure. The name does not mean that servers are not used, but that the servers are not visible to the developer.

Serverless has some overlap with PaaS, but it has a few defining characteristics [24]. First, there are no charges for idle resources: the user is only charged for the time the code is running. This is known as “scaling to zero” and is closer to the original “pay-as-you-go” promise of the cloud. Second, scaling is controlled by the provider – the user does not need to configure and fine-tune auto-scaling policies and usage metrics.

Function as a Service (FaaS) is a primary example of serverless computing [24]. In this model, a function is executed as response to a request or an event trigger. The function has a limited running time and is required to be stateless. Another example of serverless computing is Backend as a Service (BaaS). In this model, developers outsource common backend features – such as user authentication, geolocation, social media integration and push notifications – to a cloud service provider [38].

It is important to thoroughly understand the features and limitations of serverless, and carefully consider if it is the most appropriate solution for a given use case. Otherwise, serverless could result in higher costs and unexpected operational overhead [25, 29].

Eivy and Weinman [25] explored the economics of the serverless computing paradigm. First, they addressed the challenge of breaking down and understanding the complicated pricing model used by cloud providers. Then, they performed an example comparison where serverless was found to be three times more expensive than an IaaS solution. In Chapter 5, a similar approach will be used to compare the costs of the proposed system against a commercial FaaS solution.

### 2.4.2 Function as a Service (FaaS)

FaaS allows developers to execute code (more specifically, a function) in response to a request or an event. Other than the function code, everything else is controlled by the cloud provider, namely the hardware, operating system and server software [19].

FaaS offers automatic instant scaling (up and down), fully managed by the cloud provider. Moreover, it offers “scaling to zero”, meaning that when the function is not running there are no costs associated to idle resources. While this makes it a cost-effective way to run sporadic workloads, one associated issue is the so-called “cold start problem”: when a function has been scaled to zero, it takes longer to start the first time it is called again, since an execution environment needs to be provisioned first.

FaaS is well suited for independent and parallelizable tasks. Some use cases include data processing, web backends, file transcoding, image or video processing, and scientific simulation [19].

The popularity of FaaS lies on its ease of use: the developer focuses only on coding the function and the cloud provider handles everything else. However, this comes with a downside: vendor lock-in. Each cloud provider has their own conventions and it can be very difficult to migrate a FaaS project from one provider to another. However, there are some open-source projects such as Knative that aim to make FaaS portable, and allow it to run on hybrid or on-premises environments, as well as the public cloud.

FaaS is typically charged per function execution time and used resources. The charges go by the GB-s, which corresponds to 1 GB of memory running for 1 second. So, a function with 1 GB of memory running for 4 seconds costs the same as one with 2 GB running for 2 seconds.

FaaS is generally very cost-effective for low work volumes, but it can become much more expensive than a PaaS or an IaaS solution for high volumes [25]. The rest of this thesis will explore this issue and compare the costs of commercial FaaS against a custom platform built on top of IaaS spot instances.

## Hosted FaaS

All three main cloud providers have FaaS solutions: AWS Lambda<sup>7</sup>, Azure Functions<sup>8</sup> and Google Cloud Functions<sup>9</sup>. Each one has its differences, but some common features are:

- They support multiple common programming languages, including Go, JavaScript, Python, Java, among others.
- The functions can be triggered via an HTTP request or as a response to an event taking place in another cloud service, such as a database, a message queue or a storage service.

---

<sup>7</sup><https://aws.amazon.com/lambda/>

<sup>8</sup><https://azure.microsoft.com/en-us/products/functions>

<sup>9</sup><https://cloud.google.com/functions>

- The functions have a maximum running time. If a function exceeds that, an error is returned to the caller.
- Pricing: a function call is charged depending on how long it runs and the amount of resources provisioned for it.
- Free tier: there is a number of requests per month free of charge (1 million in AWS and Azure, 2 million in GCP). After that there is a small fee per million requests. There is also a monthly allowance of GB-s free of charge.

### Open-source FaaS

Open-source FaaS frameworks aim to overcome some of the downsides of their hosted counterparts, such as vendor lock-in and limited selection of programming languages [34]. Since they are not tied to any specific platform, users can run them in on-premises, hybrid or public cloud environments. The flexibility and portability, however, come with an increase in complexity, since the user is now responsible for managing the infrastructure where the framework will deploy the functions.

Some prominent projects in this regard are OpenFaaS<sup>10</sup>, Knative<sup>11</sup> and OpenWhisk<sup>12</sup>. They all run on Kubernetes, although OpenFaaS and OpenWhisk support other execution environments too. All these projects support deploying functions packaged as Docker containers, written in arbitrary programming languages. Moreover, OpenFaaS and OpenWhisk also allow to simply write function handlers, without the need of a Dockerfile, in a similar way as hosted platforms like AWS Lambda.

### Serverless containers

Cloud providers offer solutions for running containers that could be considered serverless. Some examples are AWS Fargate<sup>13</sup>, Azure Container Instances<sup>14</sup> and Google Cloud Run<sup>15</sup>.

Each of these has different characteristics, and they are not directly comparable to each other. Among them, Google Cloud Run is the only one that

---

<sup>10</sup><https://www.openfaas.com/>

<sup>11</sup><https://knative.dev/docs/>

<sup>12</sup><https://openwhisk.apache.org/>

<sup>13</sup><https://aws.amazon.com/fargate/>

<sup>14</sup><https://azure.microsoft.com/en-us/products/container-instances/>

<sup>15</sup><https://cloud.google.com/run/>

truly satisfies the attributes of serverless computing, that is, ease of use without the need to manage infrastructure, with auto-scaling fully managed by the cloud provider and with scaling to zero. Azure Container Instances fulfills the ease of use criterion, but does not auto-scale. AWS Fargate auto-scales, but not to zero, and the user still needs to provision computing infrastructure to run the containers (an ECS or EKS cluster).

It should be noted that open source FaaS solutions (such as OpenFaaS, Knative and OpenWhisk) also provide the capability to run containers. In fact, Google Cloud Run is built on top of Knative, and its pricing model is very similar to that of Google Cloud Functions. Therefore, this service can be thought of as an extension of the GCP hosted FaaS solution to support containers.

## 2.5 Related work

Most of the existing research on spot instances focuses on AWS [21, 23, 27, 33, 35, 39], perhaps in part because it is the largest cloud provider, and in part because other providers only started offering spot instances more recently. Moreover, as mentioned in Subsection 2.3.2, most of the previous research on spot instances became obsolete after the change of pricing model that AWS introduced in late 2017. Nevertheless, there is recent research worth mentioning.

