THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

A Comprehensive Analysis of Unresolved Issues and Resource Management in Serverless Computing Environments

SAKSHAM ARORA
SUMMER 2021

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Computer Science
with honors in Computer Science

Reviewed and approved* by the following:

Bhuvan Urgaonkar
Associate Professor in the Department of Computer Science and Engineering
Thesis Supervisor

Danfeng Zhang
Assistant Professor in the Department of Computer Science and Engineering
Honors Adviser

*Electronic Approvals are on file

ABSTRACT

Serverless computing has arisen as a promising new paradigm for deploying applications and services. It is a testament to the sophistication and the widespread proliferation of cloud computing, as it reflects the advancement of cloud programming frameworks and platforms. However, like any other new technology, it has its own set of challenges that clients, both new and old, are grappling with. In this thesis, we would take a closer look at the unresolved issues in serverless computing, consider the resource management problems that would be raised in the near future corresponding to monetization of data compute locality, and identify important research in the area focused on addressing the aforementioned challenge(s), with the goal of convincing the user why the study is beneficial to them.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This thesis is dedicated to those who have supported me in any capacity along my academic journey.

I extend my utmost gratitude to my thesis supervisor, Dr. Bhuvan Urgaonkar, for his patience, kindness, direction, and assistance through each step of my research analysis and writing process.

I would also like to thank my honors advisor, Dr. Danfeng Zhang, for his constant support and guidance throughout my time at Penn State. I would like to thank him for assisting me to pave my way through college.

I would like to thank my friends for their encouragement and support, without which I would be completely lost.

Lastly, I want to express my gratitude to my parents for always being there for me, for making untold sacrifices to provide a better life for me, and for inspiring me to constantly strive for excellence. Thank you so much!

CHAPTER 1
INTRODUCTION

1.1 Motivation

Serverless computing is becoming an increasingly popular cloud service due to its high elasticity and fine-grained pricing. Serverless platforms, such as AWS Lambda, Google Cloud Functions, and Azure Functions, enable users, as opposed to virtual machines, to quickly launch hundreds or thousands of lightweight tasks [16]. In serverless computing, the user is in charge of writing and packaging the code, while the cloud provider is responsible for provisioning and maintaining the infrastructure required to execute the code. The cloud provider schedules user tasks onto physical resources with the promise of dynamically scaling the tasks based on application demands and charging customers solely for the resources their tasks utilize, at millisecond granularity [17]. The current pay-per-use pricing for functions is around $0.20 per million invocations, or per AWS Lambda, making serverless a lucrative and compelling choice for end-users [30].

The novelty of serverless computing results in a steep learning curve. We will look at what serverless has to offer and what distinguishes it from other cloud service models in this thesis. Serverless, like any other new technology, has its own set of challenges that clients, both new and old, are grappling with. For instance, the exchange of intermediate data between execution stages is a significant challenge because direct communication between serverless tasks is difficult owing to its statelessness. Many cloud providers have attempted to

provide ephemeral storage solutions for saving the intermediate states in serverless, however, these solutions are either prohibitively expensive or extremely inefficient. Furthermore, transitioning from the traditional approach to serverless applications impacts the way testing and debugging are performed by developers, not to mention the cold start delays, limited lifetime, and limited user control.

Likewise, resource management issues are quite frequent with the serverless approach. The process of provisioning and allocating the appropriate amount of compute resources for an application, as well as scheduling the constituent components of that application on the specified resources is referred to as resource management in computing. Furthermore, cloud providers are required to carry out the entire resource management process while meeting the Quality of Service (QoS) goals established by the various parties involved [7]. Effective management of computing resources involves automating the entire process of resource provisioning, resource allocation, resource scheduling, resource monitoring, and scaling. The foregoing has piqued the interest of many researchers, necessitating a specialized focus on resource management under the serverless model.

Efficient resource management may be a difficult undertaking for developers, especially if they have never dealt with serverless previously. Many preliminary studies on serverless computing have identified the potential of this new computing platform, as well as the inherent challenges with the concept and ideas for addressing the said drawbacks. Certain studies have also focused on evaluating the performance of existing commercial and open-source serverless platforms to provide the user a perspective on which platform to adopt. This thesis focuses on the resource management problems that users may encounter when deploying their application on serverless, then identifies important research in the area

focused on addressing the aforementioned challenge(s), with the goal of convincing the user why the study is beneficial to them.

For many people, serverless is their first exposure to the public cloud. It has been observed that most of the time, new users settle with the norms while working with new technologies owing to the lack of available platforms and to avoid wasting too much time and resources. Therefore, I reasoned that identifying some of the key problems faced by developers, based on surveys conducted in various studies, and incorporating previous research work focused on assisting developers, both novices and professionals, might benefit them greatly. It would be a significant accomplishment if this thesis could serve as a document that users can refer to obtain a conceptual foundation on serverless architecture and to formulate research questions, hypotheses, or both.

1.2 Thesis Outline

The remainder of the thesis is organized as follows. Chapter 2 introduces serverless computing and provides a brief background on the model and its key features that distinguish serverless from other cloud service models. It also provides an overview of the application domains of this computing model, as well as some of the main drawbacks that providers and customers encounter when working with serverless computing, which leads to the motivation for this thesis. Chapter 3 discusses virtual machines and containers, in addition to serverless computing. It provides a comparison between the three technologies and highlights the advancement of the technologies. Chapter 4 focuses on resource management under the serverless architecture and identifies common resource management problems. In addition, this chapter offers an overview of research works conducted previously, focused on

addressing some of the aforementioned key challenges associated with resource management. Finally, chapter 5 summarizes the thesis and offers suggestions for future research.

CHAPTER 2

BACKGROUND INFORMATION

2.1 Serverless Computing

Serverless computing refers to the practice of delivering backend resources and services on an as-use basis [4]. The phrase "serverless" does not indicate "no servers"; it merely indicates that it requires zero configuration or maintenance from the developer. Users can write and execute code without having to worry about implementing the server or the underlying infrastructure when working with a serverless provider. Otherwise stated, it is a cloud computing execution model in which the vendor is in charge of allocating the machine resources to the users depending on their needs and managing the servers on behalf of their customers [11]. Certain services, such as AWS Lambda, Google Cloud Functions, and Azure Functions, allow customers to run their back-end code across several data centers without having to worry about picking the operating system, configuring the network, patching dependencies, or capacity provisioning.

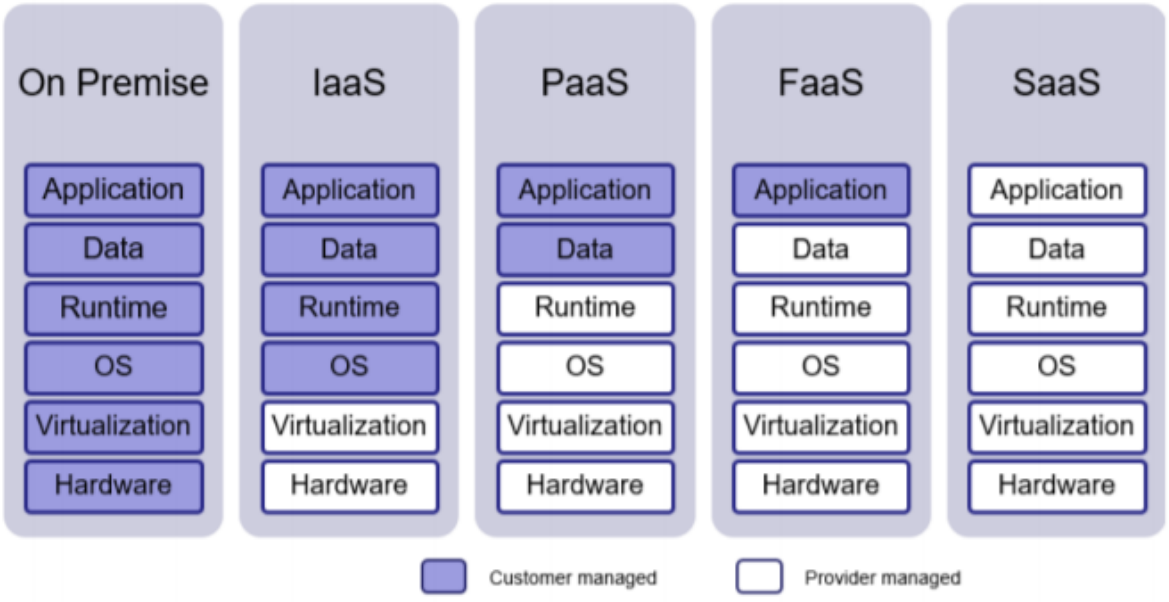

Figure 2.1: Evolution of cloud service models (On Premise, IaaS, PaaS, FaaS, SaaS), adapted from [7]

Serverless computing, as stated earlier, is an application service model in which the provider manages the server and handles all the responsibilities associated with the execution of the application throughout its lifetime. It is a rapidly growing industry that enables developers to rely on internet-based services, rather than depending on the on-premise IT infrastructure. Various cloud-based providers have begun to offer software, storage, and processing services to the clients at low costs, with minimal involvement of the user [11]. The way server infrastructure is managed for use by customers has evolved significantly over the years, starting with bare-metal servers that are maintained on-premise at redundancy. Now, the most significant change in this regard occurred around the year 2000, when the concept of cloud computing was introduced. Along with the advent of cloud computing came three different cloud service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [7].

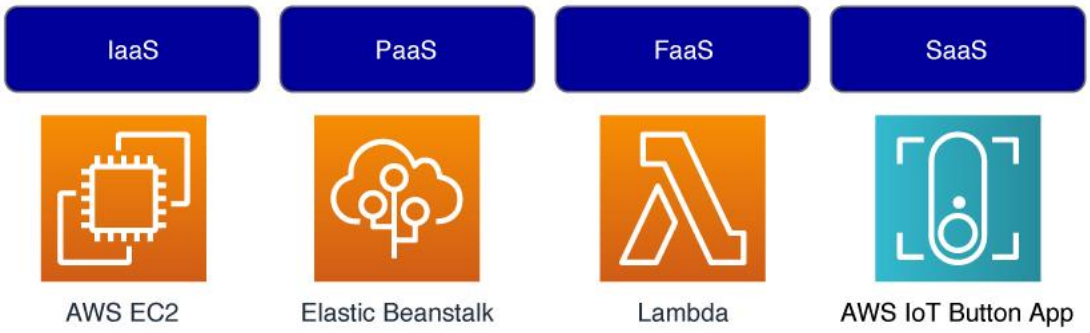

Figure 2.2: Example of AWS services offered under different cloud service models: IaaS, PaaS, FaaS, and SaaS, adapted from [23]

Infrastructure as a Service (IaaS) is a cloud computing model in which the provider rents or leases the server to the user in the cloud for compute and storage. Under the IaaS model, the provider manages hardware resources at their data centers. The users can run any OS or applications on the servers rented by the provider without having to worry about the maintenance and the operating costs of the servers [22]. The users are charged on a pay-per-use basis, which means they are charged based on how much storage or processing power they utilize over a certain time span. Amazon EC2 [Figure 2.2], Google Cloud Platform, and Azure VMs are examples of IaaS services available today.

