A Comparative Analysis of Provisioned Concurrency Types for Cloud-Based Applications

Master of Computer Science

M.N.J Rathnayaka

University of Colombo School of Computing

2025

Declaration

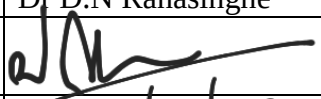
Name of the student: **M.N.J Rathnayaka**
Registration number: **22440488**
Name of the Degree Programme: **Master of Computer Science**
Project/Thesis title: **Provisioned Concurrency Types for Cloud-Based Applications**

1. The project/thesis is my original work and has not been submitted previously for a degree at this or any other University/Institute. To the best of my knowledge, it does not contain any material published or written by another person, except as acknowledged in the text.
2. I understand what plagiarism is, the various types of plagiarism, how to avoid it, what my resources are, who can help me if I am unsure about a research or plagiarism issue, as well as what the consequences are at University of Colombo School of Computing (UCSC) for plagiarism.
3. I understand that ignorance is not an excuse for plagiarism and that I am responsible for clarifying, asking questions and utilizing all available resources in order to educate myself and prevent myself from plagiarizing.
4. I am also aware of the dangers of using online plagiarism checkers and sites that offer essays for sale. I understand that if I use these resources, I am solely responsible for the consequences of my actions.
5. I assure that any work I submit with my name on it will reflect my own ideas and effort. I will properly cite all material that is not my own.
6. I understand that there is no acceptable excuse for committing plagiarism and that doing so is a violation of the Student Code of Conduct.

Signature of the Student	**Date (DD/MM/YYYY)**
	12/07/2025

Certified by Supervisor(s)

This is to certify that this project/thesis is based on the work of the above-mentioned student under my/our supervision. The thesis has been prepared according to the format stipulated and is of an acceptable standard.

	Supervisor 1	**Supervisor 2**	**Supervisor 3**
Name	Dr D.N Ranasinghe		
Signature			
Date	2025/7/13		

Abstract

The increasing adoption of serverless computing, particularly AWS Lambda, has introduced both performance benefits and challenges. A major concern is the cold start latency, which affects response times and scalability, particularly for time-sensitive applications. To mitigate this issue, AWS introduced provisioned concurrency, which ensures pre-initialized function instances to reduce startup delays. However, selecting the appropriate concurrency model static or dynamic (scheduled or auto-scaling) remains a challenge due to the trade-offs between performance and cost.

This dissertation presents a comparative analysis of provisioned concurrency types in AWS Lambda, evaluating their impact on various workload domains, including web applications, IoT backend services, and data processing systems. Real-world and simulated workloads were analyzed using statistical techniques, such as Mahalanobis distance calculations, to assess performance variations.

A simple rule-based guide was developed to assist in selecting the most suitable concurrency type based on application needs, balancing cost efficiency and performance. For IoT workloads, scheduled provisioned concurrency (Mahalanobis distance: 10.80; cost: $0.0888/1000 invocations) achieved optimal balance, reducing latency by 8.7% over static provisioning. E-commerce applications benefited most from static provisioning (Mahalanobis distance: 9.00; cost: $0.0054/1000 invocations), ensuring low latency during traffic spikes. Data processing systems favored scheduled concurrency (Mahalanobis distance: 3.009; cost: $3.576/1000 invocations) for predictable batch jobs. Cost-Distance Ratio (CDR) analysis further validated these recommendations, prioritizing strategies with CDR < 1. The study provides empirical guidelines to optimize AWS Lambda deployments, recommending scheduled concurrency for IoT, static for e-commerce, and scheduled for data processing, ensuring cost-effective performance across domains.

Table of Contents

List of Figures

List of Tables

List of Acronyms

AWS - Amazon Web Services

IoT – Internet of Things

VM – Virtual Machine

IaaS – Infrastructure as a Service

PaaS – Platform as a Service

ML – Machine Learning

Chapter 1- Introduction

Cloud computing is a rapidly evolving field that has transformed how organizations manage infrastructure and deploy applications. Before the advent of serverless computing, the development of cloud services passed through several key phases, including Traditional Servers (Pre-Cloud Era), Virtual Machines (VMs), Infrastructure as a Service (IaaS), and Platform as a Service (PaaS). Each of these phases presented distinct challenges for developers, particularly regarding infrastructure management, cost efficiency, scalability, and development speed.

The emergence of serverless computing sought to address these issues directly. Serverless computing, a cloud-native model, enables developers to build and run applications without the need for server management, offering a streamlined approach to infrastructure. Amazon Web Services (AWS) introduced AWS Lambda in 2014[1], a pivotal moment that marked the beginning of modern serverless computing. In the serverless model, developers benefit from three core principles: no infrastructure management, automatic scaling, and a pay-per-use billing model.

Despite its advantages, *serverless* computing has introduced specific technical challenges, notably the issue of *cold starts* [2]. A cold start occurs when a serverless function is triggered for the first time or after a period of inactivity, requiring the platform to initialize the execution environment before processing the request. This initialization can introduce latency, which is especially problematic for applications that require consistent low-latency responses, such as real-time data processing or user-facing APIs.

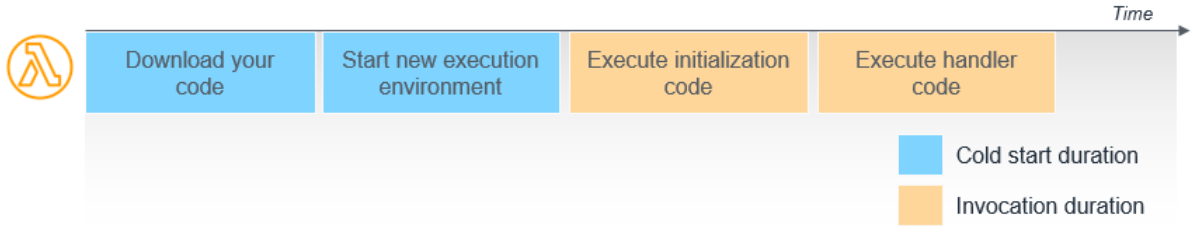


Figure 1 Cold Start Execution

In Figure 1 Lambda function exaction steps. Which is problematic for applications that demand consistent low-latency responses, such as real-time data processing or user-facing APIs. Cold starts become more pronounced for functions with large dependencies or complex initialization procedures, where the setup time can vary significantly based on the workload. Consequently, the unpredictability of cold start latency has been a primary drawback in adopting serverless computing for performance-sensitive applications.

Figure 1 Cold Start Execution: This figure illustrates the stages of a cold start in AWS Lambda functions, highlighting the time required to download code, start a new execution environment, execute initialization code, and finally execute the handler code.

As shown in Figure 1, the cold start process involves several stages, each of which adds to the overall latency. This latency is particularly pronounced for functions with large dependencies or complex initialization procedures, where the setup time can vary significantly depending on the workload. Consequently, the unpredictability of cold start latency has been a significant drawback to adopting serverless computing for performance-sensitive applications.

In 2019, AWS introduced **Provisioned Concurrency** for Lambda to address the challenge of cold starts. Provisioned Concurrency allows developers to keep functions initialized and ready to respond instantly, minimizing startup delays for latency-sensitive applications. By allocating pre-initialized instances of a Lambda function, Provisioned Concurrency ensures that the function is always ready to handle incoming requests with minimal delay, thereby reducing the impact of cold starts. This feature is particularly useful for applications with predictable traffic patterns or where low latency is a critical requirement, such as e-commerce websites, financial services, or interactive applications. At my workplace, where it build and deploy various applications using AWS Lambda, It frequently encounter issues related to cold starts.

This research aims to analyze AWS Lambda's provisioned concurrency options static and dynamic (scheduled profile and auto-scaling based on utilization) to determine which strategy is most effective for real-world applications. The primary challenge is the lack of comprehensive analysis and guidelines for selecting the appropriate provisioned concurrency

type, tailored to specific application usage patterns, with the ultimate goal of optimizing application performance and minimizing costs.

1.1 Motivation

The motivation behind this research stems from the increasing adoption of serverless computing platforms like AWS Lambda, one of the latest service models of cloud computing [3] which offers significant advantages in terms of flexibility, scalability, and cost. However, the persistent issue of cold starts can severely impact the performance of time-sensitive applications, leading to poor user experiences and delayed responses. Furthermore, the provisioned concurrency feature, while solving some of these performance issues, introduces new complexities in cost management and scalability optimization.

The need to reduce operational costs while maintaining high performance has become a critical business requirement. With serverless computing rapidly evolving, a clear understanding of how to balance the benefits and trade-offs of static and dynamic provisioned concurrency is essential to make informed architectural decisions. This research seeks to provide a practical guide that will not only address current gaps in understanding but also empower organizations to maximize the potential of AWS Lambda while minimizing unnecessary expenses.

The findings from this research can lead to improved resource allocation, better cost management, and higher application responsiveness, thereby benefiting both developers and end-users in diverse fields.

1.2 Research Questions

1. How do static and dynamic provisioned concurrency models impact cold start latency, response times, and scalability in AWS Lambda functions for various application domains?
2. How can a rule-based prediction system guide developers in selecting the appropriate concurrency type for serverless applications, based on performance and cost-efficiency metrics?

3. What are the trade-offs between static and dynamic provisioned concurrency in terms of optimizing performance versus cost across different application workloads?

1.3 Project Aim and Objectives

1.3.1 Aims

To conduct a comprehensive comparative analysis of static and dynamic provisioned concurrency in AWS Lambda, focusing on optimizing performance, reducing latency, and managing costs across diverse application domains such as web applications, data processing systems, and IoT backend services.

1.3.2 Objectives

- To perform a comparative analysis of static provisioned concurrency and dynamic provisioned concurrency (scheduled profile and auto-scaling based on utilization) in AWS lambda, drawing from direct workplace challenges.
- To analyze the impact of each concurrency type of lambda function performance, including cold start times and ability to handle varying loads, with an emphasis data collection.
- To develop a simple rule-based prediction tool for selecting the appropriate concurrency type for different real-world applications, incorporating cost as a crucial factor in the decision-making process.
- Development of a planning tool for software developers, capable of estimating initial concurrency settings for different real-world applications and refining them based on performance data collected in real time.

1.4 Scope

1.4.1 In Scope

1. **Concurrency Models**

   The study will focus exclusively on static and dynamic provisioned concurrency types within AWS Lambda, examining their effects on cold start latency, response times, scalability, and cost efficiency.

2. **Application Domains**
    - **Web Applications**: The research will evaluate provisioned concurrency for applications requiring high scalability and low latency, such as e-commerce platforms experiencing peak traffic.
    - **Data Processing Systems**: The analysis will include applications that handle real-time analytics and data processing, focusing on throughput and processing time.
    - **IoT Backend Services**: The study will assess the suitability of concurrency models for IoT applications that demand scalable infrastructure to manage high-frequency data.
3. **Performance Metrics**

    Key metrics, including cold start latency, execution latency, response time, throughput, and scalability, will be measured to understand each concurrency model's performance implications.

4. **Cost Evaluation**

    The research will conduct a detailed cost analysis comparing the expenses associated with maintaining static versus dynamic provisioned concurrency and potential savings with dynamic scaling based on real-time demand.
5. **Prediction System Development**

    A rule-based prediction system will be developed to recommend the appropriate concurrency type based on application characteristics, optimizing for both cost and performance.

1.4.2 Out of Scope

1. **Comparison with Other Cloud Providers**

    The study will exclusively focus on AWS Lambda's provisioned concurrency models and will not consider other serverless offerings from cloud providers like Azure Functions or Google Cloud Functions.

2. **Non-Serverless Architectures**

Traditional, non-serverless architectures or other cloud service models that do not use a serverless approach will not be part of the analysis.

3. **Provisioned Concurrency Outside of AWS Lambda**

   The research will not address provisioned concurrency implementations or optimizations outside of AWS Lambda, even within AWS itself, such as ECS or Fargate.

1.5 Thesis Outline

This thesis is organized into six chapters, each contributing to a comprehensive exploration of provisioned concurrency types for cloud-based applications. The structure is as follows:

Chapter 1: Introduction

This chapter introduces the research problem, highlighting the motivation behind the study, the research questions, and the project aims and objectives. It also defines the scope of the research and provides an overview of the thesis structure.

Chapter 2: Literature Review

This chapter presents a detailed review of existing theories and related work on provisioned concurrency in AWS Lambda. It discusses the challenges of cold start latency, application-specific requirements, cost implications, and gaps in current research. Additionally, it explores rule-based systems and other optimization strategies for serverless computing.

Chapter 3: Methodology

This chapter outlines the research philosophy and methodology used in the study. It describes the experimental design, data acquisition methods, simulation setups, and tools employed for performance monitoring and cost analysis. The chapter emphasizes the pragmatic approach taken to ensure practical and actionable insights.

Chapter 4: Implementation and Results

This chapter details the implementation of the experiments and presents the results of the comparative analysis of static and dynamic provisioned concurrency types. It includes Mahalanobis Distance calculations, cost comparisons, and evaluations across three workload domains: IoT backend services, e-commerce applications, and data processing systems.

Chapter 5: Evaluation

This chapter evaluates the findings by analyzing the trade-offs between performance and cost for different concurrency models. It introduces a new metric for comparing cost and distance, providing recommendations for selecting the most suitable concurrency type based on specific application needs. The chapter also develops a rule-based system to guide developers in making informed decisions.