Pham et al. [35] carried out an experimental evaluation of spot instances in AWS. For that, they issued 3,840 spot requests in three different regions, using multiple instance types. They ran different kinds of workloads in the instances (low CPU and memory, CPU intensive, memory intensive). They also tried different values for the “maximum price” parameter. The study resulted in multiple findings. A few worth highlighting are:

- No correlation was found between the waiting time and running time of the spot instances.
- Interruptions were not found to be related to the maximum price parameter nor to the workload running in the VM.
- Instances showed a minimum running time of 4 minutes before the interruption. Moreover, less than 10% of the instances were interrupted in the first 20 to 30 minutes.

Hao et al. [26] studied the VM startup times in AWS and GCP. They took into account multiple regions, operating systems and instance types, as

well as purchase models (on-demand and spot/preemptible). One of their findings was that the startup time for spot/preemptible instances does not present significant differences with that of on-demand instances.

Baughman et al. [23] explored the 2017 changes to the AWS spot pricing model. They analyzed the pricing history over a period of nine months encompassing the time before, during and after the changes. They found that price events happen more infrequently after the change; moreover, the mean and standard deviation of the prices greatly decreased. This confirms that the new pricing model is more stable. Irwin et al. [27] presented a comparison between the old and new pricing models, and analyzed the pros and cons of each of them.

There is also research about building applications on top of spot instances. Sampaio and Barbosa [39] presented an algorithm for scheduling scientific computing workflows, along with a system to deploy the tasks in containers that run on a “virtual cluster” consisting of spot and on-demand instances. The system includes a component that monitors the spot instances and launches replacements on interruption events. When an interruption happens, the container is migrated to another instance such that it can be restored and continue running. The authors argue that the container migration takes 30 to 90 seconds, so it can be achieved in the 2 minute warning period provided by AWS before an interruption.

Kadupitiya et al. [28] presented a scheduling algorithm for long-running batch jobs in scientific computing. They also performed an empirical analysis of the preemption rates of GCP preemptible VMs (that have a 24-hour running time limit) and leveraged the related findings to make decisions in the scheduling algorithm.

Ali-Eldin et al. [21] designed a platform to serve latency-sensitive web services using spot instances, seeking to maintain the service level objectives (SLO). A load balancer runs in an on-demand instance, behind which application servers are located as a mix of spot and on-demand instances. The design includes predictors for the prices and failures of the spot instances, in addition to the traffic the system will receive. An optimizer takes into account the output of the predictors and computes which instance types to deploy and on which purchase model (on-demand or spot). The system is designed to over-provision resources to a certain extent, to handle both spot interruptions and sudden increases in workload.

Okita et al. [33] propose a method for processing seismic images using spot instances. They implemented a checkpointing mechanism combined with other high performance computing techniques. They found that the overhead of the checkpoint process results in lower system performance for low interruption frequencies. In contrast, the performance significantly in-

creases for high interruption frequencies, since having checkpoints reduces the amount of work that need to be reprocessed.

Xu et al. [42] presented an in-memory storage system, where spot instances handle read-only queries, and on-demand instances handle write queries. Even though the paper assumes the previous AWS pricing model, it is still worth mentioning since it showcases the use of a replication strategy outside load-balanced web services.

## Chapter 3

# Design

This chapter presents the design of a system that leverages spot instances to run serverless functions. First, we discuss the motivation for the system, its requirements and some important assumptions. Then, we describe the system along with its different components.

### 3.1 Motivation

Researchers have designed different systems to leverage spot instances for real applications. Ali-Eldin et al. [21] proposed a system for running latency-sensitive web services, Okita et al. [33] designed a system for seismic image processing, and the works of Sampaio and Barbosa [39] and Kadupitiya et al. [28] focus on scientific computing workloads. On the commercial side, the company Spot.io<sup>1</sup> offers several products to help customers optimize cloud spending by using spot instances while maintaining availability. They use machine learning algorithms to predict interruptions and auto-replace instances. Further details about how this is achieved are not publicly available.

One direction that has not been explored to the best of our knowledge is serverless computing, more specifically serverless functions, on top of spot instances. The main selling point of commercial FaaS is the ease of use, but they suffer from several drawbacks such as: vendor lock-in, bounded execution time, limited selection of programming languages and limitations on the computational resources they can use (for example, they cannot use a GPU). Plus, serverless functions can be quite expensive at scale [25].

Therefore, there is an opportunity to leverage spot instances for running serverless functions, potentially achieving both higher flexibility and cost

---

<sup>1</sup><https://spot.io/products/>

reduction compared to hosted FaaS solutions. This is the objective of the following design.

## 3.2 Requirements

The following requirements have been identified for the system:

- The user launches a function by sending an HTTP request. Event triggers (launching a function when there is an event in another service such as a database, storage or message queue) are out of scope.
- The function is executed on a spot instance.
- The function can be written in any programming language. It is packed in a Docker image.

## 3.3 Preliminary considerations

There are a few design decisions to be made regarding the execution of serverless functions on spot instances. The first one is determining whether there will be one or multiple functions running concurrently on the same instance. In our case, we have opted to run only one function to ensure it can utilize the full resources of the VM.

The second decision is whether the spot instances will be reused for multiple function executions. In our case, we have chosen not to do that, and instead create a new spot instance for each execution (and delete it once the function completes). The reasons behind this choice are taken from the work of Pham et al. [35], which showed that spot instances in AWS had a minimum running time of 4 to 5 minutes, and less than 10% of the instances were interrupted in the first 20 to 30 minutes.

It should be noted that this data is only empirical and exclusively considers a limited number of AWS regions. Cloud providers do not officially guarantee any minimum running time. Further experiments would be desirable to increase the confidence in the findings of this study. Therefore, we do not assume that interruptions will not occur during the first few minutes. Instead, we assume interruptions will be infrequent as long as the lifetime of the spot instance is relatively short, hence our decision not to reuse the instances for multiple executions. If an interruption does happen, the request must be retried (see Subsection 3.4.4).

As a consequence of the above mentioned design decisions, every single request to our system has an overhead of tens of seconds, which is the time

required to create an instance for it. This makes our solution unsuitable for latency-sensitive workloads (such as web services), but on the other hand eases the complexity of handling spot instance interruptions.

## 3.4 System design

### 3.4.1 Components

On a high level, the system has two components:

- **Controller:** a web server that is always running, and that receives the HTTP requests from the users. It is deployed on an on-demand VM.
- **Worker:** a server that executes a function. It is deployed on a spot VM. A new spot VM is launched for each function execution, and is destroyed after the function finishes.