Platform as a Service (PaaS) is a cloud computing model in which the cloud provider provides customers with the hardware and software tools they need for application development over the internet. PaaS is halfway between Infrastructure as a Service (IaaS) and Software as a Service (SaaS). Users have access to a cloud-based environment in which they can build and deliver customized applications without having to install and work with IDEs. Users may simply purchase the resources they require on a pay-per-use basis from the cloud service provider and access those services with ease over a secure internet connection [22].

PaaS providers in today's market provide the customers with applications such as AWS Elastic Beanstalk [Figure 2.2], Azure App Services, Google App Engine, and Apache Stratos.

Function as a Service (FaaS) is a serverless way in which modular pieces of code are executed on the edge. Under the FaaS model, the developers are responsible for writing and modifying a piece of code on the fly, which may then be executed in response to an event, such as user interactions, messaging events, or database changes. The functions employed in serverless environments are known to be entirely stateless meaning that they can be scaled independently, which is why serverless computing is also sometimes referred to as function-as-a-service. The serverless model differs from other execution models in that it offers more granularity in terms of application scaling and billing schemes [21]. We know that the FaaS model is inherently scalable, thus developers do not have to worry about developing contingencies for high traffic scenarios. In such a case, the scaling concerns are handled by the serverless provider. FaaS providers, unlike traditional cloud providers, do not charge their users for idle computation time [4]. This means that instead of wasting money on over-provisioning cloud resources, the users only pay for the computation time they utilize. AWS Lambda [Figure 2.2], Google Cloud Functions, IBM Cloud Functions, Microsoft Azure Functions, and OpenFaaS are examples of FaaS providers available in the market currently.

Finally, Software as a Service (SaaS) provides the clients with applications that are accessed over the web and are managed solely by the software provider. The SaaS model relieves the users of any responsibilities associated with software maintenance, infrastructure management, network security, data availability, and other operational issues that might prevent the applications from running smoothly. Users are billed based on a variety of factors under the SaaS model, including the number of users, usage time, the amount of data stored, and the number of transactions processed [22]. Users may simply log in and utilize the

applications, which are hosted entirely on the provider's infrastructure. Current applications for SaaS include Salesforce, Dropbox, Google Workspace, Field Service Solutions, System Monitoring Solutions, Schedulers, and more. Figure 2.1, which is adapted from [7], illustrates the evolution of cloud service models, from on-premise servers to Software as a Service (SaaS), and distinguishes between the aspects managed by the customer and the provider under each service model.

2.2 Key Characteristics of Serverless Computing Platforms

Existing serverless platforms have several distinguishing characteristics that set them apart from other cloud computing platforms. These are the five most prevalent characteristics of a serverless service. It does not necessitate the user managing the server hosts or processes. There is load-based auto-scaling and auto-provisioning, which means that as requests come in, a serverless application scales to meet the demand [11]. The price is determined by how much the users use the services [20]. The users pay for the resources they consume as opposed to the provisioned capacity. Other than host size or count, performance capabilities are described in a variety of ways. Finally, the services have a high level of implicit availability. Each of these characteristics contributes to a cloud infrastructure that takes less engineering effort to design, construct, and maintain [3]. Figure 2.4 highlights the benefits of serverless architecture based on the survey [27], conducted in the year 2020. According to the survey, rapid auto-scaling, or scaling flexibility was voted the most popular because it has been the selling point of serverless computing from the beginning. This was followed by speed of development and decreased system administration, which was expected given that customers are not required to deploy any servers or manage any hardware or software,

making the entire process of deploying an application considerably easier. Pricing was also voted as a popular choice in the survey, considering the fine-grain pricing model offered by serverless.

- *Rapid Auto-scaling*: The platform may scale up or down the computing resources allocated to any application based on the application's requirements at any given time. It guarantees that an application's capability to scale resources to meet demand is always automated and immediate. When an application requires more server resources to handle a high volume of page requests or processing processes, auto-scaling comes in handy. Auto-scaling provides long-term advantages. However, it is best suited for dealing with unexpected surges of traffic [3]. All serverless platforms are equipped with container technologies that enable them to deliver thousands of instances in the blink of an eye, with minimum start-up delays. When there is no traffic to an application, the function instances scale to zero, keeping idle resources to a minimum.
- *Pricing*: The use of serverless services is known to be metered, and users must only have to pay for the time and resources they utilize while the serverless functions are running. To put it another way, pricing is done on a per-request basis [8]. This means that when a function scales to zero and no node is executing the user's code, the user incurs no cost. And the serverless platforms' ability to scale to zero instances is a distinguishing characteristic. The metered resources, such as memory or CPU, as well as the pricing model and off-peak discounts, vary depending on the providers. For certain workloads, such as those requiring parallel processing, serverless computing can be both quicker and much more cost-effective than other forms of compute. For example, a lambda function with 512 MB of memory that executes 3 million times per month and runs for 1 second each time is estimated to cost a little over $18 per month,

whereas, two EC2 t2.nano instances behind an AWS load balancer will cost the user about $28 per month [6].

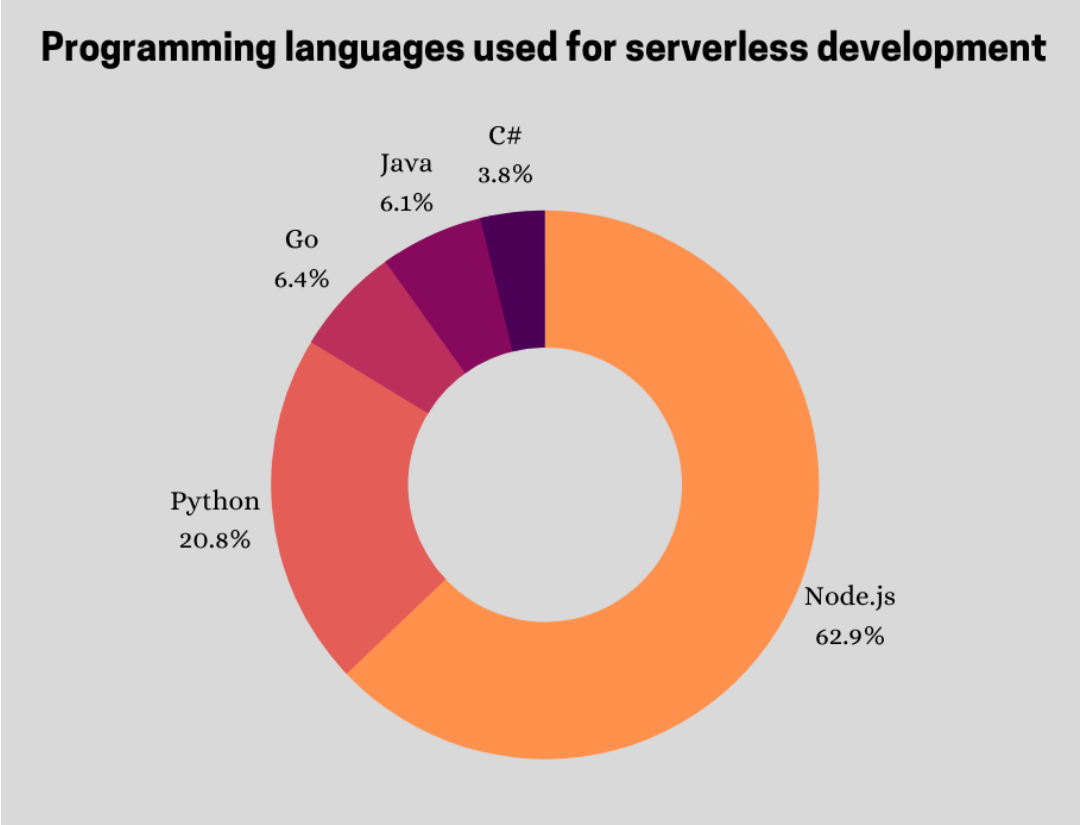

Figure 2.3: Programming languages used for serverless development according to the 2018 survey [2]

- *Programming Languages*: Most serverless platforms support a wide variety of programming languages which include C#, Go, JavaScript, Java, Python, and Swift. Furthermore, some platforms provide support for extensibility mechanisms for code written in any language, as long as it is packaged in a Docker image that supports a well-defined API [8]. AWS Lambda functions, for example, support Java, Go, PowerShell, Node.js, JavaScript, C#, Python, and Ruby. Google Cloud Functions, on the other hand, only supports Node.js, JavaScript, Python, and Go, but the functions have an infinite execution time. Microsoft Azure Functions offers similar pricing and

supports more languages than Lambda, including Bash, Batch, C#, F#, Java, JavaScript (Node.js), PHP, PowerShell, Python, and TypeScript. Lastly, Google Cloud Run, which was recently announced, broadens the language compatibility even further by allowing any language that can be run in a container, whereas AWS Lambda Layers allows developers to deploy code that is written in other languages [36]. Figure 2.3 provides a comparative analysis of the programming languages used for serverless development based on the survey [2], conducted in the year 2018. Node.js was the most preferred programming language for serverless development, according to the survey.

- *Security and Accounting*: In serverless platforms, customers are co-hosted on the same physical or virtual hardware, implying that a single instance of software provides a service to numerous customers or tenants. Because serverless platforms are multi-tenant, the providers must ensure to isolate the execution of various users' operations [14]. Container technologies take advantage of Linux kernel features like namespaces and cgroups to enable resource isolation for separate function executions [7].
- *Performance and Limits*: A serverless code's runtime resource requirements are limited in a variety of ways, which include the number of concurrent requests, maximum memory and CPU resources available to a function invocation, and an upper bound in execution time before a function instance times out. Some restrictions, such as the concurrent request threshold, can be adjusted as users' demands rise, whilst others, such as the maximum memory capacity, are built into the platforms [8].

