ISSN: 2959-6386 (Online), Vol. 2, Issue 3

Journal of Knowledge Learning and Science Technology

journal homepage: https://jklst.org/index.php/home

Implementing Serverless Architecture: Discuss the practical aspects and challenges

Prachi Tembhekar[1], Lavanya Shanmugam[2], Munivel Devan[3]

[1]Amazon Web Services, USA
[2]Tata Consultancy Services, USA
[3]Fidelity Investments, USA

Abstract

Function as a Service (FaaS) has garnered significant attention for its approach to deploying computations to serverlessbackends across various cloud environments. It simplifies the complexity of provisioning and managing resources for applications by leveraging cloud providers' capabilities, offering users an illusion of perpetual resource availability. Among these providers, the AWS serverless platform stands out, offering a novel paradigm for cloud application development, abstracting away concerns about underlying hardware infrastructure while ensuring scalability, security, and cost-effectiveness.

However, due to the absence of standardized benchmarks, serverless functions often rely on ad-hoc solutions for building cost-efficient and scalable applications. The development of the SeBS framework has addressed this gap, enabling comprehensive testing, evaluation, and performance analysis across different cloud providers. While previous research has explored serverless platforms among various providers, little attention has been given to AWS Lambda service's performance within the ARM64 architecture and its comparison with traditional x86 architectures.

ArticleInformation:
Article history: *Received:*01/11/2023 *Accepted:*10/11/2023 *Online:*30/12/2023 *Published*: 30/11/2023
DOI: https://doi.org/10.60087/jklst.vol2.n3.p580

Introduction

Cloud computing, an interconnected network of computers or servers worldwide accessed via the internet, comprises two key components: the front end and the back end. The front end facilitates user access to cloud computing systems through various devices and applications, while the back end houses the hardware infrastructure necessary for cloud computing operations [18]. Maintenance of infrastructure in the backend is streamlined as

hardware components are segregated from application development processes. This chapter elucidates the fundamental concept of serverless computing, delineates different types of cloud computing services, and outlines the structure of the thesis.

Types of Cloud Computing Services

Understanding cloud computing entails familiarity with its various deployment models, including public, private, and hybrid [7].

Private Cloud:

This model entails infrastructure exclusive to a single business, hosted either in-house or externally. While it may incur higher costs, the private cloud offers superior security, computing power, and customizability, making it an ideal choice for organizations prioritizing these aspects.

Public Cloud:

In the public cloud model, infrastructure is shared among multiple organizations. With expansive scalability and pay-per-use payment models, it is commonly provided by third-party providers such as Amazon Web Services, Salesforce, Microsoft Azure, and Google Cloud.

Hybrid Cloud:

Combining elements of both public and private clouds, the hybrid cloud offers a balance of security and cost-effectiveness. However, integrating the two models may present communication challenges.

Moreover, cloud computing services can be broadly categorized into Platform-as-a-Service (PaaS), Infrastructure-as-a-Service (IaaS), and Software-as-a-Service (SaaS) [1].

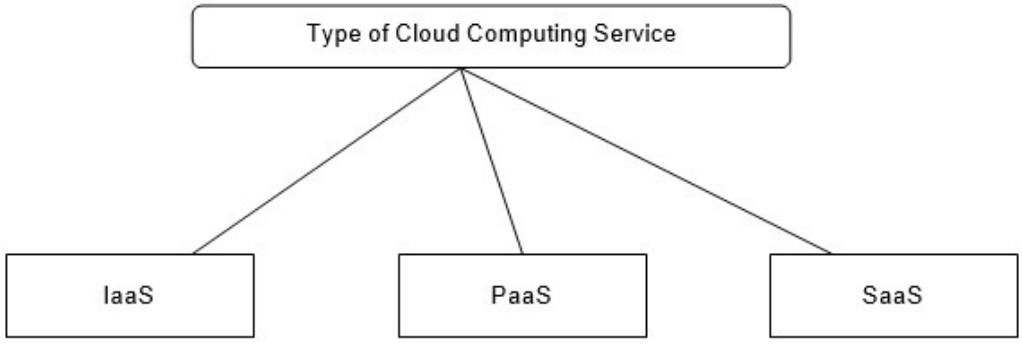


Figure 1.1. Types of Serverless Cloud Computing Services

IaaS:

Infrastructure as a Service (IaaS) offers organizations on-demand access to cloud computing resources via the internet, allowing for a pay-as-you-go model. It encompasses virtual machines, servers, storage, networks, and operating systems, all provided by cloud vendors. However, the multi-tenant architecture of IaaS introduces data security concerns.

PaaS:

Platform as a Service (PaaS) enables developers to lease cloud computing infrastructure for the entire application lifecycle, including development, testing, deployment, and maintenance phases. Designed to provide developers with a readily accessible environment, PaaS facilitates rapid development of mobile and web applications without the need to manage software infrastructure. Nevertheless, the support, reliability, and speed of PaaS solutions are heavily reliant on the vendor.

SaaS:

Software as a Service (SaaS) represents the simplest cloud computing model. Various providers grant access to their infrastructure over the cloud via APIs or web browsers, eliminating the need for installation on the host computer. Examples include Gmail, Outlook, Salesforce CRM, Jira, and Trello. While SaaS offers convenience, users must maintain network connectivity, and they relinquish control over the underlying infrastructure when using SaaS solutions.

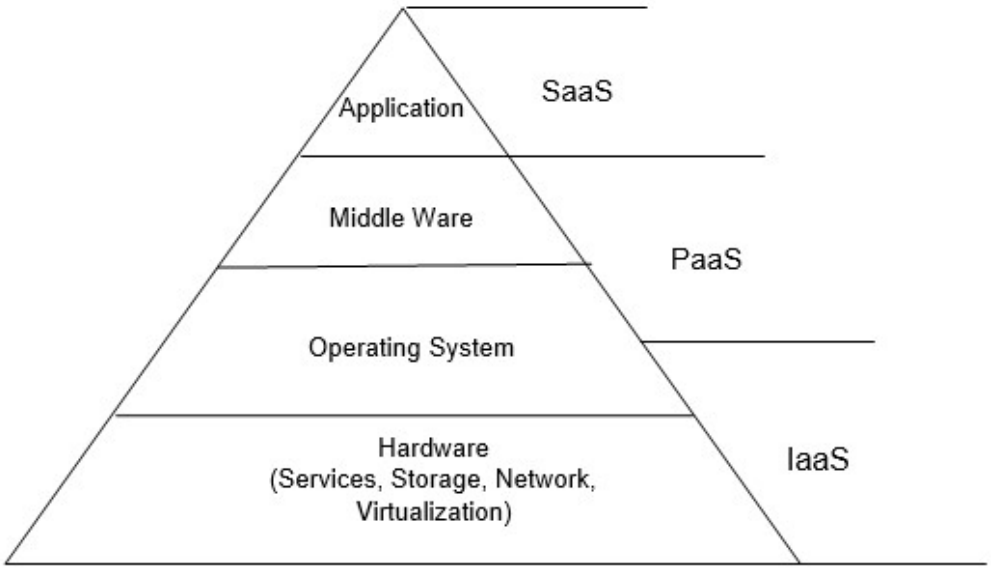


Figure 1.2. Cloud Computing Structure

Serverless

There's a common misconception that "serverless" implies the absence of servers. However, it simply means that developers focusing on business logic need not concern themselves with the server infrastructure behind the scenes. In essence, developers are relieved from the tasks of creating, maintaining, and deploying servers, hence the term "serverless" [6].