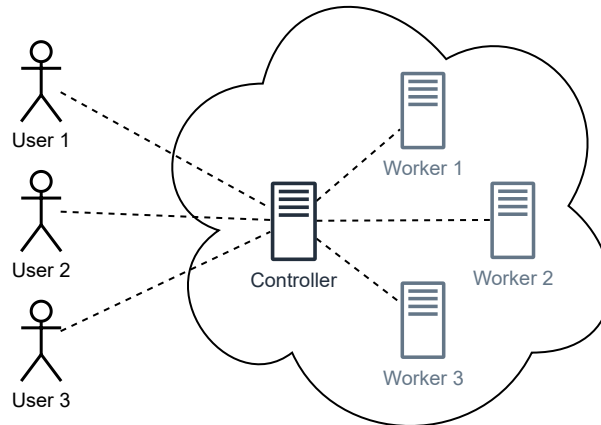


Figure 3.1: A Worker is created for each function execution.

The user-facing side of the Controller (the web server) works with a short polling mechanism. To execute a function, the user submits a POST request to the Controller, and in its body sends the values for the function parameters. When the controller receives this request, it starts processing it and sends back a request ID to the user. Subsequently, the user submits repeated GET requests with this ID until the function result is returned.

The reason for this choice is that the functions are expected to run for multiple minutes (more on this on Chapter 5). If we used a long polling strategy, the connection might close if the users are not careful to specify a long enough timeout, leaving them without a way to retrieve the function result afterwards.

To process a user request, the Controller first sends a request to the cloud provider to create a new spot VM where the Worker runs. When the VM launches and the Worker process starts, the Controller sends to it a “run job” request along the argument values to be passed to the function. With this, the Worker process launches a Docker container that contains the function, and passes the arguments to it. When the container exits, the Worker returns the function result to the Controller. Then, the Controller sends a request to the cloud provider to delete the Worker’s spot VM.

The full sequence of the workflow is shown in Figure 3.2.

### 3.4.2 User requests

The Controller’s web server has three endpoints the user can call:

Path	Method	Description
/	GET	Check if the server is running.
/job	POST	Submit a new job.
/job/:id	GET	Get the status of a given job.

The GET / request only returns HTTP status code 200. It is used as a probe to verify the web server is up and running.

The body of the POST /job request consists of the function arguments in JSON format. They are relayed as-is from the Controller to the Worker. The response to the POST request has the following JSON structure:

```
{
  "id": "string",
  "status": "string"
}
```

The response to the GET /job/:id request has the following JSON structure:

```
{
  "id": "string",
  "status": "string",
  "result": "string",
  "error": "string"
}
```

Possible values for the `status` field are: “Pending”, “In progress”, “Completed” and “Failed”. The `error` field is only present on “Failed” requests.

POST requests return HTTP status code 200. GET requests return this code too, except when the `status` is “Failed”. In this case the code can be 503 (service unavailable) if the failure was caused by a spot interruption, or 500 (internal server error) for an unknown failure.

### 3.4.3 Controller-Worker communication

There are two methods that the Controller can call in the Worker:

- Ping: to verify the Worker server is running.
- Run job: receives the arguments, executes the function and returns the result.

In principle, the Controller and Worker can communicate in any suitable way over the network. Options include plain HTTP requests (such as a REST API) or using a remote procedure call (RPC) framework such as gRPC.

The “ping” method does not receive or return any data. It is sufficient that the request succeeds (for example, returning a 200 HTTP status code). The “run job” method receives the arguments for the function as a JSON string, and returns the result of the function as an arbitrary string (it can be a JSON string but does not have to be).

### 3.4.4 Handling spot instance events

As mentioned in Section 2.3, spot instances present two trade-offs: they might not be always obtainable, and a running instance might be interrupted at any time with a short notice period. In the proposed design, both cases are handled via retry.

If a request to create a spot instance fails (meaning the cloud provider does not have capacity for it at the moment), the Controller can try again with a different availability zone, instance type or even a different region. This retry can be automatic, or alternatively, the Controller can return an error to the user, who then submits the request again with the same arguments.

For handling spot interruptions, the Worker process must monitor constantly to detect if an interruption signal is sent to the spot VM. In case this happens, an error is returned to the Controller. There are two options here as well: the Controller can retry automatically to request a new spot

instance to launch the function, or the Controller can return an error to the user, who then submits the request again with the same arguments.

For simplicity, we opted for returning an error to the users and have them retry the request, instead of making the Controller retry automatically. In either case, the Controller must keep track of the regions, availability zones and instance types it has tried, so that spot requests are not retried with the same combination that has already failed.

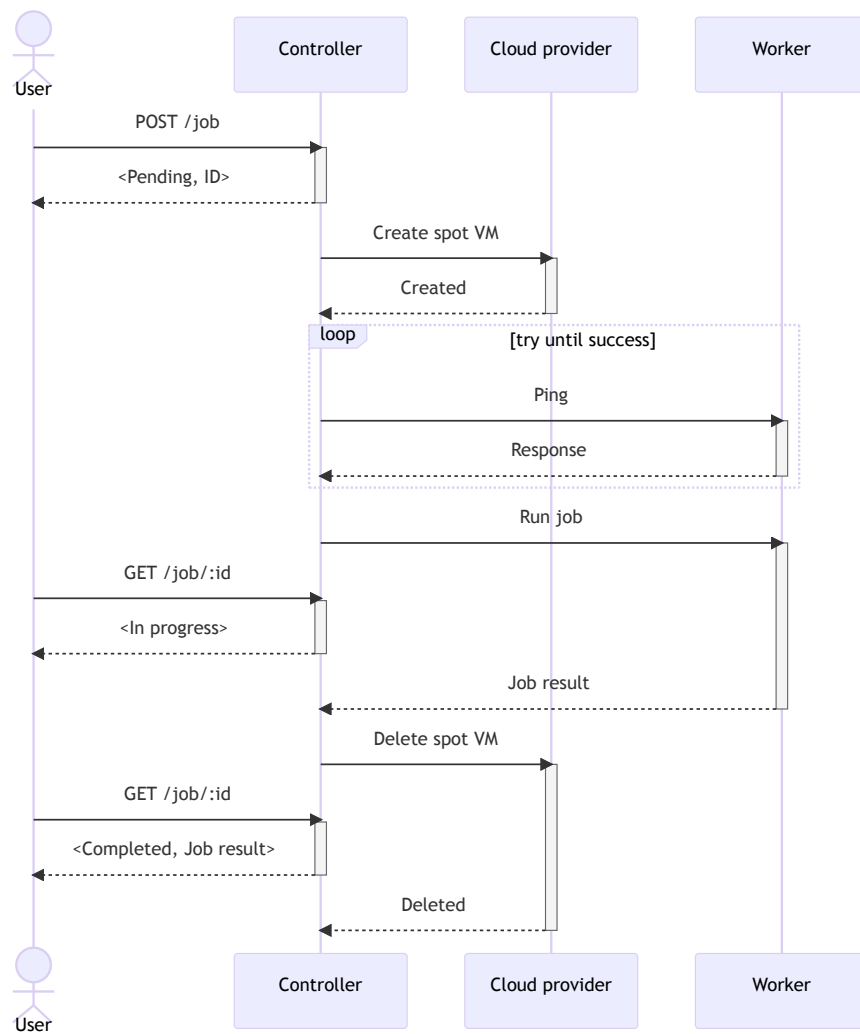


Figure 3.2: Full sequence of a function execution.