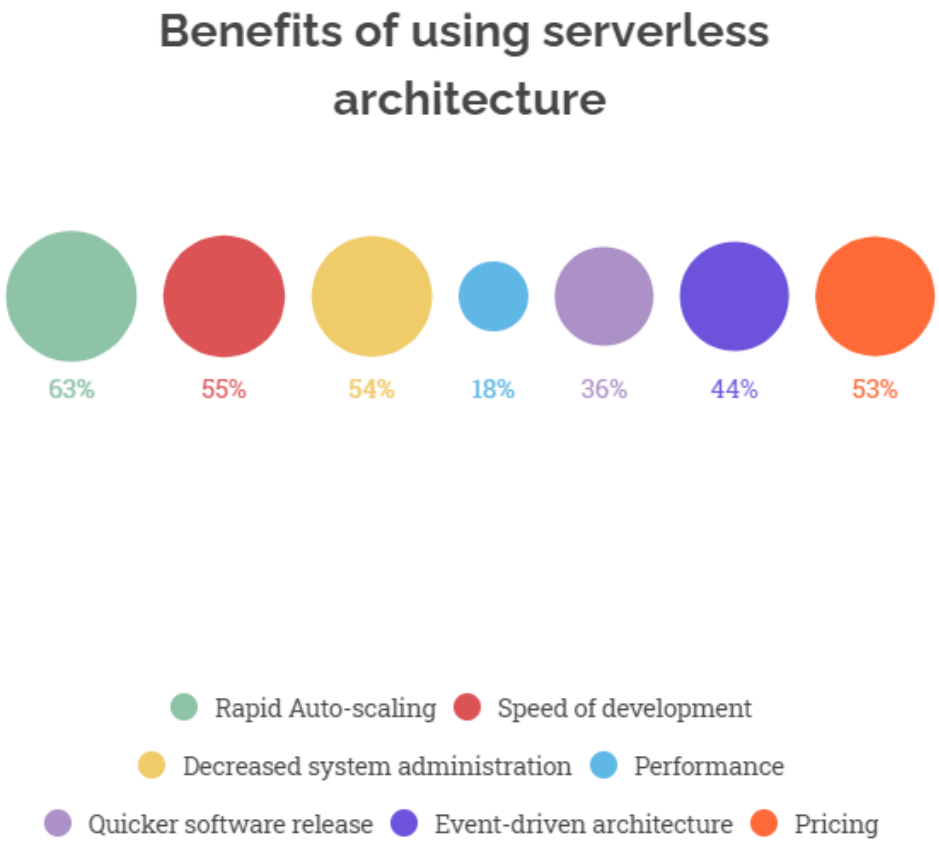

Figure 2.4: Benefits of using serverless architecture, adapted from the 2020 Coding Sans survey [27]

2.3 Application Domains of Serverless Computing

Generally speaking, stateless and ephemeral workloads that are bursty and compute-intensive might benefit more from a serverless architecture. Serverless computing is increasingly being explored for use in different fields and application domains such as web services, big data, internet of things, machine learning model training, and for large-scale mathematical computations [10]. Figure 2.5, which is adapted from [32], depicts the distribution of the domain of the surveyed use cases for serverless. Unsurprisingly,

webservices is the most common application domain in the survey, followed by data processing and mathematical and scientific computation.

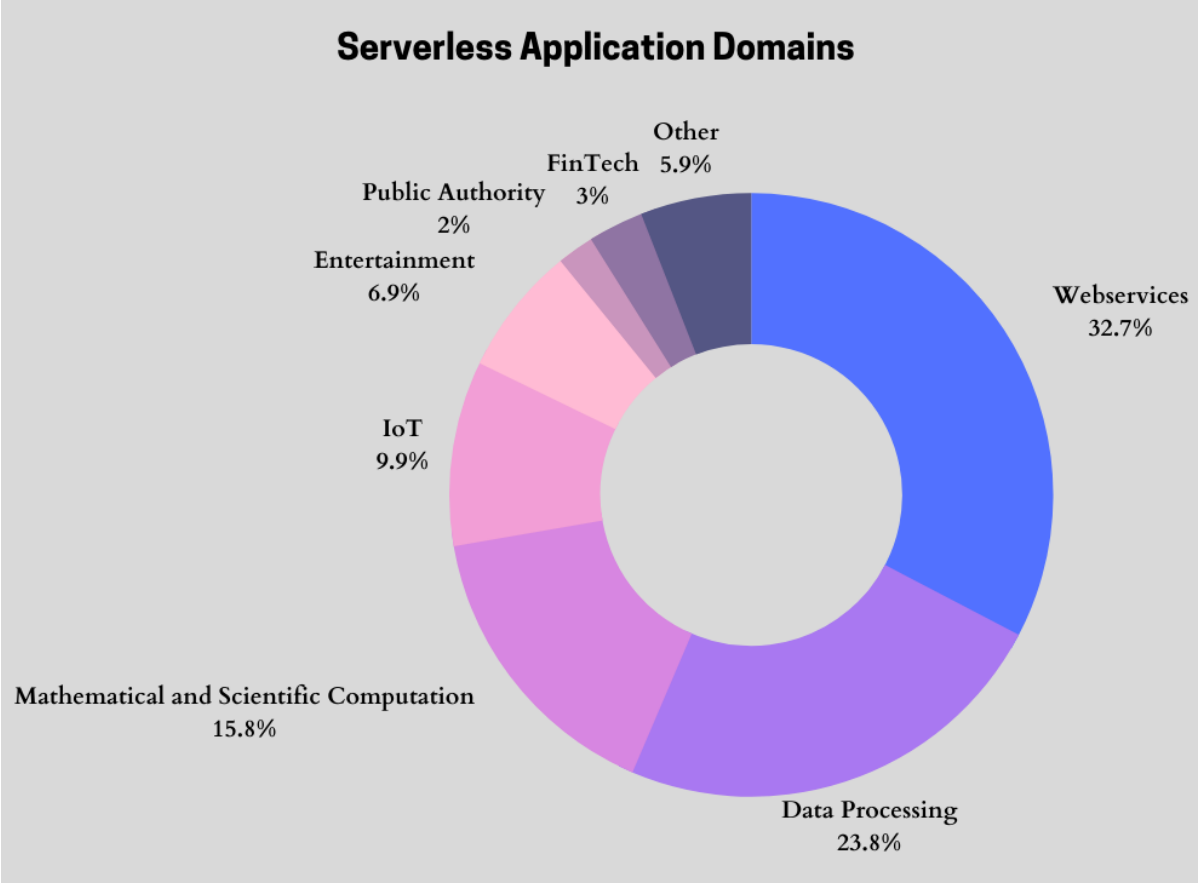

Figure 2.5: Distribution of the serverless application domains, adapted from the 2021 survey [32]

- *Web Services*: Serverless computing is said to have been developed for lightweight use cases such as delivering APIs or for providing minimal backend services [7]. According to several studies, webservices is still the most popular application domain that uses the serverless model due to its advantages.
- *Big Data Analytics*: Big data processing has traditionally been done via clusters of machines, with jobs consisting of many tasks being executed across a group of machines due to the enormous data volumes and computational needs. The major

cloud providers have recently recognized the potential of serverless computing for big data analytics [32]. AWS, for instance, now provides guidance on how to get the most out of Lambda for streaming data analytics. Moreover, IBM released IBM-PyWren two years ago to assist with data analytics utilizing IBM Cloud Functions [7]. Furthermore, a number of studies are being carried out in order to make the serverless model more advantageous and easier to implement for big data applications.

- *Mathematical Computation*: Formerly, all large-scale mathematical computations were performed on supercomputers or high-performance computing clusters connected via high-speed, low-latency networks to reduce time consumption. Given the preceding argument, it appears that serverless might not be the right approach for such applications [20]. However, when it comes to executing mathematical computations, serverless provides an advantage to the customers. Non-computer scientists can harness the power of serverless runtimes for large-scale optimization without having to worry about the infrastructure and scalability to accommodate the varying demands of resource parallelism during a computation because serverless handles it for them. Several pieces of research show that serverless computing could be a good fit for mathematical computations when computation time outweighs communication delays. Unfortunately, the high latency of external storage causes limitations for smaller problem sizes, which may pose issues for users [7].
- *Internet of Things*: Serverless computing has been used in a variety of IoT domains, including home automation and other custom-built IoT solutions, since IoT applications can be deployed as a number of lightweight functions, reducing limitations at edge devices [7, 32]. Furthermore, the serverless functions' statelessness adds to the simplicity of deploying IoT applications by allowing parts of applications

to be moved across the edge/cloud computing network without causing too many problems.

- *Machine Learning*: Serverless computing appears to be a viable method for resource provisioning issues for machine learning users due to its simplified deployment options, fine-grained resource provisioning and billing models, and the model's ability to autoscale both compute and storage resources. However, there are certain drawbacks and challenges that users face while using the serverless approach for machine learning training models at various stages [20]. Due to the limited memory and storage options, lack of effective communication between functions or fast shared storage, unpredictable launch times, and the short runtime of lambda functions, serverless platforms become less suitable for machine learning processing [7]. There is currently research being conducted to propose serverless frameworks with potential improvements to help overcome the limitations described above and make serverless feasible and beneficial for machine learning applications.

2.4 Downsides of Serverless Computing

Serverless computing provides a dynamic cloud computing execution model in which the server is run by the cloud provider. We explored several distinguishing characteristics that make serverless platforms distinct from the traditional cloud computing platforms in the preceding sections. As we know, serverless computing is a relatively new technology, and its novelty results in a steep learning curve. Even while serverless computing has a lot to offer, it also comes with certain hidden challenges and downsides. The most common challenge associated with serverless architecture is the user’s lack of control over the application and

the control being transferred to the FaaS providers and the third-party SaaS vendors. The users' lack of control makes it harder for them to troubleshoot performance issues that may emerge during the execution of the application. Not only that, but it also makes it nearly impossible to estimate expenses accurately, which ends up causing problems for providers [25].