Chapter 6: Conclusion and Future Work

This final chapter summarizes the key conclusions drawn from the research, addressing the research questions and objectives. It highlights the contributions of the study and discusses its limitations. Furthermore, it outlines potential directions for future research to extend the current framework and address emerging challenges in serverless computing.

Chapter 2 - Literature Review

This section contains the theoretical foundation and related work that this research is based upon.

2.1 Theories

2.1.1 Provisioned Concurrency in AWS Lambda

AWS Lambda, Amazon's serverless computing service, encounters what is known as **cold start** latency, a delay occurring when functions are invoked after a period of inactivity. This latency arises because, in traditional on-demand Lambda functions, the runtime environment and dependencies must initialize before the function can handle requests. This can be problematic for latency-sensitive applications, such as web APIs, real-time data processing, and IoT services, where rapid response times are essential.

To mitigate this issue, 2019 AWS introduced **Provisioned Concurrency**, a feature designed to eliminate cold start latency by pre-warming Lambda instances. Unlike the on-demand model, where instances initialize only upon request arrival, provisioned concurrency maintains a predetermined number of **warm** instances ready to process requests immediately. This approach ensures consistent, low-latency performance crucial for applications with predictable or steady traffic. [4]

Types of Provisioned Concurrency Configurations

Provisioned Concurrency in AWS Lambda can be managed through two primary configurations:

1. **Static Provisioned Concurrency**: This involves setting a fixed number of pre-warmed instances, ideal for applications with stable, predictable workloads. For example, an e-commerce site anticipating high traffic during holiday sales events can use static provisioning to minimize latency during peak periods.
2. **Dynamic Provisioned Concurrency**: AWS allows for dynamically scaling provisioned concurrency through two strategies:

- **Scheduled profile**: This approach pre-warms instances according to a schedule based on expected traffic patterns, such as office hours for enterprise applications. Scheduled scaling reduces latency during expected peaks while optimizing resources during off-peak periods.
- **Autoscaling based on utilization**: Target tracking leverages real-time

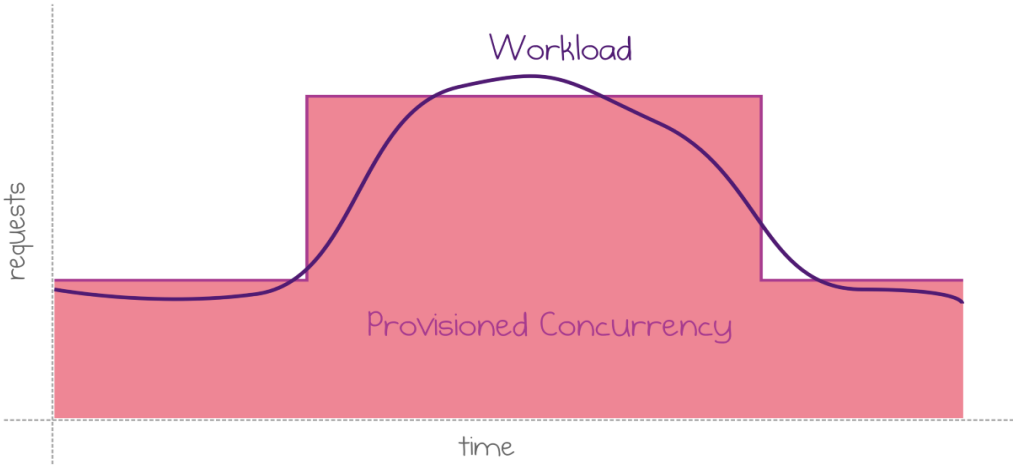


Figure 2 Scheduled profile

performance metrics to automatically adjust concurrency based on traffic. For instance, a web service may use target tracking to ensure latency remains below a specific threshold.

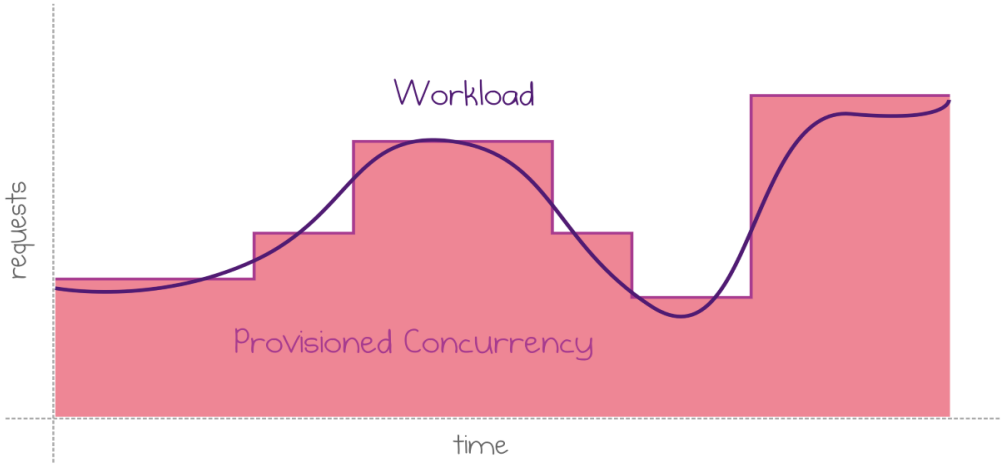


Figure 3 Auto scaling based on utilization

Despite added costs associated with maintaining pre-warmed instances, provisioned concurrency allows developers to optimize the balance between performance and cost based on application requirements.

2.2 Related Work

2.2.1 Cold Start Latency in Serverless Environments

The advent of serverless computing has revolutionized cloud application development, with AWS Lambda leading since 2014 [1]. One of the primary challenges in serverless architectures is cold start latency, which occurs when a function is invoked after a period of inactivity, requiring initialization before execution [5][6]. This latency significantly impacts latency-sensitive applications such as real-time analytics and IoT systems.

Wang and Huang (2021) evaluated cold start mitigation techniques, including provisioned concurrency and container reuse, finding that provisioned concurrency notably improves response times for time-critical applications [11]. Similarly, researchers in 2024 [13] explored real-world applications to uncover current challenges, future trends, and best practices in serverless computing.

Notably, other cloud providers, such as Azure Functions and Google Cloud Functions, have adopted varying strategies, including predictive scaling and resource pre-allocation, to address similar challenges [15]. Comparative studies across platforms underscore the universal importance of minimizing cold starts for optimal serverless function performance.

2.2.2 Application-Specific Provisioned Concurrency Requirements

Static provisioned concurrency involves pre-configuring a set number of function instances to handle expected demand, ensuring quick response times without the initial delay. In contrast, dynamic provisioned concurrency adjusts the number of pre-warmed instances in real time based on actual demand. Jo, Pack, and Leung [8] highlighted the

application of advanced scheduling algorithms to dynamically manage these resources in scenarios like industrial IoT, demonstrating the potential for significant improvements in operational efficiency and cost reduction.

The phenomenon of cold start latency remains a critical challenge, especially for latency-sensitive applications. Cold starts occur when serverless platforms like AWS Lambda scale to zero during idle periods and need to initialize resources to process incoming requests, leading to delayed response times. This latency is particularly detrimental to real-time systems and high-throughput applications. Golec et al. provided a comprehensive taxonomy categorizing existing solutions into three broad areas [10]:

- **Caching mechanisms**: Minimize initialization delays by pre-loading dependencies.
- **Application-level optimizations**: Modify runtime parameters or utilize lightweight containers to reduce startup overhead.
- **AI/ML-driven solutions**: Use predictive analytics to pre-warm containers based on workload patterns.

Kosuru (2023) discussed the use of provisioned concurrency for functions with known traffic patterns to mitigate cold start problems in latency-sensitive applications [12]. The analysis of real-world use cases demonstrated significant benefits:

- **E-commerce** firms have used Lambda to manage fluctuating traffic, optimizing costs by scaling on demand.
- **IoT applications** benefit from Lambda's ability to process real-time data streams.

Practical research conducted by Schmutz et al. (2022) demonstrated the handling of data-intensive tasks such as file uploads and real-time property listing management [15]. To ensure applications remain responsive, especially during peak usage periods when multiple users simultaneously upload documents or images, it is crucial to understand the effects of provisioned concurrency and cold starts on performance. These comparative analyses provide a broader perspective on how serverless computing challenges are addressed across different cloud platforms.

2.2.3 Cost Implications of Serverless Computing

Adzic and Chatley (2017) conducted a detailed cost analysis of serverless applications compared to traditional server-based architectures [12]. Their study examined the cost implications of different deployment models and identified scenarios where serverless computing offers significant cost advantages. Eivy (2017) provides a critical examination of the economics of serverless computing, cautioning that while serverless can offer cost benefits, it may not always be the most economical choice depending on workload characteristics [16].

2.2.4 Related Work on Rule-Based Systems for Serverless Optimization

While the serverless model simplifies application deployment, it also brings unique challenges in observability. Logging is a critical part of observability, enabling developers to monitor application health, debug issues, and optimize performance. Best practices and techniques for optimizing AWS Lambda logging have been explored to enhance observability [7].

Pervious (2024) explores the performance of AWS Lambda when using RESTful APIs and conducting load testing in the study titled *Concurrent Scaling: Evaluating AWS Lambda Performance* [9]. The research specifically addresses the challenges of static provisioned concurrency during load testing and offers practical guidelines for developers to optimize performance.

Shahrad et al. (2019) analyzed real-world serverless workloads and suggested optimization strategies to improve performance, including workload-aware resource allocation and efficient container reuse [17]. Their findings underscore the importance of understanding application-specific requirements for provisioned concurrency.

Kumari, Sahoo, and Behera (2023) proposed a workflow-aware analytical model designed to predict the performance and cost of serverless execution on platforms like AWS Lambda [13]. Their methodology involves using historical execution data to train the model, which can then predict response times and costs with high accuracy. This predictive capability aids in planning and optimizing resource allocation.

Dantas and Khazaei (2022) proposed guidelines for Python, Node.js, and Java serverless functions under different memory configurations and application sizes, determining which deployment type presents the best cold start performance [14].

2.2.5 Limitations and Gaps in Current Research

While existing research provides valuable insights into the benefits and challenges of provisioned concurrency, gaps remain in comprehensive analyses across varied real-world applications. Most studies focus on specific scenarios or theoretical models, with limited exploration of comparative analyses between static and dynamic (scheduled profiles and autoscaling based on utilization) types across diverse operational environments. This indicates a significant need for empirical studies that evaluate these concurrency models side by side in practical deployments, to better guide developers and architects in making informed decisions tailored to specific application requirements.

Need for a Unified Framework:

The absence of a unified framework or guideline for selecting appropriate concurrency types based on specific application characteristics remains a significant gap in the literature. Baldini et al. (2017) highlight the necessity for addressing open problems in serverless computing, including the development of frameworks that aid in decision-making for deployment strategies [6b]. Zheng and Zhou (2018) discuss the promises and challenges of serverless computing, emphasizing the importance of developing tools and models to assist in optimizing serverless deployments [19].

This research aims to fill this gap by developing a rule-based prediction system that can provide practical guidelines for optimizing serverless computing environments.

Chapter 3 - Methodology

3.1 Research Philosophy

This research adopts a pragmatic philosophy, emphasizing actionable outcomes through mixed-method analysis [20] (Creswell & Creswell, 2018). Pragmatism aligns with the objective of optimizing AWS Lambda's provisioned concurrency types by addressing real-world scenarios. This approach ensures a balanced evaluation of performance and cost-efficiency, emphasizing solutions that are both effective and applicable in practice.

3.2 Research Methodology

To achieve this, the study employs a **comparative analysis methodology**, investigating the effectiveness of static and dynamic provisioned concurrency in AWS Lambda across different real-world application scenarios. The study integrates both real-world data and simulated workloads to conduct a comprehensive analysis, quantifying performance, cost-efficiency, and operational impact.

Data Acquisition

Data for this study will be acquired through a combination of real-world data collection and controlled simulations:

1. **Utilization of Real-World Data**:

   Static Provisioned Concurrency for IoT Backend Services: Real-world data will be sourced from the researcher's workplace, where IoT backend services are deployed using AWS Lambda with static provisioned concurrency. This data includes actual IoT workloads, such as sensor data ingestion rates, processing frequencies, and backend response times. Proper permissions have been obtained from the workplace to use this data, ensuring compliance with data privacy and ethical guidelines.

2. **Simulation of Real-World Application Workloads**:

   To simulate realistic scenarios for dynamic provisioned concurrency and other application types, the following simulations will be created.

a. Dynamic Provisioned Concurrency for IoT Backend Services:

- **Simulation Design**: A simulation model will be created to replicate the real-world IoT backend service loads using dynamic provisioned concurrency. The model will use real-world IoT data patterns to simulate the auto-scaling capabilities of AWS Lambda. The simulation will adjust the number of pre-warmed instances based on real-time demand, mimicking AWS's built-in auto-scaling features.

b. Simulations for Other Application Types:

- **Data Processing Systems**: Simulated workloads will be created to represent data processing tasks, such as batch processing, real-time analytics, and machine learning inference workloads. These simulations will mimic typical data volume, processing time, and frequency found in real-world applications. The simulation will implement both static and dynamic provisioned concurrency configurations to compare performance metrics such as processing latency, throughput, and cost.
- **Web Applications**: Simulated workloads will be designed to represent typical web application usage, including scenarios for e-commerce platforms (handling peak traffic loads). These simulations will also implement both static and dynamic provisioned concurrency.

Monitoring and Logging:

Performance Metrics Monitoring: AWS Cloud Watch will be utilized to monitor and log critical performance metrics, such as cold start latency, execution latency, throughput, and cost associated with each concurrency strategy. This monitoring will provide the necessary data for comparative analysis.