## Chapter 4

# Implementation

This chapter presents a proof of concept of the design introduced earlier, implemented using the Go programming language and Google Cloud Platform (GCP). The source code is available on GitHub<sup>1</sup>.

### 4.1 Code structure

For this proof of concept, we have chosen gRPC<sup>2</sup> for the Controller-Worker communication. gRPC is a mature, efficient and reliable framework for server-to-server communication. It uses Protocol Buffers<sup>3</sup> – a format for serializing structured data, supported on multiple programming languages – to send data between the participants.

The source code of the application is structured as a Go workspace with three modules: `controller`, `worker` and `proto`, the last one of which contains the Protocol Buffer definitions.

The Go compiler is used to produce the executable binaries of the `controller` and `worker`. These binaries are uploaded to Google Cloud Storage. The Controller and Worker VMs download them from there and directly execute them without the need of an interpreter.

### 4.2 Deployment considerations

A Virtual Private Cloud (VPC) in GCP is a virtual network that spans all regions [15]. The Controller and Worker VMs must be in the same VPC so

---

<sup>1</sup><https://github.com/jpvg10/spot-faas>

<sup>2</sup><https://grpc.io/>

<sup>3</sup><https://protobuf.dev/>

that they can communicate with each other. Moreover, the network must have firewall rules in place that allow traffic to the ports where the web and gRPC servers are running. In our case, the web server runs on port 8080 of the Controller, and the gRPC server runs on port 50051 of the Worker.

The multi-region nature of VPCs could be leveraged to deploy Workers in multiple regions. Doing so could result in cost savings (find regions with lower prices) and higher availability (try different regions to increase the probability of finding available capacity when needed), and the increase in latency due to longer geographical distances would be negligible since the system is handling requests that run in the order of minutes anyway. Nevertheless, for this proof of concept we have only employed the region *europa-north1*, located in Finland.

Both the Controller and the Worker instances need to be authorized to access different GCP resources. This is achieved by using a Service Account [8], a mechanism that enables GCP instances to access other GCP resources without needing secret keys or user credentials. Using the default Compute Engine service account is enough for our purposes. This service account is attached to the VMs on creation. In addition to the Service Account, it is necessary to specify the scopes the VM can access. Both VMs need the `storage-ro` scope to access Cloud Storage, where the executable binaries are stored, and additionally the Controller needs the `compute-rw` scope to be able to create and delete VMs.

### 4.3 Controller

The web server was implemented using the Gin framework<sup>4</sup>. When the server receives a POST `/job` request, it generates an ID for it and returns it to the user. The ID, along with the request data, are stored on an array in memory. This array is used to look up the request by its ID on GET `/job/:id` requests. The array is updated with the function result when the Worker sends it.

The image chosen for the Controller VM was the standard GCP image *ubuntu-minimal-2204-lts*. This image has the *gcloud* CLI tool pre-installed, which is needed for pulling the Controller binary from Cloud Storage, and for sending the requests to GCP to create and delete the Worker spot VMs.

---

<sup>4</sup><https://gin-gonic.com/>

## 4.4 Function container

The function is packed in a Docker container. The only requirements imposed to the code are:

- The arguments to the function are passed as a JSON string in an environment variable called *FN\_ARGS*
- The result of the function must be written to the *standard output*

Other than that, the user can choose any programming language and implement any code structure freely.

A sample function in Node.js looks like this:

```
const fnArgs = JSON.parse(process.env.FN_ARGS);
const { a, b } = fnArgs;
console.log(a + b);
```

The Dockerfile for this function is:

```
FROM node:lts-alpine
COPY index.js index.js
ENV FN_ARGS "{ \"a\": 0, \"b\": 0 }"
CMD ["node", "index.js"]
```

The Docker image must be stored in some registry, such as Docker Hub<sup>5</sup> or Google Artifact Registry<sup>6</sup>. In this proof of concept we used Docker Hub and set the image as “public”.

## 4.5 Worker

The Worker VM needs to have Docker installed. To avoid installing Docker every time and speed up the VM startup, we created a custom image with Packer<sup>7</sup>. It is based on the standard GCP *ubuntu-minimal-2204-lts* image and has Docker installed in it.

The Worker VM has a *startup script* (that is, a script that is executed automatically when the VM starts) that pulls the **worker** binary from Google Cloud Storage and executes it. With this, the Worker server starts automatically when the VM is created. For doing this, the VM needs the *gcloud* CLI tool. It is already installed since the VM image is based on the *ubuntu-minimal-2204-lts* image. The startup script is the following:

---

<sup>5</sup><https://hub.docker.com/>

<sup>6</sup><https://cloud.google.com/artifact-registry/>

<sup>7</sup><https://www.packer.io/>

```
mkdir /program
gcloud storage cp \\\
    gs://spot-thesis-files-2994/worker /program/worker
chmod +x /program/worker
/program/worker >> /program/log 2>&1
```

When the gRPC server receives a request, a Docker process is spawned. It uses the specified Docker image and passes the arguments as a JSON string in an environment variable called *FN\_ARGS*. An example of the command used to spawn the process looks like this:

```
docker run -e FN_ARGS="{\"a\": 5, \"b\": 3}" jpvalencia/sum-fn
```

The output of the Docker process, which is the result of the function, is captured by the Worker and returned as a string to the Controller.

To create and run the container, Docker must first pull the desired image (in this example, *jpvalencia/sum-fn*) from Docker Hub (or the registry where it is stored). This process may take some time, depending on how big the image is, adding to the time overhead that our system imposes to each function execution.

On spot interruptions, GCP first sends an “ACPI G2 Soft Off” signal to the VM [12]. If the instance has a *shutdown script*, it is executed when this signal is received. 30 seconds after the Soft Off signal, GCP sends an “ACPI G3 Mechanical Off” signal, shutting down the instance.

To handle the spot interruptions, the Worker VM has a shutdown script that finds the process ID (PID) of the Worker process and sends a *SIGTERM* signal to it. The process captures this signal and immediately sends a reply to the Controller with **status** “Failed”, an empty **result**, and an **error** saying “Worker interrupted”. The shutdown script is the following:

```
pid=$(pidof worker)
kill -s SIGTERM $pid
```

## Chapter 5

# Evaluation

This chapter presents an evaluation of the system implementation along different dimensions, including costs and performance. Specifically, we perform an analysis of the benefits and drawbacks of the proposed system.