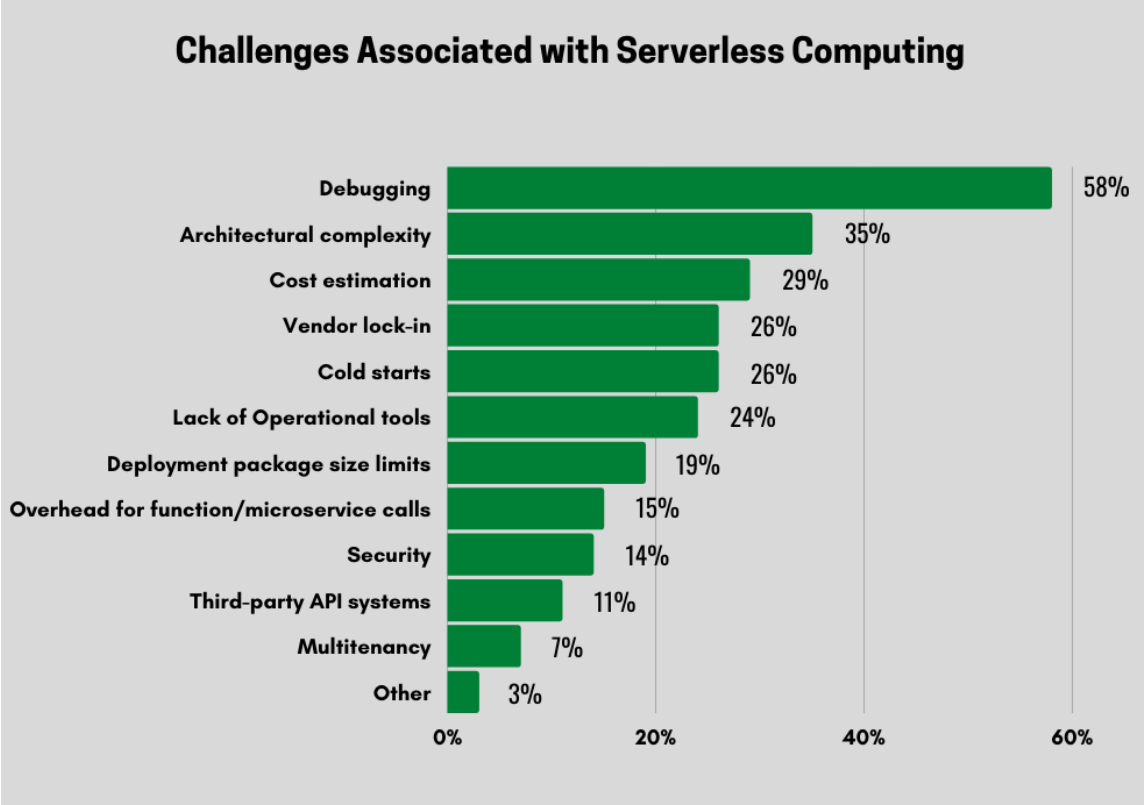

Figure 2.6: Challenges of using serverless computing according to the 2020 Coding Sans survey [27]

Some of the major challenges associated with serverless computing are as follows:

- *Testing and Debugging Challenges*: When transitioning from the traditional approach to serverless applications, it impacts the way testing is performed by developers. Traditionally, the developers would conduct local testing on application components in the same manner the application might be deployed in production. However,

because the infrastructure is abstracted away inside the serverless vendor's platform, performing end-to-end testing incorporating production-like error handling, logging, performance, and scaling characteristics in the serverless approach becomes exceedingly challenging [13]. Local testing and debugging are extremely difficult and time-consuming, according to many developers. They also stated that serverless applications that primarily employed lambda functions and API Gateway had much lower quality controls and practically no unit tests [12].

- *Cold Start Delay*: Due to the auto-scaling feature of the serverless platforms, resource creation must occur on the fly, which results in setting up new resources for a function execution to take a long time to complete. Because of this initial delay, applications frequently suffer performance degradations, which becomes significant for functions that have relatively short execution times [5].
- *Limited Lifetime and User Control*: The maximum execution timeout for a Lambda function is 15 minutes, which implies that Lambdas are terminated after 15 minutes. As a result, this makes them unsuitable for jobs that require more time to finish. In terms of user control, the user does not have any control over the order in which the provider starts interacting Lambdas [6]. This indicates that this order could be different from the user's invocation order.
- *Limited Resource Capacity*: Lambda containers lack sufficient memory to support the needs of many workloads. Even for moderately memory-intensive applications, garbage collection causes considerable overheads after only a few minutes of Lambda execution, according to [5]. This occurs due to the Lambda containers' relatively modest memory allocation. AWS Lambda provides each instance 512 MB of disk space in the /tmp directory to store the intermediate state [5].

Figure 2.6 illustrates the findings of the survey [27], conducted in the year 2020, regarding the most common problems that users face when working with serverless. According to the survey, testing and debugging are by far the biggest challenges with serverless. The above is caused by architectural complexity, which was voted as the second biggest challenge that users encounter when working with serverless. This is followed by cost estimation, which is another significant problem because it comes down to actual consumption, which is difficult to estimate ahead of time. Vendor lock-in was voted as the fourth biggest challenge in the survey because different serverless providers implement technologies and deliver services in different ways, making it exceedingly difficult and expensive for the customers to migrate from one platform to another. This occurs due to the lack of standardization in the serverless model [41].

CHAPTER 3

VIRTUAL MACHINES VS. CONTAINERS VS. SERVERLESS

3.1 What is a Virtual Machine and How Does It Work?

A virtual machine imitates a real computer and it can execute programs and applications without having direct interaction with any physical hardware. VMs function by running on top of a hypervisor, which is layered on top of either a host computer or a bare-metal host. A hypervisor is also referred to as a machine monitor, and it can either be a piece of software, firmware, or hardware that enables users to build and run virtual machines [42]. Each VM runs its own unique guest operating systems, allowing users to have a group of virtual machines sitting next to each other, each with its own unique operating system. A user could, for example, have a UNIX VM sitting alongside a Linux VM without any issues [24]. VMs allow users to experiment with another OS without actually installing the OS on their system. Users might use VMs to mess around with a different OS to determine whether it is the right choice for them. When they have finished working or playing around with the OS, they may simply delete the VM, which will have no effect on the system. Not only that, but users can also download and run applications from another OS. For example, if a Windows user were to run MacOS applications on their system that they would not have access to on their OS, they can use a VM for that purpose [38].

3.2 What is a Container and How Does It Work?

A container is a lightweight, standalone executable package of software that allows users to deploy individual applications inside portable and isolated environments. It packages the code and all of its dependencies, which include the system libraries, machine resources, runtime, and settings, to ensure that the application executes smoothly and can be moved from one computing environment to another without any hassles [40]. Containerization is a technology used to virtualize applications in a lightweight manner and it has seen significant use in the management of cloud service applications. Containers are more powerful and efficient as compared to virtual machines because they share the system resources with the host server instead of mimicking a virtual operating system. Additionally, containers allow frictionless application development by isolating the deployed application from the external host environments [37]. As stated earlier, containers have their own private space for processing, executing commands, mounting file systems, and private network interface and IP addresses, making them similar to VMs. There is, however, a difference between the two. Containers share the host's OS with other containers, whereas VMs imitate the host and do not share the host's OS. It is crucial to know that each container comes packaged with its own userspace that enables multiple containers to operate on the same host [24]. Furthermore, because the operating system is shared across all the containers, the only components that need to be developed from scratch are binaries and libraries, which can be simply added using a Docker image, making the entire process much easier.

3.3 Transitioning from Physical Servers to Virtual Machines to Containers to Serverless

Traditionally, if a company wanted to deploy an application or run a service, the DevOps team had to go out and purchase a physical server. For example, if a company wanted to set up an internal application, the DevOps team would determine what the company needed, order the necessary hardware, set it up, install the OS on the hardware, install the appropriate libraries and components required for the application to run, and then work on the application. Setting up a physical server, as seen in the preceding example, was extremely time-consuming and inefficient [29]. In addition, scaling the applications was a difficult task because it would necessitate the purchase of additional hardware by the company. Figure 3.1 illustrates how software has evolved over time, progressively enabling the levels of abstraction, and thereby increasing the focus on the code to be written rather than the infrastructure to be implemented.

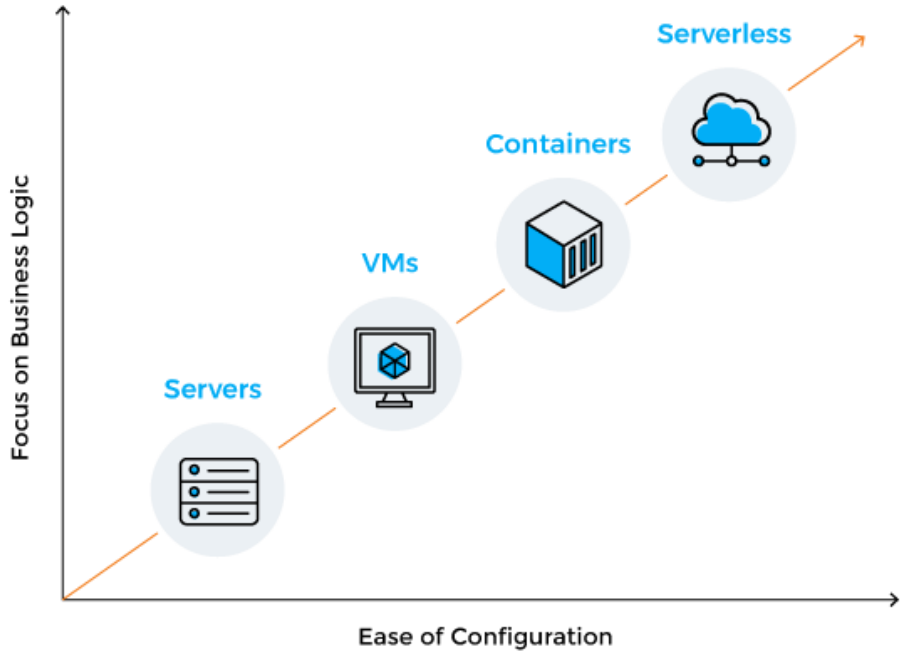

Figure 3.1: A graph illustrating the evolution in software based on ease of configuration, from physical servers to virtual machines to containers to serverless

With the advent of virtual machines, companies were able to provision resources more rapidly and were also able to allocate those resources depending upon the requirements of the applications. Virtualization was a huge turning point because the resources could be added, removed, or scaled as the demands of the application changed. However, as application development picked up the pace, deploying these applications on virtual machines became inefficient owing to the overhead of the hypervisor and VM guest operating systems. Another issue that developers encountered was that what worked on one VM on one hypervisor might not work on another, preventing the applications from being portable [29]. Figure 3.2, which is adapted from [1], offers a comparison of the services available to the customers under VMs, containers, and serverless, highlighting the responsibilities of the customer and the service

provider under each model. In VMs, for example, the customer is in charge of managing the applications and libraries, instances, networking, drivers, and the operating system.

