NORTHEASTERN UNIVERSITY
Department of Electrical & Computer Engineering

EECE 5643 – Simulation and Performance Evaluation

Performance Evaluations on Cold/Warm Starts of Serverless Functions (AWS Lambda)

Instructor: **Prof. Ningfang Mi**

Author's name: Tianqi Huang
Yuchen Li

Semester: 2025 Spring

Abstract - Serverless computing, represented by Function-as-a-Service (FaaS) platforms such as AWS Lambda, has become increasingly popular due to its simplicity, elasticity, and cost efficiency [1][2]. However, a significant challenge of serverless computing is the cold start latency — the delay caused by initializing execution environments when functions are invoked after a period of inactivity. This overhead can negatively impact user experience and is difficult to predict due to highly variable invocation patterns [1]. In this paper, we present a detailed evaluation of cold and warm start performance on AWS Lambda. We build on prior studies that have characterized production-scale FaaS workloads [1], and research that has explored dynamic pre-warming strategies [2], but focus on AWS Lambda as the target platform. Our contributions will include measurements across different runtimes and memory configurations, analysis of invocation interval impact on cold starts, and suggestions for adaptive resource management policies for serverless computing on AWS Lambda.

1. Introduction

Serverless computing, also known as Function-as-a-Service (FaaS), has rapidly evolved into a popular computing paradigm that allows developers to deploy and run fine-grained functions without the need to provision or manage servers. Major cloud providers such as Amazon Web Services (AWS), Google Cloud, Microsoft Azure, and Alibaba Cloud have launched their own FaaS platforms, with AWS Lambda being one of the most widely adopted services [3]. These platforms provide developers with the flexibility to focus solely on writing application logic while the cloud provider handles resource management, autoscaling, and infrastructure maintenance [4].

The appeal of serverless computing stems from its promise of elastic resource allocation and cost efficiency. Users are charged based on execution time and resource consumption, rather than paying for pre-allocated server capacity [5]. Furthermore, the ability to scale from zero to thousands of concurrent function instances within seconds has made serverless platforms ideal for handling unpredictable, bursty workloads.

However, the convenience of serverless computing comes with challenges. One of the most significant issues is the "cold start" problem — the overhead incurred when an instance must be created and initialized before serving a request [6]. Cold starts lead to additional latency, which can significantly impact applications requiring low response times. In contrast, warm starts reuse pre-initialized instances, providing faster execution but requiring the platform to balance memory and resource allocation. The heterogeneity of invocation patterns, along with resource contention, makes optimizing cold/warm start management a complex challenge for cloud providers [7].

Several studies have characterized the cold start phenomenon. Shahrad et al. [3] analyzed Azure Functions workloads and found that most functions are invoked infrequently, but a small fraction of functions are invoked orders of magnitude more often, making one-size-fits-all policies ineffective. Roy et al. [4] introduced IceBreaker, a dynamic keep-alive system that reduces service time and costs by leveraging heterogeneous hardware and intelligently predicting when to keep functions warm. Additionally, Wang et al. [5] highlighted the scalability challenges faced by Alibaba Cloud Function Compute, which must handle thousands of requests per second while dealing with large container startup times.

In this project, we focus on AWS Lambda, the industry-leading FaaS platform, to evaluate the performance differences between cold and warm starts. We aim to measure cold start latencies under various configurations, such as different memory allocations, runtime environments, and invocation patterns. Furthermore, we will study how AWS Lambda's resource isolation, bin-packing scheduling, and scaling strategies affect cold start times and resource efficiency, as previously observed by Wang et al. [6].

By combining measurement-based experimentation with prior research insights, this project will contribute to a more comprehensive understanding of the cold/warm start trade-offs in AWS Lambda. Ultimately, our goal is to inform better function deployment strategies and guide future optimization research for serverless systems.

2. Background and Related Work

Serverless computing platforms, such as AWS Lambda, Azure Functions, and Google Cloud Functions (GCF), have become popular paradigms that allow developers to deploy functions without managing infrastructure. These platforms abstract away resource provisioning, scaling, and server management, enabling developers to focus on writing application logic while the provider dynamically handles execution environments and scaling demands [9].

