



MASTERARBEIT | MASTER'S THESIS

Titel | Title

Exploring the use of serverless functions for compute-intensive efforts

verfasst von | submitted by

Roxana-Teodora Mafteiu-Scai

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of
Master of Science (MSc)

Wien | Vienna, 2024

Studienkennzahl lt. Studienblatt | Degree
programme code as it appears on the
student record sheet:

UA 066 921

Studienrichtung lt. Studienblatt | Degree
programme as it appears on the student
record sheet:

Masterstudium Informatik

Betreut von | Supervisor:

Univ.-Prof. Dipl.-Ing. Dr. Siegfried Benkner

Acknowledgements

I would like to express my gratitude to my supervisor, Univ.-Prof. Dipl.-Ing. Dr. Siegfried Benkner for his unparalleled support throughout the completion of my master studies in the Computer Science Department at University of Vienna. Univ.-Prof. Dipl.-Ing. Dr. Siegfried Benkner offered me the opportunity to work on such an interesting problem, and closely supervised every step towards the completion of my thesis. Each result of this work is based on the guidance and motivation he gave me during our inspiring discussions. Additionally, I had the privilege of working as a Staff Tutor on various teaching teams within the Scientific Computing research group, under his supervision. This role allowed me to deepen my knowledge of parallel computing and gain valuable experience in the field. I am sincerely grateful for these opportunities, which have significantly enriched my academic and professional journey.

I want to thank my family for raising me to follow my dreams and constantly supporting me. Our daily endless and cheerful conversations have been an enormous help, notably during the past months. Moreover, I am grateful to my parents for awakening my passion for Computer Science.

Last but not least, I would like to extend my wholehearted thankfulness to my educators and friends who encouraged and supported me.

Abstract

In the current landscape of software development, cloud computing has emerged as a dominant solution for hosting applications that face increasing demands on scalability, data storage, and security. This thesis explores serverless computing, a rapidly evolving cloud service model that enables applications to scale effortlessly, with infrastructure management fully handled by cloud providers. Serverless computing, also known as Function as a Service (FaaS), is particularly advantageous for event-driven applications and short-lived tasks, offering benefits like ease of deployment, automatic scaling, and resource-usage-based billing.

This work presents *flexi-serverless*, a C++ library designed to facilitate seamless function invocation on both serverless platforms and local environments. The framework supports cloud providers such as AWS Lambda and Google Cloud Functions, allowing developers to experiment with and evaluate applications in both environments, especially for compute-intensive and parallel tasks. Through three application scenarios, the library demonstrates its utility in simplifying transitions between local and serverless computing, making serverless solutions more accessible and efficient. Additionally, the thesis provides a comparative analysis of serverless and traditional computing models, highlighting factors such as cost, performance, and scalability. This research contributes to the field by addressing challenges in serverless adoption and offering a versatile tool for developers aiming to optimize their cloud computing strategies.

Kurzfassung

In der heutigen Softwareentwicklung hat sich Cloud Computing als dominierende Lösung für das Hosting von Anwendungen etabliert, die wachsenden Anforderungen an Skalierbarkeit, Datenspeicherung und Sicherheit gerecht werden müssen. Diese Arbeit untersucht serverloses Computing, ein schnell wachsendes Cloud-Service-Modell, das Anwendungen ermöglicht, mühelos zu skalieren, wobei das Infrastrukturmanagement vollständig von Cloud-Anbietern übernommen wird. Serverloses Computing, auch als Function as a Service (FaaS) bekannt, ist insbesondere für ereignisgesteuerte Anwendungen und kurzlebige Aufgaben vorteilhaft und bietet Vorteile wie einfache Bereitstellung, automatische Skalierung und Abrechnung auf Basis der Ressourcennutzung.

Diese Arbeit stellt *flexi-serverless* vor, eine C++-Bibliothek, die eine nahtlose Funktion-saufrufmöglichkeit sowohl auf serverlosen Plattformen als auch in lokalen Umgebungen bietet. Das Framework unterstützt Cloud-Anbieter wie AWS Lambda und Google Cloud Functions und ermöglicht Entwicklern, Anwendungen in beiden Umgebungen zu testen und zu evaluieren, insbesondere bei rechenintensiven und parallelen Aufgaben. Anhand von drei Anwendungsszenarien zeigt die Bibliothek ihren Nutzen bei der Vereinfachung von Übergängen zwischen lokalem und serverlosem Computing, wodurch serverlose Lösungen zugänglicher und effizienter werden. Zudem bietet die Arbeit eine vergleichende Analyse von serverlosen und traditionellen Rechenmodellen und hebt dabei Faktoren wie Kosten, Leistung und Skalierbarkeit hervor. Dieses Forschungsvorhaben trägt zur Optimierung von Cloud-Computing-Strategien bei, indem es Herausforderungen bei der Einführung serverloser Architekturen adressiert und Entwicklern ein vielseitiges Werkzeug zur Verfügung stellt.

Contents

Acknowledgements	i
Abstract	iii
Kurzfassung	v
1 Introduction	1
1.1 Motivation	1
1.2 Contributions	2
1.3 Outline	3
2 Background	5
2.1 Characteristics of Serverless Platforms	5
2.1.1 Strenghts of Serverless Platforms	5
2.1.2 Drawbacks of Serverless Platforms	6
2.2 Serverless Platforms	7
2.2.1 Hosted Serverless Platforms	7
2.2.2 Open-Source Serverless Platforms	9
2.3 Data Flow in Serverless Platforms	11
2.3.1 Small Data	12
2.3.2 Large Data	13
2.3.3 Motivating Examples	14
2.4 Application Domains for Serverless Platforms	17
2.5 Embarrassingly Parallel Serverless Computation	18
3 Related Work	19
4 Framework for Serverless Parallel Computing	25
4.1 Design and Architecture	25
4.2 Implementation Aspects	25
4.2.1 Library Structure	26
4.2.2 Creating, Deploying and Invoking Serverless Functions	27
4.2.3 Data Transfer	31
4.3 User Interface	32
4.3.1 Motivating Examples	32
4.3.2 Installation and Configuration Process	38

5 Evaluation	41
5.1 Criteria for Evaluation	41
5.1.1 Scalability Behaviour	41
5.1.2 Cost	42
5.1.3 Parallelism Granularity	43
5.2 Computational Environment	43
5.3 Monte Carlo Integration	44
5.4 String Similarity	46
5.5 Image Filtering	51
5.6 Summary	54
6 Conclusions and Future Work	57
Bibliography	59

1 Introduction

1.1 Motivation

In the current era, modern software applications are confronted with the challenge of accommodating an ever-increasing number of users, accompanied by heightened computational demands, amplified data storage requirements, and reinforced security and dependability standards. To remain competitive and responsive to this upward trajectory, cloud computing has rapidly emerged as the predominant model for hosting such applications. With its ability to provide expansive, pliable, and regionally distributed service hosting, along with instant resource allocation to cater to fluctuations in service demand, cloud computing has come to be viewed as the de facto standard for the contemporary software landscape. Following the deployment of these applications on the cloud, the concerns pertaining to underlying infrastructure devolve upon the cloud service providers.

Currently, cloud computing service models have been categorized into five distinct categories, based on the means employed by cloud providers to deliver their services. The National Institute of Standards and Technology identifies three traditional models, namely Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [MG11]. Notably, due to the benefits engendered by cloud computing, such as simplified backend infrastructure maintenance, two additional models have materialized since 2008 and 2011. These novel models are referred to as Backend as a Service (BaaS) and Serverless computing [Row11, Reh21].

Serverless computing is an evolving trend for the deployment of applications and services [JPV⁺17, QCL21]. A serverless application consists of managed stateless ephemeral computing solutions, also known as function-as-a-service (FaaS) or serverless functions, with fully provider-managed services for file storage, messaging, databases, streaming, and authentication, known as BaaS. Some well-known FaaS solutions are Amazon Web Services (AWS) Lambda [aws22b], Google Cloud Functions [goo22], as well as Apache OpenWhisk [FOU20].

The serverless paradigm offers several benefits ranging from ease of development and deployment to seamless elasticity since an application can be scaled up by executing multiple copies of the same function in parallel. Another critical feature of serverless computing is resource-usage-based billing. Additionally, there are numerous use cases of FaaS such as Internet of Things (IoT) data processing, automated threat detection, and image processing.

On a high-level, the serverless functions work in the following way. They are usually hosted on containers, simplifying the application packaging and the function invocation process. To process requests received from a new application on a host, there are several

1 Introduction

steps involved. First, a new container is launched. Then, the runtime environments are set up. Finally, different aspects related to the applications are initialised. The cold start latency represents the latency for these steps [WLZ⁺18a, JPV⁺17].

Since the FaaS platforms are responsible for the automatic configuration and maintenance of the function execution context and their connection to the cloud services without a server provision [YBLW19], the terms “serverless computing” and FaaS can be used interchangeably, hence “serverless computing” is referred to FaaS in this work.

Deciding whether to adopt serverless computing or traditional on-premise/local computing in compute-intensive applications depends on several key factors and considerations. Serverless computing is an attractive choice when scalability is crucial, as platforms like AWS Lambda and Google Cloud Functions excel at auto-scaling to efficiently handle varying workloads. It also tends to be more cost-effective for sporadic or low-traffic workloads, given that one only pays for actual compute usage. Event-driven applications, which respond to events such as HTTP requests or database changes, can significantly benefit from serverless architecture. Additionally, serverless is well-suited for short-lived tasks that fit within the platform’s time limits and when there is a need for seamless integration with managed services.

On the other hand, opting for on-premise or local computing makes sense in scenarios involving predictable workloads, resource-intensive tasks exceeding serverless limits, reliance on legacy systems, or the necessity for complete control over the computing environment. During development and testing phases, local computing can streamline debugging and iteration processes, offering more control and flexibility. The ultimate decision should align with your application’s specific requirements, performance goals, scalability needs, and cost considerations. In some cases, a hybrid approach may be the most appropriate solution, combining the advantages of serverless computing with the control and customization options of local computing for different components of the application.

Developers may find it necessary to experiment with both serverless and on-premise computing solutions to determine which one best fits their specific needs in applications, especially parallel applications. This experimental approach allows for a more comprehensive evaluation of factors such as performance, cost-effectiveness, scalability, and ease of management. By testing both options, developers can make well-informed decisions that align with their application’s unique requirements and constraints, ensuring the chosen solution optimally meets their goals and objectives

1.2 Contributions

We propose a solution to the aforementioned evaluation. This solution solves the problem of invoking functions on a serverless platform or locally for evaluation purpose. The key contribution of this thesis is a framework, called *flexi-serverless*. This is implemented as a C++ static library designed for simplified local and serverless function invocations across various cloud providers (Google Cloud and AWS). The architecture places a premium on ensuring smooth transitions and fostering extensibility for future integrations, thereby

rendering serverless computing both accessible and highly efficient for developers. To exemplify the library, we provided three application scenarios suitable for compute-intense, parallel efforts: Monte Carlo Integration, String Similarity and Image Filtering. We evaluated them in terms of scalability behaviour, cost and parallelism granularity. A second contribution is the discussion of multiple works related to embarrassingly parallel serverless computations, useful for our work.

1.3 Outline

After introducing the motivation and contributions of this thesis, we provide a short overview of the chapters to come: Chapter 2 introduces the minimal context for the thesis. Chapter 3 reviews related work. Chapter 4 presents an overview of the proposed framework, including the implementation. Chapter 5 presents the application scenarios using the framework and its evaluation. Finally, Chapter 6 summarizes the work and provides directions for future work.

2 Background

This chapter contains a brief overview of the concepts that this thesis builds on. For brevity, most of the following sections simplify and only include needed information.

2.1 Characteristics of Serverless Platforms

To have a better overview of serverless computing platforms, we present the main characteristics as follows, based on the comprehensive review done by Wen et al. [WCL22]. These are divided into strengths and drawbacks.

2.1.1 Strengths of Serverless Platforms

Auto-scaling

One key characteristic in serverless computing is auto-scaling of the function instances. Depending on application workload dynamics, the amount of resources allocated to that function is adapted accordingly. There are different ways of scaling as follows: horizontal scaling involving scaling-in (i.e. launch new function instances) and scaling-out (recycle old instances); respectively vertical scaling involving scaling-up (adding) and scaling-down (removing) the amount of resources used. One issue concerning auto-scaling could appear when the number of instances is scaled to zero in case there are no subsequent requests and the serverless platforms scale down the resources. In this case, all the allocated resources are to be released to a usage-based billing model without paying for idle times. Even though this capability decreases the cost, it leads to the cold start problem.

Usage-based billing

The usage in serverless computing is billed based on the allocated or consumed resources for the serverless application per fine-granular execution unit. The advantage of this characteristic is that the idle resources are not billed compared to classic cloud computing, where the users need to pay for the resources even in stand-by mode. Compared to non-serverless- or cloud-based computing, developers do not shoulder all expenses, including the purchase and installation of physical servers and the maintenance costs of engineering personnel. On top of that, the development process does not tend to be protracted due to non-uniform architecture and lower product maturity, resulting in delayed market releases.

2 Background

Separation of storage and execution

The separation of storage and execution gives one major adaptation in serverless computing. The separation of storage means the cloud providers provide cloud storage services for storing the data of the serverless functions, whereas the execution represents the stateless serverless functions. In other words, this allows the resource provisioning and usage-based billing being considered and scaled independently.

Functionality and no operations

Another beneficial characteristic of serverless platforms is given by the possibility of choosing the most preferred programming language, such as JavaScript and Java, for implementing the function-level code of the serverless application. Additionally, serverless platforms offer their own integrated development environments (IDEs) and a user-friendly deployment of the application code. In non-cloud-based software development, developers face a multifaceted process that involves selecting technology stacks, configuring local environments, and handling complex backend functionalities, resulting in relative inefficiency. In cloud-based development, they gain control over the cloud infrastructure and applications but still grapple with setup and security complexities. However, serverless-based development offers a streamlined approach where applications are constructed using event-triggered serverless functions and optional cloud services. These functions are ephemeral and stateless, and cloud services like storage can store their data. Additionally, Backend-as-a-Service (BaaS) simplifies backend functionality development, presenting a more efficient and user-friendly development paradigm.

Abstract performance capabilities

While serverless platforms do provide performance configuration options, such as memory allocation and the number of process instances triggered, it is essential to maintain this configuration abstracted from the underlying instance responsible for executing user applications or processes

2.1.2 Drawbacks of Serverless Platforms

Several constraints

Despite having abstract performance capabilities, there are several constraints established by the cloud providers for the usage of serverless functions in order to maintain the auto-scaling. These constraints range from function execution timeout, deployment package size to even memory size. Depending on the serverless platform, there might be different constraints.