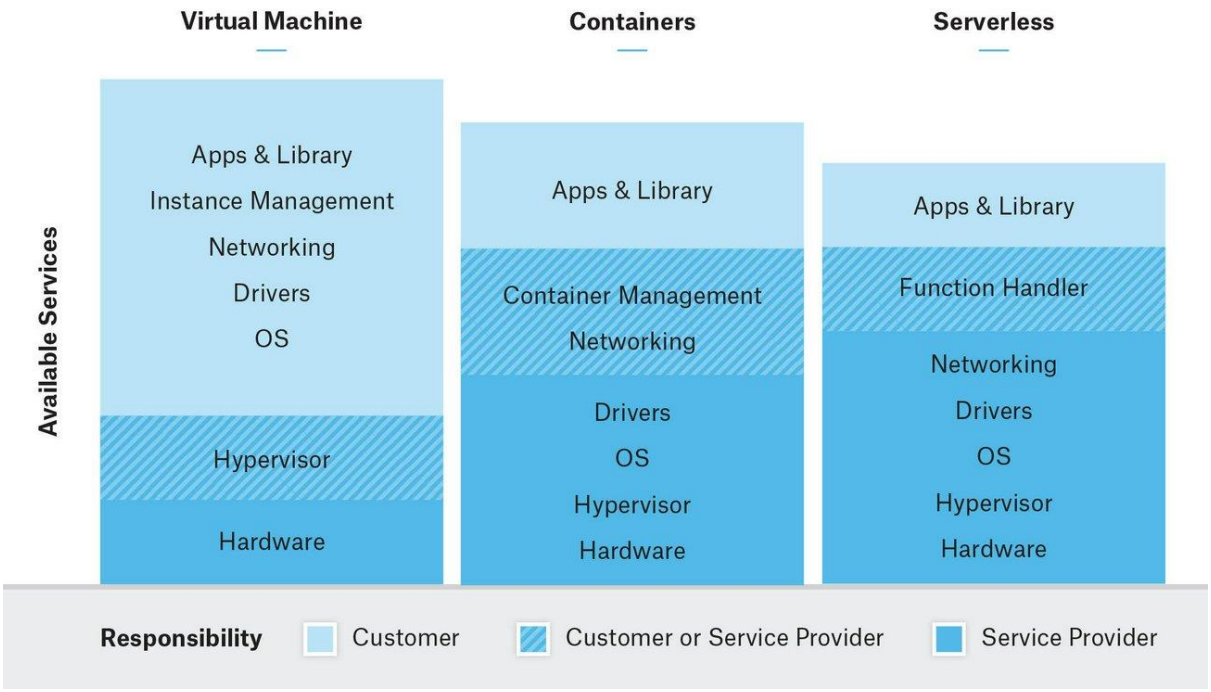

Figure 3.2: Comparison of the services available to the customers under VMs, containers, and serverless

Containerization eliminated the majority of the drawbacks associated with virtual machines. As stated earlier, a container standardized the process by bundling an application as well as the required binaries and libraries into a single package [29]. Containerization made it possible for these "contained" applications to run in any environment, irrespective of the underlying infrastructure. Containers are incredibly portable and lightweight, as well as highly efficient since they share resources, unlike virtual machines, which require the complete guest operating system to themselves. It is essential to remember that containers still require physical infrastructure to run on, but customers do not have to worry about

managing the hardware thanks to the available cloud computing services like AWS and Azure [29].

Lastly, serverless allows users to write and execute code without having to worry about implementing the server or the underlying infrastructure. The phrase "serverless" does not indicate "no servers"; it merely indicates that it requires zero configuration or maintenance from the developer. Users can run their code on AWS Lambdas without having to provision or manage any underlying infrastructure, which means they don't have to deploy VMs or containers, only their code. Moreover, Lambdas are event-driven, which means that they are triggered when an event occurs, therefore the application flow is mainly driven by events. Users can instruct Lambdas to execute their code in response to a certain event. As a result, users can build entire applications using serverless without having to manage any of the resources. The advantages of using AWS Lambda for event-driven serverless architecture include easier operational administration, quicker innovation, and lower operational expenses [31].

Table 3.1: A comprehensive comparison of virtual machines, containers, and serverless based on a variety of criteria

	Virtual Machines	**Containers**	**Serverless Computing**
Management	Extensive management	Flexible	Lightweight
Isolate workloads	Yes	Yes	Yes
Host environment	Physical "host" machine	Cloud or on-premises	Mostly cloud
Cost	Involves server costs, storage array, hypervisor maintenance, OS, server maintenance, network connectivity, and other costs.	Free, if you set them up yourself	Billed by the cloud provider, based on the resources utilized by the user
Stateful functions/apps	Stateful by default	Yes	In most cases, no
Supported Languages	Non-exhaustive list; depends on the virtual machine	Virtually any	Limited, but can be packaged in a Docker image that supports a well-defined API
Deployment times	Deployed in minutes	Deployed in seconds	Deployed in milliseconds
Ease of implementation	Requires more administrative work	Infrastructure management for container-based systems is relatively easier	No hardware or software to manage

Table 3.1 compares virtual machines, containers, and serverless based on a variety of criteria, including management, host environment, cost, deployment times, and ease of implementation, among other considerations. Virtual machines require extensive management from the customer, whereas containers and serverless are more flexible and lightweight [40]. In addition, whereas containers need developers to manage the infrastructure to some level, serverless does not necessitate any hardware or software management on the part of the developer. Serverless takes abstraction to a whole new level, which means that the user is

responsible for only deploying their code in serverless, and the cloud provider bills the client based on the resources used. It may appear like serverless is the ultimate choice, but as we shall see in subsequent sections, it has its drawbacks, which is why we explore hybrid solutions, which turn out to be superior alternatives.

3.4 Will Serverless Computing Make Containers Obsolete?

Developers are concerned that the adoption of serverless computing would obliterate docker containers. Many experts believe this isn't the case because containers require a minimal amount of conversion effort. For example, you could simply take an existing application, throw it inside a container and your work is done. This enables any team to transition to containers without encountering any significant challenges. Serverless computing, on the other hand, necessitates a fundamental redesign [33]. You'll need to rewrite all of your existing applications in order to transition to serverless. Most businesses, particularly those that are completely committed to cloud computing, are unwilling to devote considerable resources to serverless unless it provides a significant advantage, which serverless does not now.

In addition, serverless computing is appropriate for applications that need to perform tasks on the go but don't need to be running all of the time. Developers believe that serverless is the way to go when development speed and cost minimization are critical, and scalability issues need to be avoided. And, containers are better suited to larger, more complicated, and long-running programs that require a high level of control over the system environment [24]. As an example, containers would be ideal for hosting an e-commerce website that has several components, such as inventory management, product listings, transaction processing, and so

on, because you can bundle each of the above-mentioned services using containers without worrying about memory constraints and runtime limits. Building an application like the above using serverless computing would be a headache, as coordinating all of the serverless functions would be extremely tough [34].

Serverless technologies and container orchestration, notably via Docker and Kubernetes, are revolutionizing cloud computing, according to Dean Hallman, Founder, and CTO of Wiresoft. He claims that a new model, which he refers to as “serverless container orchestration” is gaining traction [10]. Serverless services are relatively cheaper, but as we will see in Chapter 4, the cost of Lambdas increases linearly as the scale and the throughput of our application increase. As a result of the foregoing, Kubernetes will eventually become more cost-effective than serverless. It should be noted that Kubernetes is not truly serverless. On the other hand, its management, scalability, and recovery capabilities, have the potential to limit ongoing maintenance for a long enough period of time to provide a reasonable approximation. Cloud platforms that lack certain QoS components, such as the FaaS runtime itself, might benefit from a Kubernetes-based strategy. In addition, vendor lock-in may be reduced or eliminated using hybrid solutions. They can also provide a more flexible architecture that allows for the application to be redeployed from a single source to more cost-effective serverless runtimes as the traffic grows [10]. Hence, it is reasonable to conclude that serverless and containers can operate together and provide their users with a gratifying experience.

CHAPTER 4

RESOURCE MANAGEMENT UNDER THE SERVERLESS MODEL

4.1 Resource Management Challenges

In a serverless environment, resource management refers to the general and overall aspect of efficiently managing the resource requirements of an application workload and the available system resources with minimal user interaction. Because of the autonomous nature of the expected resource management process in the serverless environment, each stage of the process necessitates extra attention for improved performance of the serverless applications and the system. One of the challenges associated with resource management strategies that users experience while working in a serverless environment is 'cold start delay,' which was already discussed in Section 2. Other challenges associated with resource management strategies will be discussed further below.

- *Diverse Workload Management*: As we know that there is minimal user involvement in the server resource management process, the serverless systems must develop an understanding of the application and the workload characteristics on their own. This is critical because the serverless systems are expected to deliver results in a flash, but this becomes challenging because of the diverse nature of applications being deployed on serverless platforms [7]. However, a lack of understanding of the resource

requirements of the application, application model, and workload arrival patterns could cause significant resource set-up delays, increased resource interference effects, and other issues, all of which could lead to customer dissatisfaction.

- *Resource Efficiency*: The fine-grained serverless billing model, in contrast to a generic cloud computing billing model, charges customers only for the resources allocated or utilized during the execution of the serverless application. The underlying infrastructure, on the other hand, may be maintained by the provider for longer periods of time. Due to the above, it necessitates paying extra attention to resource efficiency methods on the host nodes [7].
- *Lack of Desired Properties in Existing Storage Systems for Ephemeral Storage in Serverless Analytics*: The ephemeral storage system may make use of the unique characteristics of ephemeral data. Ephemeral data is transient and can be readily recreated by rerunning the tasks of a job. Traditional long-term storage systems cannot provide low data durability guarantees, whereas ephemeral storage solutions can [16]. Because most ephemeral data is written and read only once, the storage system may optimize capacity usage with an API that allows users to indicate when data should be deleted immediately after it is read. Serverless applications can consist of hundreds or thousands of lambdas in one execution stage and only a few lambdas in another. Now, in order to fulfill the I/O demands of these applications, the ephemeral storage system must have high elasticity, throughput, and IOPS [15]. Because serverless applications need to store both small and large objects, data access granularity varies significantly. This is the reason why ephemeral storage should be low-cost and high-performance at the same time. As discussed in the previous sections, the storage services should auto-scale resources depending on the load and relieve users of the burden of maintaining

storage clusters. Based on this approach, the users will only be charged for the bandwidth and the capacity they use, essentially extending the serverless abstraction to storage services. Existing serverless analytics ephemeral storage solutions are reported to be slow (Amazon S3), and in-memory datastores are relatively expensive (ElastiCache Redis) [16].

4.2 Prior Studies Aimed at Addressing the Majority of Resource Management Problems

The following section will focus on prior studies that were conducted with the goal of solving the aforementioned difficulties. These researches address issues like cold start delay, insufficient support for sharing of the intermediate state in Lambdas, runtime resource limitations, and Lambdas charging a greater price per unit resource than virtual machines, resulting in higher expenses for long-lasting resource demands.

4.2.1 Pocket: Elastic Ephemeral Storage for Serverless Analytics

As already mentioned in Section 2, serverless platforms were initially designed for web microservices and IoT applications, but their elasticity and fine-grain pricing make them appealing for data-intensive applications such as interactive data analytics. Exchanging intermediate data between execution stages in an analytics job is a significant challenge because direct communication between serverless tasks is difficult and inefficient [15]. Naturally, developers store such ephemeral data in a remote data store. That being said, the existing storage solutions are not designed to fulfill the demands of serverless applications in terms of elasticity, performance, and affordability [16].

Table 4.1: Comparison of existing storage systems and desired properties for ephemeral storage for λs in serverless analytics

	Elastic Scaling	**Latency**	**Throughput**	**Max Object Size**	**Cost**
S3	Auto, coarse-grain	High	Medium	5 TB	Low
DynamoDB	Auto, fine-grain, pay per hour	Medium	Low	400 KB	Medium
Elasticache Redis	Manual	Low	High	512 MB	High
Aerospike	Manual	Low	High	1 MB	Medium
Apache Crail	Manual	Low	High	Any size	Medium
Desired Properties for λs	*Auto, fine-grain, pay per second*	*Low*	*High*	*Any size*	*Low*

Note: This table is adapted from [16].