Evolution of Serverless:

The rise of containers and the availability of on-demand cloud computing infrastructure from various cloud providers have propelled the evolution of serverless architecture and serverless computing in tandem. Tracing the evolution of serverless reveals three distinct phases.

In the "Serverless 1.0" phase, numerous limitations rendered it unsuitable for general computing tasks. It primarily supported HTTP and a few other resources, with limited execution times (typically 5-10 minutes), lacking

orchestration capabilities, and offering minimal local development experiences. The subsequent "Serverless 1.5" era emerged with the introduction of Kubernetes, enabling auto-scaling of containers in various serverless frameworks. This phase is characterized by Kubernetes-based auto-scaling, microservices, function-based architectures, simplified debugging, and local testing capabilities, while also emphasizing portability.

The ongoing "Serverless 2.0" era represents the most current phase, marked by advancements in state management and integration. Many cloud providers have devised solutions to render serverless architectures more suitable for general-purpose workloads. This phase incorporates elements of enterprise Platform-as-a-Service (PaaS) and features improved state handling, enterprise integration patterns, and more [12].

Serverless Architecture:

Serverless architecture deviates from traditional cloud computing models by placing the responsibility for scaling applications and managing infrastructure squarely on cloud computing providers. Serverless applications deployed within containers are automatically scaled up or down based on demand. One of the simplest serverless architecture patterns in AWS is illustrated in Figure 1.3. Here, the API gateway serves as an asynchronous invoker.

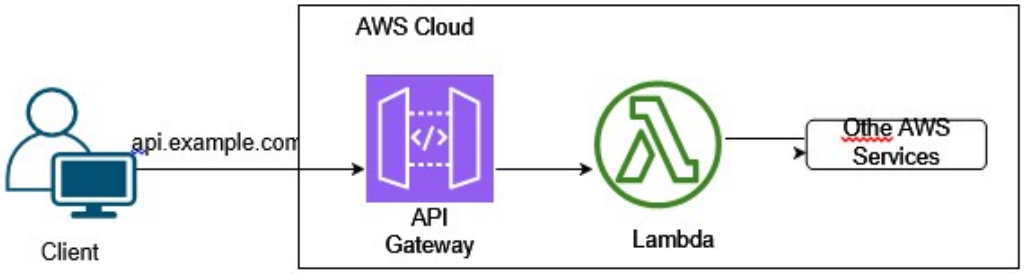


Figure 1.3. A Simple Serverless Architecture

Backend-as-a-Service (BaaS) and Function-as-a-Service (FaaS)

Serverless computing encompasses Backend-as-a-Service (BaaS) and Function-as-a-Service (FaaS). BaaS grants developers access to a variety of third-party applications and services, such as authentication services, encryption, cloud-accessible databases, and more. APIs play a crucial role in invoking serverless functions within the BaaS model. However, when developers refer to serverless computing, they predominantly focus on the FaaS model. In FaaS, developers write custom server-side logic while cloud providers handle the underlying infrastructure. FaaS facilitates a streamlined approach to serverless computing, allowing developers to concentrate solely on coding and delivering business value. Moreover, it operates on an event-driven execution model, executing functions within stateless containers

Pros and Cons of Serverless Computing

Serverless computing offers several advantages and disadvantages, as outlined below:

Pros:

- Increases developer productivity and reduces operational costs by allowing developers to focus on application development rather than managing servers and provisioning resources.
- Facilitates the adoption of DevOps practices by minimizing developers' involvement in defining infrastructure

requirements.
- Streamlines application development through third-party BaaS offerings, optimizing the development process.
- Reduces operational costs significantly by only paying for resources when they are in use.

Cons:

- Lack of control over server-side logic, as it is managed by cloud providers.
- Absence of persistent state, as functions are triggered by events.
- Flexibility and customizability constraints, as cloud providers may impose limitations on their components.
Similarly, in the BaaS environment, developers can access services but have limited control over the underlying code.

Motivation and Thesis Goal

While several cloud platforms exist, AWS stands out as the most popular, commanding a significant share of the global market. Notably, AWS has recently introduced a new feature, "ARM64," a 64-bit processor architecture, in its Lambda service, a serverless feature. This addition complements the existing x86 architecture, with AWS asserting that ARM64 offers superior price-performance.

Despite numerous studies comparing serverless platforms from different cloud providers, no research has yet examined AWS Lambda's performance in ARM64 architecture and compared it to the existing x86 architecture. This presents an opportunity for research, motivating the focus of this thesis. The goal is to conduct a comparative analysis between x86 and ARM64 architectures within AWS Lambda, utilizing various tested workloads and BSD-3 clause licensed benchmarks. These benchmarks will be invoked synchronously in AWS using HTTP API Gateway.

Thesis Layout

The structure of the thesis is as follows: Chapter 2 reviews background information and related work, detailing the data collection methodology employed for this research. Chapter 3 delves into Amazon Web Services (AWS), Lambda functions, and their invocation mechanisms. Chapter 4 outlines the process of building Lambda functions in AWS and details the data collection process for various workloads using different metrics. Chapter 5 presents the results obtained and conducts analysis. Finally, Chapter 6 discusses the findings, provides conclusions, and outlines potential future research directions.

Additionally, appendices are included to support the thesis work. Appendix A explains how to obtain AWS secret keys necessary for invoking benchmarks in AWS Lambda. Appendix B provides additional information on host requirements and environment variable setup. Lastly, Appendix C discusses workload invocation, experiment execution, result processing, and includes additional plots supporting the obtained results.

Background

In serverless computing platforms, an application comprises one or more functions, which are typically standalone, small, and stateless components designed to execute specific tasks. These functions consist of code written in specific programming or scripting languages and execute within dedicated instances known as function instances. A function instance can be a container or sandbox with limited memory and CPU resources provided by public cloud providers [15]. When invoked by a function request, one or more function instances are launched to execute the function. After processing the request, function instances become idle and may be reused to handle subsequent requests to avoid delays in launching new instances. However, unused or idle function instances may also be terminated, along with any associated non-persistent local disk used for temporary data storage, resulting in cost savings for users, as they are only charged when function instances consume resources [3].

Serverless computing providers like AWS, Azure, and Google Cloud manage the execution of application environments, support backend computers or servers for functions, and dynamically allocate resources to ensure availability and scalability during failover and high demand scenarios. Unlike traditional Infrastructure-as-a-Service

(IaaS) platforms, where hardware needs to remain turned on for functions even when not in use, modern serverless computing platforms launch function instances or servers only when a specific function is invoked. Once a request is serviced, function instances are promptly put to sleep, minimizing costs by charging users only on a per-invocation basis, without requiring payment for idle or unused resources. This efficient resource utilization is a key aspect of modern serverless computing, originally designed for handling low duty-cycle workloads, such as processing requests in response to rapid changes in cloud storage files [20].