### 5.1 Cost analysis

Next, we will carry out a cost comparison between the proposed system (implemented in GCP) and Google Cloud Run. As a reminder, that is the GCP service for running containers in a serverless fashion.

Since our system runs the functions as containers, we chose to compare against Cloud Run instead of Cloud Functions. However, the pricing model and the actual prices are the same for both services as of June 16, 2023, except for the free tier allowance.

#### 5.1.1 Assumptions

As explained in Section 3.3, our system has an overhead of tens of seconds in each request since we are creating a new spot instance for every function execution. Consequently, we assume that our system executes latency-insensitive jobs that potentially run for multiple minutes, and that can tolerate the overhead that our system incurs. For the following cost analysis, we assume that a job runs for two minutes. This is an arbitrary duration, but serves to emphasize that we are not dealing with high-traffic systems where requests are served in the order of milliseconds, and it also provides an easy way to map number of requests to total running time.

The resources provisioned for Cloud Run are 8 GB of memory and 2 vCPU. For our proposed system, we assume the instance type of the Con-

troller and Workers is the same: *e2-standard-2*. This instance type also has 8 GB of memory and 2 vCPU.

The GCP region will be *europa-north1*, which is located in Finland. All the reported prices are in US Dollars, taken from the GCP website on June 16, 2023<sup>1,2</sup>.

### 5.1.2 Google Cloud Run

Cloud Run pricing is divided between CPU and Memory:

- CPU: 0.000024 \$ / vCPU-second
- Memory: 0.0000025 \$ / GB-second

Furthermore, Cloud Run has a monthly free tier of 180,000 vCPU-seconds and 360,000 GB-seconds.

The number of requests also influences the pricing. The free tier includes 2 million requests per month. Beyond that, there is a charge of \$ 0.4 per million requests. This might seem like a small price, but it can add up very quickly for high-traffic services [25]. However, we do not take this factor into account since our analysis focuses on much lower request frequencies, as it will be apparent later.

An additional factor that affects the pricing is the egress traffic, that is, transferring data to the public internet or to resources in different GCP regions. We assume that the functions do not incur in these charges.

Under these assumptions, we can calculate how much our container will cost in a month depending on how many requests it receives. Remember that we assumed that one request takes two minutes. Let  $r$  be the number of requests received in a month. Let  $s$  be the number of seconds our container runs in a month:

$$s = r \cdot 2 \text{ minutes} \cdot \frac{60 \text{ seconds}}{1 \text{ minute}} = r \cdot 120 \text{ seconds}$$

Let  $c$  and  $m$  be the amount of CPU (vCPU-seconds) and memory (GB-seconds) consumed by our container in a month. Remember we assumed it has 2 vCPUs and 8 GB of memory:

$$\begin{aligned} c &= 2 \cdot s \\ m &= 8 \cdot s \end{aligned}$$

---

<sup>1</sup><https://cloud.google.com/run/pricing>

<sup>2</sup><https://cloud.google.com/compute/vm-instance-pricing>

Then, to calculate the total monthly cost, we subtract the CPU and memory free tier from  $c$  and  $m$  respectively, and multiply by the CPU and memory costs, taking into account that if the difference is negative (i.e. our usage is below the free tier) we take 0 instead.

$$\begin{aligned} \text{Total cost} = & \max(c - 180000, 0) \cdot 0.000024 \\ & + \max(m - 360000, 0) \cdot 0.0000025 \quad (5.1) \end{aligned}$$

Applying this formula to some sample values of  $r$ , we obtain the following costs per month:

<b>r</b>	<b>Total cost</b>
0	0
1,000	2.94
2,000	11.10
5,000	35.58
10,000	76.38
15,000	117.18
20,000	157.98

Table 5.1: Cost by number of requests on Google Cloud Run.

### 5.1.3 Proposed system

In our proposed solution, the Controller is an on-demand instance, and the Workers are spot instances. The instance type is *e2-standard-2*, which has 2 vCPUs and 8 GB of memory. The prices for this instance type are:

- On-demand price for one month: \$ 53.85648
- Spot price for one hour: \$ 0.024398

We assume that the Controller will be running for the whole month, therefore, our system has a monthly fixed cost of \$ 53.85648. The cost of the Workers will depend on how many requests our system receives.

Remember that we assumed that one request takes two minutes. Let  $r$  be the number of requests received in a month. Let  $h$  be the number of hours our container runs in a month:

$$h = r \cdot 2 \text{ minutes} \cdot \frac{1 \text{ hour}}{60 \text{ minutes}} = \frac{r}{30}$$

Then, the total cost of running the system for one month is calculated as the monthly price of the on-demand instance plus  $h$  times the hourly price of the spot instance:

$$\text{Total cost} = 53.85648 + (h \cdot 0.024398) \quad (5.2)$$

Applying this formula to some sample values of  $r$ , we obtain the following costs per month:

<b>r</b>	<b>Total cost</b>
0	53.86
1,000	54.67
2,000	55.48
5,000	57.92
10,000	61.99
15,000	66.06
20,000	70.12

Table 5.2: Cost by number of requests on our system.

#### 5.1.4 Comparison

Always keeping in mind we assumed that one request takes two minutes, we can see that for low values of  $r$ , Cloud Run is very cost-effective, and many times cheaper than our proposed system. Due to the use of the Controller VM, our system has a fixed monthly cost even when there are no requests. On the other hand, Cloud Run exhibits the “scaling to zero” behavior: if we do not get any requests, we do not pay anything.

However, Cloud Run costs start to add up quickly as the number of requests per month increases, as shown in Figure 5.1. It is clear that Cloud Run pricing scales much more steeply than our system. Beyond approximately 8,000 requests per month our system becomes cheaper than Cloud Run. At 20,000 requests per month, our system is already less than half the price of Cloud Run (i.e. 44% of the cost).

