DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master Thesis

Defining SLOs for FaaS Functions in Serverless Computing

Gor Safaryan

DEPARTMENT OF INFORMATICS

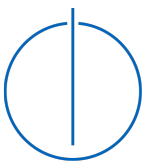
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master Thesis

Defining SLOs for FaaS Functions in Serverless Computing

Definieren von SLOs für FaaS-Funktionen im Serverless Computing

Author:	Gor Safaryan
Supervisor:	Prof. Dr. Michael Gerndt
Advisor:	MSc. Anshul Jindal
Submission Date:	15th September 2021



I confirm that this master thesis is my own work and I have documented all sources and material used.

Munich, 15th September 2021 Gor Safaryan

Acknowledgments

I want to thank M.Sc. Anshul Jindal, who supervised and mentored me during the whole research project. I am really grateful for the time he invested in our collaboration and in the project itself. It has been my pleasure to attend every meeting and discuss the results we have achieved every week. I really appreciate not only his engagement in the process, but also his friendly and ready to help attitude.

I also thank Prof. Dr. Michael Gerndt for hosting the research in his chair and allowing me to work on this topic as my master thesis.

Abstract

Serverless computing paradigm has become more ingrained into industry, as it offers a cheap alternative for application development and deployment. This new paradigm has also created new kinds of problems for the developer, who needs to tune memory parameters balancing cost and performance. Many researchers have addressed the issue of minimizing cost and meeting Service Level Objective (SLO) requirements for a single cloud function, but there has been a gap for solving the same problem for an application consisting of many cloud functions, which create a complicated tree of function calls.

In this work, we designed a python based tool called SLAM to address the issue. SLAM uses distributed tracing to detect the relationship among the cloud functions within a serverless application and by modeling each of them, it estimates the duration for an application call at different memory configurations. Using these estimations SLAM uses different optimizers to find the best configuration given the optimization constraint (SLO, minimal cost, or overall execution time). We demonstrate the functioning of our tool on AWS Lambda by testing on multiple applications. Our results have shown that the suggested memory configurations guarantee that more than 95% of requests are completed within the predefined SLO.

Index Terms— serverless, cost optimization, memory optimization, SLO, serverless applications

Contents

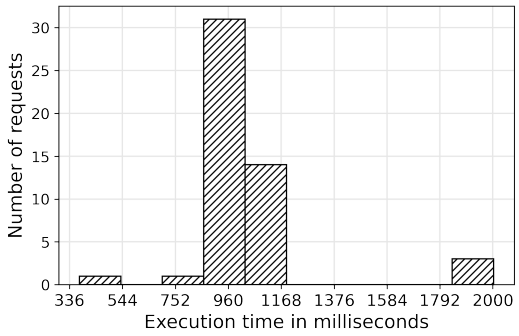
1 Introduction

Significant progress has been made in different domains [17, 14, 47, 31, 27] based on the idea of *serverless computing* since its launch by Amazon as AWS Lambda in November 2014 [10]. Serverless computing is a cloud computing model that abstracts server management and infrastructure decisions away from the users [51]. In this model, the allocation of resources is managed by the cloud service provider rather than by *DevOps*, thereby benefiting them from various aspects such as no infrastructure management, automatic scalability, and faster deployments [19, 43]. Function-as-a-Service (FaaS) is a key enabler of serverless computing [51]. In FaaS, a serverless application is decomposed into simple, standalone functions that are uploaded to a FaaS platform such as AWS Lambda [9], Google Cloud Function (GCF) [18], and Azure Functions (AF) [5] for execution. Most of the public cloud providers in their FaaS offerings allow users to configure memory allocation for the functions [4, 18, 11].

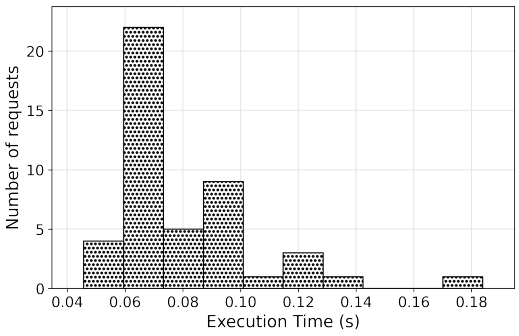
Despite having many advantages, serverless computing suffers from some pain points that obstruct its wide adoption [12, 22]. The most commonly known is optimally configuring the memory of the FaaS functions within the application based on the required the Service Level Objective (SLO). The difficulties lie in the following aspects:

Cold start: It is mainly connected with loading the FaaS function into the main memory of the executing server and preparing the execution environment for the target code (starting up the VM/container, loading libraries, loading function code, etc.) [37, 13]. The cold start phenomenon combined with the heterogeneity of the cloud environment makes the function execution time quite unpredictable. Figure 1.1a shows an execution time distribution for a sample compute-intensive function having a high variance when deployed with 128MB memory configuration on AWS Lambda.

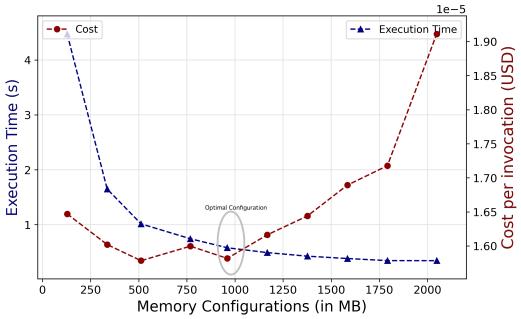
FaaS functions integration with BaaS services: The FaaS functions are usually closely integrated with other services, e.g., cloud databases, authentication and authorization services, and messaging services. These services are called Backend-as-a-Service (BaaS) [33]. These services also do influence the execution time of the FaaS functions, thus adding the variance in the time. Figure 1.1b shows an execution time distribution for a sample function querying DynamoDB having a high variance when deployed with 128MB memory configuration on AWS Lambda.



(a) Execution time variance due to cold start problem.



(b) Execution time variance due to BaaS service (DynamoDB).



(c) Performance vs cost trade for finding the optimal configuration.

Figure 1.1: Various factors making it difficult to optimally configure the memory of the FaaS functions within a serverless application.

Trade-off analysis between performance and cost: Users need to define memory configuration for their FaaS functions: a low-level information which directly influences the performance and cost of the serverless application [9, 48, 50]. Thus the user has to do a trade-off analysis between them to define the right configuration for their required SLOs [24]. Figure 1.1c shows an execution time vs the cost graph for a sample compute-intensive function when deployed with different memory configurations on AWS Lambda. We can observe that it's not trivial to find the optimal configuration where the overall cost and execution time are both optimal.

Complex application workflows: Usually, the serverless applications comprise dozens if not hundreds of small FaaS functions and these connect together to form complex event-driven workflows. Furthermore, the SLOs are usually defined at the

application level instead of the function level and thus based on the required application SLOs configuring the memory of the FaaS functions within the application even becomes more challenging since a change in one can influence the others.

In this case the whole system is exactly the sum of the parts. Translating the application SLO-s to lower level components such as the cloud functions would be enough to satisfy the whole application SLO. Currently if one wants to tune the application to meet certain SLO requirements, then the only way to approach the problem is to tune individual functions. Thus it would really help to translate the overall application SLO to individual functions.

The aspects above highlight some factors that make it difficult for the users to optimally configure memory for serverless applications based on the required SLOs. However, there are many other factors such as I/O and network bandwidth, and co-location with other functions affecting the performance and cost which the users are not aware of [50]. Many researchers have addressed the issue of optimizing the memory and cost for meeting SLO requirements for a single cloud function [1, 23, 21], but there has been a gap for solving the same problem for a serverless application consisting of many FaaS functions, which create a complicated workflow of function calls. To this end, we develop **SLAM**, a python-based tool that can automatically find the optimal memory configurations for the FaaS functions within the given serverless application based on the specified SLO. Our key contributions are as follows:

- We develop and present a novel tool called **SLAM** that automatically determines the optimal memory configuration for the FaaS functions within the given serverless application based on the specified SLO requirements (§3). To the best of our knowledge, this is the first work that find and configure FaaS functions with optimal memory configurations within a serverless application based on the specified SLO.
- We propose and implement an optimization algorithm along with its variants for various optimization objectives (minimum cost and minimum overall time) in addition to the SLO requirements in finding the optimal memory configuration for the given serverless application (§23).
- Although our approach is generic and *SLAM* can be easily extended to support other commercial and open-source FaaS platforms, we demonstrate the functionality of *SLAM* with AWS Lambda (§5) on four serverless applications comprising of various number of functions.
- We evaluate the performance of the *SLAM* on 3 different aspects: 1) Estimation time accuracy (§5.1), 2) Configuration finding accuracy (§5.2), and 3) Configu-

ration finding efficiency and scalability of *SLAM* (§5.3). From the experimental evaluation, the suggested memory configurations guarantee that more than 95% of requests are completed within the defined SLOs.

2 Background

2.1 Function as a service

2.1.1 Overview of cloud FAAS

In essence, FAAS(Function as a service) is a cloud computing service that abstracts away from the developer the infrastructure details to deploy, run and debug applications. It helps the user seamlessly upscale and downscale the application without worrying about the messy details regarding the types and the sizes of the servers.

FAAS offering is also commonly known as serverless computing, but don't be misled by the term. Serverless doesn't indicate that no actual servers are running under the hood. The term only suggests that there is no need to manage those, and all the complexities are handled by the provider.

It provides a neat API where the user can upload the functions in most of the popular programming languages such as NodeJS, Go, Java, Python. Most of the cloud providers also allow uploading a docker image which extends the possibilities for the choice of programming language. The underlying runtime environment will run the function as soon as it is triggered.

Application architecture

FAAS also imposes a certain type of architecture for the application, which might not be suitable for every task at hand. In essence, FAAS expects the developer to structure the application into small, fully functional units which can be run separately. This architecture is well known as "microservice architecture", but when developing for a FAAS platform one might need to take the process to an extreme, where the functional units are functions themselves. Generally when developing a serverless application, one might need to change the perspective from traditional server design to an event-driven design. An application developer usually writes the function in response to some "external event" such as receiving a request from a user, a delivery of a message in a queue, or an expiry of a timer. Such events trigger an action from the cloud provider which in turn runs your function in response to that event. The functions are only run as a response to an external event, which is usually defined by the developer.

Stateless

The serverless platform also has its limitations and the most noteworthy of them is that functions are stateless. They do not share any information with the previous invocation or with other functions. Nevertheless, the developer can use other external mechanisms for communication such as databases, queues, etc. It is expected that all the state necessary for the function is present as an input or can be extracted from an external source.

Most developers are accustomed to programming stateful web applications as traditional backend servers are stateful. Being stateful is quite natural and easy to program, but statelessness enables the cloud provider to seamlessly scale cloud functions. Stateless functions don't need to synchronize their states in different servers, in between function calls and there is no need to scale the infrastructure for bookkeeping. So one can see that the stateless nature is a blessing as much as it is a curse.

Latency

As explained in previous paragraphs, not all applications are meant for FAAS platforms. FAAS platforms being stateless impose a certain structure that cannot fit every application. Yet another downside for FAAS platforms is the relatively high latency and high variance in execution time. When the application downscales to zero the subsequent invocations of the function usually take longer than normally would. This phenomenon is widely studied in FAAS literature and is commonly referred to as a cold start. [37, 13]. This has to do with the fact that the function image needs to be loaded to the RAM of the server, the context needs to be established and some libraries might need some time for initialization. Another factor, which contributes to the latency and execution time variance, is the unpredictability of the execution environment. Cloud functions can not only be relocated from one CPU core to another but also another server. Such movements in the cloud make CPU caching less efficient. Also, different servers can have different types of CPUs thus contributing to the high variance of execution time.

Execution model

Sometimes cloud providers differentiate between pull and push execution models, but in essence, both of them work on the same principle. In the pull execution model, the function polls on a data source to check if any new record is added. In a push model, the function is executed once an event is registered. Both are executed in a response to the event, but the pull model has a built-in mechanism, which continually internally checks on the source for a new record and when it arrives creates an event for the function.

Invocation type

Another important aspect to consider, when calling such a serverless function is the invocation type. Usually, cloud providers allow the user to specify synchronous, asynchronous, and dry-run modes of invocation. When invoking a serverless function in the synchronous mode the client keeps the connection open and waits for a response or times out. The response usually contains metadata regarding the call and, in case the call was successful, the return value from the function. In case the function is called in the asynchronous mode, the client doesn't wait for a response. If the function is unsuccessfully called multiple times, the calling event is stored in a dead letter queue. When calling the function in a dry-run mode, the function is not executed, but only the input parameters and permissions are checked for possible execution.

Error handling and retry behavior

Naturally, errors can occur in any type of application and serverless applications are no exception. Most cloud providers have their own error codes for various errors that can happen, ranging from memory overflows to permission errors. Logging and monitoring solutions can help the developers to dig deep into the problems and find the appropriate solutions. Nonetheless, sometimes the problems are transient and can be alleviated by a simple retry. That is why most serverless functions have built-in retry mechanisms. For the synchronous mode of execution, the cloud provider leaves the retry logic to the client that initiated the request. It should decide on its own how to handle a certain error as it keeps the connection open and waits for the result of the function. For the asynchronous mode of execution, the user can specify a retry behavior for the function, in case the call fails. The function call can be retried automatically a couple of times until it is finally considered that the event has failed. The user can also specify a queue for such events to be stored for later analysis which usually is referred to as a dead letter queue or topic depending on a provider.

Fault tolerance

One of the advantages of using cloud functions is the built-in fault tolerance that comes from the nature of the architecture. The standard practice for developing fault-tolerant applications is to have a duplicate installation of the same servers and synchronize the state among those. The practice is widely used especially in databases where the master node in case of failure is replaced with one of the standby nodes which have a copy of the internal state of the master node of some consistent checkpoint. There are many replication strategies such as async, synchronous, or semi-synchronous replication [34], with many different topologies such as chain, star, etc.

Luckily, when developing a FAAS application the user gets fault tolerance out of the box due to the stateless nature of the functions. Most of the cloud providers also provide high availability guarantees since they can schedule the function to run not only on different servers but also in multiple zones of the same region to cope with even data center failures. [4]

Security

Security is of paramount importance for any application. In serverless environments usually, the security boundaries are enforced via roles and permissions assigned to the functions. Each function has its separate role and thus boundaries of resources that it has access to. When interacting with other services and functions, the role assigned for the function or the network VPC is enough to gain access to other resources inside the cloud provider environment. When interacting with the outside world, one might need access credentials or tokens specific to that service. An easy way could be to provide the location of those keys as an environment variable, but those are tied with the specific version of the function. For that reason, cloud providers devised secret management tools (Secret manager or parameter store), which keep your keys and other sensitive information in an encrypted form. The user can have access to those at any time using their respective APIs.

Resource utilization

FAAS platforms also allow the user to have fine-grained control over resource utilization. In a usual setting where most of the code is deployed in a handful of servers, scaling one of the servers inevitably scales one or more of the functionalities of the application. It might be the case that one functionality of the application is compute-intensive and would benefit from disproportionately large servers than the other parts of the application. This can easily be achieved in a FAAS environment where each function can be hand-tuned to have a certain amount of RAM and CPU share. For example, a compression workflow in an application might need a higher CPU and RAM than the storage/notification workflow. In a standard application, those two could be merged into a single service and result in relatively low resource utilization, while in a FAAS environment those two can be separated and better utilized. Another important point in resource utilization is the inherent capacity of running thousands of concurrent functions in a FAAS setting. The user not only can increase the concurrency level in a high-demand setting but also lower it once the peak is over. Such an elastic burst of computing resources has been utilized in developing tools such as gg [26], where the thousands of functions are used for a short amount of time to parallelize a lengthy

compilation of a C++ program.