Related Work

Function-as-a-Service (FaaS) has gained exponential popularity for its ability to deploy computation to serverlessbackends in the public cloud, shifting the complexity of resource provisioning and allocation to cloud providers. Mohammed et al. characterized the FaaS workload of Azure Functions at the production level and proposed a resource management policy to reduce the number of cold start functions while minimizing resource usage. Their data collection encompassed various sets, including first-trigger per function, invocation counts per function, and execution time per function, allowing for a comprehensive analysis [15]. Wang et al. introduced the SIREN framework, which leverages stateless functions on the cloud to achieve higher parallelism and elasticity, using deep reinforcement learning to control the memory and number of stateless functions. They compared the cost and performance of training machine learning models on AWS EC2 clusters versus AWS Lambda using the SIREN framework [19].

In addition to private cloud providers, research has also been conducted on open-source serverless computing platforms like OpenWhisk. Yu et al. introduced Freyr, a Resource Manager for serverless platforms that dynamically harvests idle resources to increase resource efficiency. Utilizing a deep reinforcement learning algorithm, Freyr monitors resource utilization and safely harvests idle resources to optimize performance [21]. Furthermore, several serverless application developers have conducted experiments to measure CPU usage, function instance lifetime, cold start latency, and other metrics in AWS Lambda. Liang et al. conducted extensive experiments across leading cloud providers to evaluate resource management efficiency and performance, providing insights into resource utilization, cold start latency, and scalability [20].

However, Anthony et al. experimented on a MicroFaaS prototype against traditional serverless platforms to compare results between x86 and ARM-based single-board computers. Their prototype aimed to demonstrate that serverless functions are better suited for low-overhead and smaller execution environments than conventional infrastructures. They conducted a comprehensive evaluation and cost analysis, finding that energy efficiency increased by 5.6x, and total cost decreased by 34.2% [3]. Nonetheless, optimization of this model in the real market context remains a topic of discussion. Therefore, conducting experiments on AWS Lambda to compare results between x86 and ARM64 architectures could provide valuable insights that may attract more users or organizations to this platform.

Amazon Web Services (AWS) is a leading public cloud platform renowned for its cost-effectiveness, reliability, and scalability, making it a preferred choice for users and organizations over competitors like Azure and Google Cloud. Offering on-demand operations such as database storage, content delivery, and compute power, AWS assists businesses in expanding and scaling their operations. Its applications span various domains, including storage/backup, web applications, online gaming, and mobile, web, and social applications [16]. This chapter delves into AWS Lambda, exploring its architecture, benefits, and invocation mechanisms.

Advantages of AWS:

Easy to Use: AWS boasts a user-friendly Management Console accessible post sign-up, enabling the instant launch of numerous services without the need for on-site servers, thereby facilitating the prompt deployment of applications or entire IT ecosystems.
Security: Despite common misconceptions regarding data vulnerability in public clouds, AWS stands out for its comprehensive, reliable, and secure cloud platform, offering security tools at competitive rates.
Global Availability: With 84 availability zones spread across 26 geographic regions worldwide, AWS ensures global availability and reliability. Recent expansions include plans to add 24 more availability zones and eight additional AWS regions, with each region housing one or more data centers called availability zones.
Scalability and Flexibility: AWS offers unparalleled scalability and flexibility on demand, allowing organizations to

plan infrastructure roadmaps on a subscription basis without long-term commitments. Additionally, users only pay for the resources they utilize.

AWS Services:

AWS provides a wide array of services for cloud applications, including compute, storage, databases, monitoring tools, security tools, and developer tools. This chapter focuses on the services utilized during our research.

AWS CloudWatch: A monitoring tool that oversees resources and applications running on the AWS platform, CloudWatch aggregates operational data in the form of logs, offering system-wide visibility into application performance and resource utilization. It was instrumental in debugging issues during the invocation of Lambda functions and conducting experiments.
AWS S3: Amazon Simple Storage Service (S3) is a versatile cloud-based storage service known for its data availability, scalability, performance, and security. Used for tasks such as backing up online data, storage, invoking Lambda functions, and data retrieval, it played a crucial role in storing necessary objects during Lambda function invocation and experiment execution.

AWS Lambda:

AWS Lambda is an event-driven compute service tasked with executing user application code without the need to manage backend servers. Dubbed "serverless," Lambda eliminates the need for server maintenance, allowing users to focus solely on application logic. Lambda can handle various computing tasks, including data stream processing, web page serving, and integration with other AWS services.

How does AWS Lambda operate?
AWS Lambda functions require their containers to run. Each time a Lambda function is invoked, AWS Lambda packages the function into a new container and executes it on backend servers managed by AWS. This process can be visualized through the containers depicted in Figure 3.2. Before initiating the function's execution, the container is provisioned with the necessary CPU and RAM. As the function executes, the allocated RAM is multiplied by the time spent executing the function. Users or customers are then charged based on the function's runtime and allocated memory. Figure 3.1 provides a visual representation of this process.

AWS handles the entire infrastructure layer of AWS Lambda, providing users with limited visibility into the system's underlying operations. Users need not concern themselves with tasks such as updating underlying machines or managing network congestion, as AWS takes care of these aspects. Moreover, users are relieved of operational tasks as AWS Lambda provides full management of the service.

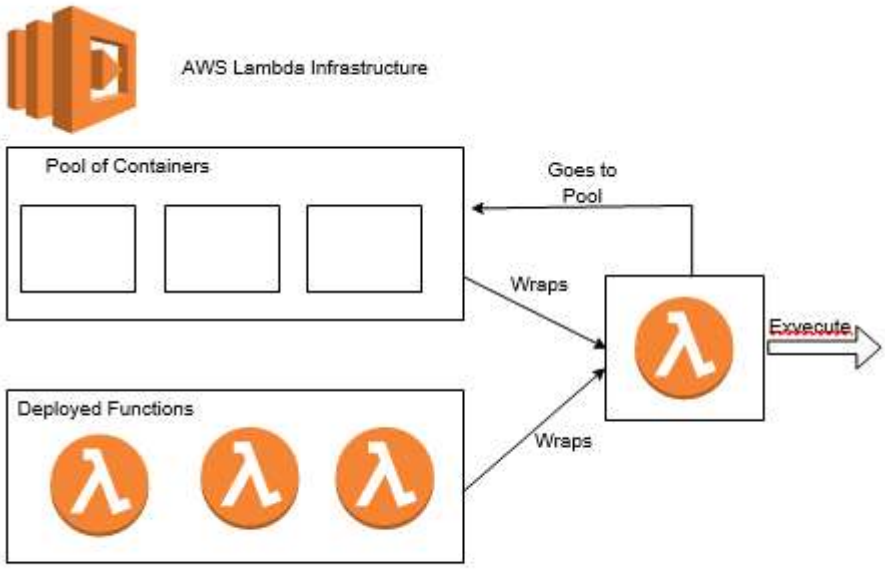


Figure 3.1. AWS Lambda Infrastructure

Why is Lambda an essential component of serverless architecture?

Lambda serves as a cornerstone in building serverless applications, forming a crucial part of the serverless stack alongside compute, database, and API gateway services. It seamlessly integrates with various AWS services like DynamoDB and RDS, playing a pivotal role in serverless solutions. Lambda offers several advantages for developers, supporting multiple languages and runtimes, including the recent introduction of ARM64 architecture, which promises faster execution and better price performance. The table below showcases the supported runtimes and architectures for Python, a language widely used in serverless development.

Advantages of AWS Lambda

Lambda offers several advantages over traditional server maintenance:

Pay-Per-Use: Users only pay for the actual runtime of their functions, making it cost-effective for workloads that experience significant scaling during peak hours.