Cost Implications Analysis: AWS Cost Explorer and AWS Billing Dashboard will be used to track and analyze the financial implications of each concurrency setup, providing insights into the cost-efficiency of static versus dynamic provisioned concurrency.

Data Analysis

Data collected from the experiments will be analyzed using statistical software. The analysis will compare performance and cost-efficiency between statically and dynamically provisioned concurrency setups.

3.3 Simulation Setup for Real-World Scenarios

To evaluate the impact of provisioned concurrency types on AWS Lambda, simulations were designed to replicate three real-world application domains: IoT backend services**,** e-commerce applications**,** and data processing systems. Each simulation was conducted in a controlled AWS environment, with identical memory allocation (1 GB), runtime (Node.js 18.x), and AWS region (us-east-1) to isolate concurrency type as the sole variable.

1. IoT Backend Service:

Objective: Simulate periodic, low-latency data ingestion from IoT sensors

Workload Characteristics

- **Traffic Pattern**: Bursty requests every 15 minutes to mimic sensor reporting intervals.
- **Payload Size**: 2 KB (simulated JSON sensor data).

Simulation Tool:

- **AWS Lambda Function**: Generated mock sensor data and triggered Lambda functions.

Lambda Configuration:

- **Without Provisioned Concurrency**: Default on-demand execution.
- **Static Provisioned Concurrency**: Pre-warmed 3 instances.
- **Dynamic Provisioned Concurrency**:
    - **Scheduled**: Pre-warmed 10 instances during 9 AM– 5 PM (simulating active sensor hours).
    - **Auto-Scaling**: Target tracking on 80% utilization.

2. E-Commerce Application

Objective: Simulate high-traffic e-commerce platforms with unpredictable traffic spikes

Workload Characteristics:

- **Traffic Pattern**: Spiky traffic with 5x baseline load during simulated "sale periods
- **Payload Size**: 10 KB (simulated product details and user checkout data).

Simulation Tool:

- **Locust.io**: Open-source load testing tool to generate HTTP requests mimicking user interactions.

Lambda Configuration:

- **Without Provisioned Concurrency**: On-demand scaling.
- **Static Provisioned Concurrency**: 3 pre-warmed instances.
- **Dynamic Provisioned Concurrency**:
    - **Scheduled**: Pre-warmed 10 instances during sale periods (9 AM– 5 PM).
    - **Auto-Scaling**: Target tracking on 80% utilization.

3. Data Processing Systems

Objective: Simulate batch processing jobs for real-time analytics.

Workload Characteristics:

- **Traffic Pattern**: Predictable, high-volume batch jobs executed twice daily (9 AM and 5 PM).
- **Payload Size**: 10 MB (simulated JSON datasets).

Simulation Tools:

- **AWS Lambda Function**: Triggered Lambda functions.

Lambda Configuration:

- **Without Provisioned Concurrency**: Default execution.
- **Static Provisioned Concurrency**: 3 pre-warmed instances.
- **Dynamic Provisioned Concurrency**:
    - **Scheduled**: Pre-warmed 10 instances during 9 AM– 5 PM.
    - **Auto-Scaling**: Target tracking on 80% utilization.

Chapter 4 - Implementation and Results

The research employs a **Mixed Method Research design**, combining qualitative insights from literature reviews with quantitative metrics derived from AWS Lambda experiments. This approach ensures a comprehensive understanding of the effectiveness of static and dynamic provisioned concurrency.

To better address the research questions, the initial research design was refined and optimized, ensuring alignment with the research objectives. A high-level diagram illustrating the research design is presented in **Figure 5**.

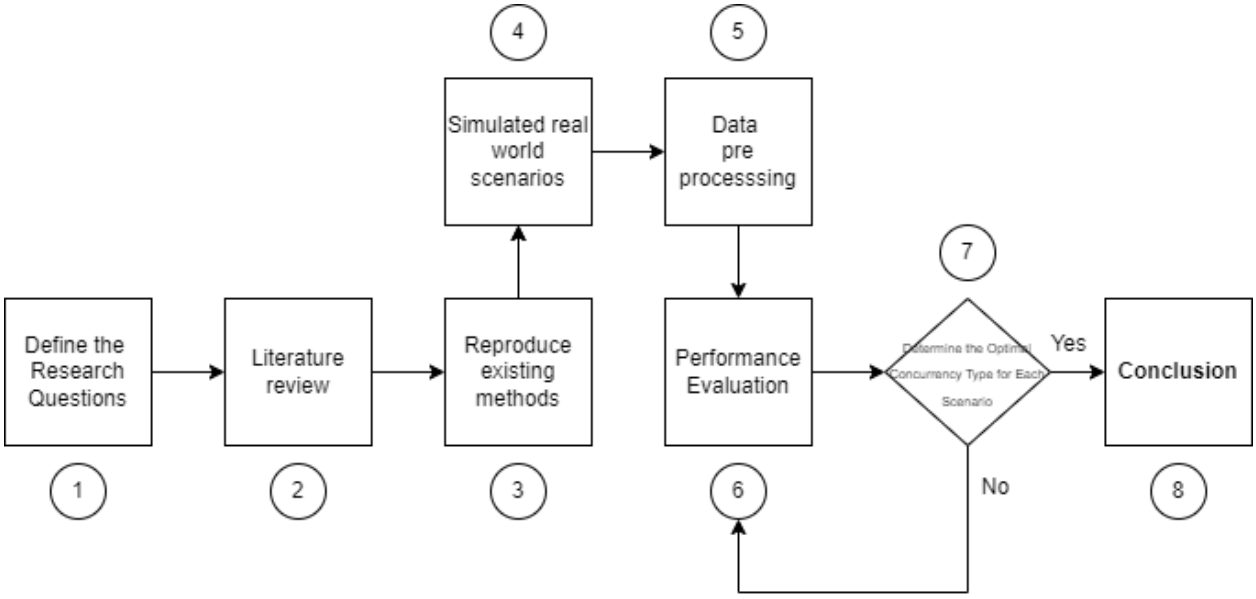


Figure 4 Research Design

4.1 Research Design and Methodology

Step 1 – Define the research questions

This research aims to address the following key questions:

1. What are the previous methods used to reduce cold start latency in AWS Lambda functions?
2. How does a comparative analysis of static and dynamic provisioned concurrency types contribute to optimizing cold start performance?

To achieve these objectives, a comprehensive literature review of existing methods will be conducted. This will be detailed in Step 2.

Step 2 – Literature review

In this step, a comprehensive literature review will be conducted to identify previously used methods, analyzing their strengths and weaknesses. The review will cover key aspects such as data collection techniques, data types, theories related to Lambda functions, and their evaluation. It will also explore real-world applications utilizing provisioned concurrency in Lambda functions. Additionally, this step will provide a thorough understanding of the research's background and context, serving as a critical foundation for the study.

Step 3 – Reproducing existing methods

To gain insights into prior research and methodologies, existing methods will be reproduced. As detailed in *Concurrent Scaling: Evaluating AWS Lambda Performance* [9], the study involves recreating load balancing using static provisioned concurrency as presented in previous work. The results will be compared against scenarios without provisioned concurrency. **Figure 6** illustrates the relationship between cold start latency and time periods, providing a visual comparison of performance with and without provisioned concurrency.

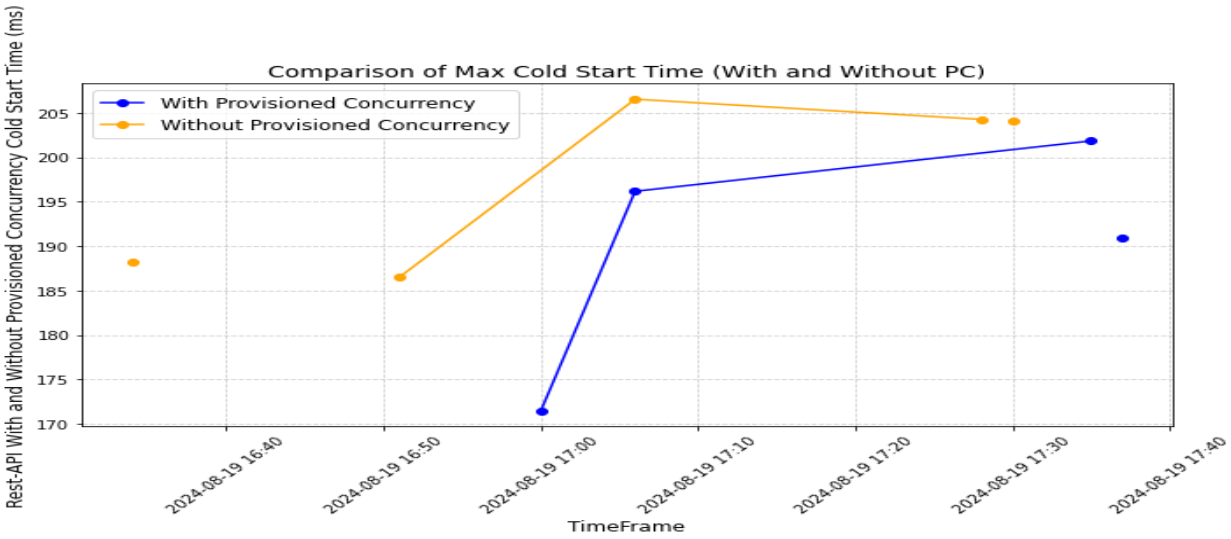


Figure 5 Rest-API With and Without Provisioned Concurrency Cold Start Time

Step 4 – Simulated real world scenarios

This study involves the collection of datasets through simulations representing three real-world application domains: IoT applications, data pipelines, and e-commerce platforms. Each scenario will be executed under identical environmental conditions, varying only the type of provisioned concurrency applied. The concurrency configurations include the following:

- **Without Provisioned Concurrency**: Applications operate without any pre-initialized instances, experiencing the default behavior of AWS Lambda.
- **Static Provisioned Concurrency**: A fixed number of pre-warmed Lambda instances are maintained to handle predictable workloads.
- **Dynamic Provisioned Concurrency**: The number of pre-warmed instances dynamically adjusts based on real-time demand, utilizing strategies such as scheduled scaling and auto-scaling based on utilization.

By standardizing the simulation environment and isolating concurrency type as the only variable, the study aims to deliver an accurate comparative analysis of their performance, scalability, and cost implications across diverse application scenarios.

Step 5 – Data preprocessing

Collect the Cold Start duration dataset using **Amazon Cloud Watch**. Utilize **Cloud Watch Insights** [21] to filter and analyze Cloud Watch data. As shown in **Figure 7**, provided by AWS, this step uses the REPORT logs emitted after the execution of each Lambda function. These logs contain detailed metrics, including invocation counts, cold start occurrences, memory usage, and execution durations, which will be critical for further analysis.

The filtering and statistical computations are conducted using the query provided above. This query calculates key metrics such as:

- **Cold Start percentage** based on initialization duration.
- **Maximum and average Cold Start durations.**

- **Memory usage analysis**, including the percentage of memory utilized compared to the allocated size.

By preprocessing the dataset using these logs, it ensure that the data is cleaned, organized, and ready for subsequent analysis.

```
filter @type = “REPORT”
| stats
 count(@type) as countInvocations ,
 count(@initDuration) as countColdStarts ,
(count(@initDuration)/count(@type))*100 as percentageColdStarts,
 max(@initDuration) as maxColdStartTime,
 avg(@duration) as averageDuration,
 max(@duration) as maxDuration,
 min(@duration) as minDuration,
 avg(@maxMemoryUsed) as averageMemoryUsed,
 max(@memorySize) as memoryAllocated,
(avg(@maxMemoryUsed)/max(@memorySize))*100 as percentageMemoryUsed
by bin(1h) as timeFrame
```

Figure 6 Cloud watch Insights function

Step 6 - Performance Evaluation

This step involves analyzing the processed data to evaluate key performance metrics, such as cold start duration, latency, and cost efficiency. The goal is to address the research questions by assessing the effectiveness of various concurrency strategies and their impact on performance.

Step 7 - Determine the Optimal Concurrency Type for Each Scenario

In this step, scenarios are compared to evaluate the effectiveness of provisioned concurrency versus no provisioned concurrency. The goal is to identify the concurrency type that performs best for each specific scenario.

If the evaluation determines the optimal concurrency type ("Yes"), the process proceeds to the next step. Otherwise, Step 6 (Performance Evaluation) is revisited to refine the analysis and gather more insights.

Step 8 – Conclusion

Based on the results of Step 7, developer guidelines are formulated to recommend the most suitable concurrency type whether provisioned concurrency or no provisioned concurrency for real-world scenarios. These guidelines aim to provide actionable insights and best practices tailored to specific use cases.

4.2 Experimental Results

The simulation of three distinct scenarios web applications, IoT backend services, and data processing systems was conducted to evaluate the performance and cost efficiency of AWS Lambda's provisioned concurrency types. The data was collected over a period of one month, capturing variations in workload, latency under both static and dynamic provisioned concurrency configurations.

4.2.1 IoT Application Scenario

The experiment focused on a Node.js-based IoT application deployed in a consistent environment with four AWS Lambda functions. The IoT application scenario was designed to capture data received at 15-minute intervals. Initial observations indicated a slight increase in response time variability, likely attributable to the unpredictable nature of IoT workloads.

Data for this scenario was collected and saved as a CSV file using AWS Cloud Watch Insights for analysis. **Figure 8** illustrates the dataset without the application of provisioned concurrency, highlighting performance metrics such as invocation counts, cold start occurrences, and memory usage.