Observability

There are many things that FAAS users get out of the box such as autoscaling, infra management, etc. One of those advantages is native metric and log collection services which work without any intervention from the developer. Traditional application developers use an open-source ELK stack to manage logs and metrics or some 3rd party solutions such as DataDog, Sumo Logic, Splunk, etc. While FAAS users get those features natively in the same cloud environment. For tracing solutions, users usually need to annotate their code with a cloud provider-specific library to enhance their application with cloud traces for better debugging or monitoring. For example, AWS lambda users can rely on Cloudwatch for metric and log collection, and on AWS X-Ray for cloud tracing. [45]

Cost

The most important selling point for FAAS platforms is the "pay only what you use" model. Cloud providers measure the runtime of each cloud function and usually round up to the nearest 100th millisecond and charge only for the execution time. Such a pricing model avoids having any idle resources thus closely approximating the cost to the load at the hand. Although the autoscaling of instances replicates this feature, we can see that the FAAS offerings are way closer to the actual load with an accuracy of 100 milliseconds. Cloud functions can also virtually downscale to zero instances when there is absolutely no load on the system. Such features can be really useful for seasonal applications, where the load spikes for only a short amount of time.[4]

Memory and CPU

One important aspect which is very specific for cloud functions is the intrinsic connection between CPU and memory allocation. The user is usually provided with an API to choose the amount of memory available for a function. This not only directly configures the RAM available at runtime, but also the amount of CPU share for the function. In essence, it trades the old complexities of optimizing resource utilization in a traditional "serverful" environment to a new optimization problem that has both CPU and RAM resources tied up in a single metric. Some cloud providers give the users option to configure memory somewhat freely from the memory, but by and large, those two metrics are usually tied together. One can see that increasing memory results in a smaller execution time for the function which reveals the connection between the two. As the graph shows the execution time plateaus at some point as increasing the

memory after that only increases the number of available virtual cores which might only help in a multithreaded environment. In this work, we are going to explore this aspect more thoroughly and help the user to pick optimal memory configurations for the cloud functions.

2.1.2 Internals of FAAS platforms.

In recent years we have observed the emergence of many FAAS platforms from different vendors such as Google, AWS, VMWare, Oracle, etc. Some of the famous ones are AWS lambda, Google cloud functions, Oracle Fn, Kubeless, Knative, OpenWhisk, etc. Many of them are open source and have huge communities which support the development and adoption of such platforms. The presence of open source FAAS projects also allows us to look under the hood of the systems and gain a better understanding of how they are designed and how they work. This also allows us to better optimize our code and create more performant applications. Nonetheless, the private cloud providers also have published white papers which describe the internals of their systems. Although those descriptions are far from ideal and leave out a lot of important details, we are still able to understand the major design decisions and the architecture of their corresponding platforms.

As a part of this section, we will take a look at Apache OpenWhisk and AWS lambda. We will try to learn their inner workings and understand the reasons for the common limitations that most cloud platforms have.

2.1.3 AWS lambda

AWS Lambda itself is a quite complex and multilayered service.[44] As of the white paper published by AWS, the service consists of two major components: the control plane and the data plane.

Control plane

The Control plane is meant to provide the user with management API such as updating function versions, creating a new function, increasing the function's memory, etc. All the communication with the client and the control plane is encrypted with TLS. AWS Lambda assures that all the relevant metadata is kept in an encrypted form using the AWS KMS service, which ensures the integrity and secrecy of the function configuration details.

Data plane

On the other hand, the data plane is designed to handle the execution of the AWS Lambda functions. When a function is triggered the data plane creates an execution environment for the function or finds an already available one. After that, it schedules the function. It also makes sure that the function return values are sent back to the user or the errors are handled according to the retry policy configured via the control plane.

An important detail regarding the execution environments is that they are unique not only to each function but also to each version of the function. So if two different versions of the same function are invoked the user is expected to get two different cold starts. Although having a warm, already ready execution environment provides quick start-up time, AWS doesn't guarantee isolation in between invocation of functions of the same version, meaning that the user might find files in the memory or the local storage of the lambda function. This poses a security challenge for the developer as they might need to separate functions that deal with sensitive data from those which interact with the external world, to avoid disclosing server-side secrets. The AWS Lambda platform also ensures that the only way that the client can interact with the execution environment is through the data plane, thus ensuring the authentication and authorization of access to those resources. This is the only API to initiate ingress or inbound traffic to the execution environment of functions.

Execution environments are continually tracked by the data plane to ensure maximal utilization of the underlying hardware. Thus the scheduler might remove the allocated execution environment to your function, in case a new function arrives, the worker has reached its lifetime, the provisioned concurrency is changed, a rebalancing of compute resources is required, or for some other reasons. This additionally makes the execution time unpredictable as discussed in the previous chapter.

When we think about the execution environment or workers, we need to think about virtual machines as in reality everything runs on normal servers despite the ill-descriptive name of such services. The one running under the AWS lambda hood is Firecracker [25], an open-source VM, which in essence is a layer on top of Linux's Kernel Virtual Machine. There are many VMs that are based on KVM such as Qemu [41] etc. The main advantage of using Firecracker is that it is a very lightweight container having only a 5MiB memory footprint, allowing AWS to deploy thousands of those in a single machine with ease. It is also very quick to start and can take less than 125ms to boot a new instance. The trick is that instead of implementing a fully functional VM they implemented the bare minimum needed for cloud function workload having developed only 5 emulated devices. That is why Firecracker provides the speed and small cold start time required by modern event-based applications. [25]

You can also read more about isolation techniques, firecracker implementation details,

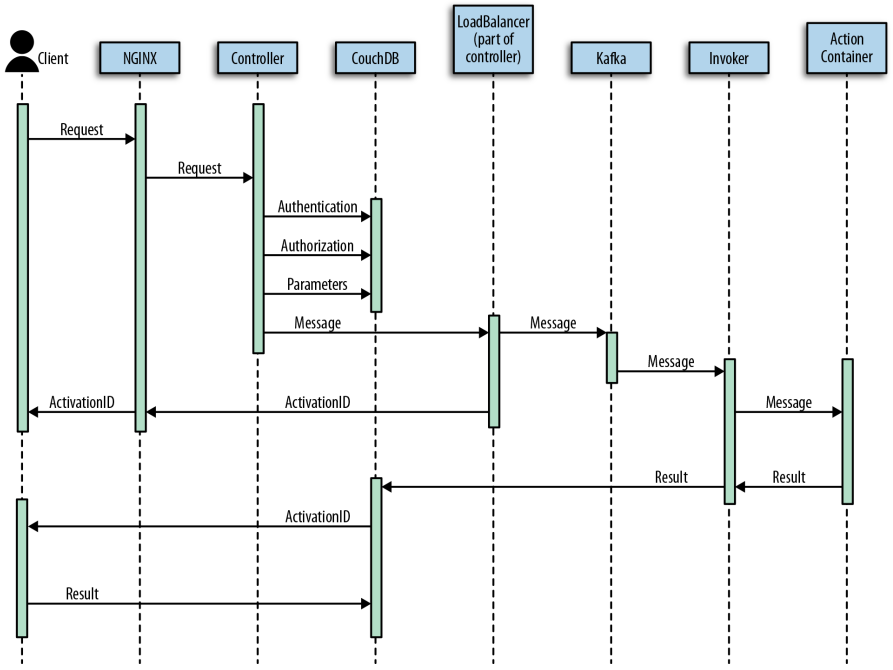


Figure 2.1: Openwhisk request workflow.[36]

or security inspection technology in AWS whitepapers [44] or in their open-source code repositories [25]. We wrap the description of the system here as the further discussion is beyond the scope of this work.

2.1.4 OpenWhisk

OpenWhisk [52] is a popular framework for deploying serverless applications in the cloud. In comparison to AWS Lambda, OpenWhisk, being an open-source project, lets us dig deep into the construction of the platform and gain a substantial understanding of the service architecture. Like any other FAAS platform, it is event-driven and everything revolves around external events generated by the clients. Fortunately, Openwhisk is built on top of well-known and commonly used components such as Kafka, Nginx, and CouchDB. Being built on top of such well-known tools makes the inner workings of the platform much more understandable for an external viewer or a novice user. The platform itself can be divided into 3 major parts: Controller, Invoker, and Action Container, which we will explore in the next chapters. [36]

Life of an event

Any event in an OpenWhisk framework comes in the form of an HTTPS request, no matter where it is generated from. Various sources can generate an event such as a web request, CLI invocation, internal invocation via API, etc. Let's take a look at an example when a client invokes a web request. The node that receives the request is a widely known reverse proxy, Nginx. Nginx not only forwards the request to the controller for further processing but also validates the requests, checks the certificates, and ensures encrypted transfer of data in and out of the OpenWhisk platform.

Controller

The controller is an integral part of the platform. First, it authenticates and authorizes the request to check if it should be scheduled on the platform. All of the metadata checks rely on the communication with CouchDB, which holds the state of the platform. After verifying that everything is correct with the request it passes it to the Load Balancer.

Load Balancer

The load balancer in OpenWhisk is responsible for choosing the right server to invoke the action. It does by looking at several factors such as call locality, load, etc. To make sure that the action is eventually invoked, despite the current state of the invoker service, as it could be busy at the moment, crashed, or restarting, the event is published in a Kafka messaging system. Kafka decouples the Load Balancer and Invoker making sure that each of them works properly despite the current state of the other service. Each event is paired with an Activation ID which is stored in CouchDB.

In case the invocation was asynchronous, the result is stored in CouchDB paired with the Activation Id. This allows the user later to poll the database to check for results. Subsequently, for any non-blocking request, the user ends up getting only the Activation Id.

In case of a synchronous invocation, the client keeps the connection open and waits for the action to complete. In this case, the result is directly passed to the user.

Invoker

Invoker is responsible for actually invoking the action. Unlike AWS Lambda where the code is executed in lightweight VMs, Openwhisk chose Docker containers as an execution environment. Although Docker doesn't provide VM grade isolation of the environment, it provides blazingly fast startup time. The Docker containers usually

host the execution environment for the supported languages for OpenWhisk. The preferable languages for such an environment are the interpreted ones that can be executed without compiling, as they skip the start-up time. On top of that, the users can not only use precompiled interpreted languages such as Java, but also they can provide the system with any ELF format compiled binary file.

Once Invoker completes the request it sends the results coupled with the initial Activation ID that it received. This way the user can later asynchronously check for the result in CouchDB.

2.2 Tracing Technology

Tracing technology is of paramount importance to any distributed application. It helps those systems become much more observable, making debugging, monitoring, and optimizing those services way easier. Origins of tracing technology can be traced back to the 90s but the real interest in the technology started in 2010 when the Dapper paper was published by Google. In that paper, the authors demonstrated how Dapper helped Google to debug services that were experiencing high latency or timing out. The tool pinpointed the server and the microservice causing the issue thus helping the developers to fix it. Although initially designed to trace "serverfull" microservice-oriented systems, it got adopted by the serverless community and is well supported by most of the platforms. In our work, we have relied heavily on tracing as it enabled us to get a detailed overview and metrics of each cloud function, thus we want to go into details about how it works.

2.2.1 General overview

At its core, tracing provides a map of services traversed by a request. It helps the user to see how the services interact with each other, the failures that happen, latencies, logs, metrics, stack traces, and much more. To achieve this the tracing technology attaches a tag to the requests to track the propagation throughout the system. The tag is just a unique string to identify the request when it traverses services. In the case of an HTTP request, it is usually a field in the HTTP header, while it is a bit more involved in the case of RPC-based communication mechanisms such as gRPC, Thrift, or Avro. It is also worthy to note that the API should be consistent across the multiple languages that the system might use and across services that it relies on. For example, if a microservice written in Python calls a Java-based service the tags should be interpretable by both languages to successfully propagate the context.

There are many different tracing technologies available on the market developed by different vendors, such as AWS, Splunk, Datadog, but by far the most advanced and

commonly accepted standard is OpenTelemetry, which is a combined effort from all those companies to create a unified standard for tracing. The problem with having 10 different technologies for tracing is portability. If there is no common standard for how requests and responses need to be annotated then an application relying on multiple different services won't be able to be traced. That is why people are working on unifying the API across all those vendors to make sure that the technology can be used interchangeably and across different platforms.

Concepts

Span or segment is the basic element of the trace. It is the elementary unit that is emitted from a server. It usually contains the span id, parent span id, host, timestamps, the request, subsegments, status codes, etc.

Trace is a collection of spans that are usually structured as a DAG. The trace also has a trace id which helps identify the path of the request from others. Using the parent span id field available in spans, one can construct the trace graph back from emitted spans.

Context is the abstract concept of the execution being part of a trace or a span. Context is usually identified by a trace id or a span id. Propagating context means passing the span id or trace id to the next execution environment so that later the trace can be reconstructed to identify that those two segments of execution came from the same request.

2.2.2 OpenTelemetry

So let's take a look at how OpenTelemetry is structured and developed. The confusing part about OpenTelemetry is that it is an umbrella term for many different projects, which come together to create a common playground for all tracing technologies.[40]

Life of an event

At the top, we have the standardized API for OpenTelemetry which can be divided into 4 parts: API-s, SDK, collectors, backend.

API

The real value for the community that OpenTelemetry provides is the standardization of APIs and the agreement on common rules on how to structure traces. Those conventions can be summarized in 4 different points:

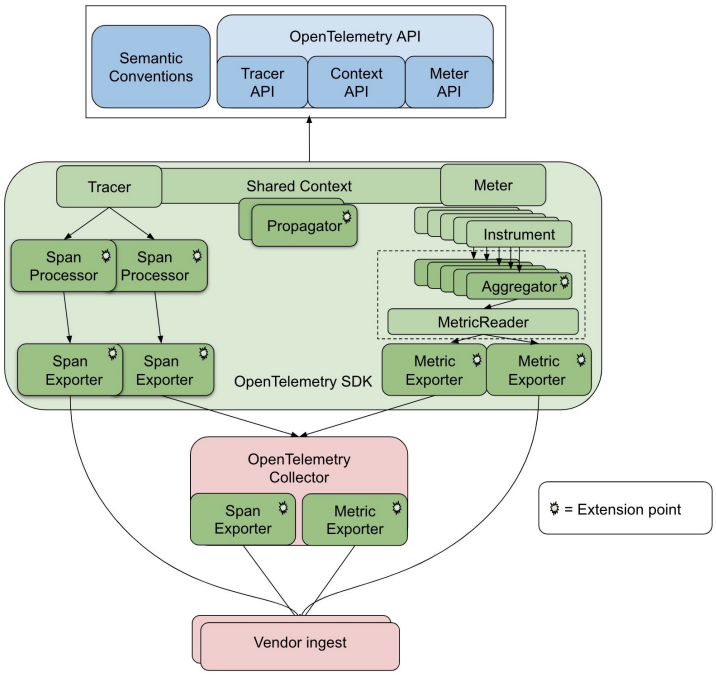


Figure 2.2: OpenTelemetry architecture.[36]

Tracer API is a convention on how the traces should be formatted and what fields should be made available by the instrumentation tools. Spans should have span id and can be assigned a trace id as well. The trace should also have metadata about the tracer type and version that it comes from.

Metric API is concerned about defining the format for providing metric instruments such as Observers and Counters. Counters are meant to count operations, while observers are designed to gauge a metric at a given point in time such as load on a server, memory usage, etc. What makes metrics of OpenTelemetry more appealing is the fact that they should have access to context, thus giving more insight to the end-user.

Context API is designed to define the conventions as to how to pass the context from one server to another in the means of HTTP, gRPC, etc.

Semantic conventions are just suggestions on how to name spans, errors, or attributes. This creates a common ground for all providers to create a unified interface and facilitate seamless trace transition from one environment to another.

SDK

Tracing pipeline mainly consists of two components: span processor and span exporter. Span processor tracks the lifetime of a span and collects the appropriate pieces from different services. It makes sure that the corresponding parts of the trace are available for the final processing. Span processor also makes sure that sampling rates are obeyed and implements different sampling strategies such as head or tail. Span exporter deals with the infrastructure necessary to export the recorded spans to the collectors. It makes sure that spans are batched to efficiently use the network traffic.

Metric pipeline consists of aggregators, readers, and exporters. Readers and aggregators record the metrics emitted by the application and depending on the type of the metric (counter or observer) the last or all records are transferred to the metric exporter. Next, the metric exporter publishes the records to the collector or the collector pulls them from the exporter depending on the specific architecture of the collector.

Context propagator is the mechanism by which the trace id or the span id is carried from one service to another, identifying the continuation of the previous execution. OpenTelemetry provides an implementation for W3C trace context propagation specification but many other specification implementations can be used.

Collector

The collector is a process deployed near the services, that collects the exported logs, metrics, and traces. Its job is to transform the OpenTelemetry formatted data to the appropriate format digestible by backend services like Jaeger, DataDog,[20, 38] etc.