Fully Managed Infrastructure: AWS handles all underlying infrastructure tasks, such as network management and OS upgrades, reducing operational costs and complexity.

Automatic Scaling: Lambda automatically scales the underlying infrastructure based on workload demand, ensuring optimal performance without user intervention.

Integration with Other AWS Services: Lambda seamlessly integrates with various AWS services like API gateway and DynamoDB, enabling developers to build comprehensive and functional applications.

Limitations of AWS Lambda

Despite its advantages, Lambda has some limitations to consider:

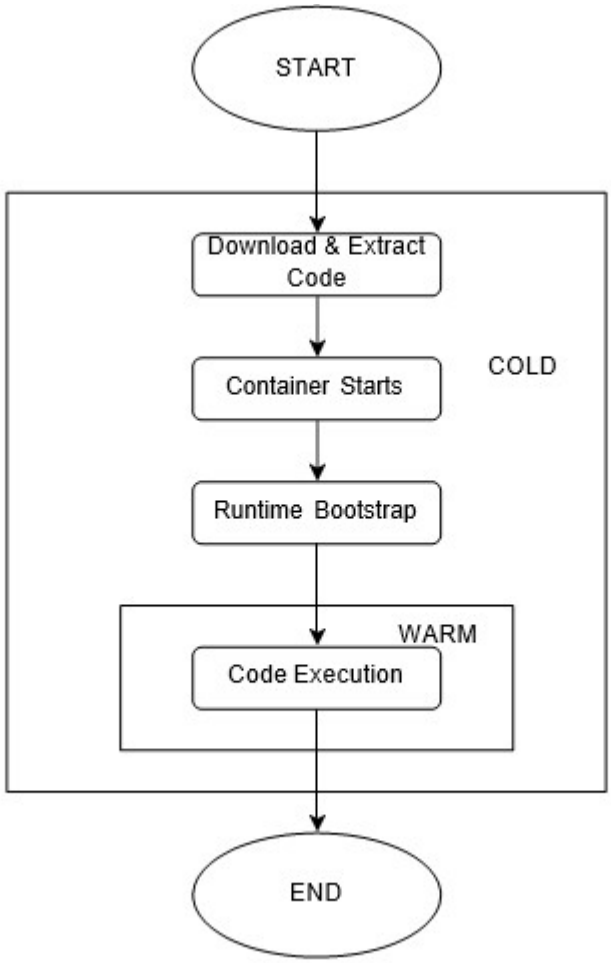
Cold Start Time: There is a latency, known as cold start time, between the event triggering a function and its

execution, which can be problematic for latency-critical applications.

Function Limits: Lambda functions have constraints such as execution timeout, memory limits, package size restrictions, and concurrent execution limits, which may impact certain use cases.

Cold and Warm Start Call of Lambda Functions

Lambda's deployment benefits are sometimes offset by inconsistent startup performance, particularly due to cold start times. When a Lambda function is triggered, AWS must spin up the necessary server resources, leading to a delay before the function code can execute. This delay, known as a cold start, can frustrate users seeking consistent performance from their serverless applications.



Asynchronous Invocation

In asynchronous invocation, Lambda functions are triggered by events and do not wait for a response. Upon receiving the event, Lambda places it into an internal queue and promptly returns a successful response without further action. Subsequently, a separate process retrieves the event from the queue and executes the Lambda function. A typical example of asynchronous invocation involves the interaction between S3, SNS, Lambda, and DynamoDB, as depicted in Figure 3.5.

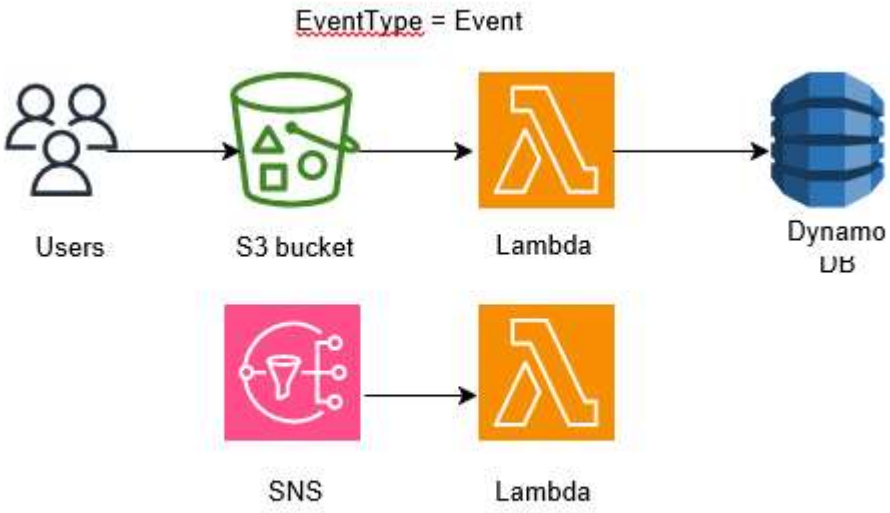


Figure 3.5. Asynchronous Invocation

Methodology

The Serverless Benchmark Suite (SeBS) offers a comprehensive and consistent approach to comparing the reliability and performance of various serverless providers across different workloads. It serves as a complete framework for developing, deploying, and invoking Lambda functions on different cloud platforms, including open-source alternatives like OpenWhisk [5]. In this study, we leverage SeBS to evaluate the performance of two AWS Lambda architectures: x86 and ARM64. This chapter elucidates the process of setting up the research environment, conducting regression tests, invoking benchmarks, executing diverse experiments, and ultimately analyzing the results.

4.1. Environment Setup

All experiments were conducted in the AWS us-east-1 region. We utilized S3 for persistent storage, HTTP endpoints as function triggers, and deployed Python 3.7 for the x86 architecture and Python 3.9 for the ARM64 architecture, respectively. Below is a simplified model or topology employed in this research.

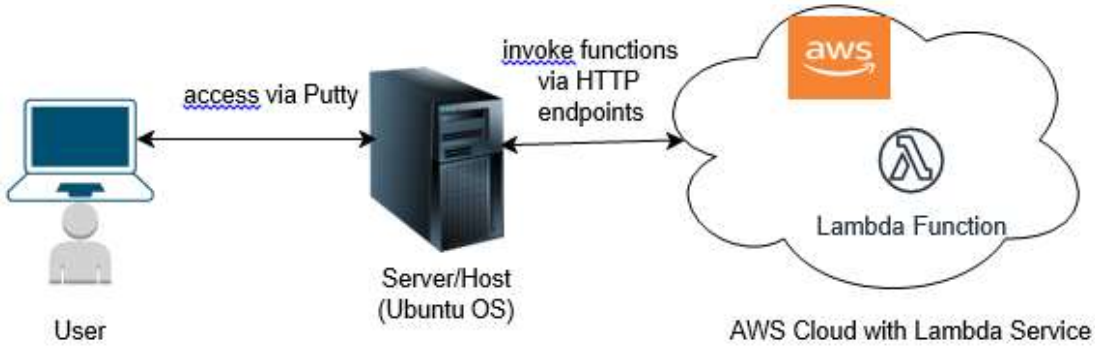


Figure 4.1. A simple Experimental Topology

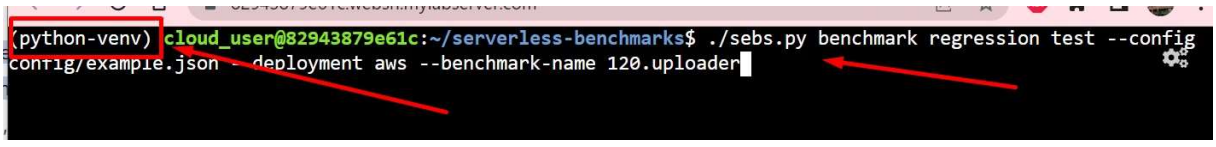
Regression Testing

SeBS facilitates three fundamental commands: benchmark, experiment, and local. Each command can be augmented with the --verbose flag to increase output verbosity. It's imperative to run all commands within the Python virtual environment. Regression testing entails executing all benchmarks on AWS Lambda. Below is an illustration of running a regression test with an input size test on AWS Lambda:

```
./sebs.py benchmark regression test --configconfig/example.json --deployment aws --verbose
```