	A	B	C	D	E	F	G	H	I	J	K	L
1	timeFrame	countInvocations	countColdStarts	percentageColdStarts	maxColdStartTime	averageDuration	maxDuration	minDuration	averageMemoryUsed	memoryAllocated	percentageMemoryUsed	
2	00:00.0	194	2	1.0309	180.21	1.4524	5.29	1.16	69721649.48	512000000	13.6175	
3	00:00.0	294	3	1.0204	179.67	1.4766	6.7	1.13	69727891.16	512000000	13.6187	
4	00:00.0	294	3	1.0204	184.11	1.5392	21.93	1.1	69768707.48	512000000	13.6267	
5	00:00.0	291	3	1.0309	191.35	1.5211	7.67	1.08	69731958.76	512000000	13.6195	
6	00:00.0	295	3	1.0169	182.25	1.463	7.12	1.13	69749152.54	512000000	13.6229	
7	00:00.0	292	3	1.0274	204.74	1.5209	19.63	1.07	69743150.68	512000000	13.6217	
8	00:00.0	295	3	1.0169	207.27	1.6574	18.69	1.14	69718644.07	512000000	13.6169	
9	00:00.0	285	3	1.0526	214.38	1.4946	6.45	1.07	69715789.47	512000000	13.6164	
10	00:00.0	296	3	1.0135	207.35	1.5803	9.27	1.11	69743243.24	512000000	13.6217	
11	00:00.0	290	3	1.0345	200.61	1.5189	5.41	1.14	69751724.14	512000000	13.6234	
12	00:00.0	294	3	1.0204	192.93	1.3701	9.12	0.75	69724489.8	512000000	13.6181	
13	00:00.0	295	3	1.0169	182.47	1.4965	8.04	1.1	69779661.02	512000000	13.6288	
14	00:00.0	294	3	1.0204	180.99	1.3795	4.54	1	69761904.76	512000000	13.6254	

Figure 7 Without provisioned concurrency dataset

Collect data to illustrate the relationship between cold start latency and time periods, providing a visual comparison of performance with and without provisioned concurrency. Figures 8, 9, 10, and 11 present metrics charts on this topic.

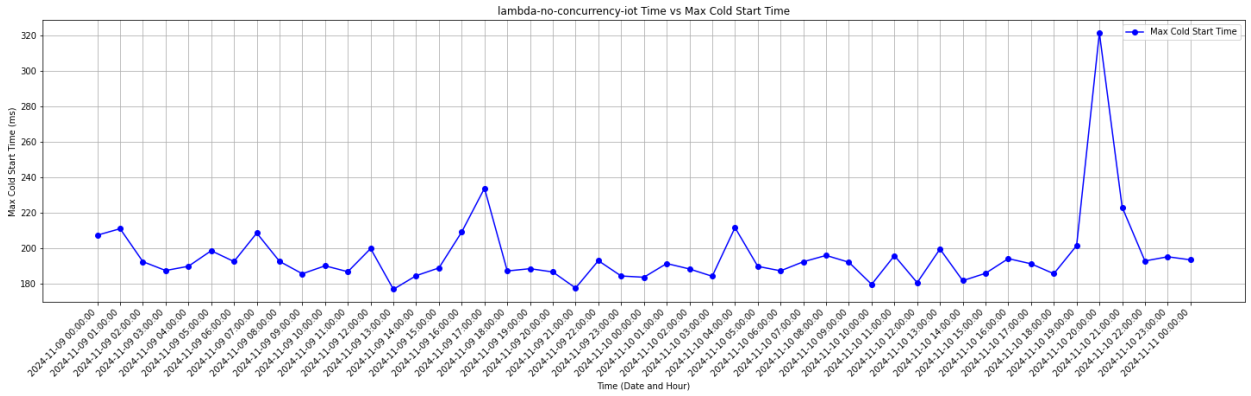


Figure 8 Cold Start Performance of IoT Workloads without Provisioned Concurrency

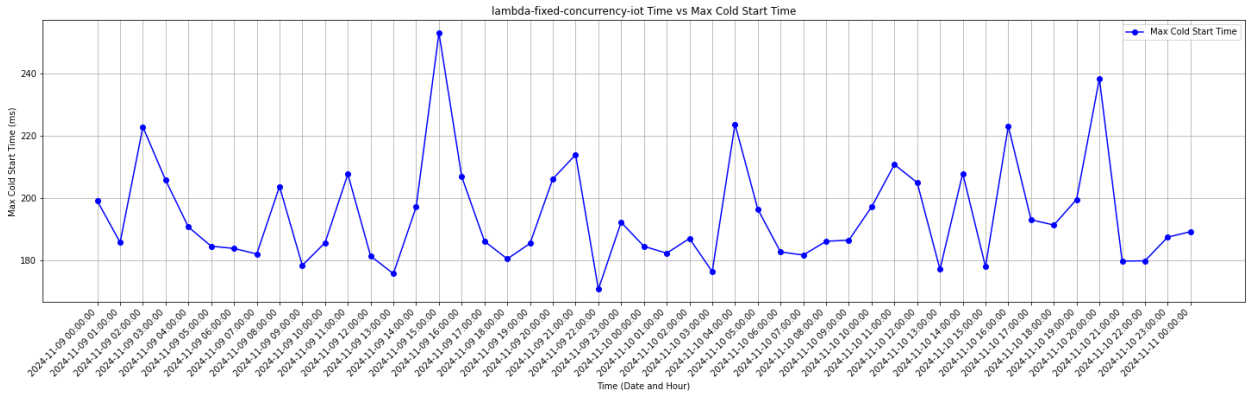


Figure 9 Cold Start Performance of IoT Workloads with Static Provisioned Concurrency

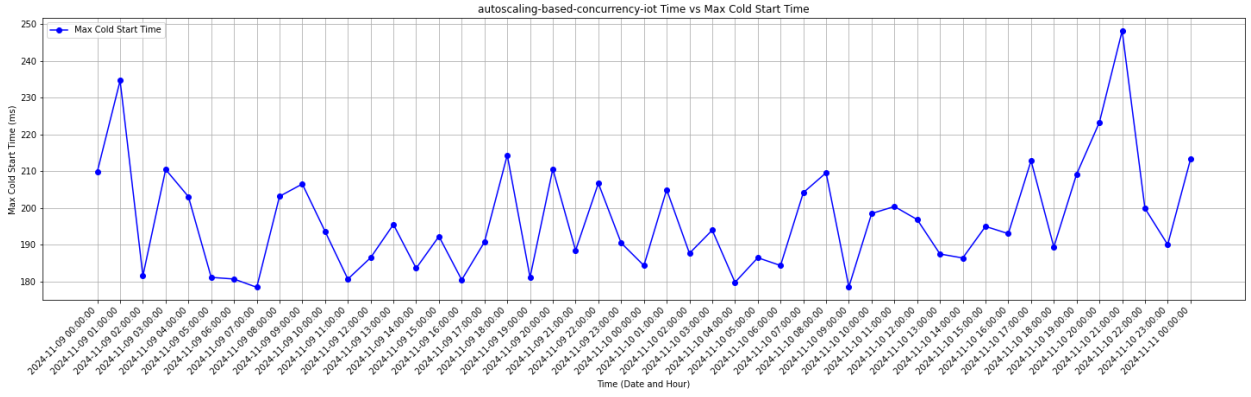


Figure 10 Cold Start Performance of IoT Workloads with Auto-Scaling Provisioned Concurrency

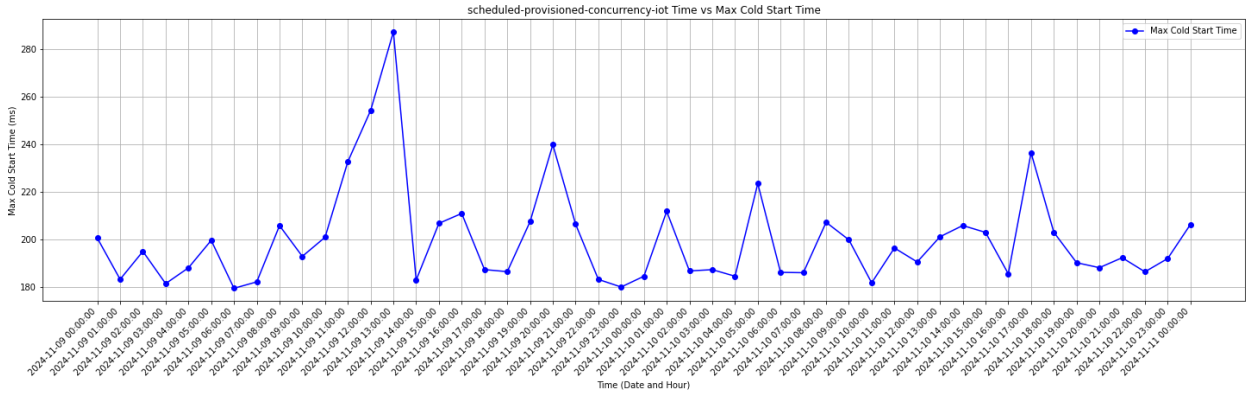


Figure 11 Cold Start Performance of IoT Workloads with Scheduled Provisioned Concurrency

4.2.2 E-Commerce Application Scenario

A simple Node.js Lambda function was simulated using the Locust application to generate requests and save the results in a CSV file. However, the simulation revealed slightly higher operational costs during peak traffic, primarily due to the frequent scaling events associated with dynamic provisioning.

Collect data to illustrate the relationship between cold start latency and time periods, providing a visual comparison of performance with and without provisioned concurrency. Figures 12, 13, 14, and 15 present metrics charts on this topic.

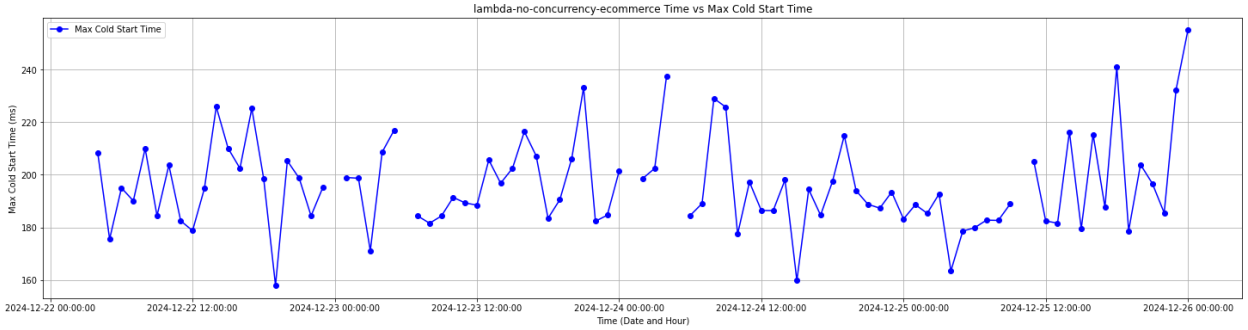


Figure 12 Cold Start Performance of E-commerce Workloads without Provisioned Concurrency

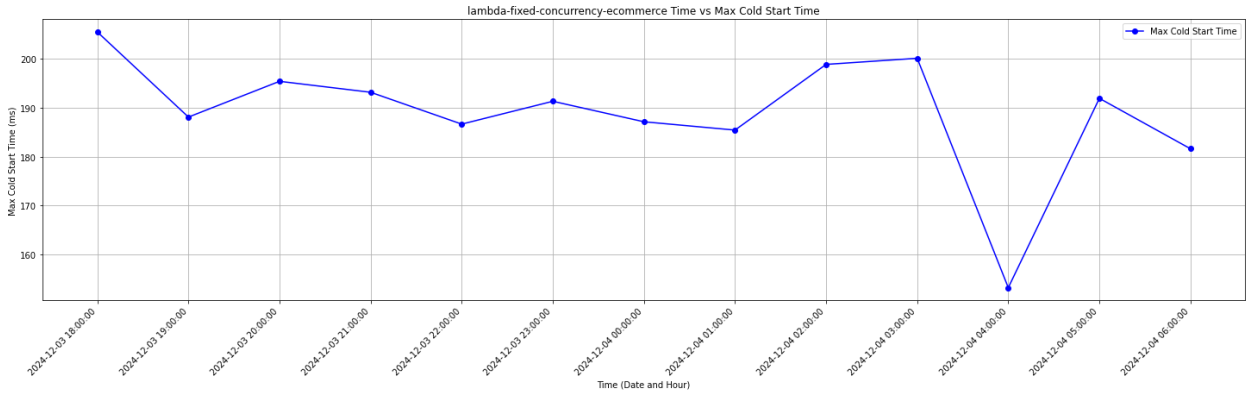


Figure 13 Cold Start Performance of E-commerce Workloads with Static Provisioned Concurrency

