National College of Ireland

A novel mechanism for the reduction of latency and cost in AWS Lambda calls

MSc Research Project
Cloud Computing

Anilgovind Kokkoori Anilkumar
Student ID: x231764558

School of Computing
National College of Ireland

Supervisor: Dr Giovani Estrada

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Anilgovind Kokkoori Anilkumar
Student ID:	x231764558
Programme:	Cloud Computing
Year:	2024
Module:	MSc Research Project
Supervisor:	Dr Giovani Estrada
Submission Due Date:	12/12/2024
Project Title:	A novel mechanism for the reduction of latency and cost in AWS Lambda calls
Word Count:	9259
Page Count:	27

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	28th January 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

A novel mechanism for the reduction of latency and cost in AWS Lambda calls

Anilgovind Kokkoori Anilkumar
x231764558

Abstract

Serverless computing, especially AWS lambda revolutionised software industry. But cold start is a significant challenge, especially for latency-critical and sporadic workloads. This research addresses this issue by proposing and evaluating novel architectures. A light weight load balancer for reducing cold start and architectures for reducing cold start were implimented in this research. We analyzed multiple strategies for minimizing cold starts, including invocation-based methods using AWS EventBridge, SNS, and CloudWatch. Among these, a CloudWatch-driven architecture demonstrated superior efficiency by selectively invoking inactive Lambdas, avoiding unnecessary overhead. Additionally, the custom load balancer consistently outperformed the AWS API Gateway in test scenarios. The solution completely eliminated cold starts. It reduced latency by up to 80% and costs by 20% compared to using API Gateway for sporadic test events. The findings have broader implications for designing responsive, cost-effective serverless applications. And the solution can adapt for cross-platform deployments to unlock its full commercial potential like vendor locking.

1 Introduction

A cloud in software industry is a collection of configurable servers, storage and services accessible via internet on demand. There is various benefits for cloud computing over the conventional software development and deployments. In a little glance these benefits spreads over cost, performance, development, deployment and scalability [1]. Developers and architects were able to create new software models and development lifecycles with the technologies of cloud computing. Event driven and microservice architectures have a ambient implementation platform in cloud [2]. Microservices and Event driven architectures are implimented in cloud via different ways using various cloud resources and its popularity is increasing [3]. A main service used for this architectures is AWS Lambda, a serverless service.

Serverless computing is introduced by AWS in 2014 [4] as its first usable implimentation in cloud. This new computing idea is revolutionasied the software architecture and designs. Event driven invocation, No infrastructure management, and pay for usage are

[1] https://www.oracle.com/ie/cloud/what-is-cloud-computing/top-10-benefits-cloud-computing
[2] https://developer.ibm.com/articles/eda-and-microservices-architecture-best-practices
[3] https://codal.com/insights/understanding-the-rise-of-microservice-architecture
[4] https://georgemao.medium.com/the-ultimate-guide-to-aws-lambda-development-6e4aae00d964

the key benefits in serverless computing [5]. Serverless computing is a general term and it can be computing, messaging, storage or gateway where the end user does not have control over management, billing , lifecycle and configuration of the service [6]. Function as a service provided by cloud service providers are for executing code in response to events. The computation space is limited in memory, cpu and maximum duration of the execution environment. But the users does not need to worry about the high level infrastructures and creation of the servers for execution. According to a survey result published by Datadog a popular log monitoring service. Over 70 percent of AWS users and 60 percent of google cloud providers are using serverless resources [7]. In FaaS environment the hardware, environment and server management are responsibility of the cloud service provider. The developers are entitled to focus on their development of their code. Availability, scalability and pay for use are the key benefits of FaaS. For a FaaS environment there wont be a dedicated server running 24 hours. The function will execute only through a external trigger called events.

AWS lambda is widely using for data processing, API backend, IoT device management, natural language processing, image and video processing etc. Where as because of there is no dedicated server running in the backend a cold start will occur in AWS lambda if there is no active environment for execution. This is generally due to AWS lambda need time to create the virtual environment for the execution. The AWS lambda execution life cycle is going through four steps. Which are downloading the code from the internal s3 bucket or ECR, creating the configured execution runtime environment, preparing the code blocks outside the handler which may involves initialising layers, packages, establishing connection to database, initialising functions and variables outside the handler. And at last stage executing the handler code. The first two stages of the execution lifecycle is not billed but the last two stages are billed according to the execution time and invocations count Figure 1.

After completing execution the environment will be frozen and will be available for a

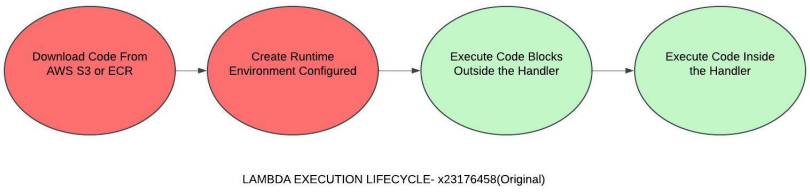


Figure 1: Lambda Lifecyle

non-deterministic time period. This environment will be reused if any event comes for execution before it is terminated else this environment will be automatically destroyed by AWS. Cold start usually varies function to function according to its size, execution environment and packages [8]. But for latency critical applications and services this is a major issue, especially for sporadic services where there is no continuous use of function code.

[5] https://www.ibm.com/topics/serverless
[6] https://www.ibm.com/topics/faas
[7] https://www.datadoghq.com/state-of-serverless/
[8] https://docs.aws.amazon.com/lambda/latest/dg/execution-environments.html

UN Sustainable Goals

Research like the one here introduced will help cloud architects to select the best strategy in such a way that minimize energy, cost, and computations. Even cloud service providers could potentially include lightweight AWS lambda services tuned to the application requirements. These are also goals of UN Sustainable Development Goals(SDGs). Target 7.3 for improvement in energy efficiency [9] and target 9.4 Upgrade infrastructure for sustainability with increased resource-use efficiency [10] are perfectly aligning with the outcomes of this research.

This research is focusing on investigating methods to keep the AWS lambda execution environment warm and reuse the existing virtual environment to minimise latency and cost. The research is successful in evaluating and designing different architectural patterns for keeping the AWS lambda warm and reusing the execution environment to reduce latency and cost due to cold start.

1.1 Research gap

While AWS lambda is the most important feature in FaaS, little is known about best strategies to keep it live. Strategies could vary with the task that has to be accomplished via AWS lambda. Especially to optimise AWS lambda, developers have to deal with cold start, cost and scalability [11]. Knowing that the active virtual environment will reduce the cold start to a great extend the current strategies to use it are complex and cost intensive. Exploiting the active virtual environment using a lightweight solution is need of the software industry.

1.2 Research question

To clarify the above mentioned research gap, a detailed study is proposed here. The key research question can be described as follows:

- What are the best ways to keep the AWS lambda environment active? Or, in other words, what could be the best cloud architecture to minimise the cold start and cost of AWS lambda?

1.3 Research objectives

In order to accomplish the research question, a number of steps have to be performed:

1. Analyse the cold start and serverless virtual environment. Then develop architectures to keep the Virtual environment active.

2. Develop a lightweight load balancer to re-use the active virtual environment to reduce cold starts.

3. Evaluated the use cases to identify the minimum viable time span to reduce cold starts. And compare the quality of lightweight load balancer with API gateway in terms of its latency and cost.

[9] `https://sdgs.un.org/goals/goal7#targets_and_indicators`
[10] `https://sdgs.un.org/goals/goal9#targets_and_indicators`
[11] `https://georgemao.medium.com/the-ultimate-guide-to-aws-lambda-development-6e4aae00d964`

1.4 Outline

This report is organised as follows. Section 2 presents a critical analysis of closely related work. Section 3 illustrates the evaluation approach to execute the research work. In Section 4, The design of the proposed work and novel idea is described The Section 5 have all the details of enactment of research question. Whereas Section 6 have the details of test results and its discussion. The Section 7 have the conclusion and future directions of the research.

2 Related Work

Advantages of using serverless cloud resources gained drastic attraction of organisations and software architects. Various researches to improve the cost, performance, security and latency of serverless FaaS environments were executed by researchers. Due to the unpredictability of the life cycle and varying cold start for function to function are major problem experienced by all of the researchers. In the section below the recent and related researches regarding lambda, load balancers and cold start issues were critically analysed.

2.1 Cold Start Related Researches

The cold start related studies, mitigating strategies and solutions are hot topics for cloud researchers. The study of cold start at different deployment strategies were explored by (Dantas et al.; 2022). The experiment and studies were focused on the container based and zip file based deployments at runtime offered by AWS. 13 different types of functions like image classifier, linear regression, factorial etc in node, python and java runtime were experimented. Its a clear depiction of the real world. Both container based and zip based deployments in arm64 and x86 architectures were tested. The experiment found that above 10 minutes of inactivity will lead to the destruction of idle virtual environment. And both java and node have faster initialisation on zip based deployments. Package size, runtime, memory configuration and deployment style are affecting the cold start. The research is highly useful for developers to choose a proper pattern according to their need. Another focused study on the behavior of node and java in AWS lambda environment were conducted by (Ferreira Dos Santos et al.; 2023). Its contradicting the above said research where it suggest node and java have faster and initialisation time. But this study concluding that node were able to reduce the startup time to 82 percent. And the study states that above seven minutes the reusing of containers seems to be drastically reducing. And the study is primarily focused on low use AWS lambda based services where cold start is a common scenario. The proposed research is also focusing on sporadic invocations at regular intervals. The clear backup of metrics and same computational implementation on both environment are justifying the recommendation of node over java runtime. But limited tests and shorter observation periods are drawbacks of this research.