Latency

One of the most significant drawbacks encountered by serverless platforms pertains to latency. Latency denotes the time elapsed between the initiation and the completion of a

function request, and the presence of latency inevitably gives rise to function timeout issues in Serverless computing.

Vendor lock-in

Another significant drawback of serverless architectures is the vendor lock-in. It arises from the lack of application portability due to tight integration with third-party services. This issue becomes particularly problematic when multiple Backend as a Service (BaaS) services are integrated, as switching cloud platforms would necessitate changing all third-party service integrations, potentially causing a major disruption to client applications.

Lack of specialised hardware

Serverless computing relies on abstracting users from the underlying environment, which can be advantageous. However, it can also pose challenges. Currently, major serverless platform providers offer users the ability to provision a time slice of a CPU hyperthread and a certain amount of RAM, but they lack mechanisms to access specialized hardware. Consequently, certain use cases that demand specific hardware, like GPU units, remain unfeasible in present-day serverless platforms. However, one serverless platform, namely Azure Functions [azu23], can be used in combination with Kubernetes [kub22] to support GPU units, according to an answer on the the platform's GitHub Issues page [azu1].

Security

According to [WLZ⁺18b], the serverless platforms are susceptible to potential security vulnerabilities.

2.2 Serverless Platforms

Based on the aforementioned characteristics, various serverless platform implementations were designed. Some platforms, such as AWS Lambda [aws22b], Google Cloud Functions [goo22] and Azure Functions [azu23], are commercial and rolled out by major cloud providers. Whereas others such as Apache OpenWhisk [FOU20] and OpenFaaS [Opee] are open-source and were developed by the serverless computing community.

Worth mentioning is that each platform has its own benefits and differences compared to the others. When selecting a serverless platform, one needs to consider different aspects regarding performance, costs, supported programming languages and model, available user documentation, function portability between different providers, deployment easiness and even what monitoring and debugging tools are available [KS18]. The characteristics of these serverless platforms were well summarized in the study [WCL22], Table 1.

2.2.1 Hosted Serverless Platforms

The commercial serverless platforms are also known as hosted and their aim is to relieve users from the responsibilities of resource management and platform configuration, as

2 Background

the provider takes care of these tasks. Additionally, they offer seamless integrations with their own products like message queues and cloud storage. In essence, hosted serverless platforms simplify serverless computing by managing infrastructure and facilitating easy integration with related services.

AWS Lambda

The first serverless platform was AWS Lambda and was developed by Amazon in 2014. Numerous interactions such as command-line interface (CLI), HTTP-based application programming interface (API), Software Development Kits (SDKs) for language languages, respectively graphical user interface (GUI) are supported by AWS Lambda. Different factors such as function execution time, allocated memory size and invocation number affect the billing of AWS Lambda. It also enables functions that can run up to 15 minutes. The CPU power is allocated in proportion to the amount of memory configured, whereas the functions have 128 MB of memory allocated by default and can support up to 10 GB. With more CPU resources, the Lambda function's duration might decrease. AWS Lambda offers multiple services and utilities useful to monitor the systems and accesses, such as Amazon CloudWatch [clo22a] and AWS CloudTrail [clo22b]. Compared to other solutions, AWS Lambda benefits from a marketplace for application development called AWS Serverless Application Repository (AWS SAR) [AWS22c]. AWS Lambda operates on several key principles. Firstly, it centers around the concept of functions, which contain the code to be executed when specific events occur. These functions can be identified by qualifiers, which come in two forms: function versions (immutable snapshots) and aliases (modifiable pointers). When invoking a function, you have the option to specify a qualifier alongside the function name. Lambda employs runtimes to bridge its service and functions, offering support for various programming languages and the flexibility to create custom runtimes when needed. Events, another crucial component, serve as input for invoked functions, carrying information relevant to the event being processed, such as messages from Amazon Simple Queue Service (SQS) queues or events detected by IoT sensors through AWS IoT Events. Different event sources generate distinct event object structures. Additionally, Lambda manages concurrency, which determines the number of simultaneous requests it can handle within a specific AWS region, with concurrency limits that can be set per function. Lastly, triggers play a pivotal role by initiating function execution in response to specific events.

Google Cloud Functions

Google adopted the serverless model later in 2016. The main drawbacks of Google Cloud Functions are the lack of IDE support and no marketplace for pre-built serverless applications. However, it has support for CLI, API, SDKs and GUI, like AWS Lambda. The billing model used by the Google Cloud Functions is connected to execution time, provisioned memory and CPU, respectively outbound network requests and data transfer fees. Google Cloud Functions offers two product versions: Cloud Functions (first generation), the original version, and Cloud Functions (second generation), the latest being the

recommended one for usage. By default, the memory allocated for a function is 256MB or 256 MiB, depending on the Cloud Functions product version. Google Cloud Functions also enables functions that can run up to 9 minutes, or even 60 minutes, depending on the product generation.

Azure Functions

Similarly to other solutions, Microsoft offers Azure Functions, which empower developers by allowing them to execute event-triggered functions without the need for intricate infrastructure management. While it is a fully managed cloud service, a portion of its source code can be accessed on GitHub [git23]. Azure Functions offers three distinct pricing models: the Consumption Plan, which relinquishes resource management to Azure and bills users based on function executions, accounting for both execution time and memory consumption; the Premium Plan, which grants customers the ability to specify pre-warmed instances and incurs charges for these instances regardless of inactivity, in addition to other resources utilized for request handling; and the App Service Plan, tailored for users of Azure App Services, enabling them to run functions without incurring supplementary costs. Function apps are initially set up within a designated runtime environment, serving as the execution environment for functions developed in the chosen programming language. Function invocations are facilitated through triggers like HTTP, timers, or queues, while bindings establish connections between functions and external resources, including queues and blob storage, supporting the integration of multiple input and output bindings. Notably, Azure Functions also facilitates local development environments, streamlining debugging procedures while maintaining seamless connectivity to Azure's cloud services

2.2.2 Open-Source Serverless Platforms

Open-source serverless platforms emerge as promising alternatives to the hosted serverless platforms' challenges such as vendor lock-in and restrictive limitations. In total, two notable platforms have been chosen for consideration based on the number of stars on GitHub [git23] and Kubernetes [kub22] support: OpenFaaS and OpenWhisk.

Apache OpenWhisk

Released by IBM and Apache also in 2016, OpenWhisk is an open-source serverless, representing a key technology of the FaaS product, called Cloud Functions, provided by IBM. The programming model of OpenWhisk can be seen in Figure 2.2.2 and consists of three primary concepts: actions (stateless functions executing code), triggers (class of events originating from different sources), respectively rules (associating a trigger with an action). The platform's architecture, as seen in Figure 2.2.2 ensures secure communication through HTTPS, employs NGINX [ngi22] for request handling, and relies on a controller, CouchDB database [Con22], Kafka streaming [Kaf22], and Docker containers [doc22] for action execution. Regarding the function deployment, OpenWhisk is focused on

2 Background

simplifying function development, offering deployment options with tools like Docker Compose [Doc13] and Ansible [Red12], with Kubernetes [kub22] being the preferred choice. OpenWhisk supports container pre-warming and reuse for efficiency according to [Tho17]. As for the function scalability, the OpenWhisk controller is responsible for managing this. While its default configuration suits development, it may require adjustments for heavy loads, with options to increase resources allocated to components. However, some system limitations may remain unchangeable. For the time being, there is no known marketplace for serverless applications in OpenWhisk. Additionally, CLI and API interactions, together with local developments and cloud environment developments are offered by OpenWhisk.

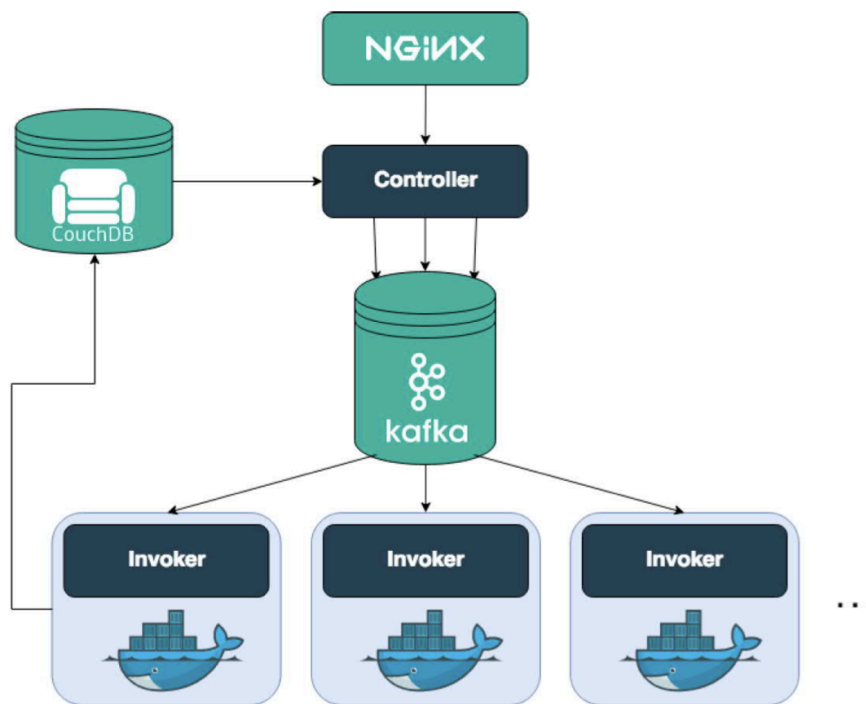
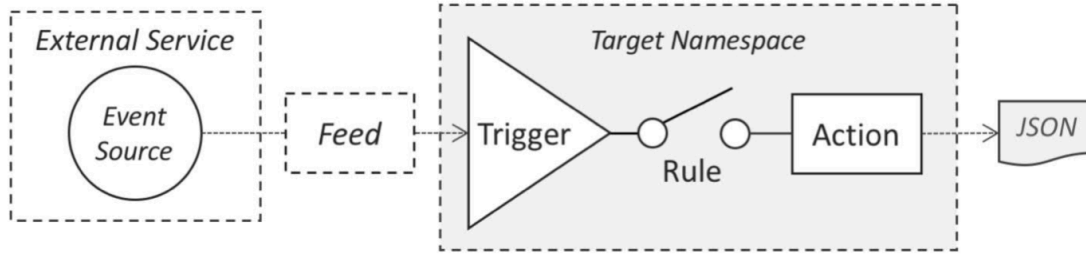


Figure 2.1: Apache OpenWhisk Architecture [oped]

OpenFaaS

OpenFaaS simplifies serverless function and microservices development, with Kubernetes and OpenShift [opea] recommended for production deployment [opea]. Users can manage functions through a UI portal or CLI, deploying them as container images. OpenFaaS offers a template store with various programming language options, streamlining function image creation. Users need to write only function handlers as templates include necessary

Figure 2.2: Apache OpenWhisk Programming model [\[oped\]](#)

files like Dockerfiles. Functions and configurations in YAML files can be packaged into container images, pushed to registries, and deployed. An additional function store allows users to choose community-contributed functions. In the Kubernetes cluster context, faas-netes serves as the official OpenFaaS provider, managing functions deployed from Docker registries. OpenFaaS provides access via CLI, UI, and REST API, accompanied by services like Prometheus for metrics collection, AlertManager for request-based auto-scaling, and NATS for asynchronous function invocations. Functions are deployed as pods with auto-scaling options. OpenFaaS includes a watchdog process in the function Dockerfile, offering two types: classic and of-watchdog. Classic forks a new process per request, while of-watchdog introduces advanced features like keeping functions warm and data caching. Auto-scaling in OpenFaaS can be request-based using Prometheus and AlertManager, or custom metrics-based via Kubernetes HPA when the default rule is disabled. Figure [2.2.2](#) illustrates the conceptual workflow of OpenFaaS on a Kubernetes cluster, depicting the interactions between various components and highlighting the auto-scaling processes.

2.3 Data Flow in Serverless Platforms

Serverless functions can accept data, process it and handle it, but the approach to doing so requires careful consideration due to the constraints and characteristics of serverless computing. The consideration can also be based on the size of the data: small or large.

One study by Winzinger et al [\[WW21\]](#) delves into the intricate data flows within serverless functions and their interactions with the environment. It introduces a measurement framework designed to assess the utilization of these data flows, aiding developers in easily detecting and quantifying data exchanges in serverless applications. While the current tool offers dynamic coverage analysis, future efforts aim to incorporate static testing support and explore practical applications, including potential interviews with domain experts and real-world evaluations.

It is highly recommended to adhere to the best practices provided by each cloud

2 Background

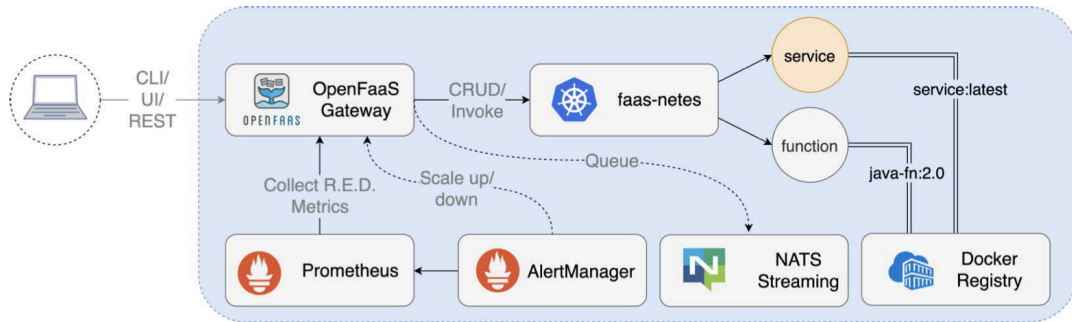


Figure 2.3: OpenFaaS Conceptual Workflow [\[opeb\]](#)

provider for handling data flow using their serverless platform. Each cloud provider has specific guidelines and services tailored to optimize data workflows within their serverless ecosystem. There are some general strategies for effectively working with data in serverless platforms, provided as follows. In summary, small data can often be processed directly within serverless functions, while large data is typically stored in cloud storage or databases, with serverless functions triggering processing tasks as needed. It is important to note that the transfer of data largely depends on the client application's needs and the cloud provider's limitations, as payload size restrictions may apply.

2.3.1 Small Data

Function Invocation Parameters

Small data can be passed as parameters to serverless functions when invoking them. These parameters are typically in the form of JSON objects or simple data types like strings or numbers.