From Equations 5.1 and 5.2 we can calculate the exact number of requests per month where both solutions have the same cost. First, let us fully expand Equation 5.1 to express it in terms of  $r$ , and for simplicity assume we are

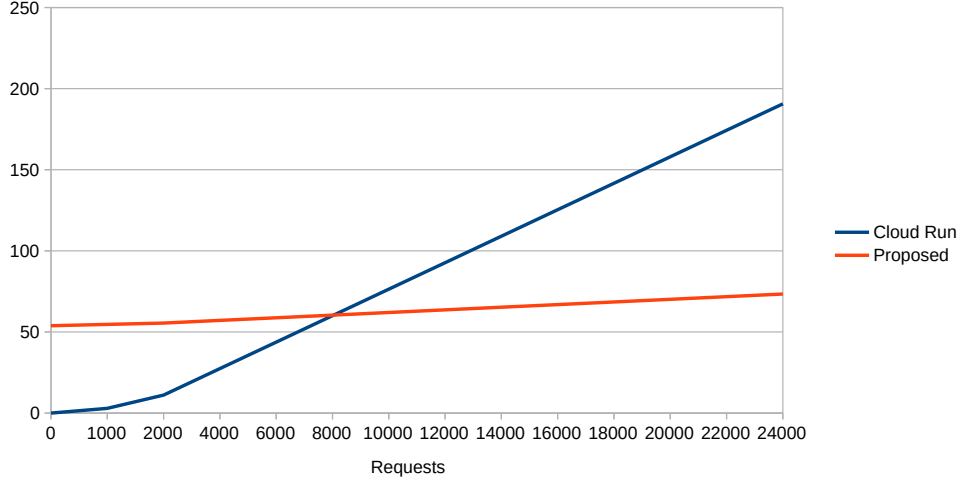


Figure 5.1: Price comparison between Cloud Run and our proposed system.

always above the free tier so that we can get rid of the max operator:

$$\begin{aligned}
 \text{Total cost} &= (c - 180000) \cdot 0.000024 + (m - 360000) \cdot 0.0000025 \\
 &= (2s - 180000) \cdot 0.000024 + (8s - 360000) \cdot 0.0000025 \\
 &= (240r - 180000) \cdot 0.000024 + (960r - 360000) \cdot 0.0000025
 \end{aligned} \tag{5.3}$$

Now, let us express Equation 5.2 in terms of  $r$  too:

$$\begin{aligned}
 \text{Total cost} &= 53.85648 + (h \cdot 0.024398) \\
 &= 53.85648 + \left( \frac{r}{30} \cdot 0.024398 \right)
 \end{aligned} \tag{5.4}$$

We can solve for  $r$  by setting Equations 5.3 and 5.4 equal:

$$\begin{aligned}
 53.85648 + \left( \frac{r}{30} \cdot 0.024398 \right) &= (240r - 180000) \cdot 0.000024 \\
 &\quad + (960r - 360000) \cdot 0.0000025
 \end{aligned}$$

$$53.85648 + 0.000813267r = 0.00576r - 4.32 + 0.0024r - 0.9$$

$$59.07648 = 0.007346733r$$

$$r = 8041.19$$

8,041 requests per month, with our assumption that one request takes two minutes, corresponds to around 268 hours, or around 11 days, of execution time. It averages to roughly 11 requests per hour, or 1 request every 5.5 minutes.

## 5.2 Performance

To test and evaluate the system performance, we issued 50 requests over the course of 3 days at different hours. The purpose was to measure the overhead introduced by each phase of the execution, and also to see if any spot interruptions occurred.

The function used for testing takes two numbers as arguments,  $a$  and  $b$ , and returns their sum. An artificial *sleep* of 2 minutes was added to comply with our assumption that the function takes this amount of time to complete.

The time events in the Controller are:

- $t_0$ : Receive POST request. Send a request to GCP to create a spot VM.
- $t_1$ : Spot VM created.
- $t_2$ : Worker server replies to “ping” request and is ready to execute the job. Send “run job” request to the Worker.
- $t_3$ : Receive job result.

Below are the statistics of the collected data, with the times calculated relative to  $t_0$ . All times are in seconds. To illustrate, a value of  $t_2 = 55$  would mean that the Worker server was ready to execute the job 55 seconds after the user sent the POST request to the system ( $t_0$ ).

Time	Average	Max	Min
$t_1$	10.1	24	8
$t_2$	54.86	66	50
$t_3$	187.6	199	181

Table 5.3: Times relative to  $t_0$ .

It is also worth looking at the times calculated relative to the previous event, e.g., a value of  $t_2 = 40$  would mean that the Worker server was ready to execute the job 40 seconds after the spot VM was created ( $t_1$ ). These statistics are presented in Table 5.4.

We can see that the creation of the spot instance ( $t_1$ ) takes on average 10 seconds. However, from Table 5.4 we observe that, after creation, it takes on average 45 seconds for the Worker to be fully ready to receive requests ( $t_2$ ). Therefore, the total average overhead of the spot creation is 55 seconds ( $t_2$  in Table 5.3).

Time	Average	Max	Min
$t_1$	10.1	24	8
$t_2$	44.76	48	37
$t_3$	132.74	135	129

Table 5.4: Times relative to  $t_{i-1}$ .

Another observation is that, even though the function has a *sleep* of 2 minutes, the function execution in the Worker takes 2 minutes and 13 seconds on average ( $t_3$  in Table 5.4). This can be explained by the time it takes to pull the container image from Docker Hub.

Taking into account the overhead of creating the spot instance and pulling the Docker image, the average time of the function execution, from the moment when the POST request is received to the moment when the function result is returned, is 3 minutes and 8 seconds ( $t_3$  in Table 5.3).

None of the the 50 function executions suffered from spot interruptions.

### 5.3 Benefits

The proposed system has multiple benefits. The first and most important is that, as it was already shown, after a certain number of requests per month it achieves large cost reductions compared to Google Cloud Run.

Furthermore, our system is very flexible, as the memory and CPU specifications of the Worker can be customized as needed. This is also possible in Google Cloud Run. However, our system can go one step further and use special instance types (for example, compute optimized or memory optimized instances), or instances with a GPU, things that are not possible with Cloud Run as of now. Also, our system could be extended to use multiple GCP regions, which is also not possible with Cloud Run.

In principle, our system does not have any limit on the function execution time, overcoming the limitation imposed by cloud providers. Though in Google Cloud Run the time limit is already high (60 minutes), in AWS Lambda, for example, the maximum execution time is 15 minutes. Having said this, very long-running functions can hinder the reliability assumptions of our system, as will be discussed further in the next section.

Another benefit of our system is that the function code inside the container is quite simple, since it reads the arguments from an environment variable and writes the result to the standard output. In comparison, in Cloud Run the Docker container receives directly the HTTP requests from

the users, so one must implement a web server in the container.

A final benefit of our system is that the handling of spot instances is made simpler than in the systems proposed by other researchers, for instance, Ali-Eldin et al. [21], Sampaio and Barbosa [39], and Kadupitiya et al. [28]. First, having a new instance for each function execution, we do not need to keep track of which instances are running a function and which are idle, or be concerned about the costs of idle spot instances. And second, handling the spot instance interruptions is done via retry, compared to the above-mentioned proposals which include models that attempt to predict the interruptions based on historical data.