A sophisticated approach to avoid cold starts were done by (Solaiman and Adnan; 2020) resulted into above 23 percent decrease in cold start of AWS environment. The solution is built with a container management architecture. The solution have queue for container states. It have states cold, warm and template. The containers will transition according to their invocation, use, state and other parameters. And pre-prepared containers as templates are kept for handling concurrent requests. The proposed solution is

implemented on OpenLambda [12] where it showed lower memory consumption, reduction in cold start occurrence and duration. But the complex architecture, scalability and performance at high load are not answered. Proper container orchestration and high level architecture are a real drawback for this research adaption. Increasing cost and platform dependent architectural designs are also a disadvantage.
The cold start in scientific workloads were addressed by (Bauer et al.; 2024). The research discussing about the sporadic and dynamic demands in serverless computing. And it outlining the demands and less exploration of the issue. This is the key area where the proposed research is focused. The paper suggesting four different empirical analysis and demonstrate trade-offs between build time, storage and cold start. The scientific computing generally needs a vast amount of storage and its use. So the study is heavily depended on Globus compute and Binder dataset which might not be the actual commercial scenario. The paper discussing about the build times, cold start and impact of warm start on containerized environments with four different strategies. The dummy workloads and functions are used at the experiment setup. This idea is integrated in the experimental functions to mimic the cold start.

A highly advanced research on container space utilisation were conducted by (Li et al.; 2023). The paper is discussing about the new architecture that can be adapted for serverless. It provides a comprehensive idea about the serverless platform and the need of its restructuring to avoid the computation power loss. The idle functions will have extra computational space for executions. The paper focusing on reusing it. Doing this the UN SDG's in serverless platform can be more efficiently implemented. The study offers a reusable resource architecture for new serverless platform designs, where less hardware can have more load to handle. The suggested solution have less cold start, cost efficiency, scalability and interoperability with existing Docker technologies. Overall the suggested solution outweighs its drawbacks. But the solution need direct control of the underlying infrastructure for end user to implement in popular CSP's. This will be a good solution to use resources efficiently for CSP's.

A application level solution is created by (Liu et al.; 2023) is identifying and separating the indispensable code from the optional code. The optional code will be loaded at demand only. This strategy is well suited for the serverless environment to reduce the latency and cost. A very big portion of the libraries and code are loaded to the execution environment at every events. The tool comprised of two separate modules one for identifying and removing unwanted files, identifying the entry points, optional functions generation. The second module is for function level rewriting of the code to optimise the on demand loading mechanism. By doing this only indispensable code will be loaded first and optional codes are loaded when necessary. The solution is able to achieve 78% of reduction in code loading latency and 42% reduction in total cold start latency. Further studies and development on this topic have great potential in cold start mitigation solutions.

2.2 Machine Learning Based Solutions For Cold Start

The time span and active containers in a execution environment were determined using using deep neural network and LSTM (Long short-term memory). And this model suggestions can be used to prewarm the AWS lambda environment. Based on this idea (Kumari et al.; 2022) have developed a ML based solution for pre warming the AWS lambda. The

[12] `https://github.com/open-lambda/open-lambda`

proposed solution had two separate layers one for reducing frequency of cold start and the second one for reducing cold start delay reduction. The dynamic approach for adjusting the pre-warmed containers reduced the latency. The matrices discussed are underlining the high performance of the adaptive solution. The solution is tested with different computation tasks in the test environment created in Openwhisk. The solution have a complex architecture thus inviting cost constraints. The heavy dependency on the log dataset will be a issue in its performance and scalability. A very similer research done by (Htet et al.; 2024) uses XGBoost model for forecasting the incoming requests. Based on this the environment is pre-warmed. The solution experimented with image extract function on AWS lambda, image classification on containerised function and image processing in step functions. The ML model was able to reduce the cold start to over 80 percent in all the scenarios. But the dataset used server hit pattern might not match with the real world scenarios. And the paper itself acknowledges the limitations of the solution in means of its cost, prediction accuracy, and the lack of testing on other CSPs like Azure and GCP. Where as the research of (Kumari et al.; 2022) were able test their system in multiple CSP's. The results received in both of these research have high discrepancy. The former one shows less than 40% improvement in average response time, whereas the later one have above 80% reduction in cold start. But a comparison of these are not possible due to difference in matrices and functions used for testing.

A more advanced solution combining light weight Kata containers[13] and pre-invoke function using ML model were proposed by (Karamzadeh and Shameli-Sendi; 2024). The suggested solution have kubernetis for managing the containers and gateway for gathering logs, predictive module for forecasting invocations, data store and a pre-invocation module for warming the function. Improved security through enhancing container isolation and mitigate risks of container escapes which is a critical vulnerabilities in serverless environments. Increased security and above 80% reduction in cold start are impressive. But still the cost is a main concern with complex architecture. The experiment is executed with low frequency requests. Though in a high frequency real world scenarios the predictive components performance will be a great concern. Another solution to reduce cold start using reinforced learning algorithm is advocated by (Htet et al.; 2024). It uses a learning agent for adaptive decisions, and the architecture is working on a feedback loop and uses Q-learning ML-algorithm. resource optimization, dynamic adaptability and improved performance are observed in this research. Whereas resource costs, training and making a capable ML model and scalability are practical issues in adapting the solution. Simplified workload, limited evaluation metric are other areas which need significant improvements. Even though with very minimal iterations the the RL agent was able to show high performance is a great sign. But the research lacks groundbreaking reduction of cold start comparing with other researches and need improvements in its overall tuning and selection of ML algorithm.

2.3 Cold Start Analysis, Studies, Reviews And Suggestions

A systematic study of cold start and possible solutions were suggested by (Alisha et al.; 2024) in their research. The research is focusing on understanding the root causes of cold start and suggesting practical solutions to overcome the latency issues. The efficiency of the solution discussed are depended on the computation task on the FaaS, workload and infrastructure capabilities and the ability to balance the resource. The solutions proposed

[13] `https://katacontainers.io/`

are not feasible for small solutions and it have complex architectures. Focus on synergy of hybrid and serverless models showing the current trends and future path for serverless computing. A very recent review paper based on cold start latency in serverless computing done by (Verma et al.; 2024) is a single point for analysing current trends, progressions and recent examinations in the serverless domain. The paper examines different other researches and found research gaps in tweaking warm-Up strategies, Optimizing Container Reuse Policies and Financially savvy Provisioning. These research gaps suggested by (Verma et al.; 2024) are trying to solve by the this research. The four different strategies to keep the AWS lambda warm and reusing containers by cost effective manner are the outcomes focused under the proposed research.

A well structured, extended and classification of existing cold start mitigation researches were done by (Ebrahimi et al.; 2024). The paper categories the existing works into four different sets. Application based, checkpoint based, invocation time prediction based and cache based. The study critically analysing the existing solutions and pointing the drawbacks. The research successfully addressed existed research gap in classification of works. The one main issue raised by this paper is the lack of benchmarks to for evaluating and studying the cold start in serverless platform. This is a real problem in conducting and classifying the researches on this particulate topic. The paper reviews some existing benchmarks like FunctionBench, ServerlessBench and SeBs and concluding that these tools are not covering all the required matrices for analysing cold start, latency and resource usage. And the author urges for setting new standards for benchmarks. the studies and suggestions in this paper is a great point for understanding the cutting edge of the cold start related researches and decision making for future studies.

2.4 Load Balancing Based Researches

Balancing load across the available servers are vital to have better scalability, reliability, distribution of workload and performance. Various researches were done on this topic to improve the above said aspects of a server. Designing and implementing a load balancer is a challenging task. The load balancers importance, concept and methodologies are described by (Mishra et al.; 2020) in their paper. The paper acknowledges key matrices that are vital for load balancers like time consumption, energy consumption, response time, throughput, scalability and migration. A load balancer should be focused on these matrices and want to have better performance on each of these constraints. The paper discussing static and dynamic load balancing algorithms and published the CloudSim simulation results. The diversity of the algorithms discussed are a proof of the comprehensive approach to address the load balancing challenges. It serve as a guide for the researchers to understand and improve the efficiency and reliability of cloud solutions. Simplistic assumption of the workload and lack of real world validation outside CloudSim seems to be a limitation.

A more structured and comparative study of load balancers were discussed and a new fault tolerant load balancing framework using ML technologies were put forward by (Shafiq et al.; 2022). Like the studies of (Mishra et al.; 2020) this paper also discussing about static and dynamic load balancers and variety of algorithms. Nature inspired algorithms are also a covered topic in the research. Overall the paper review 58 different algorithms and compares its strength, weakness. The discussed key matrices, balanced analysis of algorithms, comprehensive coverage and proposing a solution in light of the research is helpful for the development community. The need of energy efficient, fault

tolerant, reliable load balancers and its complexity are thoroughly covered in this paper. The proposed solution of fault-tolerant load balancer uses dual load balancer design this is to handle failures and fault prediction using machine learning. But the complexity of the load balancer, limited evaluation, integration with the existing systems, efficient ML model and algorithms are the area to improve and need clarity.