Table 4.1 presents a comparison of existing storage systems and the desired properties for ephemeral storage in serverless analytics, as discussed earlier in this section. According to the table, existing storage systems do not have all of the desired properties for Lambdas. Amazon S3 has significant latency, whereas ElastiCache Redis is quite expensive. The above makes it increasingly challenging for users to work with serverless and for serverless to provide its full range of services.

The research “Pocket: Elastic Ephemeral Storage for Serverless Analytics” explores an alternate storage solution for serverless analytics that possesses the desired properties for Lambdas. Pocket is a distributed elastic data store used for ephemeral data sharing in serverless analytics. It achieves high performance and cost-effectiveness by leveraging multiple storage technologies, rightsizing resource allocations for applications, and autoscaling storage resources in the cluster based on usage. It dynamically rightsizes resources across several dimensions such as CPU cores, network bandwidth, storage capacity,

leveraging multiple storage technologies to minimize costs and ensuring that applications are not bottlenecked on I/O [16]. Pocket divides the responsibilities across three different planes: the control plane, the metadata plane, and the data plane. The control plane is in charge of managing cluster sizing and data placement, while the metadata plane is in charge of tracking the data saved across nodes in the data plane. All I/O operations are designed to be simple, with latencies of less than a millisecond. Pocket's storage servers are optimized for rapid I/O and just store data, not metadata, which makes it considerably easier for them to scale up and down. The controller scales resources, based on the demand, at second granularity and regulates load by intelligently directing incoming job data, making Pocket elastic [16]. Furthermore, it provides comparable performance to ElastiCache Redis for serverless analytics applications while reducing costs by about 60%. Amazon ElastiCache for Redis is a high-performance in-memory data store that provides sub-millisecond latency that can power internet-scale real-time applications. In ElastiCache, AWS is in charge of managing the servers hosting the Redis, and it scales automatically based on the requirements, which means that it may scale up or scale down depending on the application's demands. Additionally, it deploys, runs, and scales Redis and MemCached in-memory data stores effortlessly [39].

4.2.2 SplitServe: Efficiently Splitting Apache Spark Jobs Across FaaS and IaaS

AWS Lambda and other cloud functions have been identified as ideal candidates for dealing with unexpected spikes in simple stateless workloads. This is because, as compared to virtual machines, AWS Lambda and other cloud functions offer lower startup latencies and finer-grain pricing. However, when it comes to autoscaling complex workloads that involve considerable state transfer across distributed application components, cloud functions may not

be as effective. Lambdas have a number of limitations that make integrating virtual machines and Lambdas for complex workloads extremely difficult [5]. These limitations include limited resource capacity, limited lifetime and user control, inadequate support for sharing of the intermediate state, and a steeper cost curve for longer-lasting work. The limitations ‘limited resource capacity’ and ‘limited lifetime and user control' have already been discussed in Section 2. I would now like to focus on the other two limitations for a clearer understanding of the content.

- *Inadequate Support for Sharing of Intermediate State*: Lambdas do not expose an IP address, which is why there is no direct channel for an entity, whether it is a virtual machine or Lambda, to send data to a Lambda after its initial invocation. This, along with Lambdas' limited lifetime and the user's lack of control over them when launched by the provider, implies that state transfer between Lambdas must rely on a storage facility outside of the source Lambda's container. Existing serverless analytics ephemeral storage solutions are reported to be slow, and in-memory datastores are relatively expensive [5, 16].
- *Steeper Cost Curve for Longer-Lasting Work*: VM-based deployments were traditionally used to scale resources based on the demands of the application. However, the VM-based approach presented the providers with difficulties such as higher costs or SLA violations [7]. A service-level agreement (SLA) is a contract between a service provider and a client that defines the level of service that the client can expect from the provider. It is an essential component of any technical service provider contract as it specifies the metrics used to measure the level of service provided, and the penalties that the provider would be liable to if he fails to meet the

specified service levels expectations [26]. Having stated that, functions are auto-scaled within containers with low startup latencies under the serverless model, and the users are billed per function invocation at a very granular level owing to the fine-grained pricing feature of the model. However, several studies have concluded that deploying an entire application as serverless functions is not cost-effective at times. This is due to the fact that Lambda functions are expensive, as their cost grows linearly even when there is a fixed load and the average request rate is higher. Lambdas charge a higher price per unit resource than virtual machines, resulting in higher expenses for long-lasting resource demands. Hence, it is safe to say that Lambda functions are more cost-effective when there are demand variations and average request rates are low [5]. VM-based deployments, on the other hand, are more efficient with higher arrival rates. The aforementioned is why a hybrid of VM-based deployments and FaaS-based deployments might be the solution to our problems.

SplitServe, which is an enhancement of Apache Spark, addresses these constraints by making the currently available cloud functions useful and efficient for even complex workloads. It can leverage cloud functions to efficiently bridge the gap in virtual machines if there aren't enough executors available on pre-existing virtual machines to execute a newly arriving latency-sensitive job. It can easily avoid the startup latencies of newly requested VMs in this manner. Furthermore, SplitServe has the ability to redistribute ongoing work from cloud functions to newly requested VMs, or executors on existing VMs, as they become available if it is desirable in terms of performance or cost [5]. SplitServe addresses the statelessness of serverless applications by employing a single shared high-throughput storage layer for intermediate data shuffling using HDFS, which can be accessed by both the VM as

well as the Lambda-based executors. Hadoop Distributed File System (HDFS) is a distributed file system that handles large data sets running on commodity hardware. It can scale a single Apache Hadoop cluster to hundreds (or even thousands) of nodes based on the demand of the application. SplitServe uses HDFS due to its ease of implementation compared to other alternatives, and because Spark has library support for HDFS writes [5].

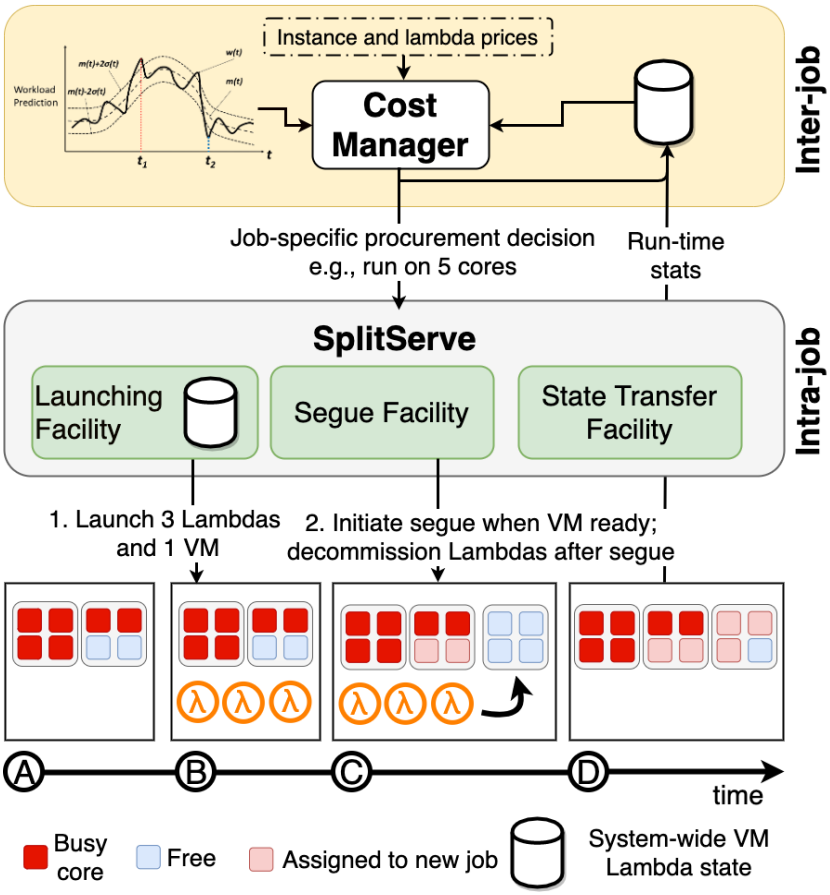

Figure 4.1: SplitServe viewed as an autoscaling system integrating resource procurement at two time scales: inter-job and intra-job decision making

SplitServe acts as an autoscaling system, integrating resource procurement at two time scales: inter-job decision-making across numerous jobs and intra-job decision-making based on resource requirement prescriptions made by the inter-job decision-making model. Figure

4.1, which is adapted from [5], offers a clearer understanding of the design of SplitServe for both inter-job and intra-job management. However, the design and operation of SplitServe are solely focused on the latter i.e., intra-job resource management. SplitServe's design for intra-job management consists of three key facilities, which have been outlined below.

- Launching Facility: The launching facility assists in sharing access to the system-wide VM/Lambda state with the cost manager, where the state tracks information about the executors currently running a job and which VM cores are available at any one time [5]. In addition, the launching facility allocates the requested number of cores for a new job from the currently available cores. If no cores are available at the moment, the facility launches new Lambdas and assigns the newly arrived job to them.
- Segueing Facility: SplitServe utilizes a segueing facility that is responsible for launching VMs in the background matching the number of cores procured through any Lambdas launched by SplitServe's launching facility. Segueing longer running tasks initially started on Lambdas to VMs is critical because of the higher cost of Lambdas after a certain period of time and the relatively small memory allocation of Lambda containers. Due to the relatively small memory allocation of Lambda containers, the JVM garbage collector is invoked more frequently, which ultimately impacts the overall workload performance of the application. It is essential to understand that the segueing facility only launches the VMs if the estimated execution time of the job exceeds the nominal VM startup delay [5]. Starting up new VMs would be absolutely pointless for jobs with estimated execution times less than the startup delay associated with VMs.

- State Transfer Facility: Finally, SplitServe implements a storage layer, as previously mentioned, which facilitates rapid transfer of state to and from Lambdas. Several Spark workloads engage in significant data transfers across stages, i.e. the shuffle operation. If an application framework allocates executors dynamically, the shuffle data is stored in local storage to prevent data loss when dealing with ephemeral executors [5]. Users must pay a steeper price per unit resource for Lambda-based executors than VMs to preserve the shuffle data, and each Lambda is provided a relatively small amount of local storage. The use of a cloud-based shared storage layer, such as Amazon S3 or SQS, and in-memory caches, such as Redis or Memcached, are two common solutions to this problem. However, the issue with these solutions is that they are either too slow or too expensive or both, which is why the state transfer facility is a better alternative.

SplitServe can employ any other similar storage facility [16] that offers a suitable cost-performance trade-off to the tenant, such as the previously mentioned "Pocket: Elastic Ephemeral Storage for Serverless Analytics". However, there is another study [35] titled “Shuffling, Fast and Slow: Scalable Analytics on Serverless Architecture” that isn't included in this thesis but may be used with SplitServe since it also offers a good cost-performance ratio.