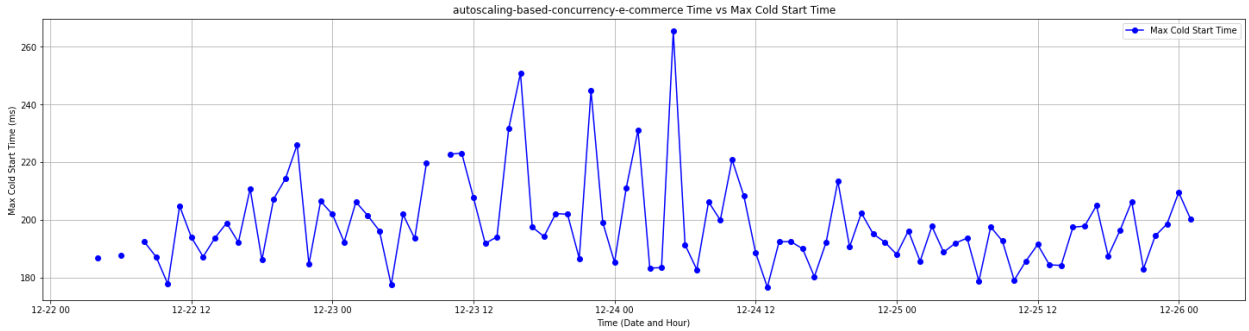


Figure 14 Cold Start Performance of E-commerce Workloads with Auto-Scaling Provisioned Concurrency

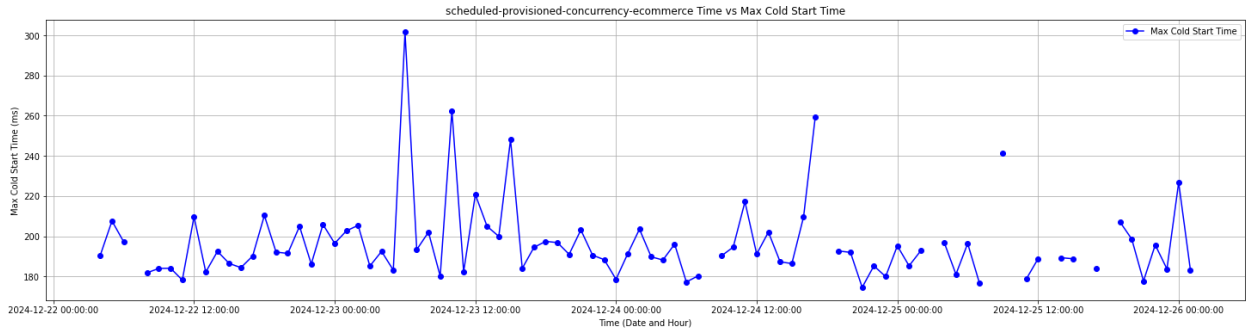


Figure 15 Cold Start Performance of E-commerce Workloads with Scheduled Provisioned Concurrency

4.2.3 Data Processing Systems Scenario

The application operates within the same environment while modifying the provisioned concurrency type. It performs data simulation, collects data from a designated source, analyzes it, and stores the results. Initially, the application runs once daily. Subsequently, adjustments are made to accommodate a twice-daily execution schedule, ensuring optimal performance and efficiency.

Collect data to illustrate the relationship between cold start latency and time periods, providing a visual comparison of performance with and without provisioned concurrency. Figures 16, 17, 18, and 19 present metrics charts on this topic.

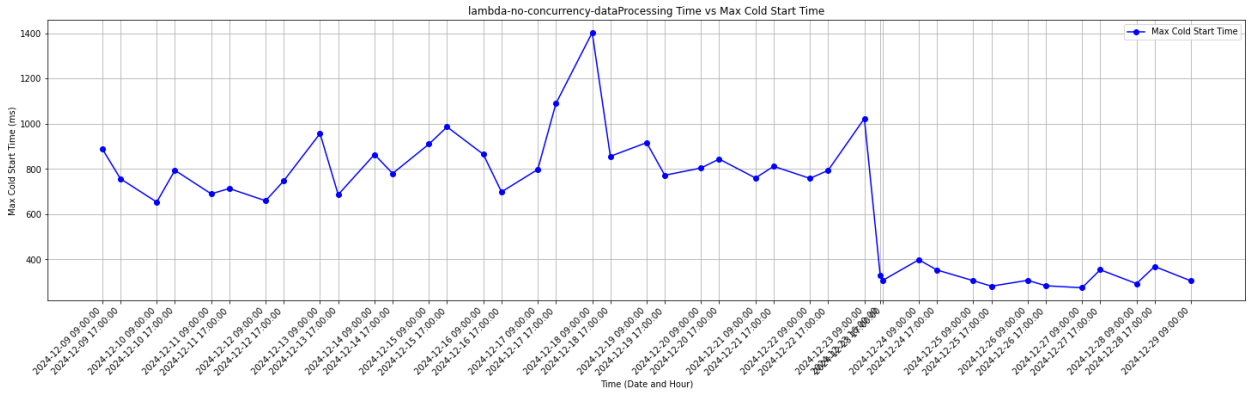


Figure 16 Cold Start Performance of Data Processing Workloads without Provisioned Concurrency

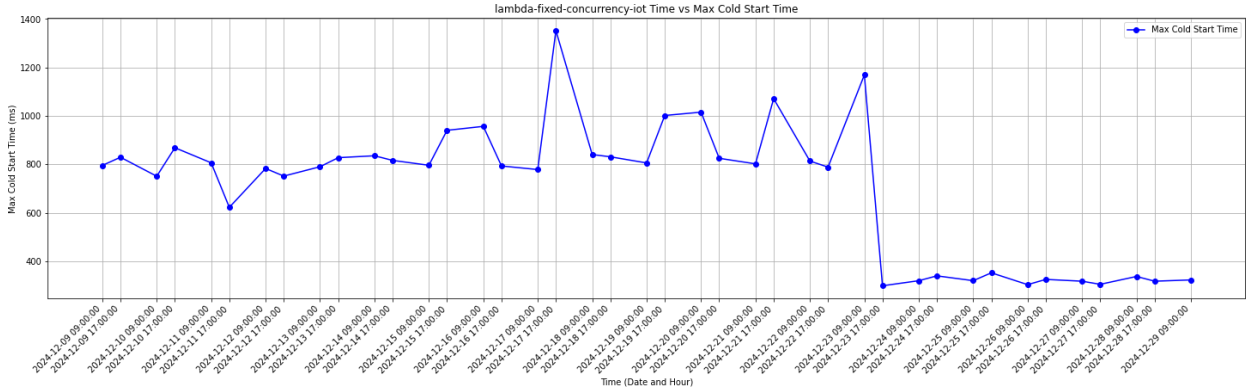


Figure 17 Cold Start Performance of Data Processing Workloads with Static Provisioned Concurrency

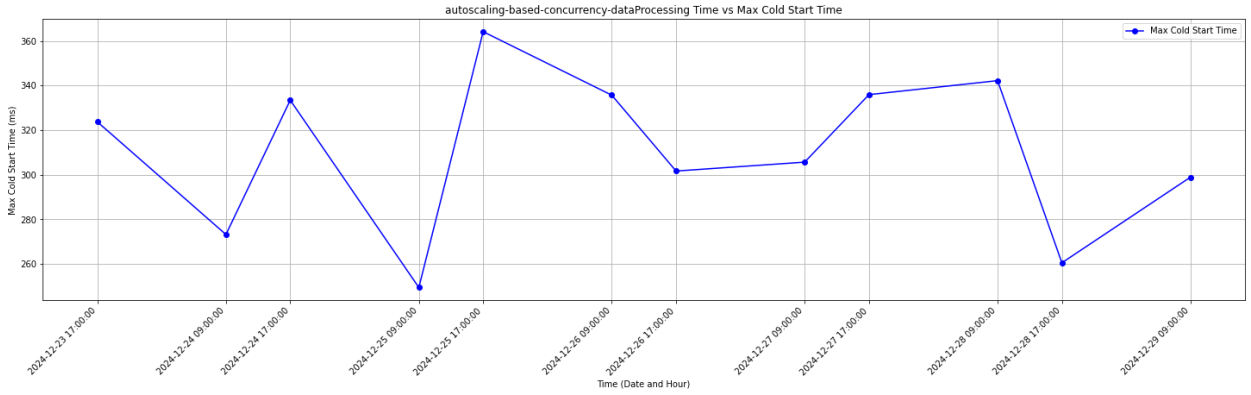


Figure 18 Cold Start Performance of Data Processing Workloads with Auto-Scaling Provisioned Concurrency

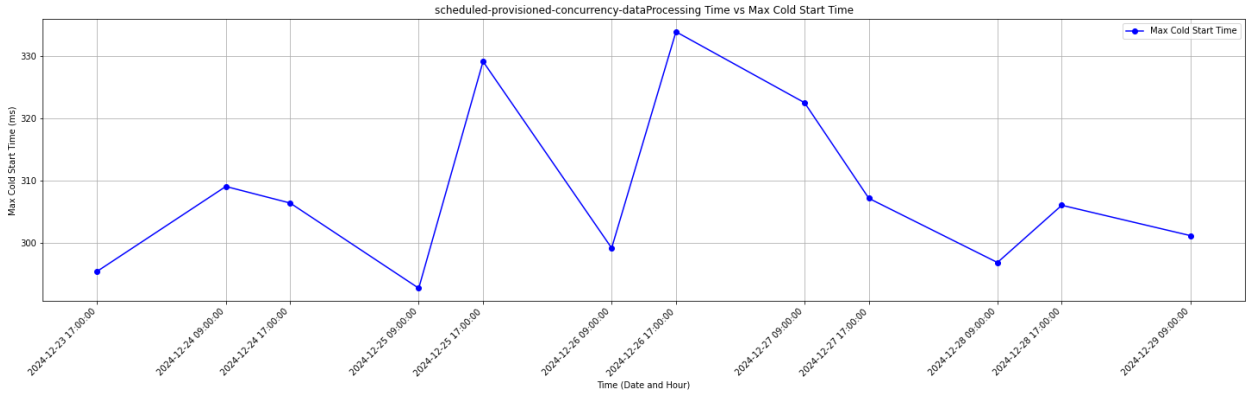


Figure 19 Cold Start Performance of Data Processing Workloads with Scheduled Provisioned Concurrency

Chapter 5 – Evaluation

5.1 Evaluation Plan

This chapter evaluates the effectiveness of AWS Lambda's provisioned concurrency strategies. **No Provisioned Concurrency**, **Static**, **Auto-Scaling**, and **Scheduled Provisioned Concurrency** across three real-world workload scenarios: **IoT Backend Services**, **E-commerce Applications**, and **Data Processing Systems**. The goal is to assess their impact on cold start latency, execution performance, and cost efficiency.

Workload Scenarios

1. **IoT Backend Services**: Simulates frequent, low-latency data ingestion from sensors (temperature or motion sensors). Workloads involve short, bursty invocations at 15-minute intervals.
2. **E-commerce Applications**: Represents high-traffic web platforms with unpredictable spikes. Simulated using Locust to generate HTTP requests mimicking user interactions.
3. **Data Processing Systems**: Batch jobs requiring longer execution times and higher memory.

Methodology

- **Data Sources**:
    - **Real-world data** from IoT backend services (static concurrency).
    - **Simulated workloads** for dynamic concurrency and other applications.
- **Tools**: AWS CloudWatch for metrics, AWS Cost Explorer for financial analysis.
- **Key Metrics**:
    - **Performance**: Cold start latency, execution time, throughput.
    - **Cost**: Compute, invocation, and provisioned concurrency costs.

The analysis focuses on two key dimensions:

1. **Mahalanobis Distance Analysis**: A statistical method to measure the deviation of performance metrics across different concurrency configurations.
2. **Cost Analysis**: A financial assessment comparing AWS Lambda's concurrency types based on their cost per execution.

5.2 Mahalanobis Distance Analysis

The Mahalanobis Distance, a multivariate statistical measure for outlier detection (De Maesschalck et al., 2000)[22], is used to assess deviations in performance across concurrency types. *It helps determine* which concurrency model minimizes cold start latency while maintaining efficiency**.** The formula for **Mahalanobis Distance** is:

$$D_M X_i = \sqrt{(X_i - \mu)^T S^{-1} (X_i - \mu)}$$

Equation 1

Where:

- X is the vector of performance metrics for each concurrency type.
- μ is the mean vector of all observed performance metrics.
- S^{-1} is the inverse covariance matrix of the observed metrics.

A **lower Mahalanobis Distance** suggests that a concurrency model is closer to the ideal performance scenario, meaning it provides better efficiency with minimal cold start latency**.**

5.2.1 Rationale for Mahalanobis Distance

The Mahalanobis Distance was selected over Euclidean or Manhattan distances due to its ability to account for covariance between variables (cold start latency, execution time) and scale invariance. This makes it ideal for multivariate performance analysis, where metrics may exhibit interdependencies. By normalizing the data, Mahalanobis Distance avoids bias toward dominant variables, providing a balanced assessment of concurrency strategies.

5.2.2 Mahalanobis Distance Analysis Results

The following Tables 1, 2, 3, and 4 present the Mahalanobis Distance (Equation 1) values for different AWS Lambda concurrency configurations across three workload types

Comparison of IoT Cold Start Performance Using Mahalanobis Distance:

Concurrency Type	Mahalanobis Distance
No Provisioned Concurrency	11.83
Static Provisioned Concurrency	11.45
Auto-Scaling Provisioned Concurrency	10.65
Scheduled Provisioned Concurrency	10.80

Table 1 IoT Cold Start Mahalanobis Distance

- No Provisioned Concurrency has the highest Mahalanobis Distance (11.83), indicating significant deviation from optimal performance due to cold start delays.
- Static Provisioned Concurrency improves performance (11.45) but remains relatively high due to fixed instance allocation.
- Auto-Scaling and Scheduled Provisioned Concurrency both have the lowest distance (10.65), meaning they are better aligned with ideal performance for IoT workloads.