Environment Variables

Serverless platforms allow you to set environment variables, which can be used to store small configuration data or constants that your functions need.

In-Function Processing

Small data can be processed entirely within the function itself, without the need for external storage. For instance, you can process small configuration files or perform calculations using data directly supplied to the function.

In-Memory Caches

Serverless functions can use in-memory caching mechanisms to store and retrieve small amounts of frequently used data for improved performance. This is especially useful for data that needs to be accessed quickly across multiple function invocations.

2.3.2 Large Data

Cloud Storage

For large data, it's common to use cloud storage services (*e.g.*, *AWS S3*, *Azure Blob Storage*, *Google Cloud Storage*) to save and retrieve data. Serverless functions can interact with these storage services to read or write large files or datasets.

Database Services

Use database services like *AWS DynamoDB*, *Azure Cosmos DB*, or *Google Cloud Firestore* for storing and querying large structured datasets. These services are designed to handle large volumes of data efficiently.

Data Streaming

Large data streams can be processed in real-time or in batches using data streaming services (*e.g.*, *AWS Kinesis*, *Apache Kafka*). Serverless functions can be triggered by these streams for processing.

External APIs

Serverless functions can call external APIs to retrieve or send large data sets. For instance, a function can fetch data from a third-party service and process it.

Serverless Data Warehouses

Serverless data warehousing solutions like *AWS Athena*, *Google BigQuery*, or *Azure Synapse Analytics* are suitable for analyzing large structured datasets using SQL queries. Serverless functions can trigger queries on these platforms.

File Processing

If dealing with large files, serverless functions can read, process, and write data back to cloud storage or other services. This is useful for tasks like file conversion or data extraction.

2 Background

Asynchronous Processing

For very large data processing, consider breaking tasks into smaller, asynchronous steps. Each step can be handled by separate serverless functions, and data can be passed between them using storage or messaging services.

Optimization for Data Size

Optimize your serverless functions for the size of the data they handle. Be mindful of memory constraints, and consider processing data in smaller chunks or batches if it's too large to fit within the function's memory limits.

Data Pipelines

Create data pipelines that orchestrate the flow of data between serverless functions and various data storage and processing services to manage large-scale data workflows efficiently.

2.3.3 Motivating Examples

A motivating example of data flow in Google Cloud Functions is provided in the Optical Character Recognition (OCR) tutorial from Google Cloud [\[Goo\]](#), showcasing the power of serverless computing for complex tasks. It walks through the process of performing OCR on Google Cloud Platform, illustrating a seamless workflow. The OCR tutorial application's data flow comprises the following sequential steps, described as follows. The overall architecture can be seen in Figure [2.3.3](#).

Image Upload

An image containing text in any language is uploaded to a Cloud Storage bucket specially created for uploading images.

Text Extraction and Language Detection

A Cloud Function is triggered, when a file is uploaded to the Cloud Storage bucket. The function invocation parameters are the Cloud Storage bucket name and the image filename. This utilizes the Vision API to extract the text from the image, based on the aforementioned parameters, and then detect the source language.

Translation Queueing

The extracted text is queued for translation by publishing a message to a Pub/Sub topic. A translation request is queued for each target language different from the source language.

Language Match Check

If a target language matches the source language, the translation queue is bypassed, and the text is sent to the result queue, which resides in a different Pub/Sub topic.

Translation

Another Cloud Function is executed when a message is published to the Cloud Pub/Sub topic specified. This receives as parameter a Pub/Sub Message object containing the extracted text, the image filename as well as the desired language. Then, it uses the Translation API to translate the text in the translation queue. The translated result is then sent to the result queue.

Storage of Translations

Yet another Cloud Function retrieves the translated text from the result queue and saves it to Cloud Storage.

Accessing Results

The translated results can be found in Cloud Storage as individual text files, each corresponding to a specific translation.

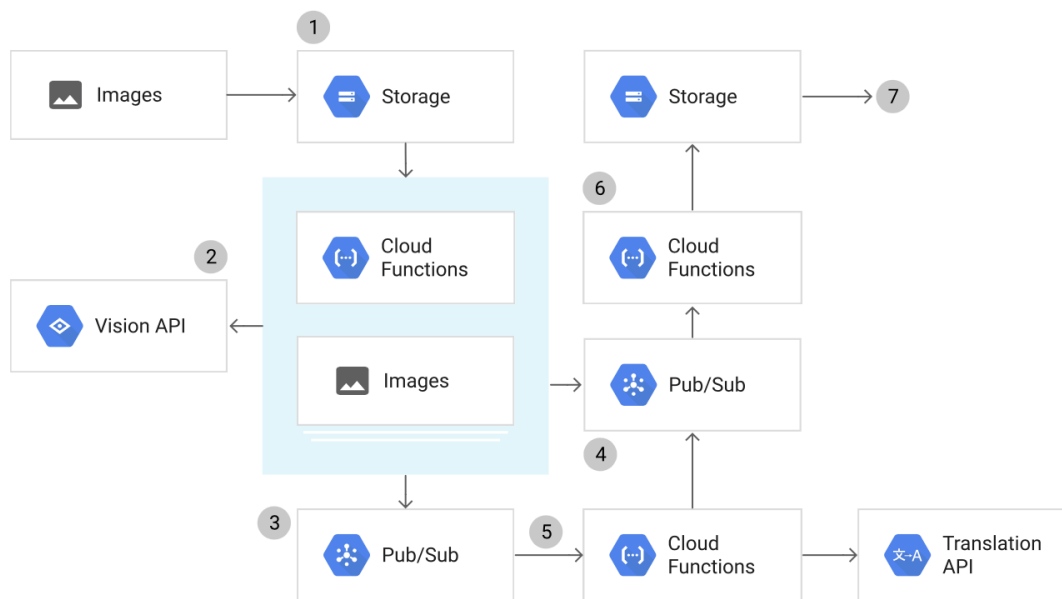


Figure 2.4: OCR architecture using Google Cloud Functions [Goo]

2 Background

Another example is provided by AWS [\[Gui\]](#), using a list manager application, illustrated in Figure [2.3.3](#). This showcases AWS Lambda's capabilities in processing records from an Amazon Kinesis data stream efficiently. Using Lambda's event source mapping, it reads records from the stream in batches and triggers a Lambda function. This function leverages the information within the records to update Amazon DynamoDB documents and stores the processed records in Amazon Relational Database Service (Amazon RDS). Records are sent to a Kinesis stream by clients, serving as a temporary buffer for data awaiting processing, similar to a queue. Notably, records in a Kinesis stream are not deleted after processing, allowing multiple consumers to work with the same data. Furthermore, records are processed in order, ensuring a sequential flow, unlike traditional queues where items can be delivered out of order. Records are eventually removed from the stream after 7 days. Additionally, the application features a secondary function for handling administrative tasks related to the database. It operates within a private Virtual Private Cloud (VPC) and connects to Amazon RDS, utilizing a VPC endpoint to interact with Amazon DynamoDB. The application also makes use of AWS Secrets Manager, AWS X-Ray, and AWS CodeDeploy for enhanced functionality and security.

List manager application

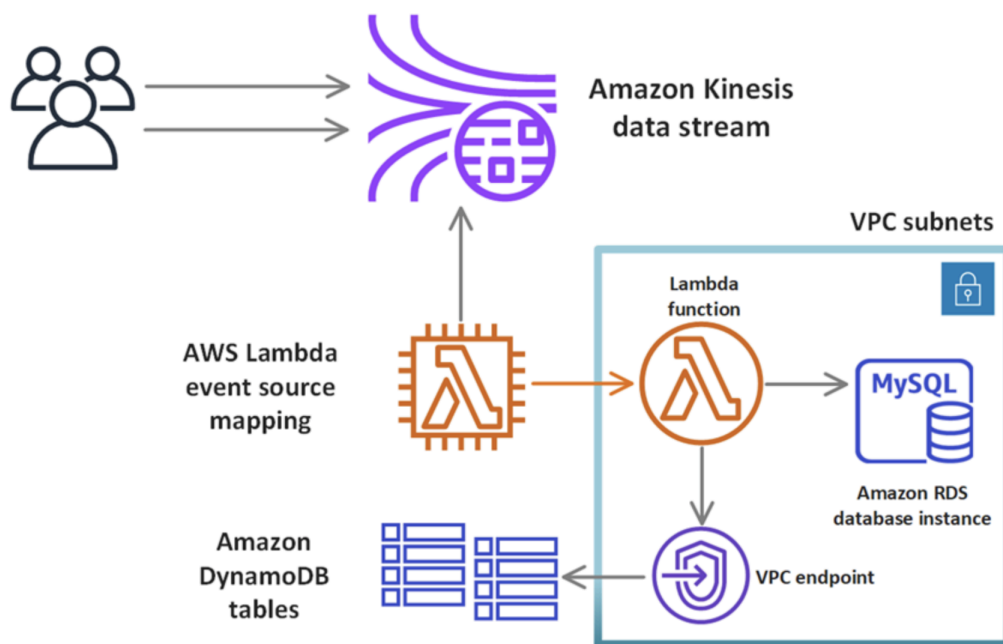


Figure 2.5: List manager architecture using AWS Lambda Functions [\[Gui\]](#)

2.4 Application Domains for Serverless Platforms

Serverless computing can be used in several application domains. It is particularly well-suited for applications and services driven by events, especially when they experience fluctuating or intermittent request rates. Some of these are exemplified as follows:

Compute-intensive Applications

One important domain is the compute-intensive applications including machine learning and scientific computing applications. These applications typically have finite execution periods, making the pay-as-you-go model of serverless computing an attractive choice. One relevant example is the NumPyWren system [SKV⁺20], which is designed for linear algebra tasks and built upon a disaggregated serverless programming model. Remarkably, when compared to optimized server-centric MPI implementations, NumPyWren achieved comparable performance in completion time, often within a factor of 2, while boasting up to 15% greater compute efficiency in terms of total CPU-hours.

Internet of Things (IoT)

A widely used domain for serverless computing can be found in IoT, as suggested by the literature [WKB⁺19, PDSF18]. For instance, a surveillance camera is installed in a home, for which a couple of serverless functions by the OpenWhisk [FOU20] platform are responsible for performing the camera-captured image processing. In case an object is detected by the camera system in a frame image, the serverless platform receives the frame for further processing. This could also extract certain features with the help of a serverless function and provide the results to the client [HR19].

Security

Another domain is security, more precisely for automated threat detection applications. Such an example is introduced with the help of serverless computing and Kubernetes [Kub22], where Kubernetes is used to automatically deploy and manage application containers [FH20]. The application handles the threats such as insecure configurations depending on the user-defined policies. Newly user-deployed application containers are registered with the threat scanner component, for which a scanner agent is installed. Scanner-detected threats trigger notifications to be sent to the OpenWhisk [FOU20] component activating a serverless function for further actions against the threats. Once the actions are performed, a Kubernetes API extension is invoked by OpenWhisk, which later returns the results to the client for further handling [BDK⁺17].

File Processing

File processing applications are another domain where serverless computing can be used, as seen in [ZZL19, PMCC19]. For example, Perez et al. in [PMCC19] proposed a serverless-architecture-based model for highly parallel file processing applications. There

2 Background

are different ways to process a file according to the model. One of them is by using the API gateway. User-specified files are sent via an HTTP request invoking the API gateway for calling an AWS Lambda function to process the file. An example of such a file could be the image that needs to be processed for surveillance cameras.

2.5 Embarrassingly Parallel Serverless Computation

Embarrassingly (Pleasantly) Parallel Computation is considered the ideal computation as it can be divided into a number of completely independent parts, each of which can be executed simultaneously by a separate processor. This approach to parallelizing problems requires no special techniques or algorithms, and simply involves distributing data and starting processes. There is no communication required between the separate processes, and each process needs different data and produces results from its input without any need for results from other processes. This approach results in maximum speedup and is often used when the independent parts of the computation are identical computations. The Single Program Multiple Data (SPMD) model is appropriate in such cases, and data is not shared, but copied to each process if necessary.

Nearly Embarrassingly Parallel Computation, on the other hand, requires results to be distributed, collected, and combined in some way. Initially and finally, a single process must be operating alone. The computation can be parallelized using a Master-Slave organization with either dynamic or static process creation. Even if the slave processes are all identical, we may not get the optimum solution if the processors are different. This approach differs from embarrassingly parallel computation in that communication is required between the separate processes, and the results need to be combined in some way. Nevertheless, this approach can still result in significant speedup, and techniques such as load balancing can be used to optimize the performance of the computation.

Embarrassingly serverless parallel computation is a recent extension of this concept, where cloud computing services provide a serverless platform for executing embarrassingly parallel workloads.

3 Related Work

In this chapter, we summarise several works where serverless computing model has been proven ideally suited for parallel computing tasks, and related to our work.

Occupy the Cloud: Distributed Computing for the 99%

The work [JPV⁺17] by Jonas et al. proposed that using serverless execution model with stateless functions one can build an elastic and straightforward data processing system. In this regard, they developed an open-source system, called PyWren, with Python as the programming language and AWS Lambda as the serverless provider. The approach for building a data processing system is to inject code into these stateless AWS Lambda functions dynamically. This is done in order to overcome the limitations. The model behind the approach is fairly simple: stateless functions are submitted by users to be executed in a remote container; the shared remote storage handles the state, as well as input, and output.

A claim is also made regarding the performance degradation from using such an approach being negligible for several workloads. Several applications were implemented in the paper, e.g. Terasort, using the PyWren MapReduce. Since Terasort produces a high number of intermediate files to shuffle in between, the authors argued that Amazon Simple Storage Service (Amazon S3) [aws22a] becomes a bottleneck and used AWS elastic cache, a Redis in-memory key-value store as alternative, showing that PyWren can sort 1TB data in 3.4 minutes using 1000 Lambda workers.

As for the limitations, the authors also mentioned that the PyWren model of using stateless functions with remote storage might not be a good fit for applications, such as particle simulations, requiring high coordination between long running processes.

ProPack: Executing Concurrent Serverless Functions Faster and Cheaper

Roy et al in their work [BRPL⁺23] proposed ProPack to address the scalability challenges in serverless platforms. ProPack is a solution that provides higher performance and yields cost savings for end users running applications with high concurrency.

The core concept involves packing multiple functions into a single function instance and concurrently spawning these instances. This technique efficiently reduces the "effective" number of concurrently spawned function instances, alleviating scalability bottlenecks. However, this approach introduces two key challenges. First, packing functions together can impact their individual performance as they run in parallel, necessitating a balance between scalability gains and execution time. Second, it affects cost considerations since longer execution times per function instance may increase expenses, although reducing

3 Related Work

the overall number of concurrent instances can lower costs. To navigate these challenges effectively, ProPack employs a sophisticated analytical model that determines the optimal packing degree, striking a balance between performance and cost. This model leverages insights that scaling time is independent of the executed code and optimizes co-located functions' interference, offering versatility to meet varying performance and cost objectives across different applications.