## 5.4 Drawbacks

The most significant drawback of our system is the per-request overhead. As already explained in Section 3.3 and quantified in Section 5.2, creating a new spot instance for each function execution takes tens of seconds, which makes our system unsuitable for latency-sensitive workloads.

Another drawback is that initial setup is needed to deploy our system. Namely, one must create the on-demand VM, deploy the Controller program and set the correct GCP permissions so that it can create and delete the Worker spot instances. In comparison, there is little to no initial setup in Cloud Run.

Furthermore, our system does not solve the vendor lock-in problem. While the Docker container that holds the function is platform-independent, the Controller and Worker code is tied to the cloud provider, in this case GCP. It can be argued that Cloud Run is more portable than our system, since it is compatible with Knative.

An additional issue is that the Controller is a potential bottleneck. In our cost analysis we assumed for simplicity that it has the same computational resources as the Worker. However, it must be assessed what are the real computational needs of the Controller. Scaling vertically the resources of the Controller is a possible first solution. Another strategy that can help reduce the load on the Controller is to implement an asynchronous mechanism for the Controller-Worker communication. This frees up resources in the Controller, since it would not need to keep the connection with the Worker open while it is waiting for the function result. It would be important to test if this strategy actually increases the capacity of the Controller, especially since implementing the asynchronous communication with a mechanism such as a message queue would result in an additional cost for the system.

Another problem with the current implementation of the Controller is

that it is storing the job information (ID, status, function arguments, function result) in memory. A reboot of the Controller VM, or even a restart of the Controller process, will result in the loss of all the job information. This can be addressed by saving the information on persistent storage. An initial solution could be co-locating a database server in the Controller instance, saving the data to the persistent disk attached to the VM. This, however, would affect the performance of the Controller, possibly requiring an increase of its resources. An alternative would be to locate the database in an external server. It should be assessed which is the best solution, since both result in increased costs.

Finally there is the issue of the retries due to spot events. First of all, this adds complexity to the user experience, although it could be mitigated by having the Controller perform the retry automatically, hiding it from the user. Either way, retrying the request means that the previous work was lost, which translates in increased waiting time for the user to get the function result, and importantly, increased costs for the system. Without a proper understanding of the interruption frequencies, it is hard to estimate the potential cost burden of the retries.

We base our assumptions about the interruption frequencies on the study by Pham et al. [35], which found that spot instances in selected AWS regions had a minimum running time of 4 to 5 minutes, and less than 10% of the instances were interrupted in the first 20 to 30 minutes. However this is not enough, since it only considers AWS and it was conducted more than five years ago. These numbers might be completely different in other cloud providers, or may not hold at all (for instance, not having a minimum running time). An updated study on this topic would be valuable, even more if it includes other cloud providers such as Azure and GCP. Nevertheless, even if the assumptions still hold true, it means that our system has an upper limit of the maximum function execution time, because if we go beyond this time we might start running into frequent interruptions, having negative impacts both in costs and performance. This is a fundamental limitation of our design.

## Chapter 6

# Conclusion

This thesis explored the possibility of leveraging spot instances for running serverless workloads. We started by providing a comprehensive description of spot instances and their characteristics, drawing information from cloud provider documentation as well as from published research. This included a section dedicated to the pricing of spot instances, which is important because the availability of the spot instances has been closely tied to the spot price and the bids the users placed for the instances, but this is not the case anymore. Understanding this fundamental change allowed us to shift the research focus away from pricing. We also reviewed research about different kinds of applications built on top of spot instances, such as latency-sensitive web services [21], image processing [33], in-memory storage systems [42] and scientific computing workloads [28, 39].

Then, we introduced the concept of serverless computing and Function as a Service (FaaS). The main advantage of this model is the ease of use, since it allows the developers to focus only on the code, and lets the cloud provider handle the execution and scaling of the functions. This, however, comes with some drawbacks such as vendor lock-in and being subject to the restrictions imposed by cloud providers in terms of available programming languages, bounded execution time and computational resource limitations. In addition to this, serverless functions can become very expensive at scale.

Having established this background, we then presented the design for our proposed system. The objective was to leverage spot instances to run serverless functions, and potentially achieve cost savings and higher flexibility compared to commercial hosted FaaS solutions. The design consists of two main components: the Controller, which is deployed on an on-demand instance and receives requests from the users, and the Workers, which execute the serverless functions and run on spot instances.

The outlined design was then implemented as a proof of concept using

Go and GCP. This implementation was evaluated in terms of performance and costs, compared against Google Cloud Run, which is a service that offers a similar functionality as our system.

On one hand, our system was found to achieve significant cost savings after around 8,000 requests per month. For low request frequencies, our system is more expensive than Cloud Run due to the usage of an on-demand instance for the Controller, but, for example, at 20,000 requests per month, the cost of our system is less than half the cost of Google Cloud Run. On the other hand, our system introduces an overhead to every request since it creates a new spot instance for each function execution. For a function that runs for 2 minutes, the average execution time in our system was 3 minutes and 8 seconds. This makes our system unsuitable for latency-sensitive applications. The evaluation concluded with a detailed analysis of the benefits and drawbacks of the proposed solution.

Some possible directions of future work have been identified. First of all, it would be valuable to study in detail the interruption frequencies of spot instances in the major cloud providers, especially regarding the initial minutes of execution. This would benefit not only the current work but also the whole field of research on spot instances. For the proposed system, having better estimates for the interruption frequencies would allow factoring into the analysis the cost of retrying requests.

Regarding the proposed system, the main improvement would be designing and implementing a persistent storage solution for the job information, taking into account performance and costs. Another idea worth exploring would be using a message queue for the Controller-Worker communication, assessing if it helps to reduce the load in the Controller, and weighing the performance gains against the increased costs for the solution.

Finally, it would be important to assess what are the actual computational resource requirements of the Controller, especially when receiving multiple concurrent requests that imply maintaining multiple open connections to Worker servers for several minutes.

# Bibliography

- [1] 2023 state of the cloud report. <https://info.flexera.com/CM-REPORT-State-of-the-Cloud>. [Online; accessed 5-July-2023].
- [2] Azure retail prices overview. <https://learn.microsoft.com/en-us/rest/api/cost-management/retail-prices/azure-retail-prices>. [Online; accessed 5-July-2023].
- [3] Azure spot virtual machines pricing. <https://azure.microsoft.com/en-us/pricing/spot-advisor/>. [Online; accessed 5-July-2023].
- [4] Best practices for EC2 spot. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-best-practices.html>. [Online; accessed 5-July-2023].
- [5] Machine families resource and comparison guide. <https://cloud.google.com/compute/docs/machine-resource>. [Online; accessed 5-July-2023].
- [6] New Amazon EC2 spot pricing model: Simplified purchasing without bidding and fewer interruptions. <https://aws.amazon.com/blogs/compute/new-amazon-ec2-spot-pricing/>. [Online; accessed 5-July-2023].
- [7] Preemptible VM instances. <https://cloud.google.com/compute/docs/instances/preemptible>. [Online; accessed 5-July-2023].
- [8] Service accounts. <https://cloud.google.com/compute/docs/access/service-accounts>. [Online; accessed 5-July-2023].
- [9] Spot instance advisor. <https://aws.amazon.com/ec2/spot/instance-advisor/>. [Online; accessed 5-July-2023].
- [10] Spot instance pricing history. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-spot-instances-history.html>. [Online; accessed 5-July-2023].