Similarly, a regression test can be executed on a single benchmark. Here's an example:

```
./sebs.py benchmark regression test --configconfig/example.json --deployment aws --benchmark-name 120.uploader –verbose
```



By default, all Lambda functions are created and invoked via HTTP on x86 architecture. Before running Lambda functions on ARM64 architecture, two factors must be considered.

Result and Analysis

This chapter delves into the detailed results obtained from executing experiments across multiple workloads. To

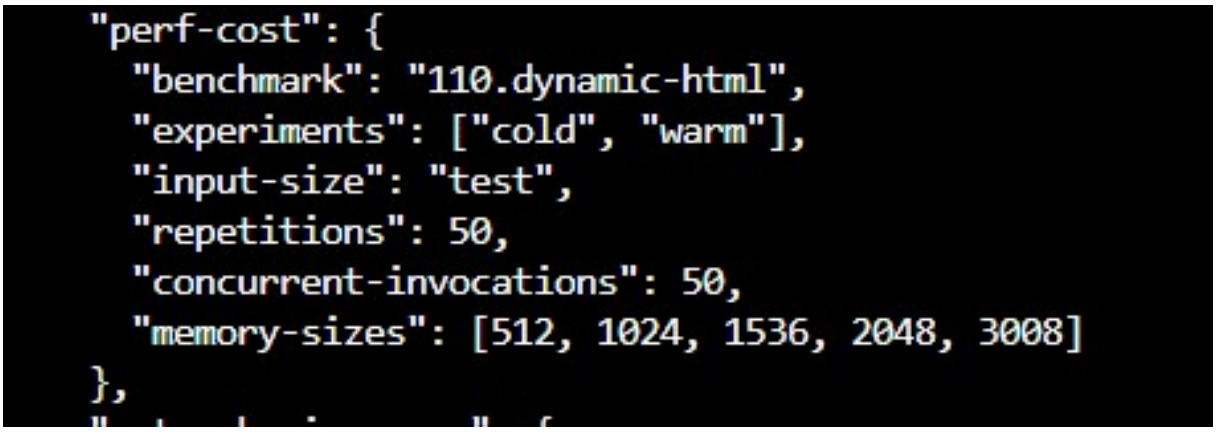
ensure clarity and accuracy in the obtained results, we established separate lab setups for each architecture, utilizing two AWS accounts. Our analysis primarily focuses on three key aspects: perf-cost, latency, and overhead.

5.1. Perf-Cost

Given the limited visibility into the cloud metrics due to the black-box nature of the FaaS system, our perf-cost analysis centers on metrics such as execution time, client time, provider time, and memory usage. Before initiating the experiments, it's crucial to connect to the server/host where the SeBS framework is installed via Putty. Subsequently, we activate our Python virtual environment using the following command:

At the outset, we initiated the experiment on the x86 architecture for the Dynamic-HTML workload using the following command:

However, to acquire the desired perf-cost metrics effectively, we utilized a configuration file named example.json to pass the inputs. This configuration file streamlined the process, enabling us to obtain the results of the inputs efficiently. The configuration file, depicted in Figure 5.2, outlines various parameters including the experimental nature (cold and warm), workload or benchmark name, memory sizes (ranging from 128 to 3008MB), as well as the number of repetitions and concurrent invocations applicable during the experiment.



The acquired results, initially in JSON format, were further processed to extract the desired metrics and their corresponding values, which were then formatted into CSV format. This transformation was achieved through the

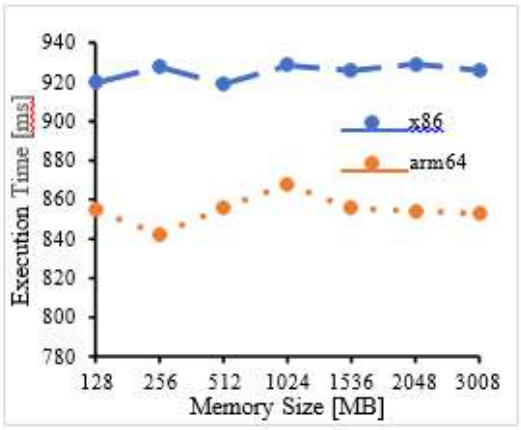
following command:

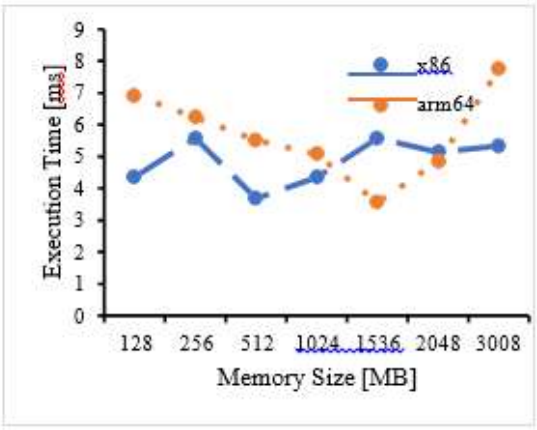
Similarly, to obtain results for the same workload, Dynamic-HTML, on ARM64 architecture, we first modified the Python runtime to 3.9 in the configuration file, example.json. Subsequently, we utilized the AWS Management Console to switch the architecture from x86 (default) to ARM64. The aforementioned process was then repeated to retrieve the desired cloud metrics values for the Dynamic-HTML workload on the ARM64 architecture.

Moreover, we replicated the aforementioned steps on both x86 and ARM64 architectures for other workloads such as uploader and compression, by adjusting the benchmark name within the configuration file - example.json. To conduct a comprehensive analysis of perf-cost, we subdivided the topic into several sub-topics, as elaborated below.

At different memory sizes, the behavior of x86 and ARM64 architectures during cold and warm start calls for the Dynamic-HTML workload was examined. Dynamic-HTML, a straightforward web application, delegates dynamic features to a serverless backend, generating Dynamic HTML from a predefined template. The execution time encompasses the duration the backend server expends executing in the AWS cloud, encompassing the work performed by the function. To specify cold start, the "experiments" parameter in the example.json file required the "cold" option, while the "warm" option was utilized for warm start, as depicted in Figure 5.2.

Results were collected for 50 to 100 concurrent invocations of the Lambda function, with a fixed memory allocation ranging from 128MB to 3008MB. To analyze the behavior of x86 and ARM64 architectures concerning the Dynamic-HTML workload, a graph (Figure 5.3) correlating execution time and memory allocation was plotted for both cold and warm starts.