The research of (Devine et al.; 2000) is describing a dynamic load balancer for parallel applications named Zoltan. The tool is empowered with multiple load balancing algorithms based on geometric methods and graph based methods. The object oriented call back functions is qualifies the tool to have easy integration with wide variety of applications. The static high frequency workload and the dynamic capabilities can cause issues in this load balancer. A research on optimising load balancer on multi path routing network is done by (Yoheswari; 2024). The optimised solution is highly beneficial for high volume data intensive applications. The research choose Genetic Algorithms (GA) and Particle Swarm Optimization (PSO) as the candidates. Incorporating these algorithms with multi path routing protocols improved the the nature of load balancer to a dynamic one from the traditional static or heuristic methods. Enhanced efficiency, reduced latency and scalability are the achievements of this research. This research is also underlines the complexity of computation overhead, parameter tuning, scalability and integration complexity of designing a load balancer.

An serverless focused load balancing tool named Hedgi was designed and implemented by (Aslanpour et al.; 2024). Hedgi is a heterogeneous serverless edge computing platform for handling AI based data intensive edge computing. The authors introduced a method to classify the edge nodes based on their performance using different metrics such as throughput, response time, cost and energy consumption. It helps to understand the capabilities and limitation of each node in the system. Using this data the weighted round-robin load balancing is implemented. The research is focused on edge computing and its difficulties in handling the serverless platform. It gives valuable information about the difficulties in handling different capacity edge nodes using serverless. A load balancer having varying capacity nodes to handle has to face performance variability, dynamic workloads, bottlenecks if any and scalability issues. These are key points to consider while developing a load balancer.

An extensive survey and analysis of cloud based load balancing researches and algorithms were done by (Mishra and Majhi; 2020). The paper analyses a wide variety of algorithms including traditional, heuristic, meta heuristic, and hybrid approaches. And these algorithms were analysed using parameters like throughput, fault tolerance, , response time and consumption of energy. It provides a single source for developers and researchers for validating, comparing and understanding existing works. The paper suggesting the importance of future work on enhancing AI based hybrid algorithms, minimising energy consumption and validating algorithms with real world deployments. These suggestions are still need to be addressed properly, especially the energy efficient solutions. Whereas the taxonomy provided for load balancers and survey of survey framework introduced in this research are useful for future classification studies too.

The study of (Dantas et al.; 2022) and (Ferreira Dos Santos et al.; 2023) are contradicting in terms of its active AWS lambda environment time span. This is probably due to the region difference. Finding the best viable time through experiments is a responsibility of this research in us-west-1. The suggestion of node (Ferreira Dos Santos et al.; 2023) for reducing cold start will consider for this research and solution will implement in node. The research of (Bauer et al.; 2024) states the less exploration of sporadic events.

TOPIC	POSITIVE	NEGATIVE
(Dantas et al.; 2022)	Developers can take decisions for their solution deployment, runtime selection according to their code domain	Tests were executed for a short amount of time. Only on AWS with limited versions of runtimes. Future runtimes may change these findings.
(Ferreira Dos Santos et al.; 2023)	Infrequent applications will benefit from Node.js due to its reduced cold start latency.	At high frequency and low-frequency high throughput environments, the behaviour is not addressed.
(Solaiman and Adnan; 2020)	WLEC uses approximately 50% less memory while maintaining performance.	Additional components increase system complexity and require resource-heavy components
(Kumari et al.; 2022)	Workloads with predictable load will have an added advantage.	Handling unpredicted load scenarios will be an issue.
(Alisha et al.; 2024)	Hybrid models enhance overall system responsiveness.	Maintaining warm instances in hybrid setups will incur higher costs.
(Htet et al.; 2024)	Consistent invocation patterns will benefit from prewarming.	Computation overhead of ML models and dependency on model prediction accuracy during unpredicted load.
(Li et al.; 2023)	Reusing idle or underutilized container resources ensures efficient resource utilization.	Additional layer of monitoring and resource management is cost-consuming.
(Karamzadeh and Shameli-Sendi; 2024)	Increased security of the execution environment by mitigating container escapes.	Configuring and managing Kata Containers, Kubernetes, and CRI will add cost and maintenance.

Table 1: Cold Start Related Research Positives And Negatives

Contributing to this area are need of the industry. Similar to this research we too need to mimic the cold starts to study its implication in possible way. All the ML related cold start mitigation strategies high architectures which need extra computation and cost. A light weight solution will be a better idea. The research of (Verma et al.; 2024) found research gaps in AWS lambda warm up techniques, one of the main objective of this research. As (Ebrahimi et al.; 2024) states proper benchmark is a problem for studying cold start on AWS lambda. The research need to implement its own testing functions. The load balancing papers describing the complexity of its implementation via different algorithms. Our system is focusing on utilising the active virtual environment. The normal load balancers are not fit for this solution. Because our design has to be aligned with AWS lambda life cycle. The recap of literature review is organised in Tables 1 and 2.

TOPIC	POSITIVE	NEGATIVE
(Devine et al.; 2000)	Adaptive capacity in dynamic systems is great for a load balancer.	The complexity in the migration.
(Shafiq et al.; 2022)	It groups 58 existing load balancing algorithms, providing a platform for understanding various studies.	Experiments extensively used simulation tools, which do not fully depict real-world scenarios.
(Mishra et al.; 2020)	Dynamically reallocating tasks will improve system reliability and scalability.	Dynamic load balancing introduces computational and communication overhead.
(Mishra and Majhi; 2020)	Sets a best foundation for developing efficient, scalable, and energy-efficient load balancers.	Proposed solutions lack practical implementation and validation in real-world environments.
(Yoheswari; 2024)	Best for real-time applications like video streaming.	Integration complexity and computation overhead are key issues.
(Aslanpour et al.; 2024)	Able to overcome central bottleneck issues such as scalability and single points of failure in IoT.	Centralized load balancers fail to perform well under high workloads.

Table 2: Pros and cons of load balancers

3 Methodology

According to CRISP [14] understanding the business needs and AWS lambda working, generating a solution, implementation, evaluation, testing and fine tuning the solution are part of this research methodology. The research has to fine tune via different iterations by understanding the AWS lambda, research objectives and test results.

3.1 Business understanding

The main disadvantage in the serverless function is its cold start time. This is a major drawback of the FaaS services in high throughput environments and sporadic environments. AWS lambdas which are failing to scale and having high latency in software solutions will end up in business loss and unhappy customers. Though handling the cold start is became developers responsibility for latency critical applications.

3.2 Analysis How AWS Lambda Works

Active AWS lambda container reuse and strategies to keep the container active are the two focus points of this research. For sporadic requests if multiple requests are incoming to a AWS lambda in a short time span there is high chance of creating new execution environments and resulting cold start. Once the AWS lambda get its first event and

[14] https://www.datascience-pm.com/crisp-dm-2

it started execution of the first request, any subsequent request hitting the same AWS lambda after starting the first lambda can reuse the VM. A simple background test is done with node by hitting the lambda. Via this test able to understand that generation of new execution environment will create a new log stream in AWS cloudwatch. The initial invocation will have a Separate log in the AWS cloudwatch. The initial "REPORT" log entry will have "Init Duration". series request resulted reusing the AWS lambda VM. The experiment was able to understand three scenarios where cold start is highly likely to happen:

1. when a code change or new deployment occur;
2. when the inactive time between two events to the AWS lambda is higher;
3. when the AWS lambda is scaling.

3.3 Potential solution

Different architectures keeping lambda warm and best fir time frame has to be found for the Ireland region. Then a light weight solution has to be build and it has to be tested with API Gateway services to meet the research gap. A dynamic load balancing solution is considered initially with ML technologies. But from the literature review it is clear that ML based solutions have scalability issues and complex architecture.

From the initial experiments and research it is clear that the AWS lambda need to be invoked at regular intervals to keep it warm. This can be done using different architectures according to the implementing solution. There is a number of ways to keep the lambda warm and active so that incoming requests will be served fast [15]. An optimised architecture to keep lambda warm by controlled periodic invocation will be a great solution. It could be implemented as part of a circuit breaker pattern[16], or a customised retry with back off design pattern[17].

To reuse the active virtual environment requests should be blocked from accessing the lambda runtime. By handling it one after one will result improvements in latency, cost, computation power and carbon footprint. For reducing the cold start it is better to hold the request dynamically by assessing the load. In simple words we need to efficiently serialize the requests which might cause a scaling cold start. A lightweight load balancer for this can be created with existing cloud tools. The solution must be tested and to ensure its reusing the active environment via section 3.4 and redesigned if fails to accurately follow CRISP.