Backend

Backend services provide the real business value to the user. They show the service graph, metrics anomalies, latency distribution, etc. They also provide a zoomed-in look into separate traces for further analysis and debugging. There are many different providers for backend such as Jaeger, Zipkin, Datadog, New Relic, etc. [53, 20, 39]

2.2.3 AWS X-Ray

AWS X-Ray is a proprietary technology owned by Amazon which essentially provides the same services as Opentelemetry. It has got both instrumentation libraries for all languages, collectors as well as a backend to visualize the collected data. We have relied heavily on AWS X-Ray for our implementation of SLAM and it is specifically tailored to the conventions used in X-Ray, but it can be easily imported to any other tracing framework. The advantage of using the X-Ray service is that it can be used out

of the box without setting up any collectors or backend services. Everything is ready to be used. It is also well integrated with other AWS services such as DynamoDB, SNS, SQS, etc. This helped us to measure the latency not only on cloud functions but also on other services used by our test applications.

The downside is that when doing an external call into another service not supporting AWS X-Ray headers, we lose the trace. That is why having a common standard across different platforms would bring huge observability benefits for the whole ecosystem. AWS is committed to that and for that reason, they have also developed their version of OpenTelemetry collector. [8]. The collector makes sure that the data emitted by both AWS X-Ray instrumented services and other instrumentation tools are supported.

We opted for AWS X-Ray as it was ready to use and had the least technical overhead for our project. Also, it is a fully-fledged solution that provides all the features necessary for observability.

3 System Description

In this section, we present *SLAM*, a python-based automated end-to-end serverless application memory optimization tool that learns the correlation between heterogeneous memory configurations of the FaaS functions within the application to estimate the overall cost and Response Time (RT). Using this learned correlation, *SLAM* is able to estimate the best memory configurations for Function-as-a-Service (FaaS) functions within the serverless application which not only conforms to the defined Service-Level Objective (SLO) requirements in terms of overall RT, but also meets user-specified constraints such as Minimum Overall Cost (MOC), or Minimum Memory Sum (MMS) for all functions, or least overall RT. *SLAM* can dynamically adapt to changes in the given serverless application and automatically adjust memory configurations of functions. *SLAM* can be incorporated into a Cloud Service Provider (CSP) FaaS platform and then leveraged by application developers for optimizing the memory configuration of FaaS functions within their serverless applications.

To understand the variations in the serverless application's execution time caused by the heterogeneous memory assignments to the functions within an application when deployed on a FaaS platform, we have developed *SLAM (SLAM)*, a python-based automated end-to-end serverless applications memory optimization tool.

The objective of the project is to determine the right configuration for a cloud application consisting mainly of cloud functions to make sure that it conforms to Service Layer Objectives. There are 9 important steps to achieve the goal and we will go through each of them one by one.

Figure 3.1 provides an overview of our developed *SLAM* tool and the interaction between its components in a typical usecase. *SLAM* assumes that the serverless application which is to be configured is already deployed by the application developer on a FaaS platform (AWS Lambda in our case) within a CSP and is instrumented with a middleware tracing library (such as AWS X-Ray) to trace the incoming and outgoing requests to other functions, or other Cloud components/services.

SLAM takes the SLOs requirement for the application as the input (step ❶). It generates a minimal amount of user workload to the application's public endpoint (step ❷) and collects application trace logs (step ❸) and various monitoring metrics data (step ❹). The collected logs are used to construct an application dependency call graph (step ❸) and this dependency call graph along with the monitoring metrics

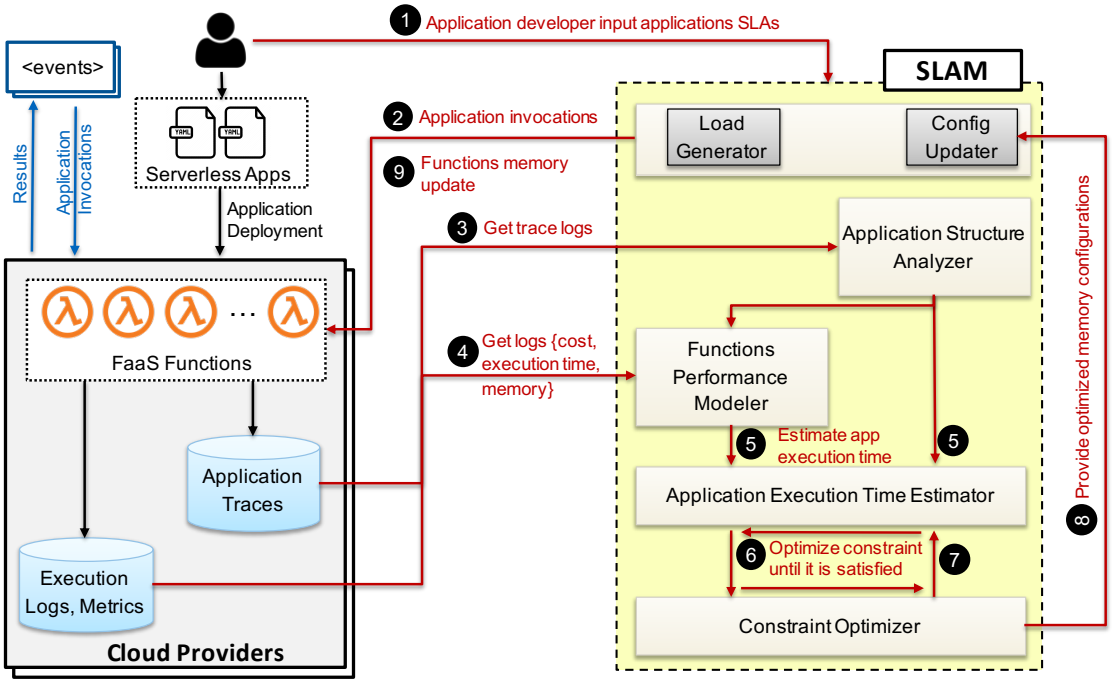


Figure 3.1: High-level architecture of the *SLAM* and the interaction between its components in a general use case.

are used for building the application's functions performance models. These models along with the application dependency call graph are then further used for estimating the overall application response time based on the different memory configurations for the functions such that it conforms to the given SLOs (step ❺). The estimated time, memory configurations, and cost are examined for the user-specified constraint satisfaction (step ❻). If the constraint is not satisfied, *SLAM* tries different memory configurations (step ❼) and continues the process until the constraint is satisfied (steps ❻ - ❼). Once the constraint is satisfied, the functions memory configurations are updated (steps ❽ - ❾). Next, we discuss the six major components of our *SLAM* tool.

3.1 Load Generator

This component is responsible for generating user workload to the deployed application. It takes one parameter as the input: the total number of requests to the application. It then based on it generates the given amount of total user workload requests synchronously to the deployed application. This user workload generation allows to

create application traces (§4.4) and collect various metrics data (§4.4) used by the other components of the *SLAM*.

3.2 Application Dependency Call Graph Builder

This component is responsible for building the application dependency call graph involving the application functions and Backend-as-a-Service (BaaS) services such as database, storage, and queues. *SLAM* relies on external middleware tracing libraries (such as AWS X-Ray) instrumented by the application developer allowing to trace the incoming and outgoing requests to other functions, or BaaS services. The tracing library creates a "segment" for each request to the components (other functions, or BaaS services), and completes the segment as soon as the request is over. This segment describes a node in the call graph consisting of a host, request, response, start/end time, sub-segments, and errors that occurred during the process.

The combination of these segments is called a trace for a request. It is important to keep in mind that the segments can also have an unfinished state, which can be observed if the user query for a trace for an incomplete request, or the traces have not been synchronized with the main storage when querying for the trace. This kind of async behavior sometimes results in problems when trying to request the trace Ids and traces, as the traces might not have been ready to be parsed.

This component with the help of *Load Generator* component generates a small amount of user workload requests (§4.5) to the deployed application. The application traces generated during the process in the form of JSON objects stored in the monitoring solution of the CSP or external database are parsed to generate the application dependency call graph involving all the functions and BaaS services within the application.

Afterwards, the component filters out BaaS services as it is out of the scope of this work to tune or configure them. Moreover, it is assumed that these BaaS services provide high scalability and serve the user requests within the defined SLOs. As a result, after this step, we get the simplified dependency call graph for the deployed serverless application along with the composing functions. In case the user already has the application dependency call graph and wants to skip this step, *SLAM* has the provision to allow the user to input manually the dependency call graph of the application. This also increases the testability of the *SLAM*, while developing it.

SLAM also incorporates similar visualization tools to AWS to help the user better understand the topology of the application and debug any issues that might come up during the execution of the system. You can see example visualizations in the evaluation section.

3.3 Functions Performance Modeler

After building the dependency call graph of the application and knowing its composing functions, the next step is to estimate the execution time of each function at different memory configurations. This is done in two steps mentioned next.

3.3.1 Create traces and metrics data for building models

This component with the help of *Load Generator* component first generates a small amount of user workload requests (around 20 invocations) to the deployed application when all of its composing functions are deployed with a default same memory configuration (128MB). Based on the composing functions found by the *Application Dependency Call Graph Builder* component, it then requests *Config Updater* component for updating the memory configurations of those functions based on the default list of memory configuration values (mem_config_list in Table 5.1) and *Load Generator* to again generate the same amount of user workload requests to the updated application. The process is continuously repeated for all the memory configurations (mem_config_list in Table 5.1) and in the end application traces and various metrics data (§4.4) is created for estimating execution time for each function within the application.

SLAM has a default list of memory configuration values (mem_config_list in Table 5.1), that it chooses from when generating memory configurations for the application. Although, the list comprehensively covers the whole range of memory values that can be configured for functions at AWS, it can be optimized by changing the values in the list depending on the requirements. For example, if the function only uses a single thread and doesn't use a considerable amount of RAM, then the list can be limited to 2GB as at that point AWS stops increasing the portion of the allocated VCPU and increases the number of available VCPU, which will then not used by the application.

3.3.2 Estimation of execution time for each function

Traces are parsed and metrics are analyzed to create a distribution of execution time for each function and each memory configuration. An example of such a distribution for a test function, when deployed with 128MB memory configuration on AWS Lambda, is shown in Figure 3.2.

One can observe that there is a high variance in the execution time of the function running under the same configuration due to the uncertainties from the underneath virtualized cloud infrastructure such as co-location of functions, cold-start, hardware failures, resource-overuse, etc. Therefore, to overcome this inherent variance, we choose a hyperparameter α representing the n^{th} percentile (Table 5.1) of the distribution as

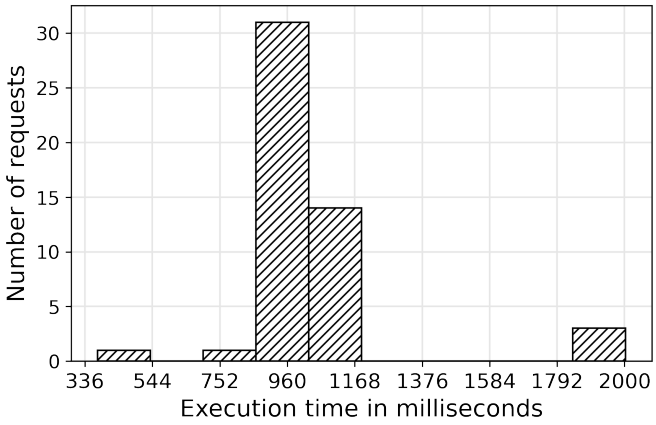


Figure 3.2: Execution time distribution for a sample function when deployed with 128MB memory configuration on AWS Lambda.

a representative for the execution time for the given function at a particular memory configuration. α is configured automatically by SLAM by calculating prediction accuracy of execution time at multiple values of it and selecting the one which results in a minimum mean squared error. We have also developed a small tuning module for the hyperparameter α. The module has a default list of hyperparameter values (50,75,90,99) that measures the mean squared error from the prediction and sets the value of the hyperparameter the one with the lowest deviation from the prediction.

Thus, in the end, a list of representative values for execution time for each function and memory combination is created, and how they are combined to form the overall execution time of the application is presented next.

3.4 Application Execution Time Estimator

Given the execution time of each FaaS function comprising the serverless application estimated by the *Functions Performance Modeler* at certain memory configurations, it is the responsibility of this component to combine them to estimate the overall application execution time based on the application dependency call graph.

It takes the application dependency call graph as the input and the functions invoked by a function in the application dependency call graph can either be processing all the invocations in parallel or one after the other in a sequence or a combination of both. Therefore, from the application dependency call graph first, it determines which functions are executing in parallel to others by using the functions' start and end timestamps available from the traces. The tool then divides all functions into groups of

sequence groups, where all the functions in each group are executed in parallel to other functions in the same group, and each group is executed in sequence to other groups. Since all the functions in a group are invoked in parallel, therefore to estimate the execution time of a group we take the maximum of the execution times of all functions in the group. In the end, we sum the execution times of each group to get an estimate of overall application execution time.

Mathematically, if we have an application consisting of N functions configured with certain memory configurations and defined as $F = \{f_1, f_2, f_3, ..., f_N\}$, with them being divided into S sequence groups defined as $G = \{g_1, g_2, g_3, ..., g_S\}$, then the execution time of the whole application is given by:

$$T(G) = \sum_{x=1}^{N} F(g_x) \tag{3.1}$$

where for some group i:

$$F(g_i) = \begin{cases} max(T(\bar{g}_1^i), \ldots, T(\bar{g}_U^i)), & \text{if } g_i \neq \text{function.} \\ \text{function execution time,} & \text{if } g_i = \text{function.} \end{cases} \tag{3.2}$$

where $\bar{g}_j^i$ ($1 \leq i \leq S$ and $1 \leq j \leq U$) being the sub-sequence group within g_i and U is the total number of sub-sequence groups within g_i.

3.5 Config Finder

Given the estimated execution time for each FaaS function comprising the serverless application provided by the *Functions Performance Modeler*, it is the responsibility of this component of *SLAM* tool, Config Finder, to find the right memory configurations for all functions such that the overall application execution time adheres to the defined SLOs and the specified optimization constraints. We first present the four optimization constraints that can be used as part of *SLAM* tool and then we introduce optimal memory configuration finding algorithms.

3.5.1 Optimization Objectives

We first assume that we have an application consisting of N functions defined as $F = \{f_1, f_2, f_3, ..., f_N\}$ and there are a total of M memory configurations which a function ($f_i \in F$) can be allocated with. Following are the four optimization constraints that can be used as part of *SLAM* tool along with the defined SLOs:

- **Minimum Memory Sum (MMS)**: As part of this constraint, the idea is to find a configuration (amount of memory allocated to each function within the application) which would result in the minimum sum of the memory allocated to each function comprising the serverless application. This is given by:

$$minimize(\sum_{x \in M} \sum_{f \in F} m_f(x)) \tag{3.3}$$

where $m_f(x)$ is the memory allocated to function f at memory configuration x.

- **Minimum Overall Cost (MOC)**: The objective of the constraint is to find a configuration which would result in minimum cost for each invocation. This is given by:

$$minimize(\sum_{x \in M} \sum_{f \in F} t_f(x) \times p_f(x)) \tag{3.4}$$

where $t_f(x)$ is the execution time of the function using memory configuration x, and $p_f(x)$ is the price (cost per unit time) for running the function using memory configuration x. Our calculation only counts for the costs associated with the function execution and does not take into account the data transfer, storage, and other costs associated with the invocation of functions. $t_f(x)$ is estimated by *SLAM* and to calculate the aforementioned execution cost, we used the data provided by AWS [**(8)**]. Though they provide pricing only for a limited number of memory configurations, we interpolated the cost as there was a linear relationship between allocated memory and cost.

- **Minimum Overall Execution Time (MOET)**: The objective of the constraint is to find a configuration that would result in minimum overall execution time. Suppose there are total of K possible memory configurations set for the serverless application defined as $C = \{C_1, C_2, ..., C_K\}$ such that $C_j = \{m_1^j, m_2^j, ..., m_N^j\}$ $(1 \leq j \leq K)$ is a memory configuration set for F adhering to the defined SLOs and $m_i^j \in M$ $(1 \leq i \leq N)$ is the memory allocated to i^{th} function in the j^{th} configuration set. This constraint is then given by:

$$minimum(\{T_1, T_2, ..., T_K\}) \tag{3.5}$$

where T_j $(1 \leq j \leq K)$ is the overall application estimated time by *Application Execution Time Estimator* when the application is configured with C_j configuration.