The authors evaluated their solution on various serverless platforms, including Amazon Lambda where accuracy in determining optimal packing degrees was significant, exceeding 95% in most cases. Notably, as concurrency levels increased, ProPack's effectiveness improved significantly, reducing service time by 85% on average at a concurrency level of 5000. Even accounting for the overhead in building its performance model, ProPack reduced end-user expenses by 66% at high concurrency. Compared to Pywren [JPV⁺17], ProPack outperformed with a 52% reduction in service time and a 78% reduction in expenses on average. Additionally, ProPack managed to mitigate scalability bottlenecks in the FuncX framework, a recent on-premise serverless execution platform for parallel applications [CBL⁺20], and proved its portability.

ProPack faces concerns regarding its dependence on cloud provider scalability measures and potential security issues when extended to multi-user scenarios. Future work should focus on adapting ProPack to dynamically interact with evolving cloud provider scalability solutions, addressing security challenges in multi-user environments, and exploring ways to optimize function packing for diverse function characteristics. Additionally, investigating how function packing can benefit cloud providers by improving resource utilization is a promising avenue for further research.

Stateful Serverless Computing with Crucial

Barcelona-Pons et al presented in [BPSSA⁺22] a system to program highly-parallel stateful serverless applications, called Crucial.

Crucial introduces the concept of "cloud threads," which link traditional threads to cloud function invocations, enabling the manipulation of global shared state through a distributed shared objects (DSO) layer. This DSO layer offers strong consistency guarantees and fine-grained synchronization, facilitating efficient parallel processing. Objects in DSO can be ephemeral or persistent, with the latter stored durably. Implemented using state machine replication and an efficient in-memory data store, Crucial's cloud threads can run on standard serverless platforms.

The framework provides a simple programming model, allowing multi-threaded stateful programs to execute as cloud functions through annotations and constructs. Global state management, treated as remote shared objects, becomes highly adaptable.

The authors reported in their evaluation better results than Apache Spark in case of 2 ML algorithms, and also better than storage alternatives (e.g Redis) for data sharing across cloud functions. The authors proved with Smile, a state-of-the-art multi-threaded ML library, that Crucial allowed to move traditional single-machine, multi-threaded Java programs to serverless. Crucial's strength lies in its provision of conventional multi-threaded abstractions, ensuring that only a small fraction of application code

needs modification when transitioning to the framework—less than 6%, according to the evaluation.

As for the limitations, Crucial operates over two separate layers, the FaaS platform and DSO, which could benefit from being jointly integrated into the same infrastructure for enhanced performance. Another limitation is the lack of direct communication between cloud threads, as this design choice was made to ensure compatibility with any FaaS platform. However, future work could explore the possibility of enabling direct function-to-function communication without relying on external storage. Additionally, the passage mentions the trade-off between explicit and implicit data sharing in Crucial, suggesting that further exploration in this area could lead to improvements in both programmer simplicity and system performance. Closing the gap between cloud threads and conventional threads is also identified as a potential avenue for simplifying the transition of single-machine programs to serverless environments.

From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers

Another work [ERI⁺19] by Fouladi et al. proposed an open-source framework called gg. This aims to help developers build and execute parallel applications using thousands of parallel threads on a cloud function service.

The framework consists of an intermediate representation (IR), front-end code generators, and back-end execution engines. The IR is the abstractisation of the application logic expressed by the application developers for details of placement, scheduling, as well as execution. The frontends emit gg IR and range from a C++ SDK, a Python SDK, to a group of command-line tools and a series of model substitution primitives that can infer gg IR from software build systems. As for the backends, gg IR expresses the application against an abstract machine that requires two components (united by coordinator program): an execution engine for forcing the individual thunks (ephemeral, closed containers that may contain pointers to other thunks, or output their own thunks), and a content-addressed storage. In this way, it is taken care of the execution and runtime features for dependency management, failure recovery, for example. Several applications have been used and tested on gg including video encoding. The example given for video encoding is ExCamera [FWS⁺17], which was ported into “front-end-only” version that targets gg IR.

The critical difference between gg’s backend and ExCamera’s backend is the fault tolerance provided by gg, i.e. this is achieved because the workers are warmed up, and the raw video is kept in their local filesystem.

As for the experiments, the authors used these applications ported in gg and executed their jobs on different compute engines (e.g. local machines, cloud functions, cluster of warm virtual machines). Their experimental results show higher performance. For example, running “cold” on AWS Lambda (with no preprovisioned compute resources), gg was almost 5 times faster compared to an existing solution, called icecc, running on a 48-core or 384-core cluster of warm VMs (i.e., not including the time to provision and boot the VM).

3 Related Work

The authors also mentioned several limitations, including the one that gg is bound only to CPU programs. This is because there is no mechanism in the IR to allow efficient switching between memory-mapped files and GPUs (or other accelerators), and scheduling the loading and unloading of configuration states from external resources is potentially expensive.

A programming model and middleware for high throughput serverless computing applications

In paper [PMCC19], Perez et al. proposed a High Throughput Computing Programming Model alongside an update of an existing middleware (called SCAR [PMCC18]) to create highly-parallel event-driven serverless applications.

The model uses containers which allow users to create customised runtime environments. In this way, the provider limitations imposed in their execution environments are bypassed. AWS Lambda was utilised as a framework for deploying and testing the programming model. There are several assumptions made in order to have the model stateless as the functions: a set of files has to be processed and stored in a storage service or a local machine; respectively, after the function execution, the output files will be transferred to a storage service not part of the memory storage belonging to the functions.

Two case studies were carried out to evaluate how feasible the proposed programming model is: a massive medical image analysis service, respectively a video analysis service. The results provided for the first case showed two aspects. First, AWS Lambda often outperformed the other platforms when the level of parallelism was high. Secondly, the model was not efficient enough for an increased workload due to the lack of scalability as in any other sequential models. Then, the results given for the second case showed that the corresponding function had bottlenecks. Each video was processed by exactly one invocation of this function, i.e. the more keyframes extracted, the more time the function required. Hence no improvement using the parallel model is noticeable. Worth mentioning is that the authors managed to avoid the cold start of the Lambda functions by warming the Lambda functions with an initial synchronous call.

Additionally, the authors identified numerous limitations, including the amount of memory allocated to each function invocation and, the most limiting one, the maximum execution time.

Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads

In the paper [FWS⁺17], Fouladi et al. presented a system called ExCamera for leveraging the emerging microservice frameworks to provide low-latency video processing. The authors designed a framework to run 5,000-way parallel jobs with inter-thread communication on AWS Lambda and a purely functional video codec for massive fine-grained parallelism running in this framework.

There are three essential components in ExCamera’s framework besides the Lambda platform: the coordinator long-lived server, the rendezvous long-lived center and the

workers. As for the fine-grained parallel video encoding, the insight is that the one can divide the work of video encoding into fast (parallel) and slow (serial) parts. The functions used are encode, decode, encode-given-state, and rebase. Using encode, the algorithm first encodes each chunk in parallel (with the help of work of tiny threads). Then, in a sequential mode, it rebases each output on top of the state left by the previous chunk. Finally, in order to concatenate the overall video, video-codec states must be communicated between workers.

The experimental results given by the authors show similar performance to Google's encoder vpxenc in terms of the bitrate of the compressed output and quality of the output, compared with the original. The benchmarks used were two open-source movies with the length between 12-15 minutes. Also worth mentioning is that similar results were obtained when using fine-grained parallelism (with rebasing) and coarser-grained parallelism (without rebasing).

Serverless Computing: Design, Implementation, and Performance

In work [MB17], McGrath et Brenner illustrated comparable performance enhancements achieved through parallelization as in [FWS⁺17], specifically in the context of a media management service application. This application employed parallelization techniques for image resizing, ensuring that the execution time remained consistent regardless of the number of simultaneous resize operations being conducted

Performance testing of serverless computing platforms, including AWS Lambda, Azure Functions, Google Cloud Functions, and Apache OpenWhisk, revealed significant insights. The concurrency test showcased AWS Lambda's near-linear scaling and superior throughput at 15 concurrent requests, while other platforms exhibited varying performance. In the backoff test, Azure Functions and the prototype displayed similar cold start times after a few minutes of inactivity, OpenWhisk had shorter cold start times and container deallocation after approximately 10 minutes, and AWS Lambda and Google Cloud Functions seemed less affected by function idling.

The limitations include challenges related to managing warm containers with a First-In-First-Out (FIFO) queue, synchronous execution limitations, constraints in Windows containers, and security concerns when hosting user code. Future work suggestions encompass exploring alternative strategies like "warm stacks," developing efficient asynchronous execution mechanisms, enhancing Windows container support, improving container security measures, and defining comprehensive performance metrics for assessing serverless platform performance, including network latencies, language runtimes, and CPU allocation scaling.

4 Framework for Serverless Parallel Computing

In this chapter, we present an overview of our framework, called *flexi-serverless* and implemented as a C++ static library designed for simplified local and serverless function invocations across various cloud providers (Google Cloud and AWS).

4.1 Design and Architecture

The *flexi-serverless* library is designed for simplicity, flexibility, and ease of use. Its core idea is to enable easy execution of functions, either locally or as serverless invocations, across various cloud providers. The architecture prioritizes seamless transitions and extensibility for future integrations, making serverless computing accessible and efficient for developers.

The architecture can be seen in Figure 4.1. The main architectural components consists of a C++ client application, which integrates the second component, and namely the *flexi-serverless* library, a static library which executable is called *flexi.a*. The integration provides the mean to switch The library component utilizes other two components that facilitate the serverless function invocations, last components: the AWS CLI [Ama23] for the AWS Lambda functions and GCloud [Clo23] for the Google Cloud functions.

The internal architecture of the *flexi-serverless* library including the functions used, within the AWS Lambda context, can be seen in Figure 4.2. This revolves around the *invoke* function called by the client application with the necessary parameters to invoke an AWS Lambda function. To determine which serveless platform shall be used, the *getEnv* function is called. Once this information is available, the *invokeAWS* function is triggered to streamline the invocation process by utilizing the *AWS CLI* tool. Within this function, a call to the *exec* function is made to construct a command string encompassing function details and payload for the AWS CLI tool and to execute it via the system shell. The response is captured and returned as a JSON-formatted string.

4.2 Implementation Aspects

In this section, we explore the implementation aspects in three primary contexts: library structure, creating, deploying and invoking serverless functions, respectively data transfer.

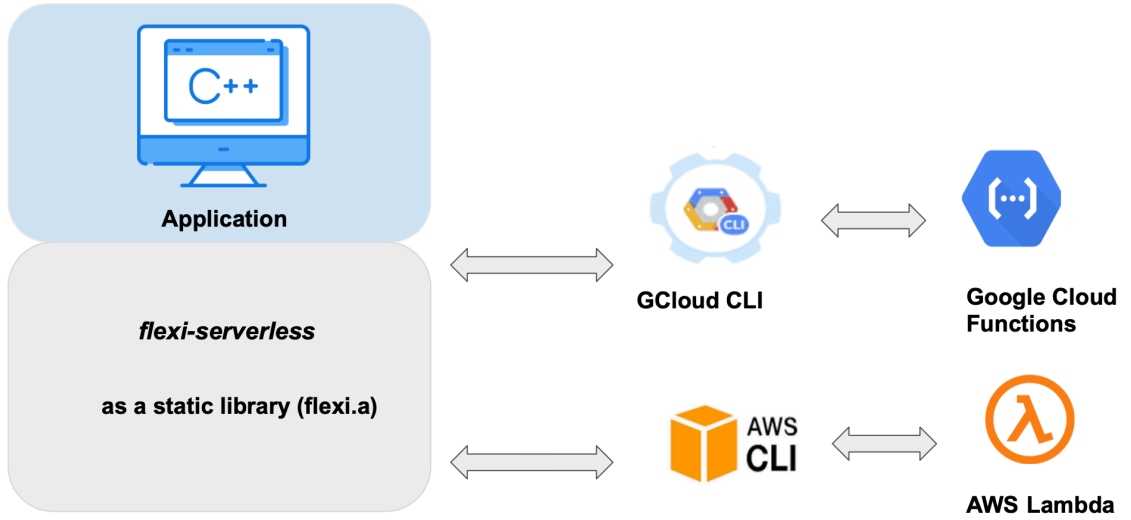


Figure 4.1: Architecture of *flexi-serverless*

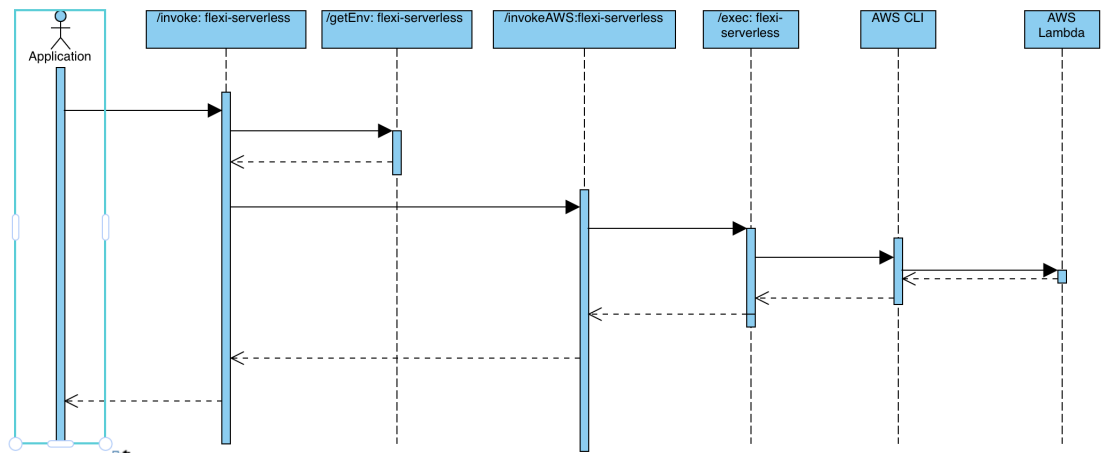


Figure 4.2: Internal Architecture of *flexi-serverless*

4.2.1 Library Structure

The *flexi-serverless* library's structure consists of .cpp file and separate .h. This approach promotes modularity and separation of concerns, making it easy for developers to integrate and use the library in their applications.

The .cpp file contains the implementation details, such as the logic for invoking serverless

functions based on the specified serverless platform and managing the communication between the application and the serverless environment.