SplitServe's experimental evaluation used four different workloads (either on a mixture of VM-based executors and cloud functions or simply cloud functions), and it demonstrated that SplitServe when compared to only VM-based autoscaling, improved the execution time by up to 55% for workloads with small to moderate amounts of shuffling and up to 31% for workloads with large amounts of shuffling. Furthermore, SplitServe-Spark,

with its unique and novel segueing techniques, can help save up to 21% of costs while still reducing execution time by nearly 40% [5].

4.2.3 Prebaking Functions to Warm the Serverless Cold Start

FaaS platforms, as we know, promise a simpler programming model for cloud computing, in which the developers focus on developing applications while platform providers handle resource management and administration. Platform providers ensure that idle resources are not kept running because FaaS users are only billed for the time the functions are executed. However, there is a shortcoming with this strategy. This might result in the cold start delay issue discussed previously, in which a function's execution is delayed owing to the lack of accessible resources to perform the function's execution. Cold starts can take hundreds of milliseconds to seconds, which can be prohibitively slow and a severe disadvantage for many applications [9]. The research “Prebaking Functions to Warm the Serverless Cold Start” focuses on decreasing the function process start-up time using a cloning technique based on Checkpoint/Restore In Userspace (CRIU).

The prebaking technique focuses on decreasing the function start-up time by restoring snapshots of previously started functions runtimes. Now, before a function is actually ready to serve requests, it must perform a complex series of steps that include creating a new process to host the runtime, bootstrapping the runtime, which involves initializing the function’s data structure and auxiliary services, and loading the function code. We derive from the research that restoring a function snapshot is faster than re-running all the start-up steps mentioned above. The above-mentioned checkpoint/restart approach is commonly employed in high-performance computing to tolerate faults in long-running applications [9]. High-performance computing is the practice of combining computing resources in order to handle complex

issues in the domains of research, engineering, or business, with the goal of achieving substantially greater performance than a conventional desktop computer or a workstation can provide [18]. For instance, instead of restarting from scratch when a failure happens, we could simply resume the application from the periodically generated snapshots. In the research, the snapshot generation is done differently from the high-performance computing case; the prebaking technique creates function snapshots only when a new function version is deployed by the user.

According to the research, using process cloning to launch serverless functions eliminates the overhead of the JVM start-up, resulting in a 40% increase for a NOOP function, and a 47% to 71% improvement for more representative functions. It also enables platform administrators to interact with the process prior to persisting its state. The prebaking approach successfully reduces not just the JVM startup overhead, but also the overhead caused by loading and compiling the code (JIT). Furthermore, the study demonstrates that these gains are proportionate to the function's code size [9].

4.2.4 ENSURE: Efficient Scheduling and Autonomous Resource Management in Serverless Environments

Last but not least, resource scheduling is a significant challenge for developers working with serverless environments.

- *Resource Scheduling*: Mapping workloads to appropriate host nodes based on their resource requirements is an essential task and an important challenge that is of importance and interest to both the cloud providers and developers, as is the efficient utilization of available resources. When demand for resources exceeds the available resource capacity, resource scheduling must additionally consider determining the

> order in which applications should be executed. Resource scheduling should address the challenges of decision-making associated with resource provisioning, resource allocation, and scheduling of function invocation requests in a serverless environment. To think about it, a comprehensive resource scheduling approach must employ performance prediction algorithms to determine the optimal level of resource allocations and the scheduling policies to fulfill the consumer and provider expectations [7].

There are certain existing approaches to achieve resource efficiency, for example, schedulers built for VM placement or web load balancers, but it turns out that they are not well suited for serverless scheduling. Virtual machines require the users to provide a specification of the resource request, such as the number of cores in a virtual machine. However, that is only limited to VMs. Serverless environments do not require the user to provide an input corresponding to the number of resources required, because they assume that any available server can execute an incoming request. However, if a server is already warm, which means that it has an active container of that application, it is preferred over starting a new server to avoid cold-start delays [17].

ENSURE, which is short for EfficieNt SchedUling and autonomous REsource management framework, is a function-level scheduler and an autonomous resource manager that is designed to reduce the provider's resource costs while satisfying the customer's performance expectations. It operates by categorizing the incoming function requests during runtime, and it then regulates the resource use of co-located functions on each invoker carefully. It elastically scales capacity in response to fluctuating workload traffic to avoid cold-start delays, utilizing concepts from operations research. Lastly, it schedules the

incoming requests by focusing the load on an adequate number of invokers so as to reuse active hosts [17]. This strategy helps them avoid cold starts and allows the unneeded resources to gracefully time out, without causing any difficulties.

ENSURE addresses the issue of how user functions must be scheduled and resource managed on bare-metal servers to lower the provider's expenses at scale while providing acceptable latencies. Addressing the aforesaid issue carefully required the researchers to focus on placing the incoming workload at the hosts efficiently and scaling the serverless platform's capacity elastically so they could reduce the provider's capital and operating expenses in the presence of dynamic workload traffic. As we are well aware, the provider, not the customer, is responsible for making these decisions in serverless environments. It can achieve high resource efficiency within and across containers. ENSURE classifies serverless functions into categories based on their diverse resource consumption and lifetime patterns within a container. It then scalably determines how the functions should be placed depending upon the inferred class of the incoming function after considering the aforementioned [17]. Not only that, but it also alleviates the resource contention across colocated, and perhaps diverse functions by dynamically regulating their cpu-shares at runtime to provide acceptable latencies, as previously indicated. Beyond a single container, ENSURE relies on two key components to achieve resource efficiency. The first component ENSURE relies on is their autoscaling component, FnScale, which dynamically adjusts the number of containers and hosts in response to fluctuations in the workload traffic. And, the second component it relies on is their scheduling component, FnSched, which distributes function requests carefully across hosts in order to minimize the SLO (Service Level Objective) violations.

More about ENSURE, it is implemented on Apache OpenWhisk, which is an open-source, distributed serverless platform that executes functions in response to events at any

scale. OpenWhisk handles the infrastructure, servers, and scaling with Docker containers, allowing the users to focus on building the applications [19]. Unlike the existing resource managers in OpenWhisk and Kubernetes, it delivers acceptable latencies across a diverse set of serverless applications. Its performance is evaluated on a 34-VM serverless cluster in AWS, and it turns out that ENSURE manages the incoming traffic with 18% to 52% fewer hosts and with around 60% fewer cold starts when compared to the existing scheduling policies. To conclude, ENSURE is designed to prevent cold starts and avoid fine-grained resource contention between diverse application functions. ENSURE's architecture, which is based on queueing theoretic principles, provides high resource efficiency and does not compromise on application latency [17].

CHAPTER 5
CONCLUSION AND FURTHER RESEARCH

5.1 Conclusion

For developers seeking relief from the burden of infrastructure, serverless computing is a fantastic option. The serverless model makes it easier and faster for developers to deploy their code by abstracting away everything but a block of code, which allows smaller teams to accomplish what previously only larger businesses could [28]. Serverless computing is a rapidly evolving field that is continually seeing substantial advancements. It is increasingly being explored for use in different fields and application domains such as web services, big data, internet of things, machine learning model training, and for large-scale mathematical computations [7]. We have explored some of the major challenges that users face while working in these various sectors and implementing those services in a serverless environment in this thesis. Many researchers and major cloud providers are continually striving to make serverless more inexpensive and to overcome the significant challenges that customers often encounter when working with serverless platforms. We have also looked at and analyzed some major research works conducted previously that focus on overcoming some of the key problems mentioned in the preceding sections. Serverless, as previously said, will continue to evolve, thanks to the developers and researchers working tirelessly and diligently to improve it every day for the benefit of all users.

5.2 Further Research

Serverless is a blessing in disguise for users who are unfamiliar with deploying an entire server or for customers who lack the necessary resources to do so. They may face a number of difficulties while working with serverless, but this should not be an impediment. Current research is focused on a single resource management issue, which may pose a dilemma for users as they may be forced to compromise one aspect for the sake of the other. Therefore, it would be both challenging and fascinating if studies could focus on resolving the majority of the significant resource management and data compute issues in a single prototype. This would allow developers to concentrate on just deploying their applications on the serverless platform, rather than worrying about first understanding and analyzing the prototype before implementing it. Businesses that are considering to transition to serverless would certainly be grateful if there were inexpensive and efficient solutions available in the market to assist them to deploy applications to serverless without having to worry about the resource management concerns highlighted in this thesis and numerous other research publications.

Despite the substantial amount of research that has been or is currently being conducted in the field of serverless architecture, there are still numerous gaps. I sincerely hope that this thesis will serve as a guide for the new users. I also hope that the resources included in this thesis will serve as a starting point for researchers working on resolving resource management problems under the serverless model, and that their work could demonstrate progress in the field, therefore benefitting users interested in transitioning to serverless. The existing research works offer a lot of room for improvement, underlining the vast potential for further research. There are a number of problems that are not widely recognized or addressed, but we cannot deny that many new users encounter them when they

first start working with serverless architecture. The majority of current research studies, for instance, are restricted to specific programming languages or platforms. It would be instrumental if the studies could incorporate additional language support, allowing users to fully explore the benefits of serverless. The researchers might also devise different versions of other popular application frameworks to cater to the larger community.

BIBLIOGRAPHY

[1] "Serverless Computing: Hot or Not?" *CloudOps*, 2 July 2018, www.cloudops.com/blog/serverless-computing-hot-or-not/.

[2] Passwater, Andrea. "2018 Serverless Community Survey: Huge Growth in Serverless Usage." *Serverless Blog*, www.serverless.com/blog/2018-serverless-community-survey-huge-growth-usage.

[3] ServerlessOps. "Serverless: What Is It & Why?" *ServerlessOps*, www.serverlessops.io/what-is-serverless.

[4] Stenberg, Jan. "Characteristics of Serverless Architecture." *InfoQ*, InfoQ, 13 Aug. 2019, www.infoq.com/news/2019/08/traits-serverless-architecture/.

[5] Jain, Aman, et al. "SpIitServe: Efficiently Splitting Complex Workloads Across FaaS and IaaS." *SpIitServe | Proceedings of the ACM Symposium on Cloud Computing*, 1 Nov. 2019, dl.acm.org/doi/10.1145/3357223.3366027.

[6] Knape, Sander. "The Hidden Challenges of Serverless: from VM to Function." *The Hidden Challenges of Serverless: from VM to Function - Sander Knape*, 2 Aug. 2018, sanderknape.com/2018/08/hidden-challenges-serverless-vm-function/.

[7] Mampage, Anupama, et al. "A Holistic View on Resource Management in Serverless Computing Environments: Taxonomy and Future Directions." *ArXiv.org*, The University of Melbourne, Australia, 1 June 2021, arxiv.org/abs/2105.11592.