3.4 Test Cases and Execution

The project have two separate sections to test. First one the keep AWS lambda VM warming architecture and second one the load balancer. The first one have three different architectures. 3, 5, 7 and 10 minutes interval invocations are tested with the lambda warming architectures to find out optimum time span. The test period is determined to

[15] https://medium.com/\spacefactor\@m{}marcos.duarte242/keeping-your-aws-lambdas-warm-strategies-to

[16] https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/circuit-breaker.html

[17] https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/retry-backoff.html

be 12 hours. The test logs is gathered from the AWS cloudwatch and analysed its cold starts and warm starts. The selection of this time gap is from the literature review it is clear that 5 to 15 are the range suggested by researchers. So a best fir case of 3 and a boundary case of 5 is selected. Then test the system with 7 and 10 minutes too. If the lambda have same performance on 3 and 10 minutes we decided to increase the time according to the test result. It is clear that an accurate time is not able to find due to the non deterministic time of lambda lifecycle. Though we need to find a time span which will be a best fit to keep the environment warm.

The load balancer is tested with normal load and sporadic load. A randomised load test has to be done for the endpoints. A 1000 hits in a 1000 seconds are planned for this. The sporadic test is conducted using a input file this input file will have three arrays. In each array there is random times to hit the lambda. The array size is 5,15 and 25 respectively. The same file will be tested for each CRUD endpoint. The time period between two array of requests is set to be 5 and 15 minutes. And the test cases should be executed for varying cold start for AWS lambda. For this research artificially we introduce 1,3, and 5 seconds of cold strts to the lambda endpoints. Both API gateway and load balancer will be tested in the same fashion. The total execution time, total cold starts and estimated cost can be determined from the cloud watch report logs. All the above said tests will be executed through node scripts.

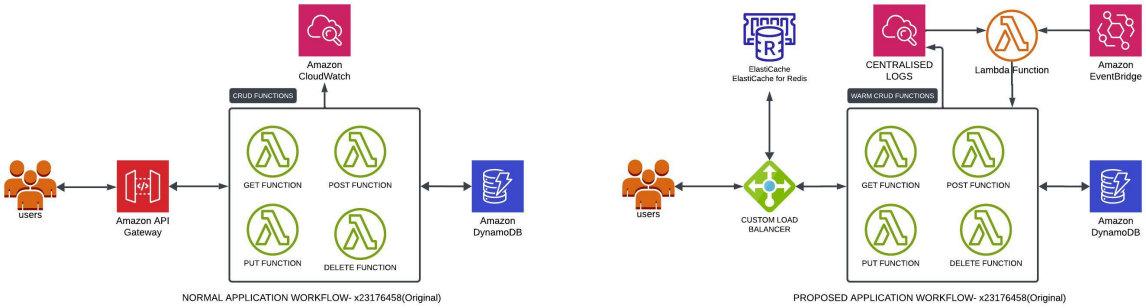
3.5 Implementation and evaluation

While implementing the knowledge gained from section 2 has to be taken in consideration. The implementation of the whole project will be on AWS due to its huge market share and cloud features. Evaluating results and solution in terms of cost, latency, scalability are crucial. The normal application architecture is depicted in Figure 2a and the newly proposed architecture is described in Figure 2b.

From section 2 it is clear that there is a lack of proper benchmark for testing the coldstart (Ebrahimi et al.; 2024). So a normal CRUD functions has to be implemented to match the real world scenario. A basic read write application in AWS lambda configured with AWS API gateway, AWS dynamoDB and AWS cloudwatch has to be implimented. The common used packages can be deployed as a seperate layer and configure in the AWS lambda. This CRUD function should be tested with AWS API gateway and suggested load balancer. Node can be utilised for as a best solution for implementing the CRUD funcrions (Ferreira Dos Santos et al.; 2023). Upon failures and undesired results various iteration on the configuration of the solution and business understanding has to be done to follow CRISP methodology pattern. The collected logs can be analysed using Google collab [18], [19]to have total response times and billed durations. A estimated cost can be calculated by from the test results using the billed duration. Any discounts or tiered pricing are not considering for cost comparison. Adding these to the solution and to the current scenario logically wont give any meaning.

[18]https://colab.research.google.com/drive/12qSmyoA2hMMqz0Hyw2vP6Bf3MDaVqcrK?usp=sharing

[19]https://colab.research.google.com/drive/1IKOHrAhAQ6fqGVY8Jezio-1jvvhgQZ4q?usp=sharing



(a) Using API Gateway

(b) Using Suggested Gateway

Figure 2: CRUD Application With API Gateway and Proposed Gateway

4 Design Specifications

Lambda is perhaps one of the most important AWS services. It is indeed the key component in FaaS. In the course if this research, it was found out a capacity problem AWS API Gateway. API Gateway gets saturated at a relatively low number of lambda functions. A comparatively simpler approach, a load-balancer, is presented as a viable and efficient alternative to the default AWS service.
The solution have implications for both performance and vendor locking of cloud providers. The widely used existing lambda warming architectures are invoking lambda at regular intervals regardless of lambda is used by user or not. Designing new architectures for keeping lambda alive and integrating it with a load balancer is an efficient solution.

5 Implementation

5.1 Tools And Services

In light of literature review it is evident that node have clear advantage on the the existing runtimes (Ferreira Dos Santos et al.; 2023) though node can be used in the lambda environments for reduced latency. Functional code can be deployed to AWS lambda. DynamoDB can be used as the datastore for the lambda CRUD functions. The lambda endpoint can be exposed and routed through API gateway. AWS Cloudwatch can use for collecting logs from AWS lambda and AWS API gateway. The in-build metrics and query capability of AWS Cloudwatch is highly useful for the project implementation in the cloud. The Ec2 instance is used for implimenting load balancer. An express app is working as a router here. Selection of express framework is purely because of its advanced routing capability and asynchronous nature of node. Apart from that lightweight, scalability and middleware support [20] also advantages of express. To make routing decision the understanding of the existing load is important. A shared storage space is mandatory for this.
AWS provides Redis ElastiCache services with very low latency for its read and write operations. This service can be used for keeping the current execution load record. Storing the details in a secret manager will be beneficial to have added security and faster easy access. AWS VPC, subnets and security groups can be implemented in the project

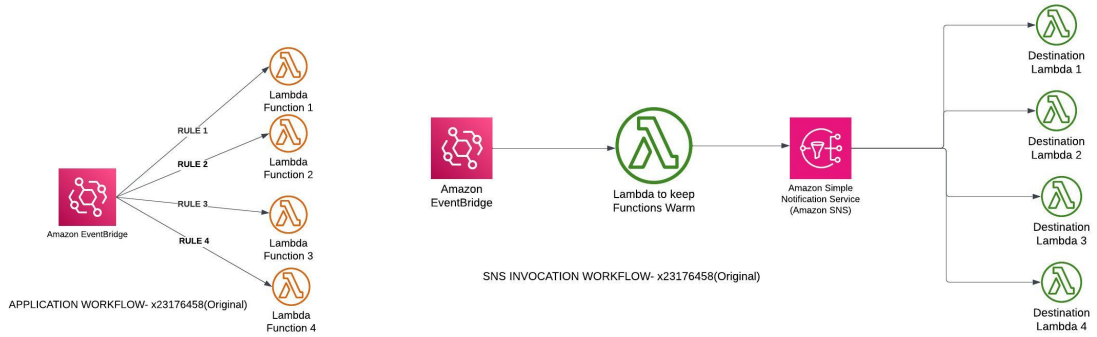
[20] `https://data-flair.training/blogs/expressjs-advantages-and-disadvantagesare`

environment to control the access and protect the application from threats. To control the development versions and code management Git hub was used [21], [22], [23]. The manual deployment will be time consuming due to the bulkiness of AWS services using. Deploying them separate and handling it is a tedious task. The serverless framework can be utilised here for automatic deployments [24]. Deployment stage controlling, creating resources and configuring resources, etc. can be done by commands and code in serverless.
Winston is used for logging. Its not only useful for configuring log level and transport options but also good for improved debugging and error handling. Moment package is used for date operations. It provides useful functions for calculating time related operations. The validator is used for validating the incoming request formats. Apart from this a random uuid generator from crypto package is used for creating specific log entries in the cloudwatch. Lambda execution environment provided aws-sdk package are used for accessing services and data inside the AWS. The above said packages are imported to functions through AWS lambda Layers.

5.2 Architectures to keep lambda warm

A number of different cloud architectures have been devised to keep lambda "warm" or "alive".

1. **AWS EventBridge Invocations.** AWS EventBridge invocations are widely used to keep the lambda warm. AWS EventBridge will send events to lambdas separately with a pre-configured payload. Figure 3a depicts the architecture of this method. AWS EventBridge provides event rules which will specifies the invocation pattern and payload to its destination.