On the other hand, the `.h` file acts as the library's interface, exposing only the essential *invoke* function to the user. This function serves as the entry point for invoking serverless functions within the client application. By keeping the implementation details separate from the interface, the library promotes clean code and enhances usability.

4.2.2 Creating, Deploying and Invoking Serverless Functions

Creating Serverless Functions

Creating a serverless function refers to creating a piece of code that can be executed on a serverless platform without managing infrastructure. This can be done generally in the following way.

1. Choose the serverless platform (e.g. Google Cloud Functions or AWS Lambda)
2. Configure the development environment, i.e. create an account and install the necessary tools.
3. Create the code for the serverless function in languages such as JavaScript (Node.js), Python, Java, Go, etc. The function should be stateless and designed to handle a specific task. For example, simple serverless functions written in Node.js on both Google Cloud Functions and AWS Lambda are shown in Listing 4.1. and Listing 4.2.

```

1 exports.helloWorld = (req, res) => {
2   res.status(200).send('Hello, Google Cloud Functions World!');
3 };
4
```

Listing 4.1: Hello word serverless function on Google Cloud Functions

```

1 exports.handler = async (event) => {
2   return {
3     statusCode: 200,
4     body: JSON.stringify('Hello, AWS Lambda World!'),
5   };
6 };
7
```

Listing 4.2: Hello word serverless function on AWS Lambda

4. Package external dependencies or libraries, if required by the function. Different platforms have various ways of handling dependencies. Depending on the serverless function and chosen language, there are different ways of handing packages.

Deploying Serverless Functions

Deploying a serverless functions refers to configuring any necessary settings for the created piece of code and making it available for execution on the serverless platform. These settings include the function name, trigger type (HTTP request, scheduled event etc.), memory allocation, timeout, environment variables, etc. Having these settings defined, the function knows how to run and respond. There are many approaches of deploying a serverless function including: SDKs, CLI tools and Infrastructure-as-a-Code (IaaS) tools. Please note that the process can be limited by the capabilities and constraints imposed by the chosen cloud provider. Each cloud provider may have its own specific limitations regarding aspects such as function size, execution time, memory allocation, and supported programming languages as mentioned in Chapter 2.

Serverless platforms such as Google Cloud Functions and AWS Lambda, offer CLI tools which allow to manage and deploy the serverless functions directly from the terminal through a manual process.

These platforms also offer SDKs, which are libraries and tools to interact with serverless platforms programmatically. This approach is useful for integrating deployment into development workflow or automating deployment pipelines. The CLI tools are GCloud [Clo23] for the Google Cloud functions, respectively [Ama23] for the AWS Lambda functions.

Infrastructure-as-Code (IaC) refers to the managing and provisioning of infrastructure through code instead of through manual processes [Red22]. There are different IaC tools that allow teams to define, configure, and manage serverless functions. This approach offers various upsides such as consistency, repeatability, version control, and improved collaboration. One popular example of a IaC tool, also used in this work, is Terraform [Ter23], which allows users to define and provision infrastructure resources in a declarative manner. By including Terraform in the framework, the user can define the serverless function's deployment configuration (e.g., timeout, memory limits, runtime, function source code), along with any associated cloud resources (e.g., IAM roles), using Terraform's domain-specific language (DSL). This configuration can be version-controlled, shared, and easily reproduced across different environments. Then, the user can then apply the defined configuration, which instructs the tool to create or modify the necessary cloud resources and deploy the serverless function. This automated provisioning process saves time and reduces the chances of configuration errors when setting up the serverless infrastructure.

Invoking Serverless Functions

Invoking a serverless function refers to triggering the function execution on a serverless platform, i.e. the function piece of code is run in response to an event (e.g. HTTP request, a schedule time etc). Invoking can be done in different ways such as by using the SDKs/CLI tools offered by the serverless platforms or other third-party frameworks including *flexi-serverless*.

In *flexi-serverless*, the exposed library function to client applications is the *invoke* function, as seen in Listing 4.3. This accepts two mandatory arguments:

- *function_name*: This is a string referring to the name of the serverless function to be invoked.
- *payload*: This is a JSON-formatted string that contains data to be passed as input to the specified serverless function.
- *index*: This is a number indexing the invocation.

The function automatically handles the cloud provider selection based on the environment variable *FLEXI_PROVIDER*, invoking the appropriate CLI tool to execute the serverless function. Finally, it returns a JSON-formatted string as the response. The response could be either an error (provider validation, errors coming from the cloud provider etc).

```

1 std::string invoke(const std::string& function_name, const std::string&
  payload, const int& index) {
2     std::string provider = getENV();
3     if (provider == "AWS") {
4         return invokeAWS(function_name, payload, index);
5     } else if (provider == "GCP") {
6         return invokeGCP(function_name, payload, index);
7     } else if (provider == "AZURE") {
8         return "Provider valid but not implemented yet";
9     } else {
10        return "Provider not found or invalid";
11    }
12 }
13

```

Listing 4.3: Invoke function

The *getEnv* function, as seen in Listing 4.4, retrieves the value of the "FLEXI_PROVIDER" environment variable. If the variable exists and is set to "AWS" or "GCP", the function returns this value as a string. Otherwise, it prints an error message and returns an empty string. In case the variable does not exist, it also prints an error message and returns an empty string

```

1 std::string getENV() {
2     const char* env_var_name = "FLEXI_PROVIDER";
3
4     const char* var_value = std::getenv(env_var_name);
5
6     if (var_value != nullptr) {
7         std::string var_value_str(var_value);
8         if (var_value_str == "AWS" || var_value_str == "GCP") {
9             return var_value_str;
10        } else {
11            std::cout << "Environment variable " << env_var_name << " is
not set to a valid value." << std::endl;
12            return "";
13        }
14    } else {
15        std::cout << "Environment variable " << env_var_name << " not
found." << std::endl;

```

4 Framework for Serverless Parallel Computing

```
16         return "";
17     }
18 }
```

Listing 4.4: getEnv function

The *exec* function, as shown in Listing 4.5, takes a single parameter, which is a C-style string representing the shell command to be executed. The function opens a pipe to the command's output using `popen` and reads each line of output into a buffer, appending it to the result string. If the command execution or pipe opening fails, the function throws a `std::runtime_error` exception. Finally, the function returns the accumulated output as a single `std::string` result.

```
1 std::string exec(const char* command) {
2     std::array<char, 128> buffer;
3     std::string result;
4     std::unique_ptr<FILE, decltype(&pclose)> pipe(popen(command, "r"),
5     pclose);
6     if (!pipe) {
7         throw std::runtime_error("popen() failed!");
8     }
9     while (fgets(buffer.data(), buffer.size(), pipe.get()) != nullptr) {
10         result += buffer.data();
11     }
12     return result;
13 }
```

Listing 4.5: Exec function

To interact with a specific cloud provider, the library utilizes the corresponding CLI tool of the chosen cloud provider, i.e. AWS CLI [Ama23] for the AWS Lambda functions and GCloud [Clo23] for the Google Cloud functions. Using CLI tools instead of linking SDKs is a valid approach, and it offers certain advantages for library development, especially concerning licensing and complexity:

- *Licensing*: SDKs often come with their own licenses, and incorporating them into a library can introduce licensing complexities. By utilizing CLI tools, which are typically command-line utilities distributed separately, the library can avoid direct linkage with the SDKs, thereby potentially sidestepping any licensing constraints associated with them.
- *Versioning and dependencies*: SDKs might have specific version requirements and dependencies, making the process of linking them into a library more complicated. By using CLI tools, the library can take advantage of the command-line utilities' stable interfaces, avoiding potential version compatibility issues.
- *Complexity*: Executing a shell command with the function name and needed payload within the library is less complex and verbose than doing the equivalent with the SDK.

For each serverless platform, there is a corresponding invocation function handling the CLI tool to execute the function call. The functions are shown in Listing 4.6, respectively Listing 4.7. For instance, the *invokeAWS* function constructs a command string for invoking the AWS Lambda function via the AWS CLI (`aws lambda invoke`) given the `function_name`, `payload` and `index` parameters. The payload is included and the response is redirected to a file named `"response[index].json"`. The function displays the constructed command in the console and employs the `system` function to execute the command within the system shell. Subsequently, the function generates a command to read the content of the `"response[index].json"` file, and utilizes the previously defined `exec` function to capture this content into a result string. The result string contains the response from invoking the AWS Lambda function and is returned by the *invokeAWS* function. As for the *invokeGCP* function, the approach is similar to the *invokeAWS* function.

```

1 std::string invokeGCP(const std::string& function_name, const std::string
  & payload, int index) {
2     std::string command = "gcloud functions call " + function_name + " --
  data '" + payload + "' >nul 2>nul";
3     std::cout << command << std::endl;
4     system(command.c_str());
5     command = "cat response" + std::to_string(index) + ".json";
6     std::string result = exec(command.c_str());
7     return result;}

```

Listing 4.6: InvokeGCP function for Google Cloud functions

```

1 std::string invokeAWS(const std::string& function_name, const std::
  string& payload, int index) {
2     std::string command = "aws lambda invoke --payload '" + payload + "'
  --function-name " + function_name + " --cli-binary-format raw-in-
  base64-out response" + std::to_string(index) + ".json >nul 2>nul";
3     std::cout << command << std::endl;
4     system(command.c_str());
5     command = "cat response" + std::to_string(index) + ".json";
6     std::string result = exec(command.c_str());
7     return result;}

```

Listing 4.7: InvokeAWS function for AWS Lambda functions

4.2.3 Data Transfer

The *flexi-serverless* library only manages in-memory payloads, which are a JSON stringified by the client application, for seamless input to each serverless function invocation. It is worth mentioning that, at present, the library does not offer the capability to specify a filename for retrieving the payload to be used as input in the function invocation.

Furthermore, the library encourages client applications to, where possible, compress data for efficient data transfer to the serverless functions. On top of that, it is important to note that the library does not perform preemptive checks to determine if a specific serverless platform's memory size limits will be exceeded before invoking the serverless function. This responsibility falls on the client application, which should ensure that data

complies with the memory constraints of the target platform to avoid potential issues during execution. For more comprehensive guidance on data handling and optimization, Chapter 2, Section 2.3 offers further insights and recommendations.

4.3 User Interface

4.3.1 Motivating Examples

Hello World

In Listing 4.8 and Listing 4.9, we provide a motivating example of how a client application can use the *flexi-serverless* library to invoke a serverless function on the AWS Lambda platform.

Firstly, the client application includes the *flexi.h* header file, which provides access to the library's interface. Behind the scene, if the clients aims to execute functions serverless, then the environmental variable *FLEXI_PROVIDER* needs to be configured to "AWS", *FLEXIUSE* preprocessor directive needs to be defined, respectively the AWS CLI tool has to be installed and configured with valid credentials.

Then, a function named *myFunction* is created to contain the logic that needs to be executed locally. This function takes a value parameter and prints a simple message concatenated with the value. Similarly, the serverless function needs to be defined on the chosen serveless platform.

If the *FLEXIUSE* is defined, the section designed for invoking the serverless function is executed. The *function_name* is set to the specific serverless function's name, a JSON-formatted payload is prepared, including the provided value, and the *nth* index of the function call order is given. The *invoke* function is then called, using the *function_name*, payload and index as parameters, to initiate the *myFunction* execution serverless. Finally, the response from the function is examined: if it indicates "Provider not found," an error message is displayed. Assuming the serverless function response is in JSON format, further actions like JSON parsing and additional processing are suggested for handling the response. In case *FLEXIUSE* is not defined, the *myFunction* function is invoked locally with the value parameter.

```

1  /* main.cpp */
2
3  #include "flexi.h"
4
5  void myFunction(std::string value) {
6      std::cout << "Hello" + value;
7  }
8
9  int main() {
10     std::string value = "World";
11
12     #ifdef FLEXIUSE
13         // Function parameters
14         int index = 1;

```

```

15     std::string function_name = "myServerlessFunction";
16     std::string payload = "{ \"key\": \"" + value + "\" }";
17
18     // Invoke the serverless function based on the provider environment
19     std::string response = invoke(function_name, payload, index);
20
21     // Process the response
22     if (response == "Provider not found.") {
23         std::cout << "Error: Invalid provider or provider not found." <<
24         std::endl;
25     } else {
26         // Process the serverless function response (assuming it's JSON)
27         // JSON parsing and further processing
28         // including error handling can be done here
29         std::cout << "Response: " << response << std::endl;
30     }
31     #else
32     myFunction(value);
33     #endif
34
35     return 0;
36 }

```

Listing 4.8: Simple example of how to use the flexi-serverless library

```

1 exports.handler = async (event) => {
2     const value = event.value
3     return {
4         statusCode: 200,
5         body: JSON.stringify('Hello ${value}'),
6     };
7 };
8

```

Listing 4.9: myFunction serverless function on AWS Lambda

Monte Carlo Integration

Another motivating and more complex example is the Monte Carlo Integration which involves estimating the integral of a function by employing random sampling. In this example, we illustrate how this application can effortlessly use the *flexi-serverless* library to invoke serverless functions for estimating the integral on the AWS Lambda platform.

In Listing 4.10 and Listing 4.11, the client code, adapted from work [McF21], and the Lambda function can be seen. In comparison to the original code, where each experiment for estimating the integral was ran sequentially, we adapted it to run multiple experiments in parallel using C++ *futures*. Each future call is parameterised with:

1. the function *monteCarloEstimateSTD*;
2. a mutex used as synchronization method for the array of experimental results;

4 Framework for Serverless Parallel Computing

3. the array of experimental results;
4. lower bound;
5. upper bound;
6. and number of iterations.

The function *monteCarloEstimateSTD* makes use of the directive *FLEXIUSE* for delimiting between the communication part with *flexi-serverless* and the actual implementation. The serverless function's payload includes the number of iterations for the particular experiment, respectively the upper bound and lower bound. The serverless function itself does the actual estimation by applying the same Monte Carlo simulation algorithm as the local execution. It returns the estimation and the standard deviation (SDT) as result, which are appended to the array of experimental results.