(a) Comparison of x86 vs. ARM64 at Cold Start for Dynamic HTML

(b) Comparison of x86 vs. ARM64 at Warm Start for Dynamic HTML

Figure 5.4 illustrates the impact on client time at various memory sizes for x86 and ARM64 architectures for Dynamic HTML. The results indicate that the ARM processor outperforms the x86 architecture in terms of scheduling and deploying a Lambda function for both cold and warm starts across all memory allocations. Therefore, in terms of client time, the ARM64 architecture appears to be the preferable choice.

Additionally, we examined the time required for cloud providers to incorporate language overhead and serverless sandbox. This duration, known as provider time, was averaged after conducting 50 to 100 concurrent invocations for both cold and warm starts at each memory allocation.

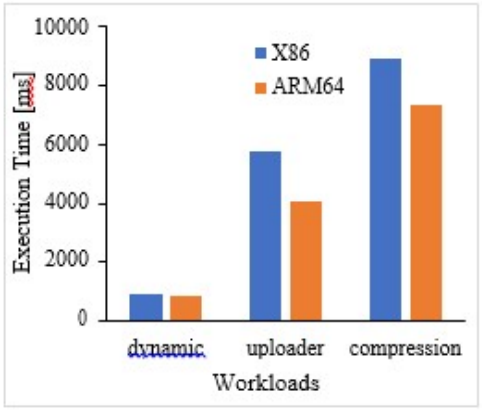
Behavior of x86 and ARM64 Architecture at Cold and Warm Start for Different Workloads

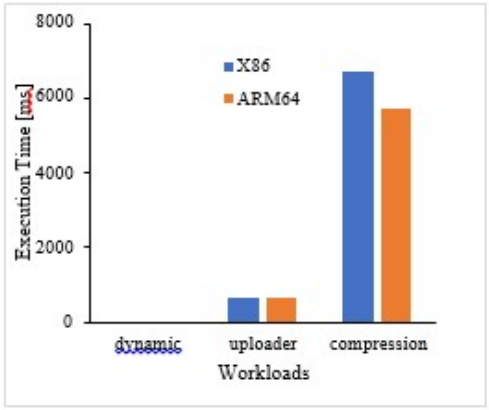
In addition to the single and straightforward workload, Dynamic-HTML, we explored the perf-cost metrics across multiple complex workloads: uploader and compression. The uploader, or storage uploader, is a web application designed to upload files from a URL to cloud storage, exhibiting higher requirements and sizes compared to Dynamic-HTML. Similarly, compression serves as a utility function compressing sets of files and returning archives to users, akin to online document text editors and office suites. These functions act as backend processing tools for web servers or application frontends facing more complex issues, with larger sizes and complexities compared to the former.

These workloads represent varying code sizes and complexities, wherein we understand that dependencies' size and complexity directly impact cold and warm start executions. Larger and more complex code packages lead to increased warm-up times for language runtimes and deployment times from cloud storage.

During the experiment, we invoked 50 to 100 concurrent functions for each workload at a memory allocation of 128MB. Figure 5.6 illustrates the impact on execution time due to code complexity.

As shown in Figure 5.6(a) for cold start, increasing code size and complexity directly correlate with increased execution time. Moreover, x86 architecture exhibits longer execution times compared to ARM64 across all workloads.





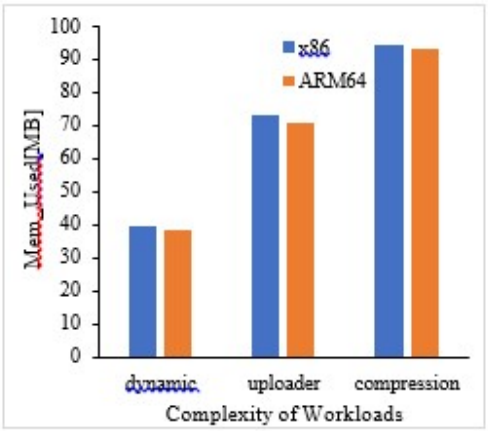
Impact of Code Complexity on Memory Usage for Different Architectures

Memory usage stands as a critical parameter impacting both performance and cost. Peak memory usage plays a vital role in configuring applications, defining billing policies, and setting execution parameters. It provides developers and cloud providers with insights to manage active or suspended containers effectively.

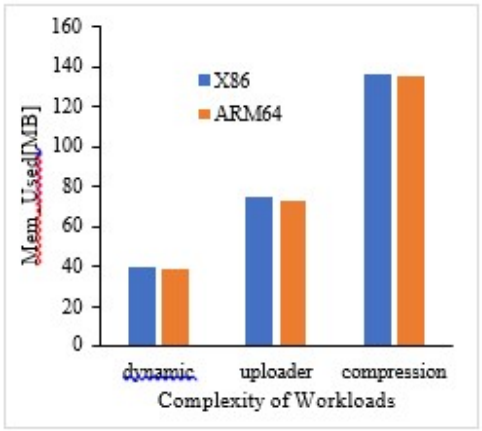
Furthermore, memory allocation dictates the amount of virtual CPU available to a Lambda function. As memory

increases, so does the virtual CPU, augmenting the overall computational power. Hence, adjustments in memory settings directly influence Lambda function performance.

Initially, we conducted experiments on x86 architecture, invoking each Lambda function 50 to 100 times, repeating 50 iterations in both cold and warm environments, with a static memory allocation of 2048 MB. Subsequently, we replicated the experiment on ARM64 architecture and plotted memory usage against workload complexity, as illustrated in Figure 5.12.



(a) x86-vs-ARM64 at cold start



(b) x86-vs-ARM64 at warm start

Discussion and Future Work

This section outlines the practical contributions of our study and outlines potential avenues for future research.

In summary, we investigated the following:

- The behavior of AWS architectures during cold and warm start calls of Lambda functions across various workloads.
- Cold and warm startup behavior on each AWS architecture.
- The impact of code complexity and size on memory usage on each AWS architecture.
- The effect of input size on performance-cost metrics and memory usage.
- Code complexity's influence on latency on each AWS architecture.
- The impact of cold startup overhead on performance.

To delve into performance-cost analysis, we segmented it into multiple sections. From our initial analysis, we observed that during cold starts, execution time, client time, and provider time were consistently higher on x86 architecture across different memory allocations ranging from 128MB to 3008MB. However, warm start calls showed varied results for certain perf-cost metrics. For instance, the execution time for Dynamic-HTML and uploader workloads was higher on ARM64 than x86 architecture.

Furthermore, our examination of different workloads revealed their varying effects on perf-cost metrics across AWS architectures. Increasing code size and complexity consistently led to higher execution time, client time, and provider time on both x86 and ARM64 architectures during cold and warm start invocations at a static memory

allocation of 128MB. ARM64 outperformed x86 in both invocation methods, holding true for other memory allocations as well.

Expanding our perf-cost analysis, we explored how workload complexity affects memory usage. At a static memory allocation of 2048 MB, we observed that increasing code size and complexity led to higher memory usage across both architectures during cold and warm starts. While ARM64 showed slightly better performance, the difference in memory consumption between architectures for specific workloads was minimal.

Additionally, we analyzed the impact of input size on perf-cost metrics for the Dynamic-HTML workload during cold startup at a static memory allocation of 512MB. Results indicated that increasing input size resulted in incremental performance metrics across both architectures. Notably, ARM64 maintained comparable performance to x86 even with larger input sizes.