(a) AWS EventBridge rule-based invocations for Lambda

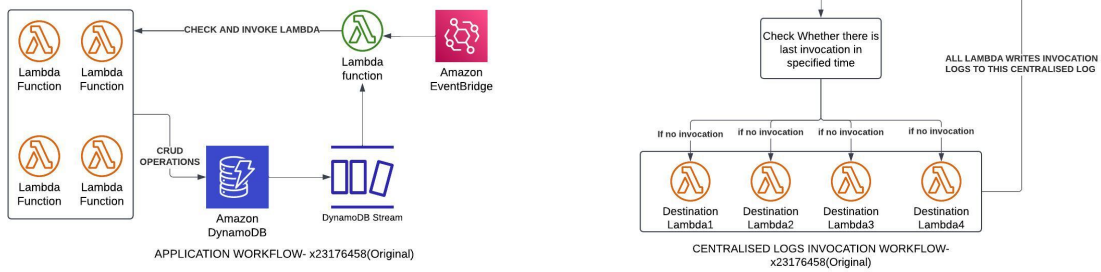
(b) Publisher and Subscriber Architecture to Keep Lambda Warm

Figure 3: Event Bridge and SNS Architectures for Lambda Invocations

2. **SNS Invocations.** SNS fan-out architecture are well suited for keeping the lambda alive. Here separate configurations for each lambda can be avoided. SNS can be acted as a publisher and lambda can act as subscriber to its events. Any event

[21] https://github.com/anilgovind-nci/ric-crud-application
[22] https://github.com/anilgovind-nci/Lambda-Layer
[23] https://github.com/anilgovind-nci/RIC-LOAD-BALANCER
[24] https://www.serverless.com/#How-It-works

passed to the SNS will broadcasted to subscribed lambdas. Though a self invoking lamnda can post a message to the SNS topic and the SNS topic trigger and pass this payload to every lambdas which are configured. The architecure is described below in Figure 3b.



(a) Invoking Lambda from DynamoDB Streams

(b) Invoke Lambda by Assessing Last Invocations

Figure 4: Invocation From DynamoDB Streams and Last Invocation Assessment

3. **DDB Stream Invocations.** The aforementioned architectures will invoke the destination lambdas even if the lambda is active. It is inefficient in terms of waste of computation and unwanted interrupt to the user functions lambdas. To avoid this a better solution is using DynamoDB stream to evaluate the last invocations and invoke the lambda if there is no invokes. For this a event bridge enabled lambda auto invokes and check the stream at regular intervals. The proposed architecture is depicted in Figure 4a. Where as comparing with other architectures this architecture cannot be used in every scenarios. The limitations are the destination lambda should want to have a DDB write operation to have a log in DDB stream. Apart from that any validation issues or internal server errors in the lambda will not effect a write operation in DDB. This scenario will also result into unwanted invoke request to the destination lambda. The GET operations will not generate DDB stream, so invoking destination GET lambdas is also not possible via this architecture. This architecture is a proposal from this research. [25].

4. **Cloud Watch Analysed Invocations** Keeping the Lambda warm from cloudwatch is depicted in Figure 4b. Here all the lambdas which need to be invoked will be writing its invocation logs to a centralised AWS cloudwatch log. AWS Event bridge is configured with a single lambda and this lambda will be invoked at regular intervals. The lambda will query through the log and distinguish whether a destination lambda have invoked or not in the configured time span. If no, the lambda will send a invocation event to the destination lambda. The destination lambda have code blocks to distinguish the normal used request and keep warm request. Keep warm requests will handle at the very starting of AWS lambda to reduce cost and execution time. This proposed architecture have clear advantage

[25] `https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html`

over any other architectures in terms of its ability to avoid unwanted triggers to the destination lambda. Thus reducing its load when the function is serving user requests. And a single point configuration for all the destination lambda is also highly beneficial.

5.3 Application Design

The AWS API gateway integrated test design is depicted in Figure 2a. Where routing, load handling, authentication, authorisation, logging, version management, etc are controlled by AWS gateway. The newly proposed architecture is depicted in Figure 5. The system is updated with a keep lambda warm architecture 4b discussed in Section 4. And in place of API Gateway the proposed light weight load balancer is implemented to reuse the containers.

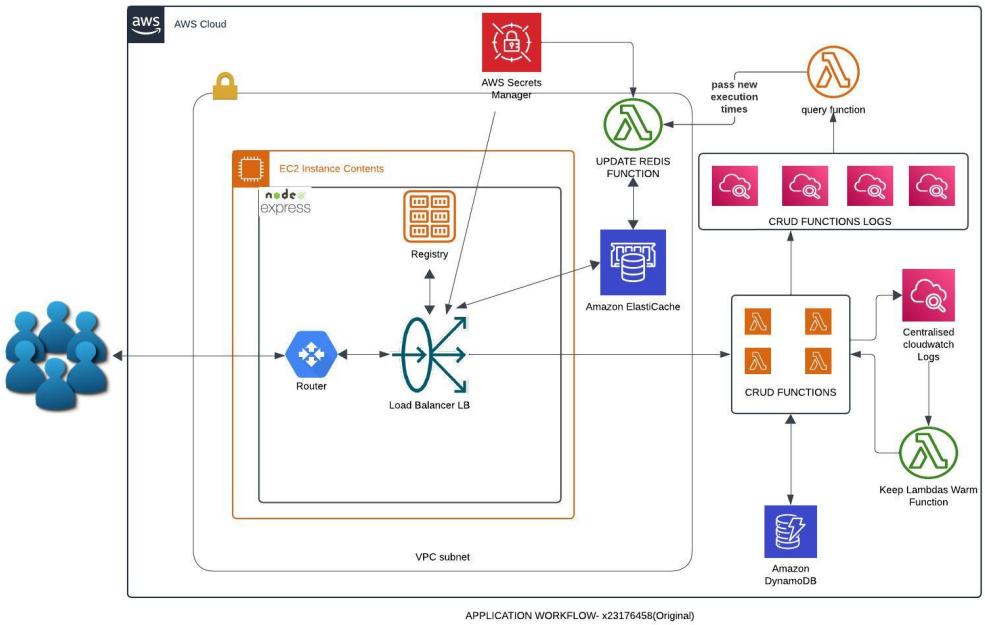


Figure 5: High Level Architecture of Application Workflow

- **User requests:** Incoming requests to the CRUD functions
- **Router:** An Express app to handle the the incoming route and add middleware like redis handler to the incoming request.
- **Load balancer functions:** It handles the incoming requests to have minimum response time. A detailed explanation of how the load balancer works is explained in section 5.4
- **Registry:** for each endpoints an associated registry will be there. This registry have credentials like lock keys to handle the incoming request. A detailed explanation is in section 5.4
- **Ec2 Instance:** This instance is the part where router, registry and load balancer is implemented.

- **Redis AWS elastic cache:** The elastic cache have records of active and deactivated lambda node records and average time of execution and average cold start time. These values are used by load balancer. The redis will be updated continuously by the update redis function in periodic intervals with new cold start average and execution time.
- **Update Redis Function:** Update redis function will be invoked by query function. This will function will read and update the redis values. Qeury function will invoke update redis funnction with keys to access secret manager and new average times. Then the function will fetch secret value from AWS secret nanager and update redis with new average time execution and new average cold start time.
- **Query Function:** This function will check every individual logs of the CRUD functions by executing a query on the logs. If the logs returned new execution average and cold start average it will call the redis update function with by passing the respective values with function secret manager key.
- **VPC:** The EC2 instance, Redis and update Redis function are configured inside a VPC and smiler security groups. This is to have increased security to the environment and the Redis is integrated with the VPC [26]. The Redis and its accessing environments is kept in same VPC.
- **Secret Manager:** For added security the credential details such as Redis endpoint, Redis keys associated with each CRUD function are kept in the secret manger.
- **CRUD Functions Logs:** There is separate logs for each of the functions. The REPORT log generated by AWS after every lambda invocation is vital important for keeping the system dynamically adjusting with the code changes and response time delays. This can be read from these logs using query function and update the redis.
- **CRUD Functions:** These are the test functions for CREAT,READ,UPFATE and DELETE from AWS dynamoDB.
- **DynamoDB:** DB service for storing data.
- **Centralised Logs:** This is for CRUD function to write its invocations logs.
- **Keep Lambda Warm Function:** The lambda will query through the centralised logs and invoke CRUD functions if needed.

5.4 Implementing a load balancer

The core part of the project is to set up a load balancer programmatically. From the literature review it is clear that none of the existing load balancer algorithms or framework wont match for the proposed load balancer. A new load balancer based in express routing and custom routing logic has to be implemented. The load balancer should be built according to the nature of lambda and its behaviour. Because its motive is to exploit lambda active environment. The router will route user request to proper destination file. For every endpoints there will be a separate load-balancing pattern via configurable parameters. The load balancing logic and workflow is same for every request. The workflow of the load balancer is depicted in Figure 6. At the server start load balancer

[26] `https://docs.aws.amazon.com/AmazonElastiCache/latest/dg/VPCs.EC.html`

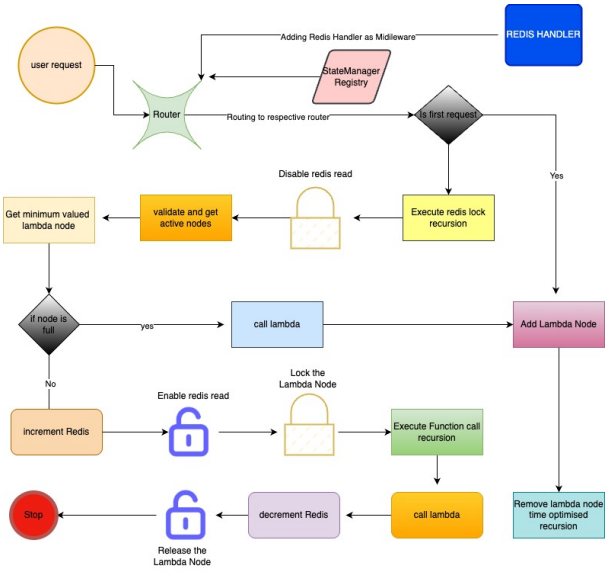


Figure 6: Load Balancer Workflow: it starts with a user request

will initialise connection with the Redis and initialise state objects for every endpoint. This object will be kept alive during the entire service life of the server. The attached middleware of Redis handler is accessible to every request for handling Redis operations. The state manager object will be attached to every incoming request at the routing stage.