- **Balanced Cost Execution Time (BCET)**: This constraint is created to find a a configuration which would result in balance between the overall cost and execution time of the application.

Algorithm 1: Generate all possible configurations.

```
Input: func_list: List[str] ,mem_config_list: List[ ], init_conf :Dict[str,int], ind : int)
Output: all_configs = Generator[Dict[str, int]]
for cur_mem in mem_config_list do
    init_conf[func_list[i]] = cur_mem
    if len(func_list) - 1 == i then
        yield copy(init_conf)
        continue
    end
    yield from generate_all(func_list,mem_config_list, init_conf,ind+1)
end
```

3.5.2 Optimal memory configuration finding algorithms

Now we describe the three algorithms for generating the memory configurations and finding the optimal memory configuration for all functions such that the overall application execution time adheres to the defined SLOs and the specified optimization constraints (§3.5.1).

- **Brute force**: This method generates all possible combinations for configurations for the functions within the application to find the configuration that conforms to defined SLOs and minimizes the given constraint. The overall complexity of this approach is given by :

$$O(M^N) \tag{3.6}$$

  This methodology has a very high overhead if we have a considerable number of functions in an application and many different memory configurations. On the other hand, for a small number of functions, it generates the most optimal solution under our estimation assumptions.

- **Binary search based**: This algorithm is based on a popular binary search algorithm where the memory configuration search space for a function is conducted by binary search method. In this case, the default sorted list of memory configuration values (mem_config_list in Table 5.1) is used.

  The pseudocode for the algorithm is shown in Algorithm 3. This algorithm initially for every function within the application begins by configuring the functions with the minimum memory i.e. 128MB. Next, we sort the functions according to their execution time in descending order. If the SLO requirements and given constraints are satisfied, then the configuration of the functions is

Algorithm 2: Brute force algorithm.

```
   Input: func_list: List[str], mem_config_list: List[ ], SLO: float)
   Output: res_config: Dict[str, int]
1  init_conf = {}
2  for func in func_list do
3  |  init_conf[func] = min(mem_config_list)
4  end
5  config_gen = generate_all(func_list, mem_config_list, init_conf, 0)
6  res_config = {}
7  for mem_cofnig in config_gen do
8  |  dur = estimate_duration(mem_config, func_list) if dur < SLO then
   |  |  // Could be cost, minimal memory sum or something else.
9  |  |  if criteria(res_config, mem_config) then
10 |  |  |  res_config = mem_config
11 |  |  end
12 |  end
13 end
14 return res_config
```

returned (Lines 1-2). If not, it then uses the binary search method to update the memory of the first function in the functions list to the middle of mem_config_list and updates the resulting configuration list (Line 11-13). During the search process, it fixes the memory of this function and recursively calls the same algorithm for the next function for trying different memory configurations of other functions to find the right configuration (Line 14). This recursive call will either return a new resulting configuration list if it is found otherwise empty. If it is empty (Line 15), it means that the memory configuration for the fixed function is low, therefore in the next iteration of the binary search process, the memory configuration assignment for the fixed function search continues in the upper half of the mem_config_list (Line 16). On the other side, if a configuration is found (Line 17), then it tries to find a lower memory configuration for the fixed function which can again satisfy the objectives by continuing the search in the lower half of the mem_config_list (Line 18-19). In the end, an appropriate resulting memory configuration list satisfying the objective is returned (Line 23). The overall complexity of this approach is given by :

$$O((\log M)^N) \tag{3.7}$$

The binary search works as long as the search criterion has some kind of inherent order. This kind of pruning of the search tree results in much bigger scalability

regarding the number of functions. Though we will find a configuration that will satisfy our SLOs, the method does not guarantee the found solution is the most optimal one. For example, if the algorithm finds a configuration for the first function as 512 MB memory then it will never look for solutions where this function has larger memory. This may result in higher due to higher execution time. However, this can be avoided by allowing the algorithm to search for all configurations that satisfy our SLO requirements, and filter out those which don't conform to the other objectives.

- **Max-Heap based**:

  Now we describe the algorithm (called *SLAM-SLO*) for finding the optimal memory configuration for serverless applications such that the overall application execution time adheres to the defined SLOs. The modified version of the algorithm for optimizing on various objectives along with the SLOs is called *SLAM-SLO-Min-Cost* for MOC and *SLAM-SLO-Min-Time* for Minimum Overall Execution Time (MOET).

  SLAM-SLO: In this approach, we leverage the max-heap data structure for finding the optimal configuration which satisfies the SLO requirements. The pseudocode for the algorithm is shown in Algorithm 4. Each function's execution time at the minimum memory configuration i.e., 128MB is calculated and is used for constructing the max-heap. We store the execution time of the function at a particular configuration as the node value and the function name and its memory configuration are further saved as the node's metadata (Line 5-8). The function at a particular memory configuration having the highest execution time will be automatically stored at the head of the max-heap tree (Line 9). We first check if this base configuration satisfies the SLO requirements. In case it does, we stop the iteration and return the configuration (Line 11-13). Otherwise, in the next step, we pop the head from the max-heap (Line 14), increase its memory to decrease its execution time (Line 16) and then push the function again back to the heap with the updated memory and execution time (Line 17-20). After this update, we check if the configuration satisfies the SLO requirements. In case it does, we stop the iteration and return the configuration (Line 11-13). Otherwise, we continue the process by popping the function at the head until a configuration is found. If no configuration is found, an empty dictionary is returned. The overall complexity of this approach is given by:

$$O(NM\log N) \tag{3.8}$$

Algorithm 3: Binary Search based Algorithm

```
Input: cur_config = Dict[str, int], curr_func_ind, func_list, mem_config_list: List[ ], SLO)
Output: result_config = Dict[str, int]
// check for objective(s) satisfaction.
if estimate_duration(cur_config) ≤ SLO then
    // other objectives can be added here.
    return cur_config;
end
// when reached end of functions list, return empty
if curr_func_ind ≥ len(func_list) then
    return;
end
// init variables for binary search
left = 0;
right = len(mem_config_list) - 1;
solution = Dict[ ];
do
    mid = int(left + (right−left)/2); // get middle memory
    old_config = cur_config; // save current config
    // update current function's memory
    cur_config[func_list[func_ind]] = mem_config[mid];
    /* call recursively this algorithm for other functions */
    new_config = binary_search(cur_config, curr_func_ind + 1, func_list, mem_config,
      SLO);
    if not new_config then
        // if the change didn't satisfies objective, update left
        left = mid + 1;
    else
        // if the change satisfies objective, update right
        right = mid - 1;
        solution = new_config;
    end
    cur_config = old_config; // restore the current config
while left ≤ right;
return solution; // return the solution config
```

This method is highly scalable and also does locally optimal steps to lower the overall execution time of function call.

SLAM-SLO-Min-Cost: We further modified the *SLAM-SLO* algorithm to take

Algorithm 4: SLAM-SLO Algorithm

```
   Input: func_list, mem_config_list: List[ ], SLO)
   Output: result_config = Dict[str, int]
1  min_mem_config = min(mem_config_list)
2  for func_name in func_list do
       // init minimum memory assignment for all functions
3      res_config[func_name] = min_mem_config;
4  end
   // prepare heap with function's exec time at min memory
5  for fname in func_list do
6      func_exec_time = exec_time(fname, min_mem_config);
7      func_heap.append(func_exec_time, fname);
8  end
9  heapify_max(func_heap); // reorder heap
10 do
       // check for objective(s) satisfaction.
11     if estimate_exec_time(res_config) ≤ SLO then
12         return res_config;
13     end
14     top_func = heappop_max(func_heap);
15     if not all_memory_config_evaluated(top_func) then
           // update memory and time, then append to heap
16         func_new_mem = update_memory(top_func);
17         func_new_exec_time = get_exec_time(top_func, func_new_mem);
18         func_heap.append(func_new_exec_time, top_func);
19         res_config[top_func] = func_new_mem; // update
20         heapify_max(func_heap); // reorder heap
21     end
22 while func_heap is not empty;
23 return ; // return the empty config
```

cost into account for finding the optimal configuration with the MOC as the objective along with the SLO requirements. Here, the algorithm uses the *SLAM-SLO* found optimal configuration as the default configuration and tries to optimize on top of it for finding minimum cost configuration. In this, every time we pop the function from the head of max-heap, we check for the following inequality at the new updated memory for that function:

$$\left|\frac{\text{new\_cost} - \text{old\_cost}}{\text{old\_cost}}\right| \leq \left|\frac{\text{old\_exec\_t} - \text{new\_exec\_t}}{\text{old\_exec\_t}}\right| \tag{3.9}$$

where new_cost and new_exec_t are the cost and execution time of an application

invocation after updating the memory of the function, and old_cost and old_exec_t correspond to the cost and execution time before the update. In case the inequality holds, we put the function back into the max-heap with the updated execution time. In case it doesn't, we fix the memory for that function in the final configuration. This also allows us to reduce the search space for finding the configuration satisfying the minimum cost objective.

SLAM-SLO-Min-Time: This modified version of the *SLAM-SLO* algorithm also uses the *SLAM-SLO* found optimal configuration as the default configuration and tries to optimize on top of it for finding minimum execution time configuration. It then leverages the binary search algorithm to find the configuration with minimum time. It uses the *SLAM-SLO* found optimal configuration execution time (β in seconds) as the maximum time and $0s$ as the minimum time. It then updates the SLO requirement to the middle of maximum and minimum time and calls the *SLAM-SLO* algorithm to find an optimal configuration. If a configuration is found, then the maximum is set to the execution time for that configuration otherwise minimum is updated to the previously found middle. This way it continues until a configuration is found with minimum application execution time. In order not to run the binary search indefinitely, we use a hyperparameter called precision (γ). When the lower and upper execution time bounds get closer than the precision hyperparameter, we stop the search and return the attained configuration. For more details check out the pseudocode for the algorithm at 5. As a default value of the parameter, we chose $\gamma = 0.01s$, which can be easily changed. The complexity of the resulting algorithm is given by:

$$O\left(NM\log N \times \log\left(\frac{\beta}{\gamma}\right)\right) \tag{3.10}$$

Algorithm 5: SLAM-SLO-Min-Time Algorithm

```
   Input: func_list, mem_config_list: List[ ], SLO, γ
   Output: result_conf = Dict[str, int], min_time
1  min_time = 0
2  max_time = SLO
   // Result is initiated as an empty configuration.
3  result_conf = {}
4  do
5  |  mid = min_time + (max_time-min_time)/2
6  |  possible_conf = SLAM_SLO(func_list, mem_config_list, mid)
7  |  if possible_conf == {} then
8  |  |  min_time = mid
9  |  |  continue
10 |  end
11 |  max_time = mid
12 |  result_conf = possible_conf
13 while max_time - min_time > γ;
   // If no configuration found.
14 if result_conf == {} then
15 |  return {}, -1
16 end
17 dur = estimate_dur(func_list, mem_config_list, result_conf)
18 return result_conf, dur
```

4 Evaluation Settings

We test the proposed *SLAM* tool on AWS Lambda, a popular serverless cloud platform. We first introduce the four developed applications, three synthetic applications having a different number of functions, and a real-world based application that we use to test our system in §4.1. Following this, we describe the testing process using these applications in §4.2. The testing environment, infrastructure, and the performance monitoring metrics are introduced in §4.3 and §4.4 respectively.

4.1 Test Applications

4.1.1 Synthetic Applications

To test the *SLAM* system we have developed an interface that can create automatically synthetic applications having a different number of functions. The input to the interface defines the application call tree containing functions that are either invoked in parallel or sequence. This way we can generate complex applications with as many functions as we like. Such a structure allows us to test the limits of the SLAM system and understand how much error is accumulated if the application contains many cloud functions with a combination of sequence and parallel invocations.

Each function within the application is a compute-intensive function that calculates the remainder for all numbers between 2 and N, where N is the parameter fixed for the function. The simplicity of the algorithm allows us to simulate test applications with heterogeneous functions requiring different compute/memory resources by scaling N. Each function within the application has a different value for N and is assigned randomly. The pseudocode of the unit function for the synthetic application can be found in 6. An example input of JSON to the interface for creating an application with three functions where one function (func-1) is invoking the other two (func-2, func-3) in parallel is shown in Listing 4.1 and its callgraph is shown in Figure 4.1a.

To better interpret the callgraphs, we decorated them with boxes that group several functions together. Functions in the same box are called in parallel to each other, while the ones on the same level are called in sequence. The directed edges show the function which has generated the invocation for the other functions on the lower level. One thing that the callgraph doesn't represent fully is the computation that each function

Listing 4.1: An example JSON for creating 3-functions test application.

```
{
  "function": "func-1",
  "N": 18000000,
  "parallel": [
    {
      "function": "func-2",
      "N": 36000000,
      "parallel": [],
      "serial": []
    },
    {
      "function": "func-3",
      "N": 27000000,
      "parallel": [],
      "serial": [
      ]
    }
  ]
}
```

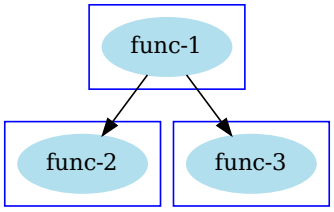
does which is separate from other function calls. Those calculations are done always serially to the calls of its children functions.

We additionally created two more synthetic complex applications containing 6 and 10 functions incorporating sequence and parallel invocations to test the *SLAM* system. Their call graphs are shown in Figure 4.1b and Figure 4.1c respectively. Such complex application call graphs allow us to estimate how well *SLAM* adapts to changing execution time when a change in a leaf function's configuration affects the execution time of the other higher-level functions. Moreover, such applications model the real-world applications and therefore allow us to give an idea of the capabilities and scalability limits of the *SLAM* system on them.

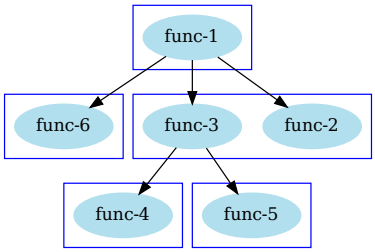
4.1.2 Real-world based Application

Since the synthetic application workloads do not fully represent the real-world use cases for serverless applications, therefore we created a pet store application based on an open-source spring-based application[1] consisting of five FaaS functions and two NoSQL databases. Its call graph is shown in Figure 4.1d. We used DynamoDB for the two NoSQL databases. This application is special in the sense that for the functions

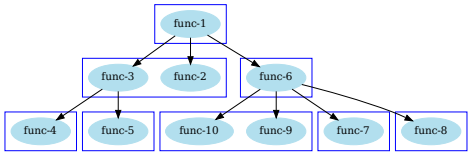
[1]https://github.com/spring-projects/spring-petclinic



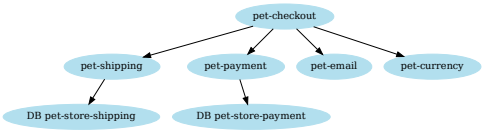
(a) 3-functions test app



(b) 6-functions test app



(c) 10-functions test app



(d) Real-world based app

Figure 4.1: Call graphs for the applications used for evaluating *SLAM*.

querying databases will not have any influence on execution time with the increase in memory for them.

In this application, when the client selecting a pet in the front-end for buying, it first automatically invokes the *pet-checkout* function which in turn is responsible for getting all the details needed for the purchase by invoking other functions. First, it calls the *pet-currency* function to convert the pet price to USD. Then it calls the *pet-payment* service for the client to pay for the pet. If the payment is successful then the *pet-checkout* function will invoke *pet-shipping* which will log the pet shipping details in the database. After successful completion of all the previous steps, the final *pet-email* function is called which generates a summary email and sends it to the client. The application itself doesn't ship anything, it is just a skeletal representation of what a real one would look like.