[8] Baldini, Ioana, et al. "Serverless Computing: Current Trends and Open Problems ..." *Research Advances in Cloud Computing*, SpringerLink, 28 Dec. 2017, link.springer.com/chapter/10.1007/978-981-10-5026-8_1.

[9] Silva, Paulo, et al. "Prebaking Functions to Warm the Serverless Cold Start." *Prebaking Functions to Warm the Serverless Cold Start | Proceedings of the 21st International Middleware Conference*, 7 Dec. 2020, dl.acm.org/doi/10.1145/3423211.3425682.

[10] Hallman, Dean. "Building Serverless- and Container-Based Applications: 7 Trends to Watch." *TechBeacon*, TechBeacon, 22 Jan. 2019, techbeacon.com/enterprise-it/building-serverless-container-based-applications-7-trends-watch.

[11] Castro, Paul, et al. "The Rise of Serverless Computing." *Communications of the ACM Volume 62 Issue 12*, 21 Nov. 2019, dl.acm.org/doi/10.1145/3368454.

[12] Swail, Paul. "The Pains of Testing Serverless Applications." *Serverless First*, 28 July 2020, serverlessfirst.com/pains-testing-serverless/.

[13] Ishani, Akila. “The Biggest Obstacles to Wider Serverless Adoption - DZone Cloud.” *Dzone.com*, DZone, 4 Dec. 2019, dzone.com/articles/what-is-the-biggest-obstacle-to-wider-serverless-a.

[14] Boylan, Clay. “How AWS Lambda Changed the Game of Multi-Tenancy.” *Narrative Science*, 22 Apr. 2020, narrativescience.com/resource/blog/how-aws-lambda-changed-the-game-of-multi-tenancy/.

[15] Klimovic, Ana, et al. “Understanding Ephemeral Storage for Serverless Analytics.” *Understanding Ephemeral Storage for Serverless Analytics | Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, 1 July 2018, dl.acm.org/doi/10.5555/3277355.3277431.

[16] Klimovic, Ana, et al. “Pocket: Elastic Ephemeral Storage for Serverless Analytics.” *USENIX*, 2018, www.usenix.org/conference/osdi18/presentation/klimovic.

[17] Suresh, Amoghavarsha, et al. “ENSURE: Efficient Scheduling and Autonomous Resource Management in Serverless Environments.” *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, 2020, doi:10.1109/acsos49614.2020.00020.

[18] *What Is High Performance Computing?*, www.usgs.gov/core-science-systems/sas/arc/about/what-high-performance-computing.

[19] “Open Source Serverless Cloud Platform.” *Apache OpenWhisk Is a Serverless, Open Source Cloud Platform*, openwhisk.apache.org/.

[20] Jonas, Eric, et al. “Cloud Programming Simplified: A Berkeley View on Serverless Computing.” *ArXiv.org*, 9 Feb. 2019, arxiv.org/abs/1902.03383.

[21] “What Is Function-as-a-Service (FaaS)? | Cloudflare.” *Cloudflare*, www.cloudflare.com/learning/serverless/glossary/function-as-a-service-faas/.

[22] “The Three Service Models of Cloud Computing.” *Open International*, Holistic CIS, Industry Articles, 14 Nov. 2019, www.openintl.com/the-three-service-models-of-cloud-computing/.

[23] Adminandru. “1.1 On Premise vs. Serverless.” *Customers Love Solutions: On Premise vs. IaaS vs. PaaS vs. FaaS vs. SaaS*, 7 Nov. 2019, customers-love-solutions.com/?p=784.

[24] Ezell, Will. “Virtual Machines vs Containers vs Serverless Computing: Everything You Need to Know.” *DotCMS Content Management System*, DotCMS, 19 Sept. 2018, dotcms.com/blog/post/virtual-machines-vs-containers-vs-serverless-computing-everything-you-need-to-know.

[25] Jauregui, Eddie. “Serverless: Benefits and Challenges.” *The Media Temple Blog*, 22 Oct. 2019, mediatemple.net/blog/cloud-hosting/serverless-benefits-and-challenges/.

[26] Overby, Stephanie, et al. “What Is an SLA? Best Practices for Service-Level Agreements.” *CIO*, CIO, 5 July 2017, www.cio.com/article/2438284/outsourcing-sla-definitions-and-solutions.html.

[27] Zöld, Gábor. “7+1 Serverless Trends You Need to Know in 2020.” *Coding Sans*, 22 Sept. 2020, codingsans.com/blog/serverless-trends.

[28] Hoff, Todd. “What Is Serverless Computing: Infrastructure Freedom for Developers.” *InfoWorld*, InfoWorld, 27 Apr. 2018, www.infoworld.com/article/3175761/what-is-serverless-computing-infrastructure-freedom-for-devs.html#tk.ifw-infsb.

[29] Rifai, Moneer. “From Physical Servers to VMs to Docker Containers. What's Next?” *Medium*, Medium, 20 Apr. 2017, medium.com/@moneerrifai/from-physical-servers-to-vms-to-docker-containers-whats-next-moneer-rifai-27bc6c77179f.

[30] “AWS Lambda Pricing.” *Amazon*, aws.amazon.com/lambda/pricing/.

[31] “Event-Driven Serverless Architecture Using AWS Lambda.” *Medium*, Cuelogic Blog, 22 Aug. 2018, cuelogictech.medium.com/event-driven-serverless-architecture-using-aws-lambda-6aef8d52ba80.

[32] Eismann, Simon, et al. “A Review of Serverless Use Cases and Their Characteristics.” *ArXiv.org*, 25 Aug. 2020, arxiv.org/abs/2008.11110.

[33] The Martec. “Will Serverless Computing Kill Docker Containers?” *Hacker Noon*, 28 Mar. 2018, hackernoon.com/will-serverless-computing-kill-docker-containers-222671bffdc4.

[34] Dhabekar, Praful. “Containers vs. Serverless: Which One You Should Choose in 2020?” *Medium*, DevOps Dudes, 17 Aug. 2020, medium.com/devops-dudes/containers-vs-serverless-which-one-you-should-choose-in-2020-38b1a0fa2b8a.

[35] Pu, Qifan, et al. “Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure.” *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, USENIX Association, 23 Feb. 2019, www.usenix.org/system/files/nsdi19-pu.pdf.

[36] Heath, Nick. “Writing Serverless Code: The Programming Languages and Everything Else You Need to Know.” *ZDNet*, ZDNet, 1 May 2019, www.zdnet.com/article/writing-serverless-code-the-programming-languages-and-what-else-you-need-to-know/.

[37] “Containers vs VMs.” *Red Hat - We Make Open Source Technologies for the Enterprise*, www.redhat.com/en/topics/containers/containers-vs-vms.

[38] Hoffman, Chris. “Beginner Geek: How to Create and Use Virtual Machines.” *How-To Geek*, 27 Oct. 2020, www.howtogeek.com/196060/beginner-geek-how-to-create-and-use-virtual-machines/.

[39] "Amazon ElastiCache for Redis." *Amazon*, aws.amazon.com/elasticache/redis/.

[40] Ramanathan, Kalyan. "Serverless vs. Containers: What's the Same, What's Different?" *Sumo Logic*, 22 Aug. 2019, www.sumologic.com/blog/serverless-vs-containers/.

[41] Mok, Kimberley. "Should We Really Be Worried about Vendor Lock-in in 2020?" *Protocol*, Protocol - The People, Power and Politics of Tech, 1 Dec. 2020, www.protocol.com/manuals/new-enterprise/vendor-lockin-cloud-saas.

[42] Sheldon, Robert, and Brian Kirsch. "What Is a Virtual Machine and How Does It Work?" *SearchServerVirtualization*, TechTarget, Mar. 2021, searchservervirtualization.techtarget.com/definition/virtual-machine.

ACADEMIC VITA

SAKSHAM ARORA

sfa5353@psu.edu

EDUCATION

The Pennsylvania State University | Schreyer Honors College **University Park, PA**
Bachelor of Science in Computer Science *Expected Graduation: August 2021*
Relevant Coursework: Operating Systems, Programming Language Concepts, Data Structures and Algorithms, Introduction to Systems Programming, Computer Organization and Design, Object-Oriented Programming with Web-Based Applications

RELEVANT EXPERIENCE

College of Engineering, The Pennsylvania State University ***January 2020 – May 2021***
Learning Assistant | CMPSC132 (Programming and Computation II: Data Structures)

- Taught in-class recitation lectures on object-oriented programming, data structures, and debugging errors in Python.
- Held office hours to thoroughly explain concepts covered in the course and assist students with additional questions.
- Assisted the instructor in crafting assignments to further students' learning of the concepts covered in the course.

Jasch Industries Ltd. ***May 2019 – July 2019***
Software Engineering Intern | Nucleonic Gauging Division

- Collaborated with other team members in engineering a software framework for radiation-based electronic gauges to automate the manufacturing process using the system-design platform LabView.

Tata Consultancy Services ***June 2018 - July 2018***
Software Development Intern | Passport Seva Project

- Worked on the Passport Seva Project as part of the Testing Team, tested the mPassport Police Android App which is used by the Indian Police during the process of verifying passport applicants.
- Optimized the app through Test Case Preparation, Execution, Defect Logging, and Analysis using Java, JBoss, and DB2.

PROJECTS

CART HRAM System

- Created a user-space device driver for a file in-memory file system that was built on top of an HRAM system.
- Maintained the correct file contents in the HRAM system special-purpose memory device and communicated with a memory controller that stored the file-system data.

Channels: Concurrent Programming

- Established communication over channels between different threads using mutex locks and conditional waits to ensure safe data transmission between threads.
- Handled communication operations associated with blocking and non-blocking signals for buffered channels utilizing concurrent programming.

Dynamic Storage Allocator

- Created a dynamic storage allocator for incorporating a customized version of malloc, free and realloc functions.
- Made the allocator organize the blocks of memory in a structured manner, use the structure to choose an appropriate location to allocate new memory, and update the structure whenever a block of memory was freed.

Writing a Caching Web Proxy

- Implemented a basic sequential policy to handle HTTP/1.1 GET requests, listen for incoming connections on a specific port number, and establish a connection.
- Cached web objects and stored local copies of objects from servers, and later responded to future requests by reading them out of its cache rather than by communicating again with remote servers.

SKILLS

Programming: Python (Pandas, NumPy, TensorFlow, Keras), Java, C/C++, MySQL, LabView, HTML, JavaScript, MATLAB

LEADERSHIP AND INVOLVEMENT

- Penn State Engineering Peer Mentor Collective, **Student Mentor** *March 2020 – March 2021*
- Penn State Engineering Orientation Network, **Head Mentor** *May 2019 – April 2020*
- The GLOBE (Special Living Option in the Schreyer Honors College)**, Vice-President** *March 2018 – May 2019*
- Global Entrepreneurship Week at Penn State, **Event Team Member** *November 2018*