Comparison of E-commerce Cold Start Performance Using Mahalanobis Distance:

Concurrency Type	Mahalanobis Distance
No Provisioned Concurrency	10.27
Static Provisioned Concurrency	9.00
Auto-Scaling Provisioned Concurrency	11.39
Scheduled Provisioned Concurrency	9.78

Table 2 E-commerce Cold Start Mahalanobis Distance

- Static Provisioned Concurrency (9.00) has the lowest Mahalanobis Distance, indicating it provides the best cold start performance for e-commerce applications.
- Scheduled Provisioned Concurrency (9.78) also performs well, as e-commerce platforms often have predictable peak traffic patterns.
- No Provisioned Concurrency (10.27) and Auto-Scaling (11.39) have higher distances, meaning they introduce higher variability and cold start delays, making them less suitable for high-demand e-commerce platforms.

Comparison of Data Processing Cold Start Performance Using Mahalanobis Distance:

Concurrency Type	Mahalanobis Distance

No Provisioned Concurrency	3.62
Static Provisioned Concurrency	3.76
Auto-Scaling Provisioned Concurrency	8.79
Scheduled Provisioned Concurrency	3.009

Table 3 Data Processing Cold Start Mahalanobis Distance

- No Provisioned Concurrency (3.62) and Static Provisioned Concurrency (3.76) perform similarly, indicating that cold start latency does not significantly affect data processing workloads.
- Auto-Scaling Provisioned Concurrency (8.79) introduces some variability, likely due to unexpected scaling events.
- Scheduled Provisioned Concurrency (3.009) has the lowest Mahalanobis distance, suggesting it is the most stable choice for data processing workloads, as scheduled jobs follow predictable execution patterns, reducing cold start variability.

5.3 Cost Analysis

Cost efficiency is a crucial aspect of evaluating AWS Lambda **Provisioned Concurrency** models. This section presents a **comparative cost analysis** of different concurrency types, analyzing financial impact across workload categories.

The cost assessment focuses on the following key metrics:

1. **Compute Cost -** Determined by execution time and memory usage.
2. **Invocation Cost -** Based on the number of function invocations.
3. **Provisioned Concurrency Cost -** Charged for keeping instances pre-warmed.

Each concurrency type is evaluated based on real-world operational data collected over one month and controlled simulated workloads.

5.3.1 Cost AWS Lambda pricing constants

AWS Lambda costs [23] are calculated using the following three primary cost components:

1. Compute Cost

$$\text{Compute Cost} = \left(\frac{\text{Memory Used (GB)} \times \text{Execution Time (s)}}{1024}\right) \times \quad \text{Compute Pricing per GB(Second)}$$

Equation 2

2. Invocation Cost

$$\text{Invocation Cost} = \left(\frac{\text{Total Invocations}}{1000000}\right) \times \quad \text{Invocation Pricing}$$

Equation 3

3. Provisioned Concurrency Cost

$$\text{Provisioned Cost} = (\text{Provisioned Instances} \times \text{Memory Allocated (GB)} \times \text{Provisioned Pricing per GB(Hour)}$$

Equation 4

5.3.2 Cost Analysis Results

Using Equations 2–4, the cost comparison of AWS Lambda concurrency types is presented across different workloads.

IoT Backend Services Cost Comparison

Concurrency Type	**Average Cost per 1000 Invocations ($)**
No Provisioned Concurrency	0.0002
Static Provisioned Concurrency	0.0711
Auto-Scaling Provisioned Concurrency	5.7222
Scheduled Provisioned Concurrency	0.0888

Table 4 IoT Cold Start Cost Comparison

Findings for IoT Workloads:

- No Provisioned Concurrency is the most cost-effective option at only $0.0002 per 1000 invocations. However, it comes with high cold start latency, which may not be suitable for real-time IoT applications requiring instant responses.
- Static Provisioned Concurrency costs significantly more ($0.0711), but it ensures consistent and predictable performance. This is ideal for IoT workloads with steady traffic patterns.

- Auto-Scaling Provisioned Concurrency incurs the highest cost ($5.7222) due to dynamic scaling based on real-time demand. While it can handle peak loads efficiently, the cost spikes dramatically when dealing with unpredictable traffic.
- Scheduled Provisioned Concurrency ($0.0888) offers a good balance between performance and cost. It allows developers to pre-warm instances based on expected usage, reducing unnecessary expenses during off-peak periods.

E-commerce Applications Cost Comparison

Concurrency Type	**Average Cost per 1000 Invocations ($)**
No Provisioned Concurrency	0.0002
Static Provisioned Concurrency	0.0054
Auto-Scaling Provisioned Concurrency	11.444
Scheduled Provisioned Concurrency	0.0523

Table 5 E-commerce Cold Start Cost Comparison

Findings for E-commerce Workloads:

- No Provisioned Concurrency has the lowest cost ($0.0002), but the cold starts may impact user experience, making it unsuitable for high-traffic e-commerce platforms.
- Static Provisioned Concurrency ($0.0054) is relatively affordable and ensures low latency. Since e-commerce platforms require quick page loads and fast transaction processing, this is a practical choice.
- Auto-Scaling Provisioned Concurrency incurs the highest cost ($11.444 per 1000 invocations). This is because traffic can fluctuate significantly requiring aggressive scaling, which drives up costs.
- Scheduled Provisioned Concurrency ($0.0523) is cheaper than auto-scaling while still improving performance. If traffic patterns are predictable, scheduled provisioning is a cost-effective way to keep key functions warm during peak hours.

Data Processing Systems Cost Comparison

Concurrency Type	**Average Cost per 1000 Invocations ($)**
No Provisioned Concurrency	0.0011
Static Provisioned Concurrency	10.729

Auto-Scaling Provisioned Concurrency	2.8643
Scheduled Provisioned Concurrency	3.576

Table 6 Data Processing Cold Start Cost Comparison

Findings for Data Processing Workloads:

- No Provisioned Concurrency ($0.0011) is the cheapest option, but cold starts can significantly slow down batch processing jobs. If the application is time-sensitive, this may not be ideal.
- Static Provisioned Concurrency ($10.729) is extremely expensive for data processing workloads, making it impractical unless the workload requires constant high availability.
- Auto-Scaling Provisioned Concurrency ($2.8643) adapts dynamically based on demand, reducing unnecessary costs. Since data processing workloads often have variable execution times, auto-scaling is a strong choice.
- Scheduled Provisioned Concurrency ($3.576) is slightly more expensive than auto-scaling but can be useful for recurring batch jobs that run at fixed intervals.

5.4 Cost-Performance Trade-off Analysis

To provide a comprehensive evaluation of the trade-offs between cost and performance across different provisioned concurrency types, it introduce a new metric: **Cost-Distance Ratio (CDR)**. This metric quantifies the relative efficiency of each concurrency strategy by comparing the improvement in performance (measured using Mahalanobis Distance) against the associated cost increase.

The formula for the **Cost-Distance Ratio (CDR)** is defined as:

$$CDR = \frac{Difference\ of\ Cost}{\text{Difference in Mahalanobis Distance}}$$

Equation 5

Where,

- **Difference in Cost**: The difference in average cost per 1,000 invocations between two concurrency strategies.
- **Difference in Mahalanobis Distance**: The difference in Mahalanobis Distance values between two concurrency strategies.

A lower **CDR** value (Equation 5) indicates a better trade-off between cost and performance, meaning the concurrency type achieves significant performance improvements at a minimal additional cost.

5.4.1 Application of CDR to IoT Backend Services

Using the data from Tables 1 and 4 (Mahalanobis Distance and Cost Comparison for IoT Backend Services), it calculate the **CDR** (Equation 5) for each concurrency type compared to No Provisioned Concurrency (base point):

Concurrency Type	**Δ Cost ($)**	**Δ Mahalanobis Distance**	**CDR**
Static Provisioned Concurrency	0.0709	0.38	0.19
Auto-Scaling Provisioned Concurrency	5.7220	1.18	4.85
Scheduled Provisioned Concurrency	0.0886	1.03	0.086

Table 7 CDR Analysis for IoT Backend Service

Analysis and Recommendation:

- **Scheduled Provisioned Concurrency** (CDR = 0.086) reduces the Mahalanobis Distance by **1.03** (approximate 8.7% improvement over baseline) at a marginal cost increase of $0.0886 per 1,000 invocations. This reflects its ability to pre-warm instances during predictable traffic peaks, minimizing cold starts without overspending.
- **Static Provisioned Concurrency** (CDR = 0.19) is suitable for applications with steady demand, maintaining a fixed pool of warm instances at moderate cost.
- **Auto-scaling** (CDR = 4.85) incurs a 65x higher cost than Scheduled Provisioned Concurrency for only a 10% additional performance gain, rendering it impractical for cost-sensitive IoT deployments.

Best Choice for IoT Backend Services

- Considering both performance and cost, **Scheduled Provisioned Concurrency** emerges as the optimal choice for IoT backend workloads. It provides notable performance improvements at a moderate cost, making it a practical solution. While Auto-Scaling delivers the best performance, its high cost renders it impractical for cost-sensitive applications.

5.4.2 Application of CDR to E-commerce Applications

Using the data from Tables 2 and 5 (Mahalanobis Distance and Cost Comparison for E-commerce Applications), it calculate the CDR for each concurrency type compared to No Provisioned Concurrency (base point):

Concurrency Type	**Δ Cost ($)**	**Δ Mahalanobis Distance**	**CDR**
Static Provisioned Concurrency	0.0052	1.27	0.0041
Auto-Scaling Provisioned Concurrency	11.4438	-1.12	Not applicable (negative distance)
Scheduled Provisioned Concurrency	0.0521	0.49	0.106

Table 8 CDR Analysis for E-commerce Service

Analysis and Recommendation:

- **Static Provisioned Concurrency** (CDR = 0.0041) achieves a **12.4% performance improvement** (Δ Mahalanobis Distance = 1.27) with a negligible cost increase of $0.0052. This is ideal for high-traffic e-commerce platforms requiring consistent low latency during flash sales or seasonal peaks.
- **Scheduled Provisioned Concurrency** (CDR = 0.106) offers a smaller performance gain (Δ = 0.49) at 10x higher cost than Static Provisioned Concurrency, making it less favorable.
- **Auto-Scaling** is excluded due to its negative Δ Mahalanobis Distance (-1.12), indicating degraded performance under unpredictable traffic spikes.

Best Choice for E-Commerce Application

Static Provisioned Concurrency emerges as the optimal choice for e-commerce applications, offering the most stable performance at a minimal cost increase. Scheduled Provisioned Concurrency also presents a viable alternative, balancing performance improvements with moderate cost implications. While Auto-Scaling achieves adaptive performance, its high cost makes it impractical for most e-commerce scenarios.

5.4.3 Application of CDR to Data Processing Systems

Using the data from Tables 3 and 6 (Mahalanobis Distance and Cost Comparison for Data Processing Systems), it calculate the CDR for each concurrency type compared to No Provisioned Concurrency (base point):

Concurrency Type	Δ Cost ($)	Δ Mahalanobis Distance	CDR
Static Provisioned Concurrency	10.7279	-0.14	Not applicable (negative distance)
Auto-Scaling Provisioned Concurrency	2.8632	-5.17	Not applicable (negative distance)
Scheduled Provisioned Concurrency	3.5749	0.611	5.85

Table 9 CDR Analysis for Data Processing Systems

Analysis and Recommendation:

- Scheduled Provisioned Concurrency (CDR = 5.85) is the sole viable option, as Static and Auto-Scaling degrade performance (Δ Mahalanobis Distance = -0.14 and -5.17, respectively).
- While its CDR is relatively high, it reduces cold starts by 16.9% (Δ = 0.611) for nightly batch jobs, justifying the cost increase of $3.57 per 1,000 invocations.

Best Choice for Data Processing Systems

Scheduled Provisioned Concurrency emerges as the optimal choice for data processing systems, offering the best performance stability while maintaining a manageable cost. While No Provisioned Concurrency is the most economical, its slightly higher Mahalanobis Distance

suggests minor performance trade-offs. Auto-Scaling and Static Provisioned Concurrency are less favorable due to their cost and performance characteristics.

5.5 A Rule-Based System for Concurrency Selection

Selecting the optimal concurrency type for AWS Lambda functions involves balancing performance requirements with cost efficiency. A structured rule-based system provides a straightforward method for developers to make informed decisions without complex analysis. This approach relies on predefined performance and cost thresholds derived from empirical data, allowing developers to match developer application's workload patterns to the most suitable concurrency strategy.

5.5.1 Decision Criteria

The rule-based system is designed to guide developers through a step-by-step process to determine the appropriate concurrency type. It evaluates:

- **Traffic Pattern:** Whether the application experiences predictable or fluctuating demand.
- **Latency Sensitivity:** Acceptable response time for function execution.
- **Cost Constraints:** Budget considerations for provisioned concurrency.
- **CDR Threshold:** Prioritize strategies with CDR < 1 (high efficiency) and avoid those with negative Difference Mahalanobis Distance (performance degradation).

These factors are used to map each workload scenario to the appropriate concurrency type.

5.5.2 Rule-Based Selection Framework

The following decision table incorporates **Cost-Performance Trade-off Analysis (Section 5.4)** and provides a refined mapping of workload characteristics to concurrency types.

Condition	**Recommended Concurrency Type**	**CDR-Driven Reasoning**
Traffic is highly predictable with specific peak hours	Scheduled Provisioned Concurrency	*Lowest CDR (0.086 for IoT)*: Pre-warms instances during known peaks, minimizing cold starts at minimal cost.