The load balancer will add a lambda node having the function details and average time to complete the AWS lambda function to Redis for the very first request in server. The physical meaning of lambda node is there is one active environment in AWS. Subsequently it will call the recursive function of remove lambda node. This recursive function efficiently handle the life of each lambda node. A lambda node have active life time of 3 minute, then it will be deactivated and destroyed after 5 minutes. This is to overcome the discrepancies that might occur in the lambda node at calculating average execution time. When the node is deactivated no new request will use this lambda node. The recursive function once activated will keep running the entire lifetime of the server and it will make sure that at least one active node is kept in the redis all the time.

For every request there is a redis read lock recursive function. This function will call recursively if the state manger value for redis read block is true. if one request is able to read the value from redis, immediately it exists and lock the redis read again. So any subsequest request will be locked in the above loop untill the flag value changes. After reading redid the response will be validated and fetch active nodes from it. At a given time there can be active and de-active nodes. Once the active nodes have fetched the node with minimum load will be selected. Code will check whether the node is already crossed its limit. The maximum number of requests that a lambda node will handle can be calculated from the following equation.

$$\text{Limit of holding requests in an active node} = \frac{\text{Average cold start time}}{\text{Average execution time}}$$

Initially global variables were used in place of Redis. But this will create a problem if the solution has to have scalability. Then DynamoDb is also considered but the latency for its read write operation will an issue in the performance of the system. Using the above mentioned equation we can find the maximum number of request in a single node

and successfully serialise the parallel concurrent requests. This is basically using the time gap difference between cold start and warm start. There is no maximum nodes set in the solution, so in scaling condition it can scale upto the limit of AWS inbuilt restrictions, but can handle more requests. If the average time for a lambda node is above its cold start limit, lambda will be called directly and redis will update with a new node. Which means one more active environment in AWS. If the condition check fails the redis will be updated with added average execution time for the current node. Then the state lock parameter will be set to false to free the earlier locked requests. The redis operations are fast comparing with lambda fetch operation. Calling lambda directly will cause a cold start. To mitigate this request have to go through one more recursion. Where it will check any previous request is already executing. This is through keeping the current executing lambda node keys in a array. If the array does not have the key, which means the execution environment is free at AWS and we can invoke it. Before this call lambda node key will be added to the state manager to lock upcoming requests. Through this function we ensuring that the lambda invoked in one after another. After getting the response the redis lambda node data for average execution time is decremented and lambda node key will be removed from the array. Then response will be send back to customer.

5.5 Setting The Environment

The project need AWS services of lambda, DynamoDb, Redis, Ec2, Cloud watch, gateway,Event bridge, Secret Manager, VPC, Security groups and Subnets as cloud resources. The test environment of CRUD applications and architectures to keep lambda warm were developed and deployed using serverless. Serverless framework create a zip of the configurations and user code then upload it to s3. From there using cloud formation template service of AWS, serverless automatically deploy all the resources to AWS. This whole process can be done using a single command from the local machine.

The VPC, Ec2, Redis, lambda layer and DynamoDB are created manually and configured in the serverless package for deployment of other services. Its mandatory that Redis and its accessing resources has to be in same VPC. VPC, associated subnets and security group were created from AWS console. While creating VPC the required subnet, internet gateway and route tables were creted. The security group is also created from the AWS console and configured inbound rules asmport 80 for http, 6379 for redis and port 22 for ssh. The outbound is set to everywhere. After creating the security group, VPC, subnet etc the AWS elastic redis cache is implimented. While configuring the redis the previously created security group, subnets and VPC is configured. AWS elasticache provides four kinds of caching service Valkey caches, Memcached caches, Redis OSS caches, Global datastores. Out of this redis is used due to its advanced datastructure storing capacity, persitance, performance and scalability. To scale the redis horizontally at peak times the cluster mode is enabled while creating redis [27].

The Ec2 is created using AWS free tier t2.micro instance. The ubuntu image is selected for server and a EBS volume of 8Gb is attached to it. A private key pair is used for establishing the ssh connection. The newly created VPC and security groups were used while setting up the Ec2 instance. The instance has to be in the VPC of redis to access redis. While developing the load balancer the redis was unable to fetch from outside VPC resources like lambda or Ec2. The public ip created and assaigned to the

[27] `https://github.com/alessandroprudencio/aws-elasticache-redis-lambda`

Ec2 instance to have stable ip and connection string. Node and required environments were installed in the AWS Ec2.

The CRUD functions were using some common packages like winston, moment, validator etc. All these packages were installed in a node project and the zip file is uploaded as a layer in lambda. Various versions of a layer can be configured in lambda. The layer arn is used for configuring the serverless file. DynamoDB is created via AWS console is configured to have provisioned read and write range of 1 to 10 and with read and write capacity auto scaling on.

6 Evaluation

In order to understand the best architectures to keep lambda warm, three architectures were implimented. Namely, using AWS EventBridge, AWS SNS, and AWS CloudWatch. The frequency of requests were further simulated as coming at regular intervals or sporadic.

6.1 Lambda Warm Experiment 1

The initial testing was conducted with invoking lambda at 3 minutes of interval. The experiment conducted for 12 hours and a total of 241 invocations were evaluated. The gained output is depicted in Table 3. The experiment expected outcome is to understand the cold start in smaller time intervals.

METHOD	COLD START RATE (%)
GET	4.15%
POST	5.40%
PUT	2.90%
DELETE	3.32%

EventBridge

METHOD	COLD START RATE (%)
GET	2.90%
POST	3.32%
PUT	4.15%
DELETE	3.32%

SNS

METHOD	COLD START RATE (%)
GET	3.73%
POST	2.90%
PUT	3.32%
DELETE	2.90%

CloudWatch Logs

Table 3: Cold Start Rates of Endpoints for 3-Minute Invocation Time Span

6.2 Lambda Warm Experiment 2

The testing of lambda warming architectures were done with invoking lambda at a time gap of 5 minutes. The results obtained is explained in Table 4. The experiment is conducted for 12 hours. Each endpoint invoked 144 times in this time span. The experiment is expected understand lambda cold start in moderate time intervals.

METHOD	COLD START RATE (%)
GET	5.56%
POST	6.94%
PUT	6.25%
DELETE	6.94%

EventBridge

METHOD	COLD START RATE (%)
GET	7.64%
POST	6.25%
PUT	4.86%
DELETE	6.25%

SNS

METHOD	COLD START RATE (%)
GET	8.33%
POST	4.86%
PUT	6.25%
DELETE	6.25%

CloudWatch Logs

Table 4: Cold Start Rates of Endpoints for 5-Minute Invocation Time Span

6.3 Lambda Warm Experiment 3

The third experiment was done with 7 minutes intervel. The experiment conducted for 12 hours and 102 invocations are evaluated. The result is depicted in Table 5. Through

this experiment the we can understand lambda cold start in moderately high intervals of invocation time

METHOD	COLD START RATE (%)
GET	100.00%
POST	100.00%
PUT	100.00%
DELETE	100.00%

EventBridge

METHOD	COLD START RATE (%)
GET	100.00%
POST	100.00%
PUT	100.00%
DELETE	100.00%

SNS

METHOD	COLD START RATE (%)
GET	100.00%
POST	100.00%
PUT	100.00%
DELETE	100.00%

CloudWatch Logs

Table 5: Cold Start Rates of Endpoints for 7-Minute Invocation Time Span

6.4 Lambda Warm Experiment 4

The experiment conducted with invoking time span of 10 minutes were depicted in Table 6. 72 invocations were evaluated and the experiment duration was 12 hours. This test is for understanding the lambda cold start in high interval invocations.

METHOD	COLD START RATE (%)
GET	100.00%
POST	100.00%
PUT	100.00%
DELETE	100.00%

EventBridge

METHOD	COLD START RATE (%)
GET	100.00%
POST	100.00%
PUT	100.00%
DELETE	100.00%

SNS

METHOD	COLD START RATE (%)
GET	100.00%
POST	100.00%
PUT	100.00%
DELETE	100.00%

CloudWatch Logs

Table 6: Cold Start Rates of Endpoints for 10-Minute Invocation Time Span

6.5 Randomised Load Test

In this experiment 1000 requests were sent to each endpoint within 1000 seconds. This test is for understanding the performance of load balancer in real world scenarios. The requests are randomised. So there is no any clear pattern for request hitting. The results are depicted in table 7.