4.2 Testing Process

After developing all the necessary applications and tools testing process was straightforward. We just run the system for the developed applications and measured the results

Algorithm 6: Synthetic application unit function

```
Input: callgraph_json: Dict[str,Any]
Output: result_sum: int
result_sum = 0
lambda = wrapWithXrayClient(AWS.lambda())
N = callgraph_json["N"]
for int i in range(2,N) do
    if N % i == 0 then
        result_sum = result_sum + i
    end
end
parallel_funcs = callgraph_json["parallel"]
serial_funcs = callgraph_json["serial"]
// Calling parallel functions and waiting for results.
futures = []
for func in parallel_funcs do
    cur_future = lambda.callAsync(func["function"], to_string(func))
    futures.append(cur_future)
end
for future in futures do
    cur_res = future.awaitResult()
    result_sum = result_sum + cur_res
end
// Calling serial functions.
for func in serial_funcs do
    cur_future = lambda.callAsync(func["function"],to_string(func))
    cur_res = cur_future.awaitResult()
    result_sum = result_sum + cur_res
end
return result_sum
```

with different benchmark standards, such as with fixed memory, optimal memory, etc.

4.3 Environment

We test the proposed *SLAM* tool for serverless applications deployed on AWS Lambda, a popular serverless cloud platform. *SLAM* tool itself was run on a machine with 8 physical cores (Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz CPU) with hyperthreading enabled and 16 GB of RAM. These conditions are similar to a typical cloud VM.

SLAM has a default list of memory configuration values (*mem_config_list* in Ta-

ble 5.1), that it chooses from when generating memory configurations for the application. As all the functions within our test applications use only one thread, we limit the maximum memory configuration to 2GB, since as at that point AWS stops increasing the portion of the allocated vCPU and increases the number of available vCPU [50], which will then not used by the application. For our experiments, the initial total number of requests for load generation is set to as 50.

4.4 Performance Metrics

We extracted following monitoring metrics from the AWS lambda[2] with the data sampling rate as one minute:

- *concurrent_executions*: The number of active function instances.
- *invocations*: The number of times the function code is executed.
- *duration*: The amount of time function code spends in processing an event.
- *memory_usage*: Function's maximum memory usage during execution.
- *allocated_memory*: The amount of memory allocated to the function.
- *function_concurrency*: The maximum number of concurrent instances allowed for processing events.

4.5 SLAM Hyperparameters

SLAM has a default list of memory configuration values (*mem_config_list* in Table 5.1), that it chooses from when generating memory configurations for the application. Although, the list comprehensively covers the whole range of memory values that can be configured for functions at AWS, the user also has the option of changing the values in the list depending on the requirements. For example, if the function only uses a single thread and doesn't use a considerable amount of RAM, then the list can be limited to 2GB as at that point AWS stops increasing the portion of the allocated VCPU and increases the number of available VCPU, which will then not used by the application.

Initial total number of requests set to as 50 Specifically, we generate around 20 calls with each memory configuration for the whole application: the point being 20 calls with all functions set to 128MB, 256MB, etc. This can also be configured from SLAM to

[2]https://docs.aws.amazon.com/lambda/latest/dg/monitoring-metrics.html

create other invocation patter but we found this to be the most straightforward and easy to scale.

We have the SLAM system for 2 different synthetic applications for evaluation. As all the functions at hand use only 1 thread we configured SLAM to look into a configuration in between 128MB and 2GB, as further increasing memory only increases the number of VCPU available and doesn't affect the clock speed. For different applications which might use multi-threading, you can configure the search space to include memory configuration to the maximum available. It doesn't change the nature of the optimization technique, but for the experiments and evaluation, we tried to keep the nature of the application simple.

5 Evaluation

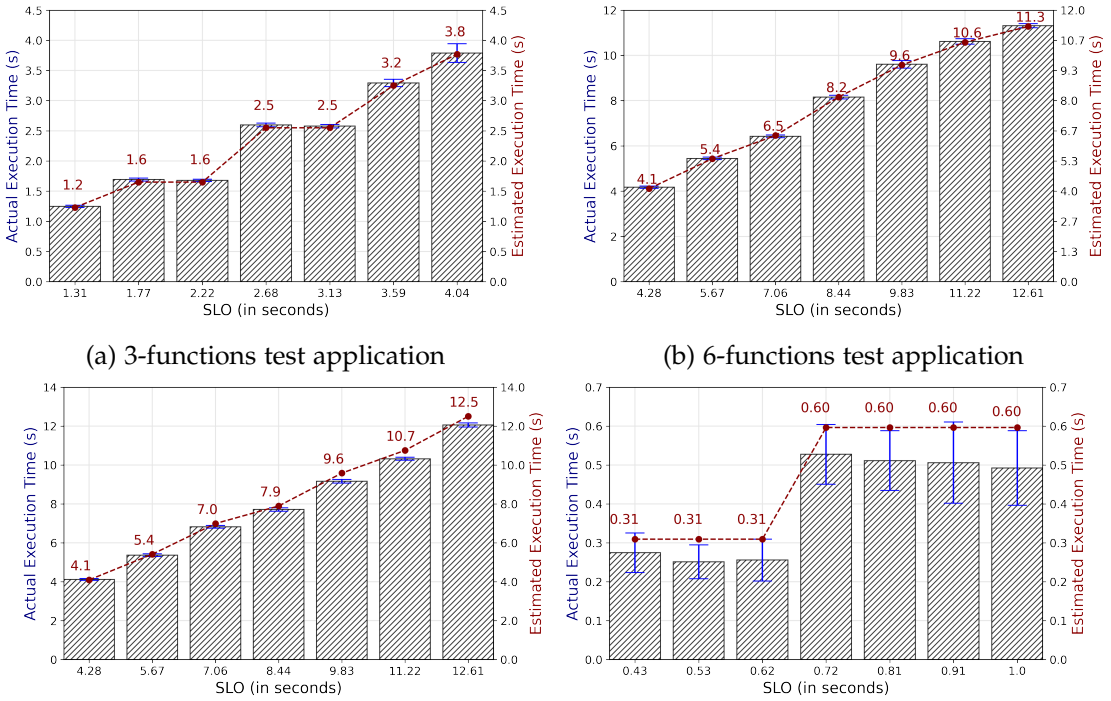
We design our experiments to answer the following questions:

- **Q1. *SLAM* estimation time accuracy**: how accurate is *SLAM* in estimating the execution time of an application for the given or found configuration at different SLOs?
- **Q2. *SLAM* configuration finding accuracy**: how accurate is *SLAM* in finding the configuration satisfying the given SLOs and objectives for an application?
- **Q3. *SLAM* configuration finding efficiency and scalability**: how efficient is *SLAM* in finding the configuration satisfying the given SLOs and objectives for an application? Additionally, how does the *SLAM* tool scale with the increase in the number of functions of the application?
- **Q4. Parameter Sensitivity**: How sensitive is the *SLAM* when the values of the parameters are changed?

5.1 Q1. SLAM estimation time accuracy

To demonstrate the effectiveness of the *SLAM* tool in estimating the execution time of the application, we test it on three synthetic and one real-world based application. For the test, *SLAM* tool is configured to find the memory configurations for the given SLOs such that the sum of the allocated memories would be the smallest i.e. Minimum Memory Sum (MMS) objective. This objective allows us to easily verify the accuracy of the tool as compared to other optimization objectives such as Minimum Overall Cost (MOC).

To begin with, *SLAM* first estimates the execution time of each function within an application at different memory configurations, and then based on the call graph it estimates the overall execution time of the application. Based on the found configuration satisfying the SLO and optimization objective, we then configured all the functions with the memory values suggested by SLAM and invoke the serverless application 100 times to get the application's execution time distribution. Figure 5.1 shows the actual experiment execution time box plot overlaid with the estimated execution time



(a) 3-functions test application

(b) 6-functions test application

(c) 10-functions test application

(d) Real-world based application

Figure 5.1: Actual experiment execution time box plot overlaid with the estimated execution time by *SLAM* tool for four test applications (three synthetic and one real-world based) at different SLOs when configured with a particular configuration.

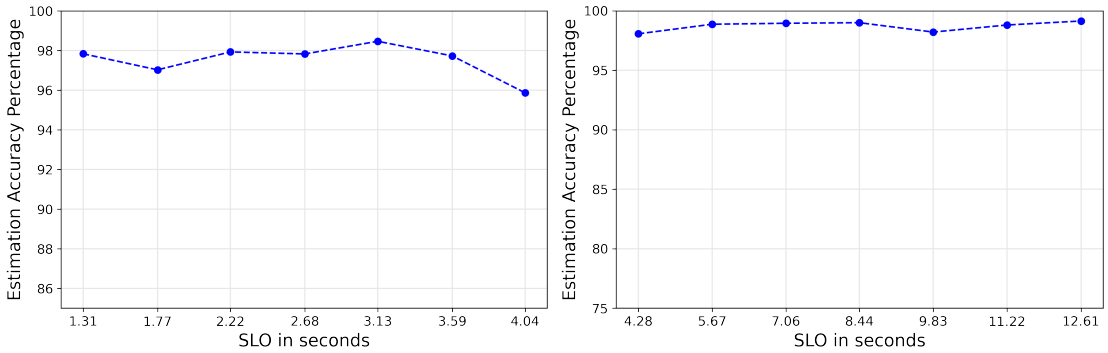
by *SLAM* tool for all four test applications (three synthetic and one real-world based) at different SLOs when configured with the found memory configurations.

Additionally, we measured the execution time estimation accuracy percentage for the four test applications at different SLOs as shown in Figure 5.2. For computing the accuracy at different SLOs, we calculate the mean squared percentage error between the estimated and actual execution time for the found configuration and then subtract it from 100.

Next, we discuss the results of the two classes of the test applications in more detail.

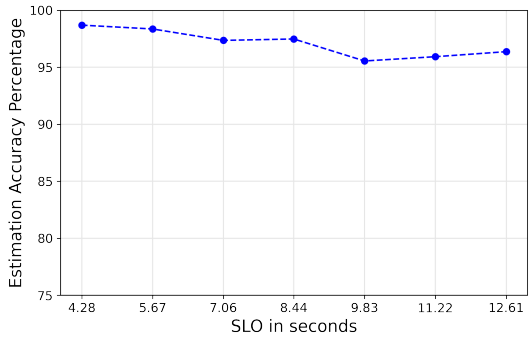
Synthetic Applications

From Figure 5.1, one can observe that in the three synthetic applications, the estimated execution time is either lower or equal to that of the specified SLOs. Additionally,

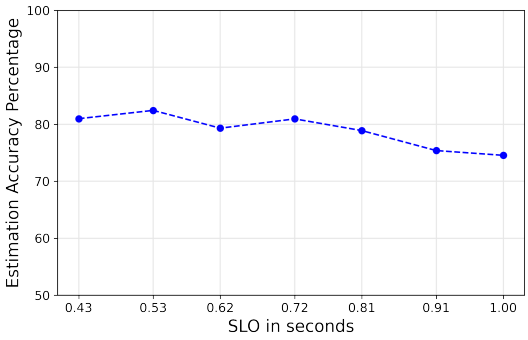


(a) 3-functions test application

(b) 6-functions test application



(c) 10-functions test application



(d) Real-world based application

Figure 5.2: Execution time estimation accuracy percentage for the four test applications at different SLOs.

from the overlaid graph of estimated execution time in Figure 5.1, we can observe that the estimated execution time is to a great extent closer to the actual execution time at different SLOs. To verify it further, we can see in Figure 5.2 that the measured execution time estimation accuracy percentage for the three test applications at different SLOs is above 90%.

In Figure 5.2a for *3-functions test application*, we perceive that, for the smaller value of SLO (i.e., 0.86s) the estimation accuracy percentage is lower (approximately 94%) as compared to other higher SLOs (greater or equal to 96%). We believe that the possible reason for this is an inherent variation in the execution time of any cloud function, connected with scheduling, availability of nodes, etc., and lower the SLO the more visible it becomes as an error, as it comprises a higher percentage of the whole call execution time.

Furthermore, from Figure 5.2b and Figure 5.2c, we can observe that after a particular

SLO (12.61*s* for *6-functions test application* and 12.61*s* for *10-functions test application*), the estimation and actual overall execution time for the applications become constant. This is because all the functions are assigned the minimum memory configuration and therefore the overall execution time of the application is highest at that configuration and cannot go beyond it. This is also evident from the configuration suggested by the *SLAM* tool for these SLOs as 128MB for all the functions.

Real-world based Application

Although, from the overlaid graph of estimated execution time in Figure 5.1d, one can observe that, the estimated execution time is a bit higher than the actual execution time at different SLOs which is also evident from the Figure 5.2d where the measured execution time estimation accuracy percentage at different SLOs is lower as compared to synthetic applications(ranging between 70% and 85%), but similar to the three synthetic applications, the estimated execution time for this application is also either lower or equal to that of the specified SLOs. This means that the configuration selected by the *SLAM* tool is good enough to fulfill the desired SLOs.

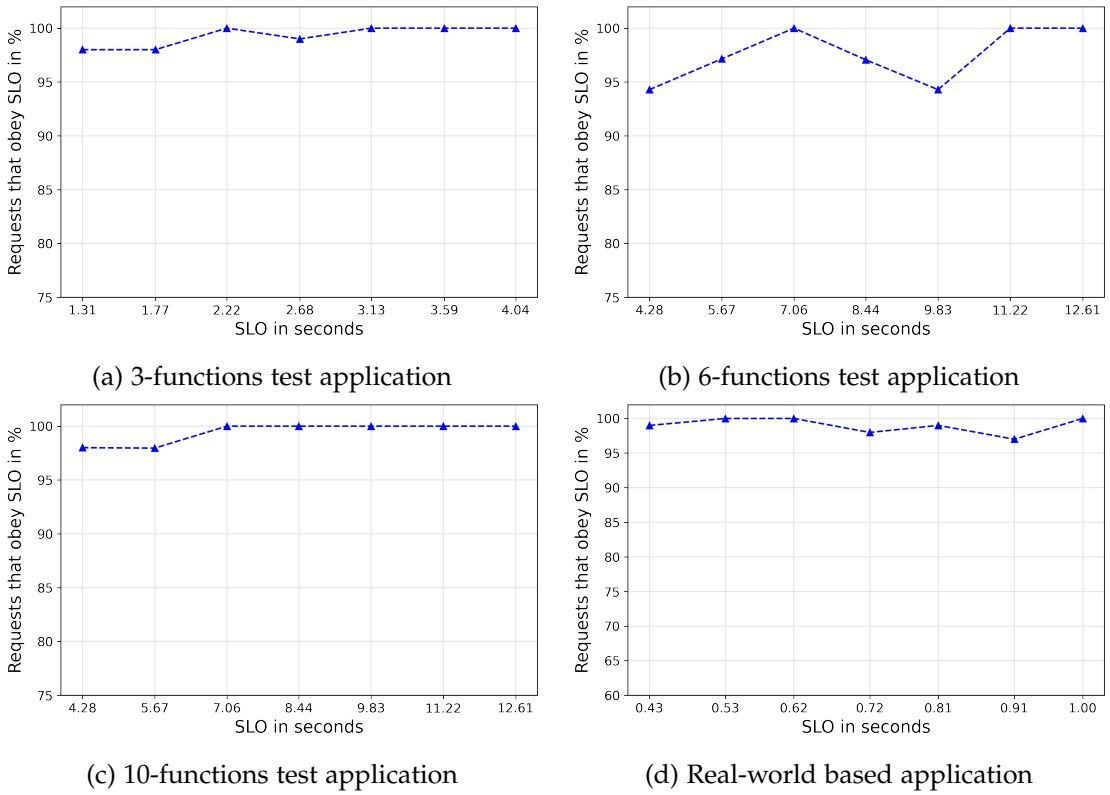
One reason for the higher estimated execution time at different SLOs could be due to the high variance in the actual execution time of the application (as seen in Figure 5.1d) because of the involvement of components such as DynamoDB which can lead to the variable execution time of the application. Moreover, the overall execution time of this application is smaller as compared to synthetic applications and thus even the small inherent variance within the application can cause high relative error rates and hence drop in the estimation of the accuracy. Nonetheless as mentioned earlier, the configuration selected by the *SLAM* tool is good enough to fulfill the desired SLOs (more details about it in).

5.2 Q2. *SLAM* configuration finding accuracy

In this experiment, for determining the accuracy of *SLAM* in finding the configuration at the given SLOs, we have considered two aspects presented next.