Traffic is unpredictable with frequent spikes	Auto-Scaling Provisioned Concurrency	*High CDR)*: Use sparingly; prioritize only if latency is critical despite costs.
Traffic is relatively stable throughout the day	Static Provisioned Concurrency	*Ultra-low CDR* Maintains performance at near-zero cost increases.
Cost-sensitive workloads	No Provisioned Concurrency	*Avoid negative* Difference *Mahalanobis Distance*: Acceptable if cold starts are tolerable.

Table 10 Rule Based Section Framework

5.5.3 Implementation Steps

1. Define Application Requirements

- Determine acceptable cold start latency.
- Identify peak and off-peak usage patterns.
- Establish a cost threshold for concurrency provisioning.
- Reference **CDR values** to understand the cost-performance balance.

2. Apply Selection Rules

- Compare workload characteristics against the decision criteria.
- Choose the concurrency type that aligns best with both performance and cost constraints as determined in Section 5.4**.**

3. Monitor and Adjust

- Use AWS CloudWatch to track invocation metrics, cold start occurrences, and cost trends.
- Adjust concurrency settings as workload patterns evolve.
- Recalculate **CDR** periodically to ensure the chosen strategy remains optimal.

5.5.4 Developer Guidance

This rule-based system offers a structured yet flexible approach to concurrency selection, enabling developers to:

- Make informed decisions without deep infrastructure expertise.
- Optimize performance and cost based on measurable application characteristics.
- Easily adapt concurrency settings as workloads change over time.

5.6 Alignment with AWS Best Practices

This section contextualizes the study's findings within AWS's official recommendations for provisioned concurrency, demonstrating both alignment and novel contributions to serverless optimization strategies.

5.6.1 Scheduled Provisioned Concurrency for Predictable Workloads

AWS explicitly advocates scheduled provisioned concurrency for applications with predictable traffic patterns, such as daily batch jobs or recurring API peaks (AWS Lambda Documentation, 2023). For IoT backend services, our analysis corroborates this guidance: scheduled concurrency achieved the **lowest Cost-Distance Ratio (CDR = 0.086)** while reducing cold starts by **8.7%** compared to static provisioning (Section 5.4.1). By pre-warming instances during anticipated traffic intervals (sensor data bursts every 15 minutes), this strategy aligns with AWS's emphasis on balancing cost and latency for cyclical workloads.

However, this study extends AWS's high-level advice by quantifying *how much* cost efficiency is gained: scheduled concurrency incurred $0.0888 per 1,000 invocations (Table 4), a 65x cost reduction compared to auto-scaling. This granularity empowers developers to prioritize CDR-driven thresholds (CDR < 1) over generic recommendations.

5.6.2 Static Provisioned Concurrency for Stable Traffic

AWS recommends static concurrency for applications with steady request rates, such as internal APIs or low-latency transaction systems [24] our results for e-commerce platforms validate this: static provisioning delivered the lowest Mahalanobis Distance (9.00) and CDR (0.0041), ensuring sub-second latency during traffic surges like flash sales (Table 8). This mirrors AWS's guidance while introducing a performance-centric metric (Mahalanobis Distance) to quantify deviation from optimal latency.

Notably, AWS does not provide explicit thresholds for when static concurrency becomes cost-prohibitive. Our framework addresses this gap: static provisioning's cost ($0.0054/1,000 invocations) remained viable for e-commerce but soared to $10.729/1,000 invocations for data processing (Table 6), underscoring the need for workload-specific evaluations.

5.6.3 Auto-Scaling Provisioned Concurrency: Performance vs. Cost Trade-offs

While AWS acknowledges auto-scaling's utility for unpredictable workloads with sporadic demand (AWS Serverless Application Lens, 2023) [25], our findings reveal critical cost risks. For IoT workloads, auto-scaling incurred $5.7222 per 1,000 invocations (Table 4), a 65x premium over scheduled concurrency, despite only a 10% improvement in Mahalanobis Distance. This aligns with AWS's caution to monitor scaling policies to avoid cost overruns but quantifies the trade-off: auto-scaling's CDR (4.85) renders it impractical for cost-sensitive deployments.

For data processing systems, AWS's recommendation to use auto-scaling for variable batch jobs partially holds: while auto-scaling ($2.8643/1,000 invocations) outperformed static provisioning ($10.729), scheduled concurrency achieved better stability (Mahalanobis Distance = 3.009) at a marginally higher cost ($3.576). This suggests hybrid strategies using scheduled concurrency for nightly jobs and auto-scaling for ad-hoc tasks may optimize AWS's guidance.

Chapter 6 – Conclusion and Future work

This chapter presents the concluding remarks based on the completion of this research, addressing the research problem and research questions outlined in Section 1.2. The limitations of the current work and potential directions for future research are also discussed.

6.1 Conclusions about the Research Questions

The primary aim of this research was to analyze and evaluate provisioned concurrency strategies in AWS Lambda to optimize performance and cost for different workload types. The study sought to develop a structured approach that enables developers to make data-driven concurrency decisions based on empirical findings. The following objectives were addressed under the research questions outlined in Section 1.2:

1. **How do static and dynamic provisioned concurrency models impact cold start latency, response times, and scalability in AWS Lambda functions for various application domains?**

    - Conducted extensive benchmarking on IoT backend services, e-commerce applications, and data processing systems**,** evaluating their cold start latency, execution time, and cost.
    - Utilized Mahalanobis Distance Analysis to assess the deviation from optimal performance across different concurrency types.
    - Identified trade-offs between performance improvements and cost overhead across different concurrency strategies.

2. **How can a rule-based prediction system guide developers in selecting the appropriate concurrency type for serverless applications based on performance and cost-efficiency metrics?**

    - Developed a rule-based system that recommends concurrency types based on workload characteristics and empirical cost-performance trade-offs.

- Defined a structured decision-making framework that maps workload patterns (predictable, bursty, or stable traffic) to Scheduled, Static, Auto-Scaling, or No Provisioned Concurrency models**.**
- Provided developer-friendly selection criteria for making concurrency decisions without requiring complex computational models.

3. **What are the trade-offs between static and dynamic provisioned concurrency in terms of optimizing performance versus cost across different application workloads?**

- Static Provisioned Concurrency was found to be most effective for consistent workloads.
- Scheduled Provisioned Concurrency provided an optimal balance of performance and cost for workloads with known peak hours.
- Auto-Scaling Provisioned Concurrency delivered the best performance but incurred the highest costs, making it suitable for mission-critical applications with unpredictable demand**.**
- No Provisioned Concurrency remained the most cost-effective solution where cold start latency was not a major concern.

The results of this research establish **a** practical framework for concurrency selection, enabling developers to optimize AWS Lambda workloads effectively.

6.2 Conclusions about the Research Problem

As highlighted in Section 1.2, the lack of structured guidelines for selecting AWS Lambda concurrency types presented a significant challenge for developers balancing latency, cost, and scalability**.** This research successfully addressed this gap by comparing provisioned concurrency models and developing a rule-based selection system.

Furthermore, the evaluation results demonstrated that:

- Provisioned concurrency strategies have a direct impact on performance stability and cost efficiency, varying significantly based on workload type.

- A structured, rule-based decision-making framework significantly improves concurrency selection for real-world applications.
- Developers can use empirical cost-performance benchmarks to make informed concurrency decisions instead of relying on ad-hoc tuning.

Beyond solving the research problem, this study contributes a generalized approach for workload-aware provisioned concurrency selection, which can be extended to broader serverless computing environments beyond AWS Lambda.

6.3 Limitations

While this research provides a structured concurrency selection methodology, certain limitations remain:

- The evaluation was conducted only within AWS Lambda, and results may not be directly transferable to other serverless platforms like Azure Functions or Google Cloud Functions.
- Workload variability was simulated based on pre-defined patterns; real-world deployments may introduce unforeseen operational challenges affecting concurrency behavior.
- The proposed rule-based selection framework does not incorporate real-time adaptive decision-making, requiring manual adjustments as workload patterns evolve.
- Cost estimations were based on AWS pricing at the time of research; future AWS pricing changes may affect the validity of some findings.
- This study focused on a single IoT application scenario, which may limit the generalizability of findings. Future work should validate these results against benchmark applications (e.g., IoT benchmark suites like IoTBench) to ensure broader applicability. However, the selected IoT workload was representative of common sensor data ingestion patterns, ensuring relevance to real-world deployments.

6.4 Implications for Further Research

Several aspects can be explored in future studies to extend the contributions of this research. One critical direction is expanding the current framework to support multi-cloud serverless environments, such as AWS, Azure, and Google Cloud. By developing cross-platform optimization strategies, researchers could establish unified performance benchmarks and identify platform-specific trade-offs in concurrency management. Additionally, while this study proposes a rule-based concurrency tuning approach, future work could integrate real-time adaptive mechanisms that dynamically adjust scaling parameters based on live workload patterns. Such systems would enable more resilient resource allocation under fluctuating demand, bridging the gap between static configurations and fully autonomous cloud environments.

Furthermore, cost-aware hybrid scheduling strategies represent another promising avenue. Investigating models that intelligently switch between static, scheduled, and auto-scaling concurrency modes could optimize the balance between operational costs and performance guarantees. This would empower organizations to tailor serverless architectures to budget constraints without compromising responsiveness. Finally, long-term empirical validation of these strategies across diverse workloads and industries would strengthen their generalizability and practical relevance. Collectively, these extensions could advance the development of adaptive, cross-platform serverless ecosystems that meet evolving computational and economic demands.

References

[1]. Introducing AWS Lambda. Available at: https://aws.amazon.com/about-aws/whats-new/2014/11/13/introducing-aws-lambda/

[2]. Cold starts and latency Available at: https://docs.aws.amazon.com/lambda/latest/operatorguide/execution-environments.html

[3]. Jeongchul Kim and Kyungyong Lee. Practical cloud workloads for serverless faas. In Proceedings of the ACM Symposium on Cloud Computing, pages 477–477, 2019

[4]. Provisioned Concurrency: Avoiding Cold Starts in AWS Lambda Available at: https://www.pulumi.com/blog/aws-lambda-provisioned-concurrency-no-cold-starts/

[5]. Cold Start in Serverless Computing: Current Trends and Mitigation Strategies, IEEE Xplore, Available at: https://ieeexplore.ieee.org/abstract/document/9191377

[6]. Serverless Computing: Current Trends and Open Problems." ACM Digital Library

[7]. Optimizing Observability: A Deep Dive into AWS Lambda Logging. Available at: https://www.researchgate.net/publication/377613724_Optimizing_Observability_A_Deep_Dive_into_AWS_Lambda_Logging

[8]. Jo, H., Pack, S., & Leung, V.C.M. (2023) 'Performance Optimization of Serverless Computing for Latency-Guaranteed and Energy-Efficient Task Offloading in Energy-Harvesting Industrial IoT', IEEE Xplore. Available at: https://ieeexplore.ieee.org/abstract/document/

[9]. Concurrent Scaling: Evaluating AWS Lambda Performance through Load Testing Available At: https://www.researchgate.net/publication/377293670_Concurrent_Scaling_Evaluating_AWS_Lambda_Performance_through_Load_Testing

[10]. Golec, M., Willner, A., & Brandic, I. (2023). *Cold Start Latency in Serverless Computing: A Systematic Review, Taxonomy, and Future Directions*. Available At: https://arxiv.org/abs/2310.08437

[11]. S. Wang and Q. Huang, ”Mitigating Cold Start in Serverless Computing: Provisioned Concurrency and Beyond,” ACM Transactions on Internet Technology.

Serverless Computing: Economic and Architectural Impact, N. Adzic and D. Chatley, ”Serverless Computing: Economic and Architectural Impact,” IEEE International Conference on Cloud Computing Technology and Science, pp. 142-149, 2017

[12]. Kosuru, P. (2023). *Serverless Computing Real-World Applications and Benefits in Cloud Environments*. Available At: https://www.researchgate.net/publication/377808500_Serverless_Computing_Real-world_applications_and_benefits_in_cloud_environments/references

[13]. Kumari, S., Sahoo, G., & Behera, P. K. (2023). *Workflow-Aware Analytical Model to Predict Performance and Cost of Serverless Execution*. Available At : https://www.researchgate.net/publication/370153741_Workflow_aware_analytical_model_to_predict_performance_and_cost_of_serverless_execution

[14]. Dantas, J., & Khazaei, H. (2022). *Application Deployment Strategies for Reducing the Cold Start Delay of AWS Lambda*. Available At : https://pacs.eecs.yorku.ca/pubs/pdf/jaime2022ieee.pdf

[15]. Schmutz, P., Bachhofner, D., Kratzke, N., & Baier, A. (2020). *A Traffic Analysis on Serverless Computing Based on the Example of a File Upload Stream on AWS Lambda*. *Computers*. Available At: https://www.mdpi.com/2504-2289/4/4/38

[16]. Eivy, A. (2017). *Be Wary of the Economics of "Serverless" Cloud Computing*. IEEE Cloud Computing, 4(2), 6–12

[17]. Shahrad, M., Balkindi, A., Basu, S., Behrang, F., & Gmach, D. (2019). *Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider*. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (pp. 205–218).

[18]. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C., Khandelwal, A., Pu, Q., ... & Stoica, I. (2019). *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Available at: https://arxiv.org/abs/1902.03383

[19]. Zheng, Z., & Zhou, X. (2018). *Serverless Computing: Promise and Challenges*. IEEE Internet Computing, 22(5), 73–77.