Beyond perf-cost analysis, we investigated latency across different workloads. For both cold and warm start invocations, x86 architecture exhibited higher connection times compared to ARM64 at any static memory allocation. Moreover, increasing code complexity correlated with increased connection times across both architectures and invocation methods, with warm invocations consistently faster than cold ones.

Lastly, we explored the effect of cold startup overhead on performance. While client time overhead showed negligible differences between x86 and ARM64 for the uploader workload, significant disparities were observed in execution and provider time overheads. Additionally, overhead decreased with increased code complexity, primarily due to higher memory consumption in larger, more complex workloads.

This study underscores the significance of perf-cost metrics, latency, and overhead in optimizing application performance and cost efficiency in the AWS Cloud. Leveraging the SeBS framework and the introduction of ARM64 architecture by AWS, we have laid the groundwork for further research to uncover additional factors impacting application performance. Our methodology provides a systematic approach to evaluating serverless computing, allowing for diverse experiments across multiple workloads.

In conclusion, evaluating serverless computing empowers users and organizations to select the most efficient configuration for their workloads in the AWS cloud. While efforts to mitigate issues related to cold and warm startups continue, new challenges are likely to emerge. A comprehensive evaluation using sound methodology can ensure optimal performance and cost efficiency across different cloud architectures.

Building upon this foundation, future research can explore heavier workloads with greater complexity and size, as well as investigate the impact of invocation overhead on application performance. Moreover, examining the potential impact of GPU architecture on various workloads, particularly in the absence of support from AWS, presents an intriguing avenue for further exploration.

References List

[1]. Shuford, J. (2023). Contribution of Artificial Intelligence in Improving Accessibility for Individuals with Disabilities. Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online), 2(2), 421-433.
DOI: https://doi.org/10.60087/jklst.vol2.n2.p433

[2]. Chentha, A. K., Sreeja, T. M., Hanno, R., Purushotham, S. M. A., & Gandrapu, B. B. (2013). A Review of the Association between Obesity and Depression. Int J Biol Med Res, 4(3), 3520-3522.

[3]. Gadde, S. S., & Kalli, V. D. R. (2020). Descriptive analysis of machine learning and its application in healthcare. *Int J Comp Sci Trends Technol*, *8*(2), 189-196.

[4]. Atacho, C. N. P. (2023). A Community-Based Approach to Flood Vulnerability Assessment: The Case of El Cardón Sector. *Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online)*, *2*(2), 434-482. DOI:https://doi.org/10.60087/jklst.vol2.n2.p482

[5]. jimmy, fnu. (2023). Understanding Ransomware Attacks: Trends and Prevention Strategies. Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online), 2(1), 180-210. https://doi.org/10.60087/jklst.vol2.n1.p214

[6]. Bayani, S. V., Prakash, S., & Malaiyappan, J. N. A. (2023). Unifying Assurance A Framework for Ensuring Cloud Compliance in AIML Deployment. *Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online)*, *2*(3), 457-472. DOI: https://doi.org/10.60087/jklst.vol2.n3.p472

[7]. Bayani, S. V., Prakash, S., & Shanmugam, L. (2023). Data Guardianship: Safeguarding Compliance in AI/ML Cloud Ecosystems. *Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online)*, *2*(3), 436-456. DOI: https://doi.org/10.60087/jklst.vol2.n3.p456

[8]. Karamthulla, M. J., Malaiyappan, J. N. A., & Prakash, S. (2023). AI-powered Self-healing Systems for Fault Tolerant Platform Engineering: Case Studies and Challenges. *Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online)*, *2*(2), 327-338. DOI: https://doi.org/10.60087/jklst.vol2.n2.p338

[9]. Prakash, S., Venkatasubbu, S., & Konidena, B. K. (2023). Unlocking Insights: AI/ML Applications in Regulatory Reporting for US Banks. *Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online)*, *1*(1), 177-184. DOI: https://doi.org/10.60087/jklst.vol1.n1.p184

[10]. Prakash, S., Venkatasubbu, S., & Konidena, B. K. (2023). From Burden to Advantage: Leveraging AI/ML for Regulatory Reporting in US Banking. *Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online)*, *1*(1), 167-176. DOI: https://doi.org/10.60087/jklst.vol1.n1.p176

[11]. Prakash, S., Venkatasubbu, S., & Konidena, B. K. (2022). Streamlining Regulatory Reporting in US Banking: A Deep Dive into AI/ML Solutions. *Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online)*, *1*(1), 148-166. DOI: https://doi.org/10.60087/jklst.vol1.n1.p166

[12]. Tomar, M., & Jeyaraman, J. (2023). Reference Data Management: A Cornerstone of Financial Data Integrity. Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online), 2(1), 137-144. DOI: https://doi.org/10.60087/jklst.vol2.n1.p144

[13]. Tomar, M., & Periyasamy, V. (2023). The Role of Reference Data in Financial Data Analysis: Challenges and Opportunities. Journal of Knowledge Learning and Science

Technology ISSN: 2959-6386 (online), 1(1), 90-99.
DOI: https://doi.org/10.60087/jklst.vol1.n1.p99

[14]. Tomar, M., & Periyasamy, V. (2023). Leveraging Advanced Analytics for Reference Data Analysis in Finance. *Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online)*, *2*(1), 128-136.
DOI: https://doi.org/10.60087/jklst.vol2.n1.p136

[15]. Sharma, K. K., Tomar, M., & Tadimarri, A. (2023). Unlocking Sales Potential: How AI Revolutionizes Marketing Strategies. *Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online)*, *2*(2), 231-250.
DOI: https://doi.org/10.60087/jklst.vol2.n2.p250

[16]. Sharma, K. K., Tomar, M., & Tadimarri, A. (2023). Optimizing Sales Funnel Efficiency: Deep Learning Techniques for Lead Scoring. *Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online)*, *2*(2), 261-274.
DOI: https://doi.org/10.60087/jklst.vol2.n2.p274

[17]. Shanmugam, L., Tillu, R., & Tomar, M. (2023). Federated Learning Architecture: Design, Implementation, and Challenges in Distributed AI Systems. *Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online)*, *2*(2), 371-384.
DOI: https://doi.org/10.60087/jklst.vol2.n2.p384

[18]. Sharma, K. K., Tomar, M., & Tadimarri, A. (2023). AI-driven Marketing: Transforming Sales Processes for Success in the Digital Age. *Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online)*, *2*(2), 250-260.
DOI: https://doi.org/10.60087/jklst.vol2.n2.p260

[19]. Gadde, S. S., & Kalli, V. D. (2021). The Resemblance of Library and Information Science with Medical Science. International Journal for Research in Applied Science & Engineering Technology, 11(9), 323-327.

[20]. Gadde, S. S., & Kalli, V. D. R. (2020). Technology Engineering for Medical Devices-A Lean Manufacturing Plant Viewpoint. Technology, 9(4).

[21]. Gadde, S. S., & Kalli, V. D. R. (2020). Medical Device Qualification Use. International Journal of Advanced Research in Computer and Communication Engineering, 9(4), 50-55.

[22]. Gadde, S. S., & Kalli, V. D. R. (2020). Artificial Intelligence To Detect Heart Rate Variability. *International Journal of Engineering Trends and Applications*, *7*(3), 6-10.