As in the previous example of Hello World, including *flexi.h* header file is required, as well as configuring *FLEXI_PROVIDER* to "AWS" or "local". Defining the *FLEXIUSE* preprocessor directive, respectively installing and configuring the AWS CLI tool are necessary.

```
1 #include <iostream>
2 #include <cstdlib>
3 #include <cmath>
4 #include <vector>
5 #include <future>
6 #include <mutex>
7 #include <thread>
8 #include "flexi.h"
9
10 #include <nlohmann/json.hpp>
11
12 using json = nlohmann::json;
13
14 double myFunction(double x);
15 void monteCarloEstimateSTD(std::mutex& m, std::vector<std::string>&
    results, double lowBound, double upBound, int iterations);
16
17 std::string function_name = "montecarlo-dev";
18
19 int main()
20 {
21
22     double lowerBound, upperBound;
23     int iterations;
24
25     lowerBound = 1;
26     upperBound = 5;
27
28
29     std::mutex m;
30     std::vector<std::string> results;
31     std::vector<std::future<void>> workers;
```

```

32
33     for(int i =1; i < 6; i++) {
34         iterations = 2*pow(4,i);
35
36         auto worker = std::async(monteCarloEstimateSTD, std::ref(m), std
37             ::ref(results), lowerBound, upperBound, iterations);
38         workers.push_back(std::move(worker));
39     }
40
41     for (auto& worker : workers) {
42         worker.get();
43     }
44
45     std::cout << "Printing all results: \n";
46     for (auto& result : results) {
47         std::cout << result << "\n";
48     }
49
50     return 0;
51 }
52
53 //Function to integrate
54 double myFunction(double x) {
55     return pow(x,4)*exp(-x);
56 }
57
58 void monteCarloEstimateSTDCloud(double lowBound, double upBound, int
59     iterations) {
60 }
61
62 //Function to execute Monte Carlo integration on predefined function
63 void monteCarloEstimateSTD(std::mutex& m, std::vector<std::string>&
64     results, double lowBound, double upBound, int iterations) {
65
66     #ifdef FLEXIUSE
67
68         std::string payload = "{\"iterations\": " + std::to_string(
69             iterations) + ", \"upBound\": " + std::to_string(upBound) + ", \"
70             lowBound\": " + std::to_string(lowBound) + "}";
71         std::string response = invoke(function_name, payload);
72
73         std::cout << "Response: " << response << "\n";
74         // Parse the JSON string
75         json jsonData = json::parse(response);
76         // Extract the body and parse it as JSON
77         std::string bodyString = jsonData["body"];
78         json bodyData = json::parse(bodyString);
79         // Extract the statsArray and convert it to an array of doubles
80         std::vector<double> statsArray = bodyData["statsArray"].get<std::
81             vector<double>>();
82
83     #else
84         vector<double> statsArray[2]; //position 0 holds the estimate,

```

4 Framework for Serverless Parallel Computing

```
position 1 holds the STD
79
80     double totalSum = 0;
81     double totalSumSquared = 0;
82
83     int iter = 0;
84
85     while (iter < iterations-1) {
86         double randNum = lowBound + (float(rand())/RAND_MAX) * (
upBound-lowBound);
87
88         double functionVal = myFunction(randNum);
89
90         totalSum += functionVal;
91         totalSumSquared+= pow(functionVal,2);
92
93         iter++;
94     }
95
96     double estimate = (upBound-lowBound)*totalSum/iterations;
97     double expected = totalSum/iterations;
98
99     double expectedSquare = totalSumSquared/iterations;
100
101     double std = (upBound-lowBound)*pow( (expectedSquare-pow(expected
,2))/(iterations-1) ,0.5);
102
103     statsArray[0] = estimate;
104     statsArray[1] = std;
105
106
107     #endif
108
109     std::string result = "Estimate for " + std::to_string(lowBound) + "
-> " + std::to_string(upBound) +
110                        " is " + std::to_string(statsArray
[0]) + ", STD = " + std::to_string(statsArray[1]) +
111                        ", (" + std::to_string(iterations) +
" iterations)\n";
112
113     const std::lock_guard<std::mutex> lock(m);
114     results.emplace_back(result);
115
116 }
117
118
119
```

Listing 4.10: Example of Monte Carlo Integration client application using flexi-serverless library

```
1 // Function to integrate
2 function myFunction(x) {
```

```

3     return Math.pow(x, 4) * Math.exp(-x);
4 }
5
6 exports.handler = async (event) => {
7     // Retrieve the payload from the event object
8     const payload = event;
9
10    try {
11        // Process the payload or perform any desired operations
12        console.log('Received payload:', payload);
13
14        let iterations = payload.iterations;
15        let lowBound = payload.lowBound;
16        let upBound = payload.upBound;
17
18        if (iterations === undefined || iterations === null || iterations
19            === ''
20            || lowBound === undefined || lowBound === null || lowBound === ''
21            || upBound === undefined || upBound === null || upBound === '') {
22            throw new Error('Iterations, lowBound, and upBound must be
23            defined');
24        }
25
26        const statsArray = [0.0, 0.0];
27
28        let totalSum = 0;
29        let totalSumSquared = 0;
30
31        let iter = 0;
32
33        while (iter < iterations - 1) {
34            const randNum = lowBound + Math.random() * (upBound -
35            lowBound);
36
37            const functionVal = myFunction(randNum);
38
39            totalSum += functionVal;
40            totalSumSquared += Math.pow(functionVal, 2);
41
42            iter++;
43        }
44
45        const estimate = (upBound - lowBound) * totalSum / iterations;
46        const expected = totalSum / iterations;
47
48        const expectedSquare = totalSumSquared / iterations;
49
50        const std = (upBound - lowBound) * Math.sqrt((expectedSquare -
51            Math.pow(expected, 2)) / (iterations - 1));
52
53        statsArray[0] = estimate;
54        statsArray[1] = std;
55    }
56 }

```

```

52     // Return a response
53     const response = {
54         statusCode: 200,
55         body: JSON.stringify({ statsArray }),
56     };
57     return response;
58 } catch (error) {
59     // Handle any errors that occur during processing
60     console.error('Error processing payload:', error);
61
62     // Return an error response
63     const response = {
64         statusCode: 500,
65         body: JSON.stringify({ message: error.message || 'An error
66 occurred during function execution' }),
67     };
68     return response;
69 };
70
71

```

Listing 4.11: monteCarlo serverless function on AWS Lambda

4.3.2 Installation and Configuration Process

The installation and configuration process for integrating the *flexi-serverless* library into a C++ client application involves a series of straightforward steps, enabling seamless interactions with serverless functions across diverse cloud providers. The steps are described as follows.

1. Install the necessary CLI tools.
 - For AWS, install the AWS CLI tool and configure it with valid credentials.
 - For GCP, install the GCloud tool and set it up with appropriate credentials.
2. Specify the cloud provider to be used by setting up the environmental variable *FLEXI_PROVIDER* as in Listing 4.12.

```

1     export FLEXI_PROVIDER=AWS .
2

```

Listing 4.12: Set up the environmental variable *FLEXI_PROVIDER*

3. Ensure that the library is properly linked to the executable of the application. This is done during the linking phase of the compilation process. Let us consider a C++ application that consists of a *main.cpp* file. *flexi-serverless* library can be linked as follows as in Listing 4.13.

```

1  g++ main.cpp flexi.a
2

```

Listing 4.13: Link the library to the executable of the application

4. Include the header directive "flexi.h" in the source code, as in Listing 4.14. This tells the *C++* compiler to include the contents of the library to provide the declarations and definitions necessary to use the *invoke* function.

```

1  #include "flexi.h"
2

```

Listing 4.14: Include the header directive "flexi.h"

5. Include the preprocessor directive *FLEXIUSE* in the source code, as shown in Listing 4.15. This checks if the function should be invoked serverless with the help of *flexi-serverless* library. If the *FLEXIUSE* macro is defined, it means the function of the *C++* application will be invoked on serverless and the code within the *#ifdef* block will be executed. Otherwise, the function is executed locally.

```

1  #ifdef FLEXIUSE
2      // serverless invocation
3  #else
4      // local invocation
5  #endif
6

```

Listing 4.15: Include the preprocessor directive *FLEXIUSE*

6. Call the *invoke* function to invoke the function serverless with the parameters: *function_name* as a string value and the payload as JSON-formatted string value, as in Listing 4.16.

```

1  std::string response = invoke(function_name, payload);
2

```

Listing 4.16: Invoke the serverless function

7. Process the response from the function call. This can be done by using a dedicated JSON *C++* library, as shown in Listing 4.17.

```

1      // Parse the JSON string
2      json jsonData = json::parse(response);
3      // Extract the body and parse it as JSON
4      std::string bodyString = jsonData["body"];
5      json bodyData = json::parse(bodyString);
6

```

Listing 4.17: Parsing the serverless function response

5 Evaluation

In this chapter, we present the criteria for evaluation, the application scenarios used for the evaluation and the experimental results. Through this evaluation, we aim to determine if, in each scenario, a serverless approach using our framework is more advantageous compared to local execution.

5.1 Criteria for Evaluation

We evaluated our framework efficiency in connection with scalability behaviour, cost and granularity which are quantified as follows.

5.1.1 Scalability Behaviour

A criterion to evaluate is the scalability behaviour. Scalability refers to the ability to handle an increasing workload of a given application scenario. One can divide the application scenarios into strong scaling applications scenarios and weak scaling application scenarios.

Strong scaling measures how much faster a fixed problem can be solved by increasing the number of processors. Weak scaling refers to how the execution time remains constant as the problem size increases proportionally with the number of computational resources.

In our evaluation, we are interested in the strong scalability. To evaluate strong scalability, we measure how the execution time decreases with the increase in the number of threads invoking the necessary number of functions. Specifically, our experiments include 2, 4, and 8 threads.

Local execution may have limitations in terms of the available computational resources, such as the number of CPU cores and memory capacity. Scaling up the local infrastructure can be costly and challenging. In contrast, serverless execution offers automatic scalability, allowing some functions to scale dynamically based on demand. This flexibility ensures that the application can handle varying workloads efficiently. Assessing the scalability requirements and considering the ability of our framework to handle load fluctuations will help determine the appropriate approach.

The execution time is computed as the total time required to execute all function calls, including the payload preparation necessary for each function, whether executed locally or in a serverless environment. An example of how this is done is shown in Listing 5.1, where we initialize the worker threads by creating each with the necessary payload and adding them to the workers vector. Then, we measure the execution time by capturing the start time, invoking the *get()* method on each worker to ensure completion, and finally recording the end time. Finally we compute the duration by subtracting the start time from the end time and converting the result into milliseconds.

5 Evaluation

```
1  for(int t = 0; t < threads; t++) {
2      auto worker = ...// create worker with payload
3      workers.push_back(std::move(worker));
4  }
5
6  auto startTime = std::chrono::high_resolution_clock::now();
7  for (auto& worker : workers) {
8      worker.get();
9  }
10 auto endTime = std::chrono::high_resolution_clock::now();
11 auto duration = std::chrono::duration_cast<std::chrono::milliseconds>
    >(endTime - startTime).count();
```

Listing 5.1: Measuring the execution time example

5.1.2 Cost

Another criteria to consider is the cost associated with executing functions locally versus serverless. Local execution typically involves utilizing your own hardware and infrastructure, which may require upfront investments and ongoing maintenance costs. On the other hand, serverless execution allows for a pay-as-you-go model, where you are billed based on the actual usage of resources. Evaluating the pricing structure and comparing the costs of both approaches will help determine the most cost-effective option for a specific application scenario.

In our framework evaluation, we define the following cost models for each application scenario. Executing the functions locally includes the initial investment in hardware and the ongoing operational cost of energy consumption, measured in kilowatt-hours (kWh) per hour. To estimate the energy consumption, we use the machine's power rating in watts (W) and the usage duration as in Equations 5.1 and 5.2.

$$\text{Power (kW)} = \frac{\text{Power (W)}}{1000} \quad (5.1)$$

$$\text{Energy Consumption (kWh)} = \text{Power (kW)} \times \text{Hours Used} \quad (5.2)$$

For serverless execution, we also incorporate the variable pay-as-you-go costs associated with AWS Lambda provider, which are roughly calculated based on the number of function invocations and execution time excluding the free tier of 1 million free requests per month. To estimate the cost of AWS Lambda function calls after the free tier is used up, two primary components are considered: request charged and duration charges [Bin24]. Request charges are calculated at \$0.20 per million requests, which can be defined as in Equation 5.3:

$$\text{Request Charges} = \frac{\text{Total Requests} \times 0.20}{1,000,000} \quad (5.3)$$

Duration charges depend on the execution time and memory allocation of the function. The cost is \$0.00001667 per GB-second, and the duration charges is computed as in Equation 5.4:

$$\text{Duration Charges} = \frac{\text{Total Requests} \times \text{Average Duration (ms)} \times \text{Memory (MB)} \times 0.00001667}{1000 \times 1024} \quad (5.4)$$

5.1.3 Parallelism Granularity

Parallelism granularity refers to the size of tasks and the number of tasks relative to available processors. In local execution, usually finer granularity can be achieved, allowing tasks to be divided into smaller units that can still be managed independently. However, if the tasks are too fine-grained, the overhead of managing numerous small tasks (such as scheduling and context switching) may outweigh the performance gains, leading to inefficiencies. On the other hand, coarse granularity, where tasks are fewer and larger, can result in load imbalance, as some processors may remain idle while waiting for others to complete their tasks.

In serverless execution, granularity is also influenced by the stateless nature of serverless functions, and limitations in execution time and resource allocation. Tasks may need to be divided in a way that balances the overhead of invoking multiple functions against the risk of inefficient resource usage due to underutilized processing power. Therefore, it is essential to evaluate how well tasks can be divided and parallelized in both local and serverless environments to determine the best execution model for the framework, considering potential performance losses from either too coarse or too fine task granularity.

5.2 Computational Environment

All experiments were conducted on a MacBook machine with 12 GB of RAM, based on an Intel Core i7 6-core processor, running macOS in 64-bit mode, with a power rating of approximately 96 watts(W).

For the experiments where the functions were invoked locally in each application scenario described in Sections 5.3, 5.4 and 5.5, we compiled the source code using the commands from Listing 5.2:

```
1 g++ -c main.cpp -o mainlocal.o -std=c++2b -pthread
2 g++ -Ofast mainlocal.o flexi.a -o mainlocal -std=c++2b
```

Listing 5.2: Code compilation of an application scenario running the functions locally

The `-std=c++2b` flag was used to enable features from the upcoming C++23 standard, allowing us to take advantage of modern language enhancements for performance and usability. Additionally, the `-pthread` flag enabled multithreading support, essential for

5 Evaluation

leveraging concurrent processing capabilities. The use of *-Ofast* ensured maximum optimization for local execution.

For instances where the functions were invoked in a serverless environment using the *flexi-serverless* library, we compiled the source code with the commands from Listing 5.3:

```
1 g++ -c main.cpp -o main.o -std=c++2b -pthread
2 g++ -Ofast main.o flexi.a -o main -std=c++2b
```

Listing 5.3: Code compilation of an application scenario running the functions serverless

In this setup, the *-DFLEXIUSE* flag was used to signal that the functions should be invoked serverless with the help of the *flexi-serverless*.