- [11] Spot placement score. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-placement-score.html>. [Online; accessed 5-July-2023].
- [12] Spot VMs. <https://cloud.google.com/compute/docs/instances/spot>. [Online; accessed 5-July-2023].
- [13] State of cloud cost report 2022. <https://www.anodot.com/blog/state-of-cloud-cost-report/>. [Online; accessed 5-July-2023].
- [14] Use Azure spot virtual machines. <https://learn.microsoft.com/en-us/azure/virtual-machines/spot-vms>. [Online; accessed 5-July-2023].
- [15] Virtual Private Cloud (VPC). <https://cloud.google.com/vpc/>. [Online; accessed 5-July-2023].
- [16] VM instance pricing. <https://cloud.google.com/compute/vm-instance-pricing>. [Online; accessed 5-July-2023].
- [17] What is a private cloud? <https://www.techtarget.com/searchcloudcomputing/definition/private-cloud>. [Online; accessed 5-July-2023].
- [18] What is cloud computing? <https://www.ibm.com/topics/cloud-computing>. [Online; accessed 5-July-2023].
- [19] What is FaaS (Function-as-a-Service)? <https://www.ibm.com/topics/faas>. [Online; accessed 5-July-2023].
- [20] Work with spot instances. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-requests.html>. [Online; accessed 5-July-2023].
- [21] ALI-ELDIN, A., WESTIN, J., WANG, B., SHARMA, P., AND SHENOY, P. SpotWeb: Running latency-sensitive distributed web services on transient cloud servers. In *HPDC '19: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (2019), Association for Computing Machinery, pp. 1–12.
- [22] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I., AND ZAHARIA, M. A view of cloud computing. *Communications of the ACM* 53, 4 (Apr. 2010), 50–58.
- [23] BAUGHMAN, M., CATON, S., HAAS, C., CHARD, R., WOLSKI, R., FOSTER, I., AND CHARD, K. Deconstructing the 2017 changes to AWS

- spot market pricing. In *ScienceCloud '19: Proceedings of the 10th Workshop on Scientific Cloud Computing* (2019), Association for Computing Machinery, pp. 19–26.
- [24] CASTRO, P., ISHAKIAN, V., MUTHUSAMY, V., AND SLOMINSKI, A. The rise of serverless computing. *Communications of the ACM* 62, 12 (Nov. 2019), 44–54.
- [25] EIVY, A., AND WEINMAN, J. Be wary of the economics of “serverless” cloud computing. *IEEE Cloud Computing* 4, 2 (2017), 6–12.
- [26] HAO, J., JIANG, T., WANG, W., AND KIM, I. K. An empirical analysis of VM startup times in public IaaS clouds. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)* (2021), pp. 398–403.
- [27] IRWIN, D., SHENOY, P., AMBATI, P., SHARMA, P., SHASTRI, S., AND ALI-ELDIN, A. The price is (not) right: Reflections on pricing for transient cloud servers. In *2019 28th International Conference on Computer Communication and Networks (ICCCN)* (2019), pp. 1–9.
- [28] KADUPITIYA, J., JADHAO, V., AND SHARMA, P. SciSpot: Scientific computing on temporally constrained cloud preemptible VMs. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (Dec. 2022), 3575–3588.
- [29] KUHLENKAMP, J., WERNER, S., AND TAI, S. The ifs and buts of less is more: A serverless computing reality check. In *2020 IEEE International Conference on Cloud Engineering (IC2E)* (2020), pp. 154–161.
- [30] LEE, S., HWANG, J., AND LEE, K. SpotLake: Diverse spot instance dataset archive service. In *2022 IEEE International Symposium on Workload Characterization (IISWC)* (2022), pp. 242–255.
- [31] LIN, L., PAN, L., AND LIU, S. Methods for improving the availability of spot instances: A survey. *Computers in Industry* 141 (Oct. 2022).
- [32] MELL, P., AND GRANCE, T. The NIST definition of cloud computing, Sept. 2011.
- [33] OKITA, N. T., CAMARGO, A. W., RIBEIRO, J., COIMBRA, T. A., BENEDICTO, C., AND FACCIPIERI, J. H. High-performance computing strategies for seismic-imaging software on the cluster and cloud-computing environments. *Geophysical Prospecting* 70, 1 (Jan. 2022), 57–78.

- [34] OLEYNIKOV, V. Overview of self-hosted serverless frameworks for Kubernetes: OpenFaaS, Knative, OpenWhisk, Fission. <https://blog.palark.com/open-source-self-hosted-serverless-frameworks-for-kubernetes/>, 2022. [Online; accessed 5-July-2023].
- [35] PHAM, T.-P., RISTOV, S., AND FAHRINGER, T. Performance and behavior characterization of Amazon EC2 spot instances. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)* (2018), pp. 73–81.
- [36] RICHTER, F. Amazon, Microsoft & Google dominate cloud market. <https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/>, 2023. [Online; accessed 5-July-2023].
- [37] ROUSE, M. Cloud sprawl. <https://www.techopedia.com/definition/31646/cloud-sprawl>, 2017. [Online; accessed 5-July-2023].
- [38] ROUSE, M. Backend as a service. <https://www.techopedia.com/definition/29428/backend-as-a-service-baas>, 2022. [Online; accessed 5-July-2023].
- [39] SAMPAIO, A. M., AND BARBOSA, J. G. Constructing reliable computing environments on top of Amazon EC2 spot instances. *Algorithms* 13, 8 (Aug. 2020).
- [40] STORMENT, J., AND FULLER, M. *Cloud FinOps, 2nd Edition*. O’Reilly Media, Inc., 2023.
- [41] WANG, S., AND CASADO, M. The cost of cloud, a trillion dollar paradox. <https://a16z.com/2021/05/27/cost-of-cloud-paradox-market-cap-cloud-lifecycle-scale-growth-repatriation-optimization/>, 2021. [Online; accessed 5-July-2023].
- [42] XU, Z., STEWART, C., DENG, N., AND WANG, X. Blending on-demand and spot instances to lower costs for in-memory storage. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications* (2016), pp. 1–9.