[23]. Chentha, A. K., Sreeja, T. M., Hanno, R., Purushotham, S. M. A., & Gandrapu, B. B. (2013). A Review of the Association between Obesity and Depression. *Int J Biol Med Res*, *4*(3), 3520-3522.

[24]. Tao, Y. (2022). Algorithm-architecture co-design for domain-specific accelerators in

communication and artificial intelligence (Doctoral dissertation).
https://deepblue.lib.umich.edu/handle/2027.42/172593

[25]. Tao, Y., Cho, S. G., & Zhang, Z. (2020). A configurable successive-cancellation list polar decoder using split-tree architecture. IEEE Journal of Solid-State Circuits, 56(2), 612-623.
DOI: https://doi.org/10.1109/JSSC.2020.3005763

[26]. Tao, Y., & Choi, C. (2022, May). High-Throughput Split-Tree Architecture for Nonbinary SCL Polar Decoder. In 2022 IEEE International Symposium on Circuits and Systems (ISCAS) (pp. 2057-2061). IEEE.
DOI: https://doi.org/10.1109/ISCAS48785.2022.9937445

[27]. Tao, Y. (2022). Algorithm-architecture co-design for domain-specific accelerators in communication and artificial intelligence (Doctoral dissertation).
https://deepblue.lib.umich.edu/handle/2027.42/172593

[28]. Mahalingam, H., Velupillai Meikandan, P., Thenmozhi, K., Moria, K. M., Lakshmi, C., Chidambaram, N., & Amirtharajan, R. (2023). Neural attractor-based adaptive key generator with DNA-coded security and privacy framework for multimedia data in cloud environments. Mathematics, 11(8), 1769.
https://doi.org/10.3390/math11081769

[29]. Padmapriya, V. M., Thenmozhi, K., Praveenkumar, P., & Amirtharajan, R. (2020). ECC joins first time with SC-FDMA for Mission “security”. Multimedia Tools and Applications, 79(25), 17945-17967.
DOI https://doi.org/10.1007/s11042-020-08610-5

[30]. Padmapriya, V. M. (2018). Image transmission in 4g lte using dwt based sc-fdma system. Biomedical & Pharmacology Journal, 11(3), 1633.
DOI : https://dx.doi.org/10.13005/bpj/1531

[31]. Padmapriya, V. M., Priyanka, M., Shruthy, K. S., Shanmukh, S., Thenmozhi, K., & Amirtharajan, R. (2019, March). Chaos aided audio secure communication over SC-FDMA system. In 2019 International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN) (pp. 1-5). IEEE.
https://doi.org/10.1109/ViTECoN.2019.8899413

[31]. Padmapriya, V. M., Thenmozhi, K., Praveenkumar, P., & Amirtharajan, R. (2022). Misconstrued voice on SC-FDMA for secured comprehension-a cooperative influence of DWT and ECC. Multimedia Tools and Applications, 81(5), 7201-7217.
DOI https://doi.org/10.1007/s11042-022-11996-z

[32]. Padmapriya, V. M., Sowmya, B., Sumanjali, M., & Jayapalan, A. (2019, March). Chaotic Encryption based secure Transmission. In 2019 International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN) (pp. 1-5). IEEE.

DOI https://doi.org/10.1109/ViTECoN.2019.8899588

[33]. Sowmya, B., Padmapriya, V. M., Sivaraman, R., Rengarajan, A., Rajagopalan, S., & Upadhyay, H. N. (2021). Design and Implementation of Chao-Cryptic Architecture on FPGA for Secure Audio Communication. In Emerging Technologies in Data Mining and Information Security: Proceedings of IEMIS 2020, Volume 3 (pp. 135-144). Springer Singapore https://link.springer.com/chapter/10.1007/978-981-15-9774-9_13

[34]. Padmapriya, V. M., Thenmozhi, K., Avila, J., Amirtharajan, R., & Praveenkumar, P. (2020). Real Time Authenticated Spectrum Access and Encrypted Image Transmission via Cloud Enabled Fusion centre. Wireless Personal Communications, 115, 2127-2148. DOI https://doi.org/10.1007/s11277-020-07674-8

[35]. Kommaraju, V., Gunasekaran, K., Li, K., Bansal, T., McCallum, A., Williams, I., & Istrate, A. M. (2020). Unsupervised pre-training for biomedical question answering. arXiv preprint arXiv:2009.12952.

[36]. Bansal, T., Gunasekaran, K., Wang, T., Munkhdalai, T., & McCallum, A. (2021). Diverse distributions of self-supervised tasks for meta-learning in NLP. arXiv preprint arXiv:2111.01322.

[37]. Gunasekaran, K., Tiwari, K., & Acharya, R. (2023, June). Utilizing deep learning for automated tuning of database management systems. In 2023 International Conference on Communications, Computing and Artificial Intelligence (CCCAI) (pp. 75-81). IEEE.

[38]. Gunasekaran, K. P. (2023, May). Ultra sharp: Study of single image super resolution using residual dense network. In 2023 IEEE 3rd International Conference on Computer Communication and Artificial Intelligence (CCAI) (pp. 261-266). IEEE.

[39]. Gillespie, A., Yirsaw, A., Gunasekaran, K. P., Smith, T. P., Bickhart, D. M., Turley, M., ... & Baldwin, C. L. (2021). Characterization of the domestic goat γδ T cell receptor gene loci and gene usage. Immunogenetics, 73, 187-201.

[40]. Yirsaw, A. W., Gillespie, A., Zhang, F., Smith, T. P., Bickhart, D. M., Gunasekaran, K. P., ... & Baldwin, C. L. (2022). Defining the caprine γδ T cell WC1 multigenic array and evaluation of its expressed sequences and gene structure conservation among goat breeds and relative to cattle. Immunogenetics, 74(3), 347-365.

[41]. Gunasekaran, K. P., Babrich, B. C., Shirodkar, S., & Hwang, H. (2023, August). Text2Time: Transformer-based Article Time Period Prediction. In 2023 IEEE 6th International Conference on Pattern Recognition and Artificial Intelligence (PRAI) (pp. 449-455). IEEE.

[42]. Gunasekaran, K., & Jaiman, N. (2023, August). Now you see me: Robust approach to partial occlusions. In 2023 IEEE 4th International Conference on Pattern Recognition and Machine Learning (PRML) (pp. 168-175). IEEE.

[43]. Gillespie, A., Yirsaw, A., Kim, S., Wilson, K., McLaughlin, J., Madigan, M., ... & Baldwin,

C. L. (2021). Gene characterization and expression of the γδ T cell co-receptor WC1 in sheep. Developmental & Comparative Immunology, 116, 103911.

[44]. Gunasekaran, K. P. (2023). Leveraging object detection for the identification of lung cancer. *arXiv preprint arXiv:2305.15813*.

[45]. Gunasekaran, K. P. (2023). Exploring sentiment analysis techniques in natural language processing: A Comprehensive Review. arXiv preprint arXiv:2305.14842.

[46]. Lee, S., Weerakoon, M., Choi, J., Zhang, M., Wang, D., & Jeon, M. (2022, July). CarM: Hierarchical episodic memory for continual learning. In Proceedings of the 59th ACM/IEEE Design Automation Conference (pp. 1147-1152).

[47]. Lee, S., Weerakoon, M., Choi, J., Zhang, M., Wang, D., & Jeon, M. (2021). Carousel Memory: Rethinking the Design of Episodic Memory for Continual Learning. arXiv preprint arXiv:2110.07276.