SOURCE	METHOD	COLD STARTS	TOTAL BILLED DURATION (ms)	TOTAL MEMORY USE (MB)	TOTAL USER RESPONSE TIME (ms)
API Gateway	GET	2	37696	104303	132300
API Gateway	POST	5	40920	111248	164686
API Gateway	PUT	2	40736	111098	134660
API Gateway	DELETE	4	39335	110331	135828
API Gateway	**Total**	**13**	**158687**	**436980**	**567474**
Load Balancer	GET	0	32040	102908	147869
Load Balancer	POST	0	34554	109130	150741
Load Balancer	PUT	0	34372	110000	153918
Load Balancer	DELETE	0	34971	111986	153003
Load Balancer	**Total**	**0**	**135937**	**433024**	**605531**

Table 7: Performance Comparison for API Gateway and Load Balancer with Cold Start and User Response Time

6.6 Sporadic Experiment 1

The below experiment is conducted with 1 second cold start latency. Experiment had 3 waves of 5, 15 and 25 requests at a time frequency of 5 minutes depicted in table 8 and 15 minute depicted in 9. The main goal for this test is to understand performance of load balancer in lower cold start functions.

SOURCE	METHOD	COLD STARTS	TOTAL BILLED DURATION (ms)	TOTAL MEMORY USE (MB)	FAILURE RATE	TOTAL USER RESPONSE TIME (ms)
Load Balancer	GET	0	1845	4500	0	5920
Load Balancer	POST	0	1663	5040	0	5984
Load Balancer	PUT	0	1806	4995	0	5860
Load Balancer	DELETE	0	1910	4950	0	5961
API Gateway	GET	2	2160	4458	0	10712
API Gateway	POST	2	2228	4979	0	10088
API Gateway	PUT	2	2320	4946	0	11401
API Gateway	DELETE	2	2444	4976	0	10120

Table 8: Performance Comparison for 1-second Cold Start Invocation Invoked in 5 Minute Frequency

SOURCE	METHOD	COLD STARTS	TOTAL BILLED DURATION (ms)	TOTAL MEMORY USE (MB)	FAILURE RATE	TOTAL USER RESPONSE TIME (ms)
Load Balancer	GET	0	1695	4526	0	5571
Load Balancer	POST	0	2009	5040	0	6233
Load Balancer	PUT	0	1848	4995	0	6251
Load Balancer	DELETE	0	1893	4950	0	5686
API Gateway	GET	19	5142	4457	0	43015
API Gateway	POST	20	4854	4831	1	46387
API Gateway	PUT	19	4520	4832	1	44463
API Gateway	DELETE	20	4847	4716	2	46803

Table 9: Performance Comparison for 1-second Cold Start Invocation Invoked in 15 Minute Frequency

6.7 Sporadic Experiment 2

The below experiment is conducted with 3 second cold start latency. Experiment had 3 waves of 5, 15 and 25 requests at a time frequency of 5 minutes depicted in table 10 and 15 minute depicted in 11. The goal of this test is to understand performance of load balancer in moderate cold start functions.

SOURCE	METHOD	COLD STARTS	TOTAL BILLED DURATION (ms)	TOTAL MEMORY USE (MB)	FAILURE RATE	TOTAL USER RESPONSE TIME (ms)
Load Balancer	GET	0	1782	4454	0	6237
Load Balancer	POST	0	1747	4905	0	7013
Load Balancer	PUT	0	1836	4950	0	6479
Load Balancer	DELETE	0	1759	4950	0	6161
API Gateway	GET	4	2675	4462	0	20356
API Gateway	POST	11	3659	4378	5	74219
API Gateway	PUT	4	2656	4947	0	21933
API Gateway	DELETE	5	2884	4937	0	34967

Table 10: Performance Comparison for 3-second Cold Start Invocation Invoked in 5 Minute Frequency

SOURCE	METHOD	COLD STARTS	TOTAL BILLED DURATION (ms)	TOTAL MEMORY USE (MB)	FAILURE RATE	TOTAL USER RESPONSE TIME (ms)
Load Balancer	GET	0	1651	4455	0	9081
Load Balancer	POST	0	1763	4905	0	9590
Load Balancer	PUT	0	1567	4950	0	8750
Load Balancer	DELETE	0	1686	4950	0	8037
API Gateway	GET	23	4979	3249	12	91408
API Gateway	POST	25	4740	3280	15	108238
API Gateway	PUT	25	4570	2953	18	125908
API Gateway	DELETE	16	4240	4523	4	63011

Table 11: Performance Comparison for 3-second Cold Start Invocation Invoked in 15 Minute Frequency

6.8 Sporadic Experiment 3

The below experiment is conducted with 5 second cold start latency. Experiment had 3 waves of 5, 15 and 25 requests at a time frequency of 5 minutes depicted in table 12 and 15 minute depicted in 13. The goal of this test is to understand performance of load balancer in high cold start functions.

SOURCE	METHOD	COLD STARTS	TOTAL BILLED DURATION (ms)	TOTAL MEMORY USE (MB)	FAILURE RATE	TOTAL USER RESPONSE TIME (ms)
Load Balancer	GET	0	1906	4410	0	5732
Load Balancer	POST	0	1796	4948	0	5781
Load Balancer	PUT	0	1780	4995	0	5287
Load Balancer	DELETE	0	1901	4950	0	5789
API Gateway	GET	15	1651	4455	6	35051
API Gateway	POST	5	1763	4905	0	39615
API Gateway	PUT	5	1567	4950	0	36024
API Gateway	DELETE	5	1686	4950	0	35995

Table 12: Performance Comparison for 5-second Cold Start Invocation Invoked in 5 Minute Frequency

SOURCE	METHOD	COLD STARTS	TOTAL BILLED DURATION (ms)	TOTAL MEMORY USE (MB)	FAILURE RATE	TOTAL USER RESPONSE TIME (ms)
Load Balancer	GET	0	1606	4449	0	5298
Load Balancer	POST	0	1740	4950	0	5817
Load Balancer	PUT	0	1652	4995	0	5340
Load Balancer	DELETE	0	1625	4950	0	5015
API Gateway	GET	25	4917	2460	20	149874
API Gateway	POST	16	3436	3292	15	93451
API Gateway	PUT	25	4614	2735	20	153225
API Gateway	DELETE	16	3463	3307	15	94077

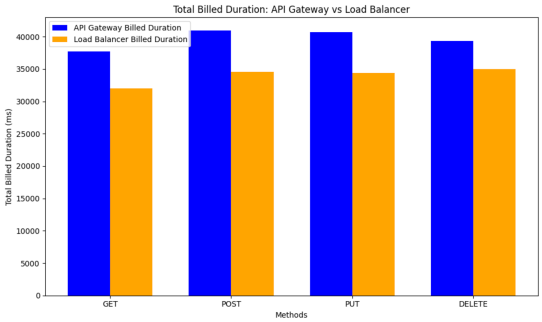
Table 13: Performance Comparison for 5-second Cold Start Invocation Invoked in 15 Minute Frequency

6.9 Discussion: Lambda Warm Test Results

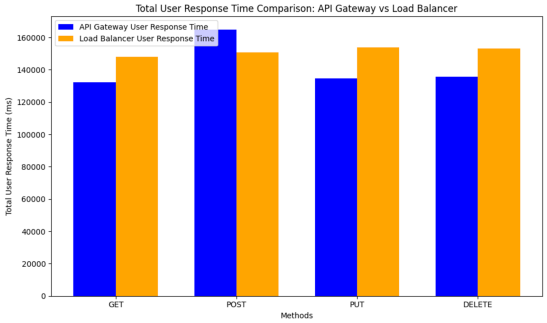
As can be seen, all the three implemented architectures were very efficient in keeping the lambda active. It was tested with varying time ranges. The outcome is depicted in Sections 6.1, 6.2, 6.3, and 6.4. From the table 5 and table 6 it is clear that above 5 minutes of inactivity the lambda environments are not active. Assessing the result it is evident that for the proposed load balancer the keep lambda module should be configured to invoke at 5 minutes interval or less than that. An advantage of the invoking lambda from checking the cloud watch is that in the testing environment according to the use of lambda unwanted triggers were avoided. Especially on the 5 minute frequency test scenarios. This was helpful to maintain a clean log at peak times and usage times. And it reduces load to the CRUD functions. A general outcome from tables 3, 4, 5, 6 is that lower the invocation time higher the efficiency to reduce cold start. We have the lowest (2.9%–5.4%) for 3 minutes invocation frequency, a slightly increased (4.86%–7.64%) for 5 minute invocation. Both are ambient for keeping lambda warm. 7 and 10 minute invocations frequency are not good for keeping lambda warm. This result is negating findings of (Dantas et al.; 2022). which says above 10 minutes reusing of containes were reduced. But aligning with findings of (Ferreira Dos Santos et al.; 2023) as above 7 minute lambda reusing capacity reduces.

6.10 Discussion: Randomized load test

The randomized Load Test result depicted in the table 7 are good place for cost and latency comparison for normal load. The insight of the table is depicted in Figure 7a, Figure 7b. This means in frequently accessed services the latency is high, but cost is lower for lambda functions with the proposed solution. On long run there is no much cold starts so the API gateway can out perform the custom load balancer. There was 13 occurance of cold start during the entire tests in API gateway but none for the implemented solution. The AWS API gateway seeming to use more memory than the load balancer which is due to its cold starts.