Furthermore, we chose AWS Lambda as the serverless platform due to its generous free tier, which offers 1 million free requests per month and 1,000,000 free compute time units. Additionally, we were granted access to the faculty account on AWS, allowing us to utilize additional resources without incurring extra costs for our work.

Our AWS Lambda functions were configured to run on the nodejs20.x runtime. There were primarily configured with the default settings, including a timeout of 5 seconds and a memory allocation of 512 MB. These configurations were generally sufficient for handling our compute-intensive, memory-light application scenarios, such as Monte Carlo Integration [5.3] and String Similarity [5.4]. However, these settings proved inadequate for the demands of Image Filtering scenario [5.5]. To accommodate the larger image sizes and substantial download transfers, we increased the memory limit to the maximum of 9,728 MB and extended the timeout to the maximum allowable 900 seconds only in the case of Image Filtering.

5.3 Monte Carlo Integration

As described in [4.3.1], in the Monte Carlo Integration application scenario, the base goal is to compute the estimation and standard deviation of the integral of a function based on the number of iterations.

For the purpose of this scenario, we varied the levels of computational load to assess the strong scalability, cost, and granularity of the serverless approach. Specifically, we tested the following combinations of iteration integrations per function invocation and the corresponding number of invocations:

(1, 10000), (50, 2000), (100, 1000), (200, 500), (500, 200).

. Each integration run maintained a fixed number of iterations, the same as the number of integration runs. Additionally, we set the lower bound of the integration range to a constant value of 100 and the upper bound to 5000.

We conducted experiments with different levels of concurrency, using 2, 4, and 8 threads to invoke the necessary serverless function calls as defined in Section [5.1]. This approach helps us to observe the performance improvements and identify potential bottlenecks as we scale the number of threads for a fixed total computational load.

The execution times and cost results for the various experiments outlined above are presented in Table [5.3].

5.3 Monte Carlo Integration

Table 5.1: Execution times and costs for different test scenarios on Monte Carlo Integration

Invocation Scenario	Threads	Invocations	Iterations	Execution time (ms)	Cost (USD)
serverless	2	200	500	260646	Request Charges: 0.00004 Duration Charges: 0.0285 Total Cost: 0.0285
local	2	200	500	110	Power Consumption: 0.0062
serverless	2	500	200	677307	Request Charges: 0.00010 Duration Charges: 0.1698 Total Cost: 0.1698
local	2	500	200	105	Power Consumption: 0.0061
serverless	2	1000	100	1481034	Request Charges: 0.00020 Duration Charges: 0.4251 Total Cost: 0.4251
local	2	1000	100	105	Power Consumption: 0.0061
serverless	2	2000	50	3131724	Request Charges: 0.00040 Duration Charges: 0.6922 Total Cost: 0.6922
local	2	2000	50	103	Power Consumption: 0.0061
serverless	2	10000	1	14232352	Request Charges: 0.00200 Duration Charges: 2.9132 Total Cost: 2.9132
local	2	10000	1	13	Power Consumption: 0.0008
serverless	4	200	500	301777	Request Charges: 0.00004 Duration Charges: 0.0576 Total Cost: 0.0576
local	4	200	500	120	Power Consumption: 0.0074
serverless	4	500	200	876489	Request Charges: 0.00010 Duration Charges: 0.1850 Total Cost: 0.1850
local	4	500	200	122	Power Consumption: 0.0075
serverless	4	1000	100	1532443	Request Charges: 0.00020 Duration Charges: 0.3935 Total Cost: 0.3935
local	4	1000	100	118	Power Consumption: 0.0074
serverless	4	2000	50	3003230	Request Charges: 0.00040 Duration Charges: 0.7108 Total Cost: 0.7108
local	4	2000	50	120	Power Consumption: 0.0074
serverless	4	10000	1	19052676	Request Charges: 0.00300 Duration Charges: 3.9076 Total Cost: 3.9076
local	4	10000	1	16	Power Consumption: 0.0010
serverless	8	200	500	534419	Request Charges: 0.00004 Duration Charges: 0.1147 Total Cost: 0.1147
local	8	200	500	149	Power Consumption: 0.0089
serverless	8	500	200	1231191	Request Charges: 0.00010 Duration Charges: 0.2325 Total Cost: 0.2325
local	8	500	200	161	Power Consumption: 0.0097
serverless	8	1000	100	2447714	Request Charges: 0.00020 Duration Charges: 0.5483 Total Cost: 0.5483
local	8	1000	100	151	Power Consumption: 0.0092
serverless	8	2000	50	4924239	Request Charges: 0.00100 Duration Charges: 1.0499 Total Cost: 1.0499
local	8	2000	50	147	Power Consumption: 0.0087
serverless	8	10000	1	18014465	Request Charges: 0.00300 Duration Charges: 3.5641 Total Cost: 3.5641
local	8	10000	1	19	Power Consumption: 0.0012

5.4 String Similarity

In the String Similarity application scenario, the objective is to compute the Levenshtein distance between two strings, which measures the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one string into the other. The scanner implementation can be seen in Listing 5.4. The function retrieves two strings from the input payload and calculates their lengths. It then initializes a dynamic programming (DP) table to store the intermediate results of the edit distances. The DP table is filled using nested loops that iterate over the characters of both strings, updating the table based on the costs of insertions, deletions, and substitutions. The results, including the computed Levenshtein distance, are returned as a JSON response.

By adjusting the granularity with minor changes for evaluation purposes, this scenario can handle different levels of task division. One could send a single pair of strings to compute their similarity, or multiple pairs in a single request, leveraging the serverless function's ability to scale and handle varying workloads.

```

1
2 exports.handler = async (event) => {
3   // Retrieve the payload from the event object
4   const payload = event;
5
6   try {
7     const str1 = payload.str1;
8     const str2 = payload.str2;
9
10    const len1 = str1.length;
11    const len2 = str2.length;
12
13    const dp = new Array(len1 + 1).fill().map(() => new Array(len2 +
14    1).fill(0));
15
16    for (let i = 0; i <= len1; ++i) {
17      dp[i][0] = i;
18    }
19    for (let j = 0; j <= len2; ++j) {
20      dp[0][j] = j;
21    }
22
23    for (let i = 1; i <= len1; ++i) {
24      for (let j = 1; j <= len2; ++j) {
25        const cost = str1[i - 1] === str2[j - 1] ? 0 : 1;
26        dp[i][j] = Math.min(dp[i - 1][j] + 1, dp[i][j - 1] + 1,
27        dp[i - 1][j - 1] + cost);
28      }
29    }
30    // Return a response
31    const response = {
32      statusCode: 200,
33      body: JSON.stringify({ distance: dp[len1][len2] }),
34    };
35    return response;
36  }
37  catch (error) {
38    // Handle error
39  }
40  }

```

```

34     } catch (error) {
35         // Handle any errors that occur during processing
36         console.error('Error processing payload:', error);
37
38         // Return an error response
39         const response = {
40             statusCode: 500,
41             body: JSON.stringify({ message: error.message || 'An error
occurred during function execution' }),
42         };
43         return response;
44     }
45 };
46

```

Listing 5.4: string-similarity serverless function on AWS Lambda

We varied the levels of computational load to assess strong scalability, cost, and granularity of the serverless approach for this scenario. The experimental setup ensures that the total number of string pairs processed remains constant at 10,000 across all serverless function invocations. Specifically, we tested the following combinations of string pairs per invocation and the corresponding number of invocations:

(100, 1000), (50, 2000), (500, 200), (200, 500).

This allowed us to assess the impact of granularity on resource utilization and performance efficiency. Furthermore, we varied the number of concurrent threads, using 2, 4, and 8 threads, to evaluate scalability and potential bottlenecks, as described in Section 5.3

For the String Similarity scenario, test data was generated using a custom C++ script (see Listing 5.5) designed to create a large number of random string pairs. The script produced 100,000 pairs of strings with a length of maximum 4 characters. These string pairs were then saved to a file for use in the experiments.

```

1  #include <iostream>
2  #include <fstream>
3  #include <vector>
4  #include <string>
5  #include <random>
6  #include <algorithm>
7
8  // Function to generate a random string of given length from the alphabet
9  std::string generate_random_string(size_t length) {
10     const std::string alphabet = "abcdefghijklmnopqrstuvwxyz";
11     std::random_device rd;
12     std::mt19937 generator(rd());
13     std::uniform_int_distribution<> distribution(0, alphabet.size() - 1);
14
15     std::string random_string;
16     for (size_t i = 0; i < length; ++i) {
17         random_string += alphabet[distribution(generator)];
18     }
19     return random_string;

```

5 Evaluation

```
20 }
21
22 // Function to generate pairs of strings, some similar or equal, others
   different
23 std::vector<std::pair<std::string, std::string>> generate_string_pairs(
   size_t num_pairs) {
24     std::vector<std::pair<std::string, std::string>> pairs;
25     std::random_device rd;
26     std::mt19937 generator(rd());
27     std::uniform_real_distribution<> distribution(0.0, 1.0);
28     std::uniform_int_distribution<> length_distribution(3, 4);
29
30     for (size_t i = 0; i < num_pairs; ++i) {
31         size_t string_length = length_distribution(generator);
32         std::string str1 = generate_random_string(string_length);
33         std::string str2;
34
35         if (distribution(generator) < 0.5) {
36             if (distribution(generator) < 0.5) {
37                 str2 = str1; // Equal
38             } else {
39                 // Similar: change a few characters
40                 str2 = str1;
41                 std::uniform_int_distribution<> char_distribution(0,
string_length - 1);
42                 for (size_t j = 0; j < std::uniform_int_distribution<>(1,
3)(generator); ++j) {
43                     str2[char_distribution(generator)] =
generate_random_string(1)[0];
44                 }
45             }
46         } else {
47             str2 = generate_random_string(string_length); // Completely
different
48         }
49         pairs.emplace_back(str1, str2);
50     }
51     return pairs;
52 }
53
54 // Function to write pairs of strings to a file, each pair on a new line
   separated by a space
55 void write_pairs_to_file(const std::vector<std::pair<std::string, std::
string>>& pairs, const std::string& filename) {
56     std::ofstream file(filename);
57     if (file.is_open()) {
58         for (const auto& pair : pairs) {
59             file << pair.first << " " << pair.second << "\n";
60         }
61         file.close();
62     } else {
63         std::cerr << "Unable to open file: " << filename << std::endl;
64     }
```

```
65 }  
66  
67 int main() {  
68     size_t num_pairs = 100000;  
69     std::vector<std::pair<std::string, std::string>> pairs =  
70         generate_string_pairs(num_pairs);  
71  
72     // Write the pairs to a file  
73     write_pairs_to_file(pairs, "string_pairs.txt");  
74  
75     return 0;  
76 }
```

Listing 5.5: Test data generation for String Similarity application scenario

The execution times and cost results for the various experiments outlined above are presented in Table [5.2](#)

5 Evaluation

Table 5.2: Execution times and costs for different experiments on String Similarity

Invocation Scenario	Threads	Invocations	Iterations	Execution time (ms)	Cost (USD)
serverless	2	200	500	0	Total Cost: 0
local	2	200	500	0	Power Consumption: 0.0062
serverless	2	500	200	0	Total Cost: 0
local	2	500	200	0	Power Consumption: 0.0062
serverless	2	1000	100	0	Total Cost: 0
local	2	1000	100	0	Power Consumption: 0.0062
serverless	2	2000	50	1823	Request Charges: 0.00040 Duration Charges: 0.0285 Total Cost: 0.0289
local	2	2000	50	0	Power Consumption: 0.0061
serverless	4	200	500	0	Total Cost: 0
local	4	200	500	0	Power Consumption: 0.0074
serverless	4	500	200	1932	Request Charges: 0.00010 Duration Charges: 0.1850 Total Cost: 0.1851
local	4	500	200	0	Power Consumption: 0.0075
serverless	4	1000	100	0	Total Cost: 0
local	4	1000	100	0	Power Consumption: 0.0074
serverless	4	2000	50	0	Total Cost: 0
local	4	2000	50	0	Power Consumption: 0.0074
serverless	8	200	500	1912	Request Charges: 0.00004 Duration Charges: 0.1147 Total Cost: 0.11474
local	8	200	500	3	Power Consumption: 0.0089
serverless	8	500	200	9422	Request Charges: 0.00010 Duration Charges: 0.2325 Total Cost: 0.2326
local	8	500	200	0	Power Consumption: 0.0097
serverless	8	1000	100	10922	Request Charges: 0.00020 Duration Charges: 0.5483 Total Cost: 0.5485
local	8	1000	100	0	Power Consumption: 0.0092
serverless	8	2000	50	12627	Request Charges: 0.00100 Duration Charges: 1.0499 Total Cost: 1.0509
local	8	2000	50	0	Power Consumption: 0.0087

5.5 Image Filtering

In the Image Filtering application scenario, the objective is to process an image to count the number of black and white pixels. The serverless function deployed on AWS Lambda can be seen in Listing 5.6.

The function retrieves the URL of the image from the input payload and reads the image. Once the image is loaded, the function initializes two counters, `blackPixels` and `whitePixels`, to zero. Due to serverless functions' limitations on data transfers, as described in Sections 2.3 and 4.2.3, we opted to send image URLs rather than transferring the entire image data directly. This approach allowed us to work within the constraints of the Lambda environment by minimizing data transfer requirements while still enabling effective processing of the images within the serverless functions.

The image is then scanned pixel by pixel using Jimp's `scan` method, which iterates over the image dimensions (width and height). For each pixel, the RGBA values are extracted from the image buffer. The red, green, and blue values are checked to determine if they are equal, which indicates a grayscale pixel. If the grayscale value is 0, the pixel is counted as white; if the grayscale value is 255, the pixel is counted as black. The results, including the counts of black and white pixels, are returned as a JSON response.