Precision of requests obeying SLO

Here we calculate the percentage of requests conforming to the defined SLOs when the functions are configured with the memory configurations suggested by *SLAM*. Experiment results on the four test applications are shown in Figure 5.3 for different SLOs when a total number of 100 requests were issued to the application at each SLO. We can observe that for all the synthetic applications the percentage of requests



(a) 3-functions test application

(b) 6-functions test application

(c) 10-functions test application

(d) Real-world based application

Figure 5.3: Percentage of the requests conforming to the given different SLOs based on the configurations suggested by *SLAM* tool at those SLOs for different applications.

conforming to the given SLOs is either equal or above 95% which means that out of issued 100 requests at least 95 requests were served within the SLO requirements. Additionally, for the *Real-world based* application as well, despite having lower estimation time accuracy as compared to synthetic applications, *SLAM* is still able to generate configurations that result in above 95% precision of requests conforming to the given SLOs. Thus we can conclude that the memory configurations suggested by *SLAM* are at least 95% precise.

Various objectives configuration finding effectiveness

In this aspect, we determine the effectiveness of *SLAM* tool when requested to optimize for various optimization objectives (§3.5.1) keeping SLOs fixed. In this regard, we calculate the overall execution time and the cost needed by one invocation of the

application when configured with memory configurations selected by *SLAM* for those optimization objectives and compared them against static minimum-memory=128MB (*min-mem*) and maximum-memory=4GB (*max-mem*) configurations to get the worst/best execution times for the applications, and also the corresponding costs. The memory configurations of *min-mem* and *max-mem* signify configurations where each function in the application is configured to that memory.

Experiment results on the four test applications are shown in Figure 5.4 and the results are averaged over 100 application invocations. We show the three objectives: *min_cost* representing Minimum Overall Cost (MOC), *min_time* representing Minimum Overall Execution Time (MOET), and *optimal* representing Balanced Cost Execution Time (BCET). It is to be noted that, since we want to find the global minimum cost and execution time, the *min_cost* and *min_time* points for each application are obtained by checking every single configuration from the initially provided set of memory list and function combinations using *Brute force* approach.

We observe following from Figure 5.4 for different optimization objectives:

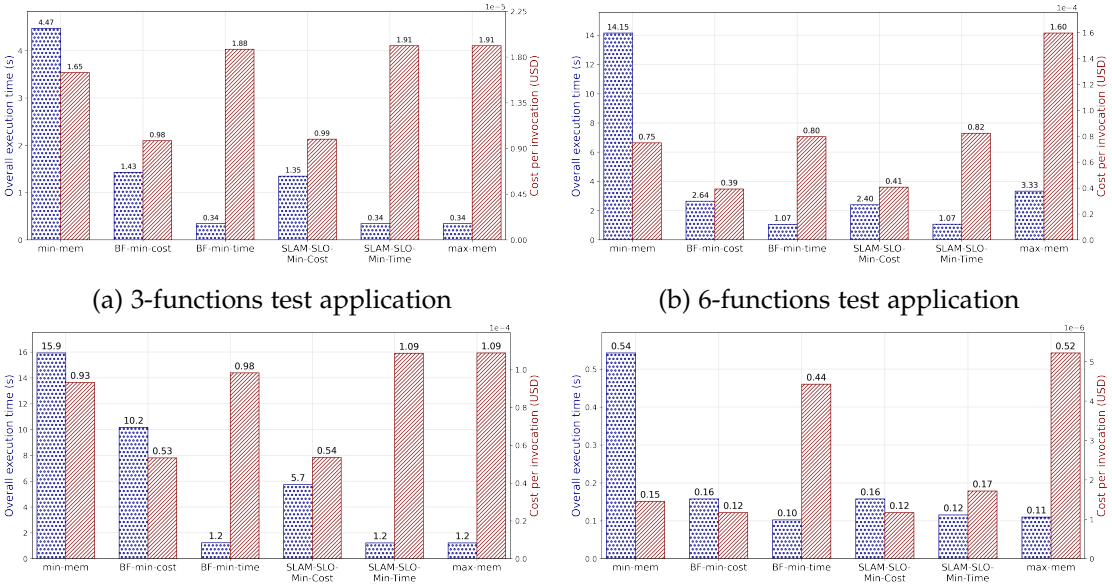
- **Minimum Overall Cost optimization objective**: From Figure 5.4, we can see that for all the applications, *SLAM* finds the minimum cost configuration (0.98×10^{-5} as seen in Figure 5.4a for the 3-functions,). However, the downside of this objective is that the minimal cost point is obtained by checking every single configuration from the initially provided set of memory lists and function combinations.

The optimal configuration is a point obtained by the cost-optimal approximation algorithm described in the previous chapter. As you can see on the graph, our approximation algorithm finds a configuration that is close to the global optima.

Figure 5.4d shows the runtime and price per invocation dependency for the given application. The graph only includes the price of the lambda execution and doesn't include things like network, database calls, etc. As you can see our optimization algorithm closely approximates the minimal cost configuration which was obtained by checking every single memory configuration. This was possible due to the relatively lower number of functions in the application.

5.3 Q3. *SLAM* configuration finding efficiency and scalability

In Figure 5.5, we show how efficient and scalable *SLAM* is in finding the optimal configurations at various objectives. In Figure 5.5a we can see the time required for



(a) 3-functions test application

(b) 6-functions test application

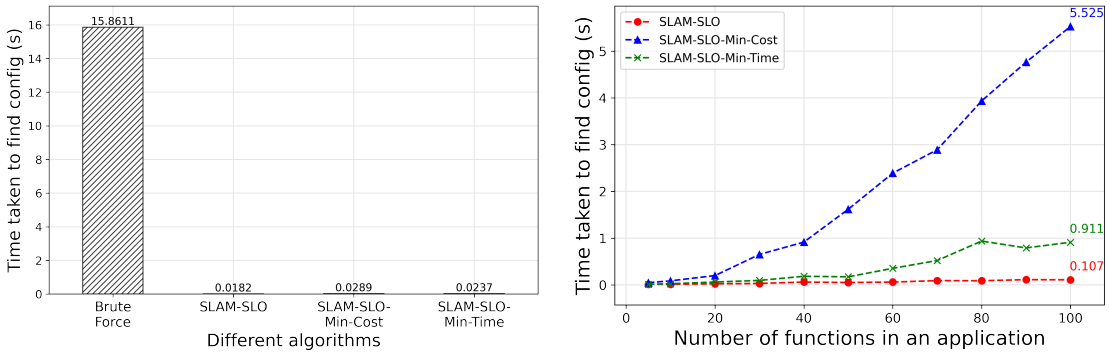
(c) 10-functions test application

(d) Real-world based application

Figure 5.4: Overall execution time and the cost needed by one invocation of the application when configured with memory configurations selected by *SLAM* for various optimization objectives.

different optimization algorithms to find the optimal configuration when run on *6-functions* application. The *Brute-force* algorithm performed worst as compared to the developed optimization algorithm (almost took 871x time more than the developed algorithm). Although, it is possible to parallelize the *Brute-force* search but it is beyond the scope of this work. When comparing *SLAM-SLO* (0.0182s) with *SLAM-SLO-Min-Cost* (0.0289s) and *SLAM-SLO-Min-Time* (0.0237s), *SLAM-SLO-Min-Cost* requires the most amount of time for this application with 6 functions. This can also be validated from the Figure 5.5b where the actual scalability of the three algorithms is tested by running it on applications containing a larger number of functions (from 1 to 100) and *SLAM-SLO-Min-Cost* requires the most amount of time. All algorithms scale linearly with the number of functions in the application but with different slops and *SLAM-SLO* having the least slope.

SLAM-SLO-Min-Cost, which has to estimate the cost at every step of the search, has to go through a higher number of configurations as compared to *SLAM-SLO* and *SLAM-SLO-Min-Time*. Nevertheless, for an application containing 100 functions *SLAM-SLO-Min-Cost* took 5.5s, which is not a lot considering the benefits the algorithm can provide in terms of cost-saving.



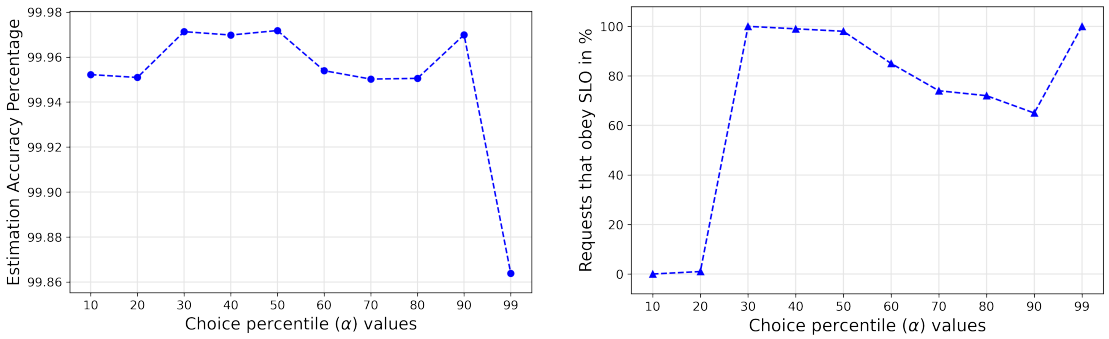
(a) Configuration finding time for different algorithms

(b) Performance of SLAM when the number of functions are scaled

Figure 5.5: SLAM configuration finding efficiency and scalability performance

5.4 Q4. Parameter Sensitivity

SLAM requires tuning of one hyperparameter: choice percentile (α), which currently can either be set manually based on the expert's knowledge or let *SLAM* choose it automatically. In this experiment, we showcase how sensitive this hyperparameter is towards estimation of application overall execution time (see Figure 5.6a) and percentage of requests conforming to the given SLOs (see Figure 5.6b) on *3-functions* test application at a fixed SLO of 1 second. Since all the functions within *3-functions* test applications do not have much variance, therefore variation in α does not have much influence on estimating the overall application execution time (it is above 99% as shown in Figure 5.6a). On the other hand, varying α does influence the percentage of requests conforming to the given SLOs (see Figure 5.6b). When setting low values of α, none of the requests conform to the given SLOs. This is due to the fact that, low values of α means that the estimation of the execution time of the function is the only representative of those a few requests and hence can be the wrong estimation. Furthermore, we see that percentage of requests conforming to the given SLOs increased to 100% till $\alpha = 50$ and then starts to drop with again increasing to 100% at $\alpha = 99$. This shows that it does play a crucial role in the overall accuracy of the *SLAM* and selecting the right value can provide higher accuracy. Therefore, we have added as part of *SLAM* to automatically determine and select the value of α based on some initial data collection as users would not know which value is best for their application.



(a) Execution time estimation accuracy percentage for the 3-functions test application at fixed SLO with varying choice percentile (α) values

(b) Percentage of the requests conforming to the given SLO for the 3-functions test application at varying choice percentile (α) values.

Figure 5.6: SLAM hyperparameter sensitivity analysis

5.5 Other optimization algorithms.

We have extensively explored the performance of the Max-Heap based optimization algorithms to find the necessary configurations to obey our objective. We spent so much effort on understanding the properties of the approach because they scale to hundreds of functions and can bring great improvements having only a runtime of seconds. Unfortunately, the time complexity of the other algorithms described in the previous section doesn't provide similar scalability. Obviously, the brute force approach becomes computationally prohibitive only after having a few functions in the cloud application. Unfortunately the same is true for the binary search approach. Nonetheless, we want to add a small report on the properties of both approaches and create a side-by-side comparison of those 2. We intentionally skipped those 2 in the previous section as the scale of the graphs for SLAM doesn't allow us to see anything meaningful for these two.

Binary search approach.

As explained in the previous sections binary search uses the fact that if there is not a configuration with a given memory satisfying the SLO requirements it means that there is not one with smaller memory either. Using this ordered relation we binary search through the space of all possible configurations trying to find the one which satisfies the requirements.

Table 5.1: Symbols and definitions.

Symbol	Default Value	Interpretation
N	50	total number of requests to the application for load generation
mem_config_list	[128, 256, 512, 1024, 2048, 4096, 8192, 10240]	a list of memory values is used when generating memory configurations for the application.
K	20	total number of requests to the application for load generation
$n^{th} Percentile$	90	n^{th} percentile of the distribution as a representative for the execution time for the given function at a particular memory configuration.

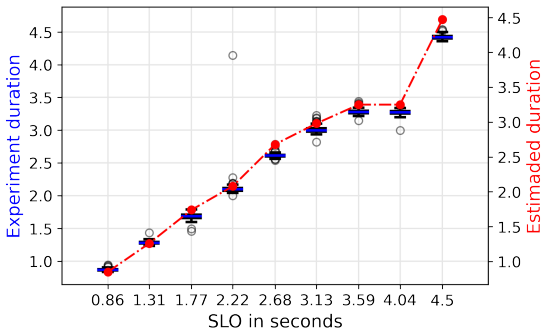


Figure 5.7: Estimation Variation

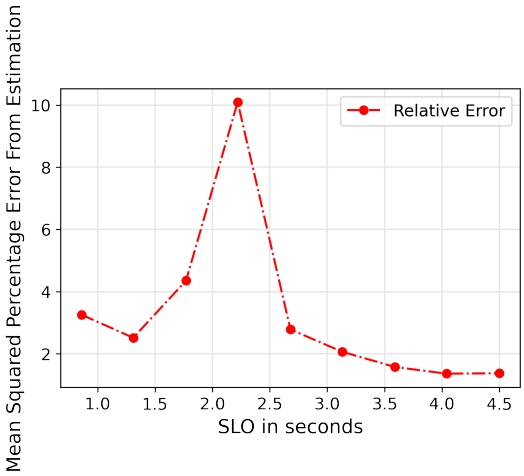
As noted in the chapter the time complexity is

$$O((\log M)^N) \tag{5.1}$$

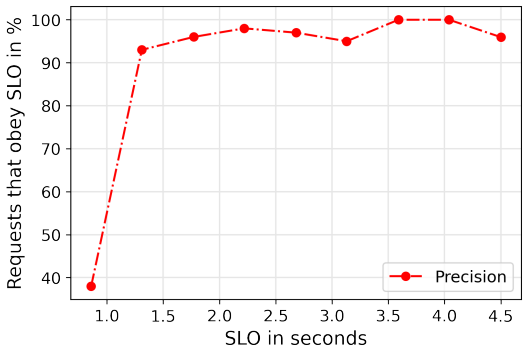
Given the exponential nature, the algorithm doesn't terminate for even 20 functions in a reasonable time. To show the properties of the approach we experimented on a synthetic cloud application comprising 3 functions. This is the same application used in the SLAM algorithm comparison.

We configured the system with the binary search optimization algorithm to generate configurations for SLO in a range between 0.86 and 4.5 seconds. The suggested configurations had different expected runtimes, which were always lower or equal to the SLO. You can see the relationship between those two in the following graph.

Then after getting the configurations we called the application 100 times to check the



(a) Error From Estimation



(b) Precision

Figure 5.8: Binary search method error and precision on 3 function synthetic application.

distribution of durations. In the following graph, we can see how close the estimated durations are to the actual distribution.

Then we measured the percentage of requests that obeyed our preset SLO requirements to gauge the precision and for accuracy measurements, we measured the Mean Squared Percentage error from the estimation. You can see both graphs below.

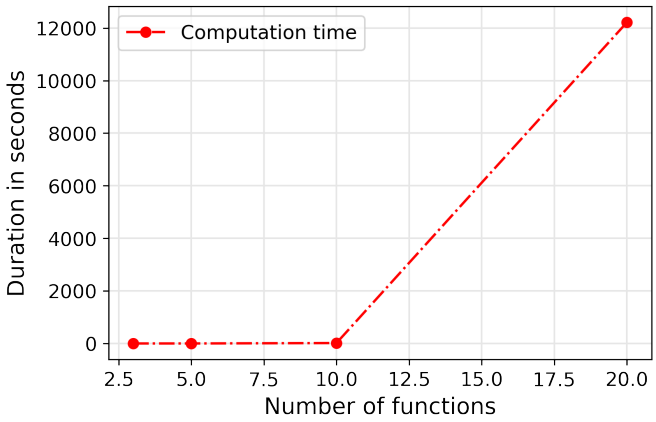


Figure 5.9: Scalability of binary search

Scalability

The real issue with the approach is the lack of scalability. As you can see on figure 5.9, the approach finds the needed configuration under a second for an application consisting of 3 or 5 functions. As we try to compute the results for 10 functions it already takes 15 seconds while for 20 functions we kept running for 3.5 hours and just stopped the experiment at some point as it didn't make sense to wait longer. The point is that it takes prohibitively long and doesn't bring a lot to the table.