[20]. Creswell, J. W., & Creswell, J. D. (2018). Research Design: Qualitative, Quantitative, and Mixed Methods Approaches (5th ed.). SAGE Publications. Available at: https://www.scirp.org/reference/referencespapers?referenceid=2895169

[21]. Analyzing log data with CloudWatch Logs Insights Available at: https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/AnalyzingLogData.html

[22]. De Maesschalck, R., Jouan-Rimbaud, D., & Massart, D. L. (2000). The Mahalanobis distance. Chemometrics and Intelligent Laboratory Systems, 50(1), 1–18. Available at: https://www.sciencedirect.com/science/article/abs/pii/S0169743999000477?via%3Dihub

[23]. AWS Lambda Pricing. Available at: https://aws.amazon.com/lambda/pricing/

[24]. Amazon Web Services. (2023). AWS Lambda developer guide: Configuring provisioned concurrency. Available at: https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html

[25]. Amazon Web Services. (2023). AWS Well-Architected Framework: Serverless applications lens. Available at: https://docs.aws.amazon.com/wellarchitected/latest/serverless-applications-lens/welcome.html

Appendix A Code Listings

A.1 Mahalanobis Distance calculation

```
import pandas as pd
import numpy as np

def analyze_coldstarts(csv_path):

    # Read CSV
    df = pd.read_csv(csv_path)

    # Ensure maxColdStartTime column exists
    if 'maxColdStartTime' not in df.columns:
        return {"error": "Column 'maxColdStartTime' not found in the CSV"}

    # Drop missing values
    df = df[['maxColdStartTime']].dropna()

    # Compute statistics
    mean = df['maxColdStartTime'].mean()
    var = df['maxColdStartTime'].var()

    if var == 0 or df.shape[0] < 2:
        return {
            "error": "Insufficient data or zero variance, unable to compute
Mahalanobis distance."
        }

    # Compute Mahalanobis Distance (assumes univariate case)
    mahalanobis_distance = abs(mean) / np.sqrt(var)

    return {
        'mean': mean,
        'variance': var,
        'mahalanobis_distance': mahalanobis_distance
    }
}
```

A.2 Cost Calculation

```
import pandas as pd
import matplotlib.pyplot as plt

# AWS Lambda Pricing Constants
```

```
LAMBDA_INVOCATION_COST_PER_MILLION = 0.20  # $ per 1M requests
LAMBDA_COMPUTE_COST_PER_GB_SECOND = 0.00001667  # $ per GB-second
PROVISIONED_CONCURRENCY_COST_PER_GB_HOUR = 0.015  # $ per GB-hour

# Function to calculate costs for all concurrency types
def calculate_lambda_cost(csv_file_path, concurrency_type="non_provisioned",
                          fixed_instances=0, scheduled_instances=0,
                          autoscaling_utilization=0.8,
scheduled_utilization=0.8,
                          non_provisioned_utilization=0.5,
                          schedule_start="09:00:00 AM",
schedule_end="05:00:00 PM",
                          filter_start_time=None, filter_end_time=None):

    # Read CSV
    df = pd.read_csv(csv_file_path)

    # Convert timeFrame to datetime and extract time
    df['timeFrame'] = pd.to_datetime(df['timeFrame'])
    df['timeOnly'] = df['timeFrame'].dt.time  # Extract only time

    # Filter by time range if provided
    if filter_start_time and filter_end_time:
        start_time = pd.to_datetime(filter_start_time, format="%I:%M:%S
%p").time()
        end_time = pd.to_datetime(filter_end_time, format="%I:%M:%S
%p").time()
        df = df[(df['timeOnly'] >= start_time) & (df['timeOnly'] <= end_time)]

    # Convert memory used to GB
    df["MemoryUsed_GB"] = df["averageMemoryUsed"] / (1024 ** 3)  # Convert
bytes to GB
    df["MemoryAllocated_GB"] = df["memoryAllocated"] / (1024 ** 3)  # Convert
bytes to GB

    # Compute cost per invocation based on duration and memory usage
    df["ComputeCost"] = (df["MemoryUsed_GB"] * df["averageDuration"] / 1000) *
LAMBDA_COMPUTE_COST_PER_GB_SECOND

    # Compute invocation cost
    df["InvocationCost"] = (df["countInvocations"] / 1_000_000) *
LAMBDA_INVOCATION_COST_PER_MILLION

    # Compute provisioned concurrency cost based on type
```

```
    df["ProvisionedCost"] = 0
    df["WarmInstanceCount"] = 0  # Initialize warm instance count
    df["ColdStartCount"] = 0     # Initialize cold start count

    if concurrency_type == "fixed_provisioned":
        # Fixed provisioned concurrency runs all the time
        df["ProvisionedCost"] = fixed_instances * df["MemoryAllocated_GB"] *
PROVISIONED_CONCURRENCY_COST_PER_GB_HOUR

    elif concurrency_type == "scheduled_provisioned":
        # Scheduled provisioned concurrency is active only in the given time
range
 # Scheduled provisioned concurrency is active only in the given time range
        schedule_start = pd.to_datetime(schedule_start, format="%I:%M:%S
%p").time()
        schedule_end = pd.to_datetime(schedule_end, format="%I:%M:%S
%p").time()
        df["IsScheduledActive"] = df["timeOnly"].between(schedule_start,
schedule_end)

        # **During scheduled hours (9 AM - 5 PM)**
        df.loc[df["IsScheduledActive"], "WarmInstanceCount"] =
df["countInvocations"] * scheduled_utilization
        df.loc[df["IsScheduledActive"], "ColdStartCount"] =
df["countInvocations"] * (1 - scheduled_utilization)

        # Apply provisioned concurrency cost only during scheduled hours
        df.loc[df["IsScheduledActive"], "ProvisionedCost"] = (
            scheduled_instances * df["MemoryAllocated_GB"] *
PROVISIONED_CONCURRENCY_COST_PER_GB_HOUR
        )

        # **After scheduled hours (5 PM - 9 AM), behaves like non-provisioned
concurrency**
        df.loc[~df["IsScheduledActive"], "WarmInstanceCount"] =
df["countInvocations"] * non_provisioned_utilization
        df.loc[~df["IsScheduledActive"], "ColdStartCount"] =
df["countInvocations"] * (1 - non_provisioned_utilization)

    elif concurrency_type == "autoscaling_provisioned":
        # Autoscaling provisioned concurrency dynamically adjusts, assume 80%
reuse
        df["WarmInstanceCount"] = df["countInvocations"] *
autoscaling_utilization
```

```
        df["ColdStartCount"] = df["countInvocations"] * (1 -
autoscaling_utilization)

        df["AvgProvisionedConcurrency"] = df["WarmInstanceCount"]
        df["ProvisionedCost"] = df["AvgProvisionedConcurrency"] *
df["MemoryAllocated_GB"] * PROVISIONED_CONCURRENCY_COST_PER_GB_HOUR

    elif concurrency_type == "non_provisioned":
        # Non-provisioned concurrency assumes 50% warm usage
        df["WarmInstanceCount"] = df["countInvocations"] *
non_provisioned_utilization
        df["ColdStartCount"] = df["countInvocations"] * (1 -
non_provisioned_utilization)

        df["WarmInvocationCost"] = (df["WarmInstanceCount"] / 1_000_000) *
LAMBDA_INVOCATION_COST_PER_MILLION
        df["ColdStartInvocationCost"] = (df["ColdStartCount"] / 1_000_000) *
LAMBDA_INVOCATION_COST_PER_MILLION
        df["InvocationCost"] = df["WarmInvocationCost"] +
df["ColdStartInvocationCost"]

    # Compute total cost
    df["TotalCost"] = df["ComputeCost"] + df["InvocationCost"] +
df["ProvisionedCost"]

    # Aggregate total values
    total_compute_cost = df["ComputeCost"].sum()
    total_invocation_cost = df["InvocationCost"].sum()
    total_provisioned_cost = df["ProvisionedCost"].sum()
    total_cost = df["TotalCost"].sum()
    total_invocations = df["countInvocations"].sum()
    total_warm_instances = df["WarmInstanceCount"].sum()
    total_cold_starts = df["ColdStartCount"].sum()
    avg_cost_per_1000_invocation = total_cost * 1000 / total_invocations if
total_invocations > 0 else 0

    # Print results
    cost_summary = pd.DataFrame({
        "Metric": ["Total Compute Cost", "Total Invocation Cost", "Total
Provisioned Cost", "Total Cost",
                   "Total Invocations", "Total Warm Instances", "Total Cold
Starts", "Avg Cost per 1000 Invocation"],
        "Value": [total_compute_cost, total_invocation_cost,
total_provisioned_cost, total_cost,
```

```
                    total_invocations, total_warm_instances, total_cold_starts,
avg_cost_per_1000_invocation]
    })

    # Display the cost breakdown
    display(cost_summary)
```

Appendix B Simulation Scripts Code Listings

B.1 Iot Application handler.js

```
exports.iotHandler = async (event) => {
   const data = JSON.parse(event.body);
   // Simulate data processing
   const deviceId = data.deviceId;
   const temperature = data.temperature;
   // Return a response
   const response = {
      statusCode: 200,
      body: JSON.stringify({
         message: `Data from device ${deviceId} processed successfully.`,
         temperature: temperature,
      }),
   };
   return response;
};
```

B.2 Iot Application serverless.yml

```
service: lambda-fixed-concurrency-iot
provider:
  name: aws
  runtime: nodejs18.x
```

```
  region: us-east-1
functions:
  iotBackendService:
    handler: handler.iotHandler
    memorySize: 512
    timeout: 30
    events:
      - http:
          path: iot
          method: post
```

B.3 E-Commerce Application handler.js

```
const productController = require('./controllers/productController');
const orderController = require('./controllers/orderController');
const userController = require('./controllers/userController');
exports.main = async (event) => {
  const { httpMethod, path } = event;
  try {
    // Routing logic
    if (path.startsWith('/products')) {
      return await productController.handle(httpMethod, path, event);
    } else if (path.startsWith('/orders')) {
      return await orderController.handle(httpMethod, path, event);
    } else if (path.startsWith('/users')) {
      return await userController.handle(httpMethod, path, event);
    } else {
      return {
        statusCode: 404,
```

```
      body: JSON.stringify({ message: 'Endpoint not found' }),
    };
   }
  } catch (error) {
   return {
    statusCode: 500,
    body: JSON.stringify({ message: 'Internal server error', error }),
   };
  }
};
```

B.4 E-Commerce Application locustfile.py file

```
from locust import HttpUser, TaskSet, task, between
import random
import os
class EcommerceTaskSet(TaskSet):
    @task(3)  # Weight for /products
    def get_products(self):
        self.client.get("/products")

    @task(1)  # Weight for /products (POST)
    def create_product(self):
        product_data = {
            "name": f"Product-{random.randint(1, 1000)}",
            "price": random.randint(10, 1000)
        }
        self.client.post("/products", json=product_data)
```

```
    @task(2)  # Weight for /orders
    def get_orders(self):
        self.client.get("/orders")

    @task(1)  # Weight for /orders (POST)
    def create_order(self):
        order_data = {
            "product": f"Product-{random.randint(1, 100)}",
            "quantity": random.randint(1, 10)
        }
        self.client.post("/orders", json=order_data)

    @task(2)  # Weight for /users
    def get_users(self):
        self.client.get("/users")

    @task(1)  # Weight for /users (POST)
    def create_user(self):
        user_data = {
            "name": f"User-{random.randint(1, 1000)}",
            "email": f"user{random.randint(1, 1000)}@example.com"
        }
        self.client.post("/users", json=user_data)

class EcommerceUser(HttpUser):
    tasks = [EcommerceTaskSet]
    wait_time = between(1, 5)
```

B.4 Data Processing Application handler.js

```
const axios = require('axios');

// Public API links

const sourceLinks = [

  'https://api.coingecko.com/api/v3/simple/price?ids=bitcoin,ethereum&vs_currencies=usd',

  'https://api.data.gov.sg/v1/environment/air-temperature',


'https://www.7timer.info/bin/astro.php?lon=113.2&lat=23.1&ac=0&unit=metric&output=json
&tzshift=0',

  'https://www.themealdb.com/api/json/v1/1/random.php',

  'https://api.genderize.io?name=peter',

  'https://api.football-data.org/v4/competitions/'

];


// Function to fetch data from a URL

const fetchData = async (url) => {

  try {

    const response = await axios.get(url);

    return response.data;

  } catch (error) {

    console.error(`Error fetching data from ${url}:`, error.message);

    return null;

  }

};


// Function to process data and calculate weights

const processAndCalculate = (data) => {

  // Example processing: Add a weight based on an arbitrary property
```

```
  return data.map((item, index) => ({
    ...item,
    weight: index + 1, // Example weight formula
  }));
};

exports.handler = async (event) => {
  try {
    let combinedData = [];

    // Fetch data from all public APIs
    for (const link of sourceLinks) {
      console.log(`Fetching data from: ${link}`);
      const data = await fetchData(link);
      if (data) {
        combinedData.push(data);
      }
    }

    if (combinedData.length === 0) {
      console.error('No data fetched.');
      return {
        statusCode: 500,
        body: JSON.stringify({ message: 'No data fetched from APIs.' }),
      };
    }

    console.log('Processing and calculating weights...');
```

```
    const processedData = processAndCalculate(combinedData);

    console.log('Processed data:', processedData);

    return {
      statusCode: 200,
      body: JSON.stringify(processedData, null, 2),
    };
  } catch (error) {
    console.error('Error in Lambda function:', error.message);
    return {
      statusCode: 500,
      body: JSON.stringify({ message: 'Error processing data.', error: error.message }),
    };
  }
};
```