5 Evaluation

```
1 const Jimp = require("jimp");
2
3 exports.handler = async (event) => {
4   const imageUrl = event.image;
5   // Assuming the image URL is passed as 'imageUrl' in the event object
6   try {
7     const image = await Jimp.read({
8       url: imageUrl, // Required!
9       headers: {}},
10
11     })
12     let blackPixels = 0;
13     let whitePixels = 0;
14
15     image.scan(0, 0, image.bitmap.width, image.bitmap.height,
16     function (x, y, idx) {
17       // x, y is the position of this pixel on the image
18       // idx is the position start position of this rgba tuple in
19       // the bitmap Buffer
20
21       var red = this.bitmap.data[idx + 0];
22       var green = this.bitmap.data[idx + 1];
23       var blue = this.bitmap.data[idx + 2];
24       var alpha = this.bitmap.data[idx + 3];
25
26       // rgba values run from 0 - 255"
27       // e.g. this.bitmap.data[idx] = 0; // removes red from this
28       pixel
29       if (red === green && green === blue) {
30         if (red === 0) {
31           whitePixels++;
32         } else if (red === 255) {
33           blackPixels++;
34         }
35       }
36     });
37     return {
38       statusCode: 200,
39       body: JSON.stringify({ blackPixels, whitePixels }),
40     };
41   } catch (error) {
42     return {
43       statusCode: 500,
44       body: 'Error processing image',
45     };
46   }
47 }
```

Listing 5.6: Serverless myFunction on AWS Lambda

By adjusting the granularity with minor changes for evaluation purposes, this scenario can handle different levels of task division. One could send a single image URL to compute

the number of black and white pixels, or multiple image URLs in a single request. We tested the following combinations of images per invocation and the corresponding number of invocations:

(5, 2000), (10, 1000), (20, 500), (50, 200).

Furthermore, we varied the number of concurrent requests, using 2, 4, and 8 threads, as described in Section 5.3.

As test data, we used a single test image from the following URL [ets24]. This image was downloaded and saved locally. For serverless function invocations, the URL was provided multiple times in the function payload, allowing the function to download and process the image as needed. For local function invocations, the saved image was read from the local storage and processed repeatedly, according to the number of images specified per function call. This ensured consistency across tests while enabling a comparison of scalability and performance between serverless and local processing.

The execution times and cost results for the various experiments outlined above are presented in Table 5.3.

5 Evaluation

Table 5.3: Execution times and costs for different experiments on Image Filtering

Invocation Scenario	Threads	Invocations	Images	Cost (USD)
serverless	2	1000	2	Request Charges: 0.00040 Duration Charges: 2.5121 Total Cost: 2.5125
local	2	1000	2	Power Consumption: 0.0062
serverless	2	2000	1	Request Charges: 0.00080 Duration Charges: 4.6550 Total Cost: 4.6558
local	2	2000	1	Power Consumption: 0.0064
serverless	4	1000	2	Request Charges: 0.00040 Duration Charges: 2.4505 Total Cost: 2.4509
local	4	1000	2	Power Consumption: 0.0078
serverless	4	2000	1	Request Charges: 0.00080 Duration Charges: 4.3161 Total Cost: 4.3169
local	4	2000	1	Power Consumption: 0.0074
serverless	8	1000	2	Request Charges: 0.00040 Duration Charges: 2.3786 Total Cost: 2.3790
local	8	1000	2	Power Consumption: 0.0080
serverless	8	2000	1	Request Charges: 0.00080 Duration Charges: 3.9727 Total Cost: 3.9735
local	8	2000	1	Power Consumption: 0.0084

5.6 Summary

Based on the experimental results, one can conclude that the local approach generally outperforms the serverless approach in terms of execution times. For example, when using 2 threads with 200 invocations and 500 iterations in the Monte Carlo Integration scenario, the serverless execution time is 260,646 ms, whereas the local execution time is only 110 ms. This trend continues across various configurations, highlighting that local execution is much faster than serverless execution.

One reason for this significant time difference is the choice of programming languages used for implementing the application client and AWS Lambda functions. The serverless implementation of AWS Lambda leverages the nodejs20.x runtime, which executes JavaScript. For example, as explained in work [Oga14], JavaScript’s dynamic typing

allows variables to hold any type, which is resolved only at runtime. This flexibility adds overhead, as the system must determine the type and appropriate operation each time. Conversely, C's static typing assigns fixed types at compile time, allowing more efficient, type-specific code execution without runtime type resolution. Moreover, C++ is a compiled language, so it avoids runtime source code parsing and does not require just-in-time compilation compared to JavaScript.

Another reason is related to the possible cold start delays of the serverless functions, occurring when a new instance of the function must be initialized, which can increase response times significantly. Local executions, by contrast, do not face cold start issues, as the environment remains available throughout the processing task, ensuring consistent performance without initialization delays.

Additionally, in the serverless context, preparing data as payloads for Lambda functions and transferring it over the network introduces additional overhead. In the case of Image Filtering application, while the process is minimized by sending URLs instead of full image data, the time required for these steps can still affect performance. In contrast, the local approach handles images directly from local storage, avoiding network latency and reducing the time spent in data handling.

From the view of thread impact on the performance, we can say that increasing the number of threads from 2 to 8 generally leads to increased execution times for serverless functions, with notable exceptions when the number of invocations is low (e.g., 1 invocation with 10,000 iterations in the Monte Carlo Integration scenario). For instance, with 8 threads and 200 invocations at 500 iterations, the execution time is 534,419 ms, compared to 301,777 ms for 4 threads under the same scenario.

The cost associated with serverless execution increases with the number of invocations and workload. For example, in the String Similarity scenario, the cost for 1,000 invocations and 100 string pairs per invocation with 2 threads is 0.0289, while it slightly increases to 0.5483 with 8 threads for the same invocation count. Conversely, local execution costs remain relatively low and consistent. The power consumption for local function invocations remains consistently low (ranging from 0.0008 to 0.0097), further emphasizing the efficiency of local processing for computational tasks compared to the escalating costs associated with serverless execution.

In the case of the String Similarity scenario, several instances executed all functions, both in local and serverless environments, in a swift time, resulting in recorded execution times of zero milliseconds. This leads to the assumption that the functions completed faster than the precision limit of our timing mechanism. There are several factors that could contribute to this: high efficiency of the functions, minimal overhead from resource allocation, how granular the timing mechanism is, and effective way of handling the concurrent executions.

6 Conclusions and Future Work

Even though there are specific challenges in serverless computing, the authors (researchers and engineers) of the analysed papers have attempted to overcome them, with each attempt being more successful than the last. However, as is the case with such challenges, there will always be room for improvement. In the case of PyWren [JPV⁺17] and ExCamera [FWS⁺17], where massively parallel computations were demonstrated, the solutions required their own ad-hoc external orchestration services for synchronising the parallel execution of functions. For instance, ExCamera relied on an external rendezvous server to synchronise the parallel executions. Furthermore, intra-communication between functions still needs to be improved, as shown by Fouladi et al. [FRI⁺19], who used intermediate functions or external coordinators that serve as brokers between functions. Additionally, the programming model proposed by Perez et al. [PMCC19] is practical for file processing applications.

Our work has critically examined the trade-offs between serverless computing and traditional on-premise/local computing for three compute-intensive application scenarios: Monte Carlo Integration, String Similarity and Image Filtering. The evaluation revealed that using a serverless approach for the three chosen application scenarios, Monte Carlo Integration, String Similarity and Image Filtering, is not a performant solution for predictable workloads or resource-intensive tasks.

The development of the *flexi-serverless* framework offers however a significant contribution to this field, simplifying the invocation of functions in both serverless and local environments. This C++ static library enables seamless transitions between cloud providers, promoting extensibility for future integrations. Additionally, the framework demonstrates its versatility through three application scenarios suited for compute-intensive parallel tasks.

Our experimental results emphasize the importance of aligning the choice between serverless and local computing with the specific needs of applications. A hybrid approach may lead to better outcomes, allowing developers to leverage the strengths of both environments.

As future work, we plan to conduct additional tests using the same programming language for both serverless functions and local executions to isolate the effects of runtime differences on performance. Furthermore, we intend to perform an in-depth performance analysis with various programming languages, similar to the empirical study conducted by [RMMR24], which evaluated how language selection impacts CRUD operations in a REST API on AWS Lambda. Their study used automated tests across Python, JavaScript with Node.js, and Java, highlighting the performance variations due to language choice. This approach allows us to further understand how programming language selection affects serverless function performance.

Bibliography

- [Ama23] Amazon. Aws cli. 2023.
- [aws22a] Amazon s3. 2022.
- [aws22b] Aws lambda, 2022.
- [AWS22c] Aws serverless application repository. 2022.
- [azu] How can i use gpu with azure function? 1579.
- [azu23] Azure functions, 2023.
- [BDK⁺17] Nilton Bila, Paolo Dettori, Ali Kanso, Yuji Watanabe, and Alaa Youssef. Leveraging the serverless architecture for securing linux containers. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 401–404, 2017.
- [BPSSA⁺22] Daniel Barcelona-Pons, Pierre Sutra, Marc Sánchez-Artigas, Gerard París, and Pedro García-López. Stateful serverless computing with crucial. *ACM Trans. Softw. Eng. Methodol.*, 31(3), mar 2022.
- [BRPL⁺23] Rohan Basu Roy, Tirthak Patel, Richmond Liew, Yadu Nand Babuji, Ryan Chard, and Devesh Tiwari. Propack: Executing concurrent serverless functions faster and cheaper. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, pages 211–224, 2023.
- [CBL⁺20] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. Funcx: A federated function serving fabric for science. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, pages 65–76, 2020.
- [clo22a] Amazon cloudwatch. 2022.
- [clo22b] Aws cloudtrail. 2022.
- [Clo23] Google Cloud. Gcloud. 2023.
- [Cou22] Couchdb. 2022.
- [Doc13] Docker, Inc. Docker compose, 2013.

Bibliography

- [doc22] Docker, 2022.
- [Ein24] Eran Einav. Escaping aws lambda costs. 2024. Accessed: 2024-10-08.
- [ets24] Image for image filtering application scenario, 2024. Accessed: 2024-10-06.
- [FH20] Dayong Fan and Dongzhi He. A scheduler for serverless framework base on kubernetes. In *Proceedings of the 2020 4th High Performance Computing and Cluster Technologies Conference; 2020 3rd International Conference on Big Data and Artificial Intelligence*, HPCCT amp; BDAI '20, page 229–232, New York, NY, USA, 2020. Association for Computing Machinery.
- [FOU20] THE APACHE SOFTWARE FOUNDATION. Apache openwhisk, 2020.
- [FRI⁺19] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association.
- [FWS⁺17] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, page 363–376. USENIX Association, 2017.
- [git23] Github, 2023.
- [Goo] Google. <https://cloud.google.com/functions/docs/tutorials/ocr>.
- [goo22] Google cloud functions, 2022.
- [Gui] AWS Lambda Developer Guide. Processing a kinesis stream with database resources in a vpc.
- [HR19] Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, IoTDI '19, page 225–236. Association for Computing Machinery, 2019.
- [JPV⁺17] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 445–451, New York, NY, USA, 2017. Association for Computing Machinery.
- [Kaf22] Kafka. 2022.

- [KS18] Kyriakos Kritikos and Paweł Skrzypek. A review of serverless frameworks. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 161–168, 2018.
- [kub22] Kubernetes, 2022.
- [MB17] Garrett McGrath and Paul R. Brenner. Serverless computing: Design, implementation, and performance. *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410, 2017.
- [McE21] Cameron McElfresh. Monte carlo integration, 2021. Medium.
- [MG11] Peter M. Mell and Timothy Grance. Sp 800-145. the nist definition of cloud computing. Technical report, 2011.
- [ngi22] Nginx, 2022.
- [Oga14] T. Ogasawara. Workload characterization of server-side javascript. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 13–21, Raleigh, NC, 2014. IEEE.
- [opea] Openfaas deployment.
- [opeb] Openfaas stack.
- [opec] Openshift.
- [oped] Openwhisk documentation.
- [Opee] OpenFaaS Community. Openfaas.
- [PDSF18] Duarte Pinto, João Pedro Dias, and Hugo Sereno Ferreira. Dynamic allocation of serverless functions in iot environments. In *2018 IEEE 16th International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 1–8, 2018.
- [PMCC18] Alfonso Pérez, Germán Moltó, Miguel Caballer, and Amanda Calatrava. Serverless computing for container-based architectures. *Future Generation Computer Systems*, 83:50–59, 2018.
- [PMCC19] Alfonso Pérez, Germán Moltó, Miguel Caballer, and Amanda Calatrava. A programming model and middleware for high throughput serverless computing applications. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC '19*, page 106–113, New York, NY, USA, 2019. Association for Computing Machinery.

Bibliography

- [QCL21] Sterling Quinn, Robert Cordingly, and Wes Lloyd. Implications of alternative serverless application control flow methods. In *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*, WoSC '21, page 17–22, New York, NY, USA, 2021. Association for Computing Machinery.
- [Red12] Red Hat, Inc. Ansible, 2012.
- [Red22] RedHat. What is infrastructure as code (iac)? 2022.
- [Reh21] Taavi Rehemägi. The origins of serverless. 2021.
- [RMMR24] Nelson Rodríguez, María Murazzo, Adriana Martín, and Matías Rodríguez. Check for updates evaluation of programming languages for memory usage, scalability, and cold start, on aws lambda serverless platform as a case study. In *Computer Science–CACIC 2023: 29th Argentine Congress of Computer Science, Lujan, Argentina, October 9-12, 2023, Revised Selected Papers*, volume 2123, page 33. Springer Nature, 2024.
- [Row11] Dan Rowinski. Mobile backend as a service parse raises 5.5 million in series a funding. 2011.
- [SKV⁺20] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 281–295, New York, NY, USA, 2020. Association for Computing Machinery.
- [ter23] Terraform. 2023.
- [Tho17] M. Thommes. Squeezing the milliseconds: How to make serverless platforms blazing fast!, April 2017.
- [WCL22] Jinfeng Wen, Zhenpeng Chen, and Xuanzhe Liu. A literature review on serverless computing, 06 2022.
- [WKB⁺19] Rich Wolski, Chandra Krintz, Fatih Bakir, Gareth George, and Wei-Tsung Lin. Cspot: Portable, multi-scale functions-as-a-service for iot. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, SEC '19, page 236–249. Association for Computing Machinery, 2019.
- [WLZ⁺18a] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, July 2018. USENIX Association.

- [WLZ⁺18b] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, July 2018. USENIX Association.
- [WW21] Stefan Winzinger and Guido Wirtz. Data flow testing of serverless functions. In *CLOSER*, pages 56–64, 2021.
- [YBLW19] Vladimir Yussupov, Uwe Breitenbücher, Frank Leymann, and Michael Wurster. A systematic mapping study on engineering function-as-a-service platforms and tools. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing, UCC’19*, page 229–240, New York, NY, USA, 2019. Association for Computing Machinery.
- [ZZZL19] Miao Zhang, Yifei Zhu, Cong Zhang, and Jiangchuan Liu. Video processing with serverless computing: A measurement study. In *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video, NOSSDAV ’19*, page 61–66, New York, NY, USA, 2019. Association for Computing Machinery.