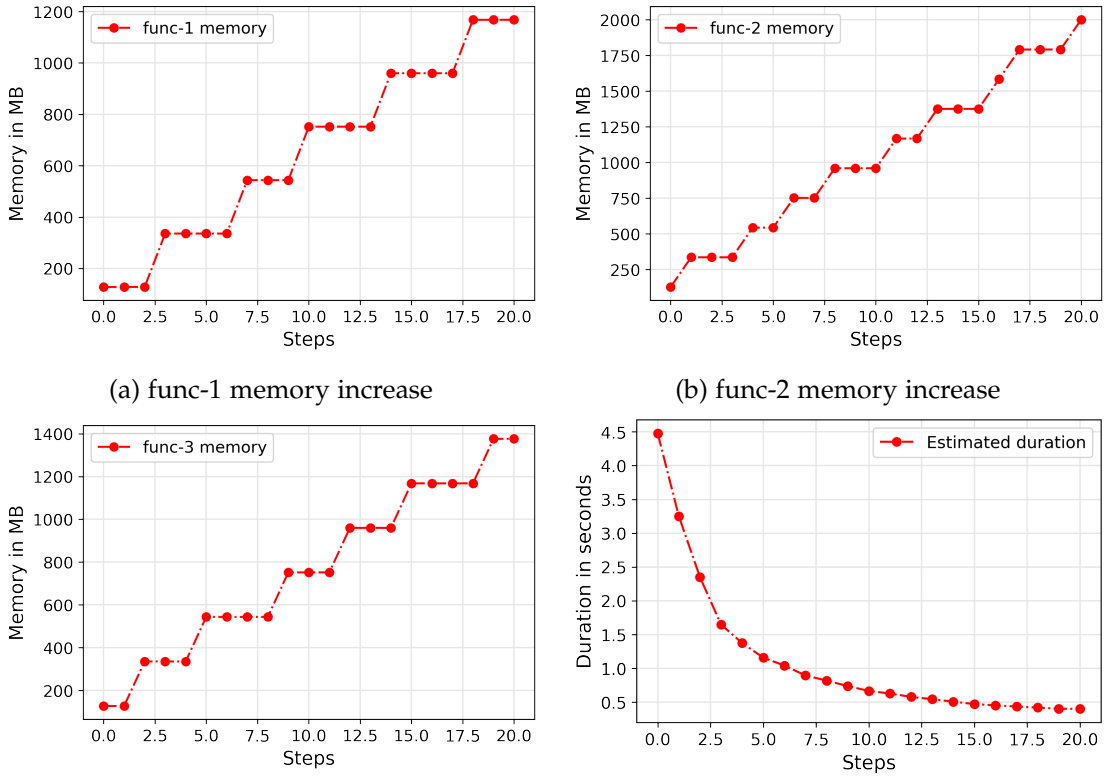
The comparison with such an approach shows how effective the SLAM Max-Heap based algorithms are and how easily they can scale with the number of functions bringing a production-ready tool for its users.

5.5.1 More insight

To gain more insight into the nature of the algorithms let's take a look at the sample run of the vanilla SLAM-SLO algorithm where the SLO is configured to 0.4 seconds. We chose the synthetic 3 function application which you should be familiar with from the previous chapters and present it on the following graphs on Figure 5.10.

First let see how each function's memory changes on each step of the algorithm. First thing you will notice is that each step alters only one function's configuration. All the steps only increment the function's memory. There is no backtracking in the choice of the memory, once a memory has been set it is never decreased.

Take a look at the estimated duration graph, where you notice that the most change is registered in the first few steps of the algorithm. The more the algorithm progresses the more steps are required to bring similar percentage change in the result.



(a) func-1 memory increase

(b) func-2 memory increase

(c) func-3 memory increase

(d) Application duration change

Figure 5.10: SLAM-SLO algorithm run on 3 function synthetic application.

6 Related Work

6.1 Overview

With the advent of serverless computing, there is a significant amount of research aimed at optimizing cloud computing resource utilization [2, 32, 29]. There has been some work on the performance profiling of various FaaS platforms. Wang et al. [50] performed an in-depth study of resource management and performance isolation with three popular serverless computing providers: AWS Lambda, Azure Functions, and Google Cloud Functions. Their analysis demonstrates a reasonable difference in performance between the FaaS platforms. Furthermore, Shahrad et al. [46] studied the architectural implications of serverless computing and pointed out that exploitation of system architectural features like temporal locality and reuse are hampered by the short function runtimes in FaaS. Chadha et al. [16] examine the underlying processor architectures for Google Cloud Functions (GCF) and determine the optimization of FaaS functions using Numba can improve performance by and save costs on average.

Furthermore, there is a significant number of research works aimed at optimizing the memory and cost for the FaaS functions. COSE [1] framework finds the optimal configurations for a FaaS function using the Bayesian Optimization algorithm while minimizing the total cost of execution. It not only models the behavior of a function but also the environment (cloud, edge) in which those functions are deployed. However, they consider FaaS functions separately and optimized based on cost. Bayesian Optimization was also used in CherryPick [3] tool for creating performance models for different cloud applications. The system provides 45-90% accuracy in finding optimal configurations and decreases cost up to 25%. But, they focused on traditional cloud applications. Another framework Astra [30], is designed to optimize FaaS function configurations for specifically map-reduce usecase.

Similar optimization tools have also been developed by Google and Amazon. Google has developed a recommendation system to help the users choose the optimal virtual machine (VM) type [28]. It currently does not support Google Cloud Functions. AWS Compute Optimizer [7] recommends optimal AWS resources for applications to reduce costs and improve performance by using machine learning to analyze historical utilization metrics. It can also be used to find optimal memory configuration for the lambda-based function. However, it can only be executed for the functions whose

allocated memory level is less or equal to 1792MB and which are invoked at least 50 times in the last two weeks. AWS Lambda Power Tuning [15] tool uses exhaustive search to identify optimal memory level for a cost, or execution time. By default, this algorithm will need to perform at least 225 requests to the function to identify the optimal memory point.

None of the aforementioned research efforts address the issue of automatically configuring optimal memory of FaaS functions within a serverless application based on the user-defined SLOs. Thus abstracting the user from defining the memory of FaaS functions: low-level information. Most of the research either addresses a single FaaS function or an application consisting of step functions that do not have complex call graph workflows. The proposed tool *SLAM* fills that gap by creating a recommendation tool that in a short time can find optimal memory configurations of FaaS functions within a serverless application given the SLOs.

6.2 Related systems

Many works are closely related to the theme of our work, but two of those are essentially trying to address the same problem but in different settings. For that reason, we will explore in more detail specifically “Astra: Autonomous Serverless Analytics with Cost-Efficiency and QoS-Awareness” and “COSE: Configuring Serverless Functions using Statistical Learning”.

6.2.1 Astra

Recap

"Astra: Autonomous Serverless Analytics with Cost-Efficiency and QoS-Awareness"[30] is a tool designed to optimize the memory configuration for cloud function applications that are specifically designed for analytics jobs, particularly map-reduce workloads. The tool heavily relies on mathematically modeling both cost and performance aspects of the workflow, specifically implementation details of each step. After modeling the whole workflow, they use a standard algorithm to find the shortest path in a graph to find the optimal configuration for their problem. The authors use Dijkstra’s algorithm as the weights in the graph were all positive and hence it is the most efficient algorithm for such problems. According to the paper they have implemented Astra for AWS lambda and have developed real-world applications to benchmark the tool. They have recorded up to 60% improvement in performance when the budget is fixed and up to 80% cost reduction without violating predefined SLOs.

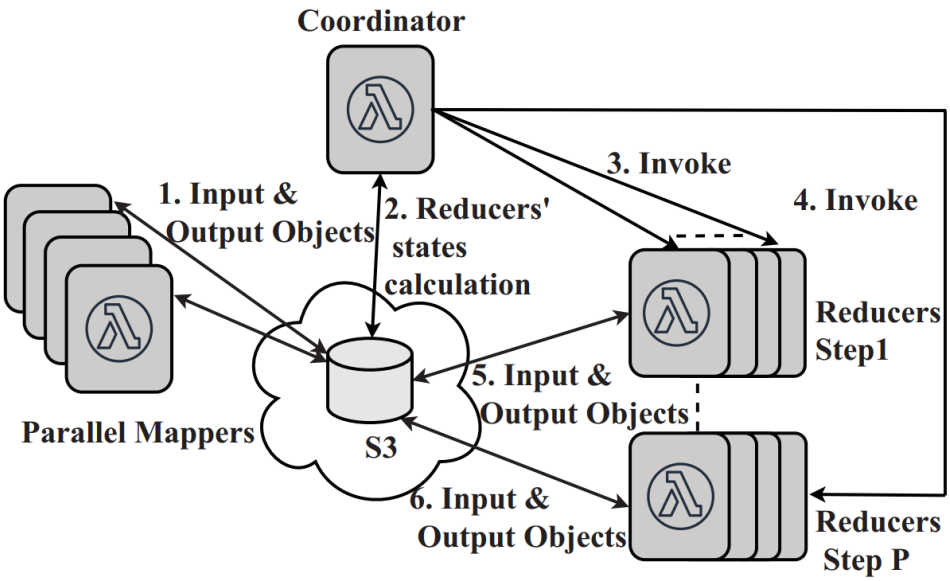


Figure 6.1: Components of Astra framework [30]

Composition of framework

Astra is built for optimizing serverless analytics workflows, so it is of utmost importance to understand the composition of such a framework as it vastly differs from the traditional server-based map-reduce setting. There are three main components of such a framework: mapping functions, reducing functions, and coordinator functions. As the name suggests mapping and reducing functions correspond to the map and reduce part of any workflow in a traditional map-reduce job. An interesting role has the coordinator function which after finishing each step rebalances the work between the appropriate number of mappers or reducers to ensure successful completion of each step. They also make sure that in case of an error or a failure the work is retried to compilation. As a communication mechanism, there are usually 2 main choices for function interaction: external in-memory storage (Memcached, Redis, etc) [35] [42] or hard disk store (bucket, networked file system, etc.). For the case of Astra, as the framework was developed in the AWS environment, the main choice was among S3, EFS, and Elasticache [6]. The authors opted for S3 as a medium of communication among mappers and reducers.

Cost and Performance modeling

One of the most comprehensive aspects of the paper was the detail relating to the cost and performance modeling. Although it would have been very useful if the authors provided the accuracy of their estimates, the results, in the end, prove that the estimates were close enough to deliver the optimization gains that they achieved. Performance: The approach that Astra takes is to divide the workflow into distinct steps and build models for each one of them. For the case of their application at hand, the whole lifetime of the application is the exact sum of the parts, which makes it easier to generalize and give an estimate for the whole duration of the application. As far as the performance is concerned the authors divide the duration of the application into 3 parts: the lifetime of mappers, the lifetime of coordinators, and the lifetime of reducers.

The lifetime of mappers is essentially the sum of the time spent downloading and uploading the files to the S3 bucket plus the time for computation. To calculate the time spent on networking the authors divide the sum of input and output files, by the bandwidth.

$$t = (in + out)/B \tag{6.1}$$

where in is the input data, out is the output data, and B is the network bandwidth. To estimate computation time, they find the time it takes for a mapper to process a unit amount of input and multiply by the actual input size. Then to find the duration for the whole phase they take the maximal value of all mappers as the phase is not over until the slowest mapper is over.

The lifetime of the coordinator is solely determined by the time taken to interact with S3 buckets. As of itself the coordinator doesn't do any computation but shuffles the data into separate files for the mappers or reducers to pick up from there. Therefore, the duration of the phase will be

$$t = P * l/B \tag{6.2}$$

where P is the number of reducers or mappers of the next phase, l is the estimated file size for each input and B is the bandwidth of S3 connection.

The lifetime of the reducer is a bit more complicated than its counterparts. Reducing is a multistep process where the output of the reducers might need to be fed into other reducers. Also, the multistep reducers need to communicate with each other, and they employ the same S3 technology as a middleman to transfer the state from one step to the other. To take into account all of these complications the authors came up with the following formula for the whole duration:

$$t = Q_P \sum_{s=1}^{L} z_s u_s \quad s \in 1, 2...L \tag{6.3}$$

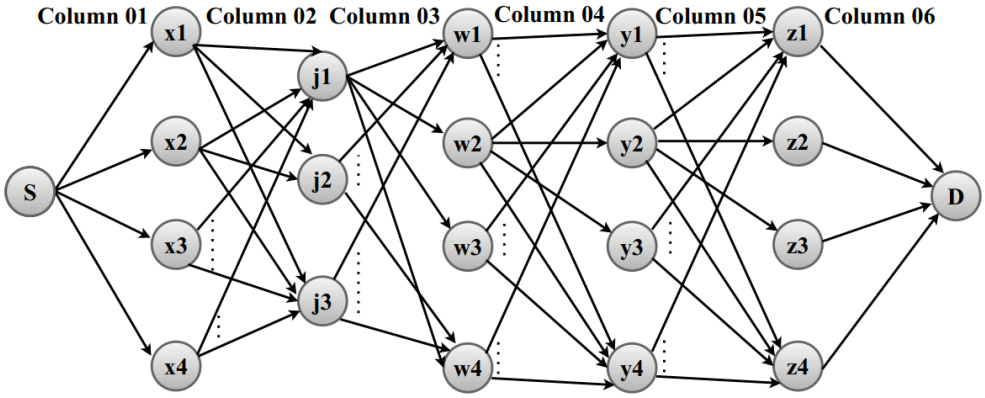


Figure 6.2: Optimization graph built by Astra framework[30]

Where Q_P is the total input size, z_s is 1 if s-th memory configuration is chosen for the reducers and 0 otherwise. u_s is the duration reducer takes for processing unit amount of data.

Cost estimation

After estimating the duration for each step the cost estimation becomes easier. The authors have divided the cost of the workflow into 3 different parts.

S3 request cost. The authors have calculated the number of requests as well the bandwidth and amount of data transfer required for the compilation of the whole workflow. **S3 storage cost.** Effectively the user needs to store the intermediate or so-called ephemeral data only in between two different phases and can be deleted once the next phase is over. The authors taking this into account calculate the cost of storage in S3 buckets. **Lambda runtime cost.** Having calculated the duration for each phase the authors now can easily multiply those by the cost of unit execution given they have chosen the lambda memory configuration and AWS execution region.

Summing these up we get the estimation for both the cost execution time for the whole run of the process.

Optimization technique

Having built this performance and cost estimation framework the authors have come up with an interesting optimization technique to balance the cost and performance. They have devised a graph that starts at a source S and ends at destination D. There are 5 layers in the Directed Acyclic Graph. Each layer correspondingly represents

one aspect of the optimization problem: the memory allocation for mapper lambdas, the number of mappers, the number of objects per reducers, the memory allocation for coordinator lambda, and the memory allocation for reducer lambdas. Each layer consists of many nodes which represent different values for that layer that it describes. For example node x_1, x_2, x_3 and x_4 represent different memory configurations for all mappers. The edges in between layers represent a choice of a certain configuration of that layer. The weights of the edges represent the corresponding completion time for the mapper, reducer, or coordinator. ' So having a source S and a destination D, a path between them completely represents a fixed configuration of the system. So the challenge is to find the minimum cost path connecting S and D. Thus using Dijkstra's algorithm one can easily find the optimal configuration. In case we want to find the path for the minimal monetary cost we can swap the edge values with the monetary value of the configuration and run the same algorithm.

Discussion and comparison with SLAM

According to the paper, Astra has reached a performance improvement of 20 to 60 percent when the budget was fixed and a cost reduction of 20 to 80 percent without violating the predefined SLOs. These are remarkable results but there are some issues left unexplained in the paper. First, the authors don't present any metrics on how good their estimations are. While developing SLAM we have encountered variance in the execution duration and that was something that needed to be addressed separately. In the paper, the authors just assume that they have a representative unit of execution duration. Secondly, the authors assume that all mappers need to be configured to the same memory. The same assumption is done for the reducers but keep in mind that the reducers' memory configuration is not tied to the mappers. The authors leave out the case when there are multiple different reducers and mappers which do different types of tasks. Thirdly, a characteristic that is shared with SLAM is the fact that Astra chooses discrete values for the mapper and reducer configuration, which can be improved to a continuous one both in SLAM and Astra. Like SLAM, the complexity of the algorithm is tightly coupled with the number of memory configurations. Fourth, the authors don't show how scalable the algorithm is. They mention that it takes only 2 seconds to evaluate the test examples that they have tried, but they don't show what would happen if we wanted to have different types of mappers and correspondingly layers in the optimization graph. This would greatly increase the credibility of the paper, as it is not obvious how much the computation time would increase.