(a) Billed Duration for 1000 requests in 1000 seconds



(b) User Response Duration for 1000 requests in 1000 seconds

Figure 7: Comparison of latency and billed duration for API gateway and Custom Load Balancer

6.11 Discussion: Load Balancer Test Results

The executed test results were illustrated in Sections 6.6, 6.7 and 6.8. An interesting fact noticed in the test result is on sporadic intervals the load balancer outperformed the API gateway in terms of its throughput handling and latency . The API Gateway is failing continuously to scale above 10 VM. This is because of the current configuration wont allow more than 10 VM of a single lambda at a given time. Whereas the custom Load balancer was able manage the load efficiently. None of the tests were experienced cold start during its execution. It is evident from tables 9 8, 11 10, 12 13 , . Whereas API gateway experienced a total of 311 cold starts during the sporadic tests. This means the solution was 100% effective in test cases explained in section 6.6, 6.7,6.8. This means complete reuse of existing virtual environment in AWS as we targeted. When cold start of a lambda increases the failure rates for sporadic events were drastically increasing in API gateway. It is evident from the tables 9 8, 11 10, 12 13. Though the proposed solution fitness will improve when cold start increases. The Custom design was able to reduce cost to nearly 20% and user experiencing latency to 80% when considering tables 8, 10, 12. A exact approximation is hard due to the failed cases in API gateway. Even though with less energy, cost and latency the system was able to do more computation. This is closely aligning with UN SDG's. Considering the invocations of both load balancer and API gateway the memory use seems to be nearly same and non deterministic. The main technologiacal advantage of this research is its ability to mitigate the maximum number of active lambda environments and reduced latency and cost of lambda. The proposed load balancer have a wide variety of use cases in software industry. Some of it are listed below.

- **Ticket and Events Booking:** In these applications high request rate and sales can occur at the release of events and ticket. To have better latency and less cost the proposed architecture can perform better.
- **Marketing Campaigns or Flash sales:** The marketing campaigns or flash sales will introduce additional load to the respective server blocks. keeping these blocks in the load balancer will be beneficial
- **Scheduled Jobs:** For scheduled jobs like report generation, event triggered high throughput jobs.

- **Iot devices:** Its quite common that IoT devices updating cloud with batch jobs. In these scenarios the custom load balancer can perform better for high cold start functions.
- **Satellite Communication:** Lambda functions which using telemetry data from a satellite, decode and store the results in a database, and trigger downstream analytics (e.g., generating weather reports or detecting environmental changes). The satellite will send data to ground stations periodically.
- **Unlocking Vendor Locks:** Using this light weight portable load balancer AWS API gateway load balancer can be unlocked. The load balancer can be in a private cloud or in other CSP.

6.12 Limitations

On high concurrent loads the proposed system is failed to work better and crashing. There is a number of recursive functions in the solution this will run and crash the application or make it non responsive. This is because of high memory consumption and frequent garbage collection. The redis read write operations and discrepancies were occurring and ending up at loosing of performance. The system has to have the capacity of handling high load concurrently. The performance on real world scenarios are still need to be tested and verified for the solution. Algorithms to change the recursive patterns is crucial due to its complexity generation in the working environment. The state manager registry kept at locally will affect the scalability of the load balancer, keeping it in a shared memory in case of scalable environments are preferred. On longer run the cold starts occurances are very less comparing with low frequency events. So for continuously using endpoints the solution is not perfect. After a change and release of new lambda the time to change the current average cold start and execution time will aslo affect the performance of the system.

7 Conclusion and Future Work

This research explored how to minimize cold start latency in AWS Lambda. The primary objective was to design and evaluate architectures to keep Lambda environments warm and reuse active environments through a lightweight custom load balancer. The results gave revolutionary practicality by filling the research gap for reducing latency and cost for sporadic as well as latency critical environments. This research successfully developed and tested various lambda warming architectures. The research validated that the latency can be reduced by carefully timed invocation strategies.And it also validated contradictions in previous researches about the invocation time span for active lambdas. By targeting only inactive lambda AWS cloudwatch driven lambda invocation is most efficient. The designed load balancer reduced cold start completely under sporadic test cases. And it outperformed API gateway in cost and latency. The light weight load balancer was able to achieve a 80% reduction of latency and 20% reduction of cost for sporadic events. For randomised load the solution was able to reduce cold start completely. Thus it fulfilling the research question and pave path for future studies in serverless. In sporadic and latency critical environment the developers can have more responsive and cost effective light weight solution without any vendor locking. The solutions can be implemented in

dedicated servers or CSP's for having multicloud architectures. The research is an alternative for API gateway for better scalability in unpredicted high traffic environments. For future work the system faced challenges in handling high load of concurrent requests due to redis bottle-neck. A advanced in-memory database will be a better solution. Implementation of the research in other CSP's and assess its effectiveness. And Configuring the light weight load balancer in a FaaS services and evaluating its performance are for future. Overall this research provides a solid foundation for advancing serverless computing efficiency.

References

Alisha, S. H., Mohasin, S. M., Sai, B. N. V. S., Vajrapu, D., Tumuluru, P. and Burra, L. R. (2024). Analyzing cloud performance optimization: Strategies to enhance cold start latency, *2024 5th International Conference on Smart Electronics and Communication (ICOSEC)*, pp. 487–492.

Aslanpour, M. S., Toosi, A. N., Cheema, M. A., Chhetri, M. B. and Salehi, M. A. (2024). Load balancing for heterogeneous serverless edge computing: A performance-driven and empirical approach, *Future Generation Computer Systems* **154**: 266–280.
URL: *https://www.sciencedirect.com/science/article/pii/S0167739X24000207*

Bauer, A., Gonthier, M., Pan, H., Chard, R., Grzenda, D., Straesser, M., Pauloski, J. G., Kamatar, A., Baughman, M., Hudson, N., Foster, I. and Chard, K. (2024). An empirical investigation of container building strategies and warm times to reduce cold starts in scientific computing serverless functions, *2024 IEEE 20th International Conference on e-Science (e-Science)*, pp. 1–10.

Dantas, J., Khazaei, H. and Litoiu, M. (2022). Application deployment strategies for reducing the cold start delay of aws lambda, *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pp. 1–10.

Devine, K., Hendrickson, B., Boman, E., St. John, M. and Vaughan, C. (2000). Design of dynamic load-balancing tools for parallel applications, *Proceedings of the 14th International Conference on Supercomputing*, ICS '00, Association for Computing Machinery, New York, NY, USA, p. 110–118.
URL: *https://doi.org/10.1145/335231.335242*

Ebrahimi, A., Ghobaei-Arani, M. and Saboohi, H. (2024). Cold start latency mitigation mechanisms in serverless computing: Taxonomy, review, and future directions, *Journal of Systems Architecture* **151**: 103115.
URL: *https://www.sciencedirect.com/science/article/pii/S1383762124000523*

Ferreira Dos Santos, P. O., Jorge de Moura Costa, H., Leithardt, V. R. Q. and Jorge Silveira Ferreira, P. (2023). An alternative to faas cold start latency of low request frequency applications, *2023 3rd International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*, pp. 1–6.

Htet, T. Y., Shwe, T., Mendonca, I. and Aritsugi, M. (2024). Pre-warming: Alleviating cold start occurrences on cloud-based serverless platforms, *2024 IEEE 10th International Conference on Edge Computing and Scalable Cloud (EdgeCom)*, pp. 66–72.

Karamzadeh, A. and Shameli-Sendi, A. (2024). Reducing cold start delay in serverless computing using lightweight virtual machines.
URL: *https://www.sciencedirect.com/science/article/pii/S1084804524002078*

Kumari, A., Sahoo, B. and Behera, R. K. (2022). Mitigating cold-start delay using warm-start containers in serverless platform, *2022 IEEE 19th India Council International Conference (INDICON)*, pp. 1–6.

Li, B., Zhan, Y. and Ren, S. (2023). A fast cold-start solution: Container space reuse based on resource isolation, *Electronics* **12**(11).
URL: *https://www.mdpi.com/2079-9292/12/11/2515*

Liu, X., Wen, J., Chen, Z., Li, D., Chen, J., Liu, Y., Wang, H. and Jin, X. (2023). Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing, *ACM Trans. Softw. Eng. Methodol.* **32**(5).
URL: *https://doi.org/10.1145/3585007*

Mishra, K. and Majhi, S. (2020). A state-of-art on cloud load balancing algorithms, *International Journal of computing and digital systems* **9**(2): 201–220.

Mishra, S. K., Sahoo, B. and Parida, P. P. (2020). Load balancing in cloud computing: A big picture, *Journal of King Saud University - Computer and Information Sciences* **32**(2): 149–158.
URL: *https://www.sciencedirect.com/science/article/pii/S1319157817303361*

Shafiq, D. A., Jhanjhi, N. and Abdullah, A. (2022). Load balancing techniques in cloud computing environment: A review, *Journal of King Saud University - Computer and Information Sciences* **34**(7): 3910–3933.
URL: *https://www.sciencedirect.com/science/article/pii/S131915782100046X*

Solaiman, K. and Adnan, M. A. (2020). Wlec: A not so cold architecture to mitigate cold start problem in serverless computing, *2020 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 144–153.

Verma, P., Goel, P. and Rani, N. (2024). A review: Cold start latency in serverless computing, *2024 Sixth International Conference on Computational Intelligence and Communication Technologies (CCICT)*, pp. 141–148.

Yoheswari, S. (2024). Optimization techniques for load balancing in data-intensive applications using multipath routing networks (1st edition), *Journal of Science Technology and Research (JSTAR)* **5**(1): 377–382.