Overall the paper achieves significant improvements over the chosen baseline and similar SLAM at some deeper level uses Dijkstra's algorithm to solve the optimization problem. Although the construction of the graph and the performance modeling are

vastly different, Astra and SLAM share the same optimization engine to solve a similar performance and cost balancing problem.

6.2.2 COSE

"COSE: Configuring Serverless Functions using Statistical Learning"[1] is a similar framework to Astra/SLAM and has the same objectives of minimizing cost without hurting performance. They show a different approach to the problem, namely using statistical learning methods to collect necessary data points and predict the behavior of the cloud functions. To predict the performance of the functions they use Bayesian Optimization, while finding the best configuration for the functions they rely on Integer Linear Programming. Unlike other similar applications such as ARIA [49] and Astra[30] (ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments), COSE can self configure on the fly and adapt to the changes in the environment. According to the paper for 95% of the time, the system requires only 5 measurements to start suggesting optimal memory/CPU configurations for the cloud application. They have tested COSE on AWS Lambda as well as on a simulated environment to check the validity of their models.

Architecture

The architecture of the system is quite similar to SLAM. It consists of 2 major parts: Performance Modeler and Config Finder. While the SLAM is described on a higher resolution resulting in more parts the main idea is the same and shared among similar systems.

Performance Modeler

While designing the performance modeler the authors have gone through several variants before choosing the Bayesian optimization model. First, they rolled out the exhaustive search which would take a prohibitive amount of time to run due to its exponential nature. Secondly, they rolled out algorithms based on parameter descent. They have constructed a specific case for the Additive Increase and Additive Decrease type of algorithms, for which the algorithm gets stuck in a loop. The main idea is that if a function gets scheduled on different machines one after the other the tool would get confused and get trapped in a loop. So they have finally chosen to rely on statistical learning particularly Bayesian optimization. The method builds a probabilistic model for the underlying characteristic function for the performance of cloud function. A nice feature of Bayesian optimization is that it dynamically adapts the next point of query based on the confidence interval. Henceforth it avoids making unnecessary queries.

Every Bayesian optimization model needs an acquisition function and the authors chose EI - Expected Improvement for the problem. The acquisition function chooses the next point of query, thus it is important to choose it wisely. It has been shown that EI outperforms its counterparts and does not have any hyperparameters to tune.

Tuning Bayesian optimization model

Vanilla Bayesian optimization does not reach high enough accuracy for that reason the authors tailored the method to serve the peculiarities of serverless functions.

Initial point of search. Having observed that the duration and memory configuration have a convex relationship between each other, the authors chose the initial point from a uniform distribution covering all of the memory range. For the experiments that they describe they chose 4 points.

Reduction of search space. To limit the computational complexity of the algorithm they chose a fixed amount of memory configuration for their AWS lambda functions. Particularly, they fixed the possible memory configurations between 128MB and 3008 MB with 64 MB of increments, resulting in 46 different points. Also because the lambda can be deployed in Edge or in Core cloud we can assume that realistically the algorithm had to choose between 92 points, as functions can be scheduled in both environments.

Noise in data. As we have noted in our research duration of a function call is not always precisely the same and has a quite variance to it. This has to do with the fact of collocation, cold-start, hardware heterogeneity, balancing, and so on. To combat the problem the authors introduce a Gaussian noise to the system with the hyperparameter α with a value of 0.01. Adapting to the change in performance. COSE-s ability to dynamically adapting to the changes in performance comes from the fact that the system discards old sampled points as the new ones are collected. The more frequent the sampling is the more recent the data points tend to be as there are only a fixed number of data points that COSE can use for its predictions.

Convergence. The convergence criterion for the system is tied to the acquisition function. For the case at hand, when Expected Improvement is below 5% mark for the next probe, the system stops collecting data and is deemed as converged.

Config finder

The cost estimation component is similar to the one we have employed in SLAM, which is just the product of function lifetime and cost given the configurations. It is no way near to the detailed estimated given by Astra where they incorporate network and storage fees as well. To find the optimal configuration the COSE relies on Integer Linear Programming. They calculate the cost and form inequality using the budget at hand

then use the CPLEX solver to find a fitting configuration. This raises the question of the complexity of the solver as it is well known that Integer Linear Programming is NP-complete. The authors also leave the scalability of the system out of the paper evaluating cloud applications that have up to 2 functions.

Evaluation and results

To evaluate the accuracy and the performance of the system they have tested the system on 4 representative functions having I/O-, CPU-, network-, memory-intensive tasks. According to the paper, COSE managed to find the optimal configuration for I/O, CPU, and Memory intensive tasks but failed at network-intensive one as changing the memory of the function had little to no impact on the running time of the function. To further look into more complex examples they simulated a cloud provider modeling the factors that affect the duration of a function call such as co-location of functions, cold starts, the lifetime of an execution environment, edge/core cloud environment, pricing, execution time, and the dynamicity of the execution nature of functions. For more details on the simulation, you can take a look at the paper. Having simulated the cloud environment the authors successfully tested the system for chained functions of a length 2 and found that COSE found the optimal configuration.

Discussion and comparison with SLAM

COSE brings statistical methods particularly Bayesian optimization into the memory optimization problem which both Astra and SLAM are aimed to solve. Remarkably it does only a few measurements to achieve convergence and bring highly accurate results. A very important aspect of the system is its dynamicity as it can adapt to the changes in the system on the fly. Nonetheless, some issues are not addressed in the paper as to how COSE handles those. First what if there is an internal dependency among functions and one waits for another to finish. This is a common pattern when developing commercial cloud applications. They cover the case of function chaining which is the easy case and has no overlap over the function runtimes.

Secondly, the choice of Linear Integer Programming as an optimization tool is debatable as the problem is famous for being NP-complete. This might be a problem when the function lifetimes are tangled like the case we look at in SLAM and running a CPLEX solver would be prohibitively costly. The paper would be much more thorough if the authors included a runtime duration analysis for applications having 50, 100, and 150 functions which would shed light on the scalability of the system.

Nevertheless, the paper is a remarkable step in the optimization problem as it brings a new approach to the table and proves it to be effective in many circumstances.

7 Conclusion and Future work

Serverless computing has abstracted most of cloud server management and infrastructure scaling decisions away from the users but configuring the memory of FaaS functions: a low-level configuration, which directly influences the performance and cost of the FaaS functions, is still left up to the users.To solve this problem, we introduced **SLAM** to find optimal memory configurations given predefined SLO requirements.

SLAM uses a max-heap-based optimization algorithm along with its variants for various optimization objectives (minimum cost and minimum overall time) in finding the optimal memory configuration for the given serverless application based on the specified SLO. It supports complex serverless application call-graph workflows and has the ability to adapt to changes in a serverless application. We demonstrate the functionality of *SLAM* with AWS Lambda (§5) on four serverless applications comprising of a various number of functions and found that the suggested memory configurations guarantee that more than 95% of requests are completed within the defined SLOs.

In the future, we plan to extend *SLAM* with other public serverless compute providers and to open source FaaS platforms. Currently, we support memory configurations provided by the user, and the complexity of the algorithms is highly coupled with the number of elements in the memory configuration list. Transitioning from a discrete search space for the memory configurations to a continuous one could be the next improvement for the *SLAM*. We would also like to create the probabilistic version of the tool, where given a configuration and SLO, the tool would give the expected percentage of requests that would complete in less time than SLO.

List of Figures

List of Tables

Bibliography

[1] N. Akhtar, A. Raza, V. Ishakian, and I. Matta. "COSE: Configuring Serverless Functions using Statistical Learning." In: *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*. 2020, pp. 129–138. DOI: 10.1109/INFOCOM41043.2020.9155363.

[2] M. Akin. *How does proportional CPU allocation work with AWS Lambda? | Opsgenie Engineering*.

[3] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. "Cherrypick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics." In: *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. NSDI'17. Boston, MA, USA: USENIX Association, 2017, pp. 469–482. ISBN: 9781931971379.

[4] Amazon Lambda. https://aws.amazon.com/lambda/. Accessed on 09/24/2020.

[5] *An introduction to Azure Functions*. URL: https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview.

[6] AWS. *AWS square enix case study*. https://aws.amazon.com/de/solutions/case-studies/square-enix/. Accessed: 2020-04-17.

[7] *AWS Compute Optimizer*. (Accessed on 06/17/2021). 2021.

[8] *AWS Distro for OpenTelemetry*. https://aws.amazon.com/otel/?otel-blogs.sort-by=item.additionalFields.createdDate&otel-blogs.sort-order=desc. Accessed: 2020-05-02.

[9] *AWS Lambda*. URL: https://aws.amazon.com/lambda/.

[10] *AWS Lambda releases*. 2020. URL: https://docs.aws.amazon.com/lambda/latest/dg/lambda-releases.html (visited on 05/31/2020).

[11] *Azure Functions hosting options, https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale*. Accessed on 02/18/2021.

[12] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter. "Serverless Computing: Current Trends and Open Problems." In: *Research Advances in Cloud Computing*. Springer Singapore, 2017, pp. 1–20. DOI: 10.1007/978-981-10-5026-8_1.

[13] R. Byrro. *Can We Solve Serverless Cold Starts?* https://dashbird.io/blog/can-we-solve-serverless-cold-starts/. Accessed: 2020-04-17. 2019.

[14] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz. "Cirrus: A serverless framework for end-to-end ml workflows." In: *Proceedings of the ACM Symposium on Cloud Computing*. 2019, pp. 13–24.

[15] A. Casalboni. *AWS Lambda Power Tuning*.

[16] M. Chadha, A. Jindal, and M. Gerndt. *Architecture-Specific Performance Optimization of Compute-Intensive FaaS Functions*. 2021. arXiv: 2107.10008 [cs.DC].

[17] M. Chadha, A. Jindal, and M. Gerndt. "Towards Federated Learning Using FaaS Fabric." In: *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*. WoSC'20. Delft, Netherlands: Association for Computing Machinery, 2020, pp. 49–54. ISBN: 9781450382045. DOI: 10.1145/3429880.3430100.

[18] *Cloud Functions Overview*. https://cloud.google.com/functions/docs/concepts/overview. (Accessed on 08/22/2020).

[19] CloudFlare. *Why use serverless computing?* https://www.cloudflare.com/learning/serverless/why-use-serverless/. Accessed: 2020/12/16.

[20] *DataDog tracing technology*. https://docs.datadoghq.com/tracing/. Accessed: 2020-05-02.

[21] S. Eismann, L. Bui, J. Grohmann, C. L. Abad, N. Herbst, and S. Kounev. *Sizeless: Predicting the optimal size of serverless functions*. 2021. arXiv: 2010.15162 [cs.DC].

[22] A. Eivy. "Be Wary of the Economics of "Serverless" Cloud Computing." In: *IEEE Cloud Comput.* 4.2 (2017), pp. 6–12. DOI: 10.1109/MCC.2017.32.

[23] T. Elgamal, A. Sandur, K. Nahrstedt, and G. Agha. "Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement." In: *CoRR* abs/1811.09721 (2018). arXiv: 1811.09721.

[24] E. van Eyk, A. Iosup, S. Seif, and M. Thömmes. "The SPEC Cloud Group's Research Vision on FaaS and Serverless Architectures." In: *Proceedings of the 2nd International Workshop on Serverless Computing*. WoSC '17. Las Vegas, Nevada: Association for Computing Machinery, 2017, pp. 1–4. ISBN: 9781450354349. DOI: 10.1145/3154847.3154848.

[25] Firecracker. https://firecracker-microvm.github.io/. Accessed on 09/24/2020.

[26] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein. "From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers." In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 475–488. ISBN: 978-1-939133-03-8.

[27] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski. "Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research." In: (Aug. 27, 2017). DOI: 10.13140/RG.2.2.15007.87206. arXiv: 1708.08028 [cs.DC].

[28] *Google Cloud Recommendations*. (Accessed on 06/17/2021). 2018.

[29] J. Grogan, C. Mulready, J. McDermott, M. Urbanavicius, M. Yilmaz, Y. Abgaz, A. McCarren, S. MacMahon, V. Garousi, P. Jamshidi, et al. "An analysis of Function-as-a-Service (FaaS): vendors, challenges and implications for software developers." In: ().

[30] J. Jarachanthan, L. Chen, F. Xu, and B. Li. "Astra: Autonomous Serverless Analytics with Cost-Efficiency and QoS-Awareness." In: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2021, pp. 756–765. DOI: 10.1109/IPDPS49936.2021.00085.

[31] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. "Occupy the cloud: Distributed computing for the 99%." In: *Proceedings of the 2017 Symposium on Cloud Computing*. IEEE. 2017, pp. 445–451.

[32] S. Kulkarni. *Optimize AWS lambda memory | Towards Data Science*.

[33] K. Lane. "Overview of the backend as a service (BaaS) space." In: *API Evangelist* (2015).

[34] *Maria DB*. https://mariadb.org/. Accessed: 2020-05-02.

[35] *Memcached*. https://memcached.org/. Accessed: 2020-05-02.

[36] Michele Sciabarrà. "O'Reilly Media, Inc.", 3 Jul 2019. Learning Apache OpenWhisk: Developing Open Serverless Solutions.

[37] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov. "Agile cold starts for scalable serverless." In: *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX Association. 2019.

[38] *New Relic tracing technology*. https://www.jaegertracing.io/. Accessed: 2020-05-02.

[39] *New Relic tracing technology*. https://newrelic.com/products/edge-infinite-tracing. Accessed: 2020-05-02.

[40] *OpenTelemetry: Future-Proofing Your Instrumentation*. https://newrelic.com/blog/best-practices/opentelemetry-opentracing-opencensus. Accessed: 2020-05-02.

[41] QEMU documentation. https://www.qemu.org/documentation/. Accessed on 09/24/2020.

[42] *Redis*. https://redis.io/. Accessed: 2020-05-02.

[43] M. Roberts. *Serverless Architectures*. https://martinfowler.com/articles/serverless.html. Accessed: 2020-04-17. 2018.

[44] Security Overview of AWS Lambda. https://docs.aws.amazon.com/whitepapers/latest/security-overview-aws-lambda/security-overview-aws-lambda.pdf. Accessed on 09/24/2020.

[45] Serverless Architectures with AWS Lambda. https://d1.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf. Accessed on 09/24/2020.

[46] M. Shahrad, J. Balkind, and D. Wentzlaff. "Architectural implications of function-as-a-service computing." In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 1063–1075.

[47] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley. "Numpywren: Serverless linear algebra." In: *arXiv preprint arXiv:1810.09679* (2018).

[48] J. Spillner. "Resource Management for Cloud Functions with Memory Tracing, Profiling and Autotuning." In: *WoSC@Middleware 2020: Proceedings of the 2020 Sixth International Workshop on Serverless Computing, Virtual Event / Delft, The Netherlands, December 7-11, 2020*. ACM, 2020, pp. 13–18. DOI: 10.1145/3429880.3430094.

[49] A. Verma, L. Cherkasova, and R. Campbell. "ARIA: automatic resource inference and allocation for MapReduce environments." In: Jan. 2011, pp. 235–244. DOI: 10.1145/1998582.1998637.

[50] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. "Peeking behind the curtains of serverless platforms." In: *2018 {USENIX} Annual Technical Conference ({USENIX} {ATC} 18)*. USENIX Association. 2018, pp. 133–146.

[51] C. S. WG. *Cncf wg-serverless whitepaper v1. 0*. https://gw.alipayobjects.com/os/basement_prod/24ec4498-71d4-4a60-b785-fa530456c65b.pdf. [Online; Accessed: 15-July-2020]. 2018.

[52] *What is Apache OpenWhisk?* 2020. URL: https://openwhisk.apache.org/ (visited on 07/20/2020).

[53] *Zipkin*. `https://zipkin.io`. Accessed: 2020-05-02.