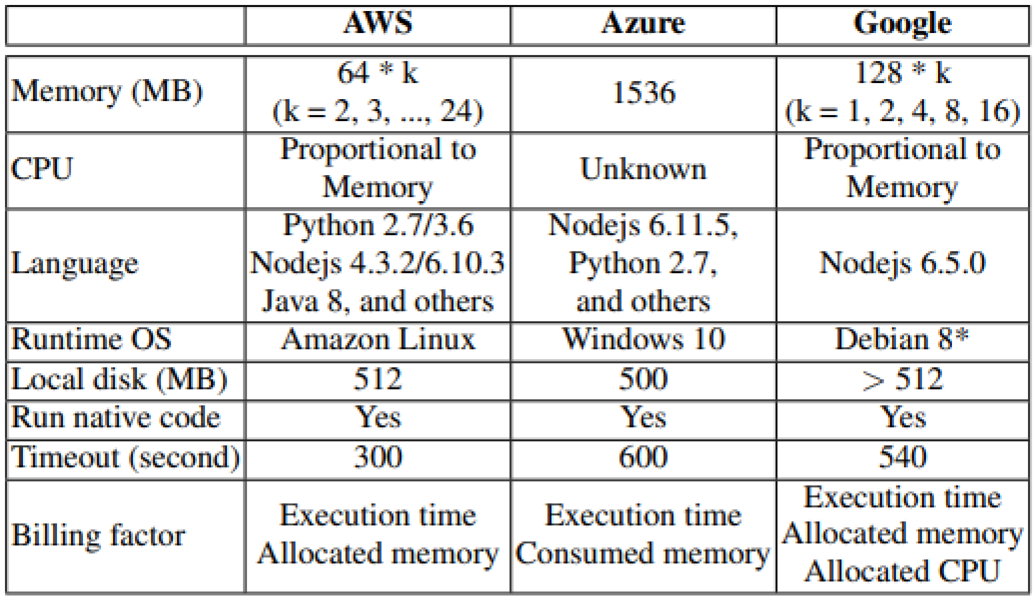
A key challenge in serverless computing is the phenomenon of cold starts and warm starts. A cold start occurs when a function invocation triggers the initialization of a new execution environment (container or VM), leading to noticeable latency as resources are allocated and code is loaded. In contrast, a warm start reuses an already-initialized environment, resulting in lower invocation latency [10]. Cold start delays can become critical for latency-sensitive applications and are especially problematic for functions with low invocation frequency.

To mitigate cold start delays, numerous strategies have been proposed. Dynamic pre-warming strategies attempt to predict future invocations and proactively keep containers warm [7]. However, static pre-warming policies can be inefficient and wasteful. One promising solution is heterogeneous infrastructure utilization, such as proposed in the IceBreaker system [9]. IceBreaker dynamically selects between high-end and low-end servers based on the probability of imminent invocation, significantly reducing both cold starts overhead and keep-alive costs.

Additionally, serverless workloads are known to exhibit extreme variability in invocation frequency and burstiness. Observations from large-scale cloud providers indicate that while many functions are invoked extremely infrequently, a small portion experiences frequent bursts of invocations [10]. Understanding these invocation distributions is essential for designing effective pre-warming and resource allocation strategies.

Previous measurement frameworks have laid the foundation for methodological inspiration in this area. The work on OpenWhisk-based measurement studies [7] provided one of the earliest frameworks to systematically study FaaS platforms, examining invocation patterns and resource isolation issues. Similarly, more recent efforts, such as Pocket [8], explored ephemeral storage scalability for serverless analytics, highlighting the interplay between workload burstiness and I/O demands.

This work builds on these insights and aims to further evaluate and quantify cold and warm start behaviors in AWS Lambda under diverse workloads, while also proposing dynamic management strategies inspired by prior studies.



	AWS	Azure	Google
Memory (MB)	64 * k (k = 2, 3, ..., 24)	1536	128 * k (k = 1, 2, 4, 8, 16)
CPU	Proportional to Memory	Unknown	Proportional to Memory
Language	Python 2.7/3.6 Nodejs 4.3.2/6.10.3 Java 8, and others	Nodejs 6.11.5, Python 2.7, and others	Nodejs 6.5.0
Runtime OS	Amazon Linux	Windows 10	Debian 8*
Local disk (MB)	512	500	> 512
Run native code	Yes	Yes	Yes
Timeout (second)	300	600	540
Billing factor	Execution time Allocated memory	Execution time Consumed memory	Execution time Allocated memory Allocated CPU

Table 1: A comparison of function configuration and billing in three services.

The table summarizes these key differences between AWS Lambda, Azure Functions, and Google Cloud Functions, highlighting their execution environments, billing factors, and resource allocation strategies [4]. These variations directly impact function performance, cold start behavior, and cost efficiency, making it crucial to analyze their implications in real-world workloads.

3. Methodology

In this section, we describe our experimental design to evaluate the cold start and warm start performance of AWS Lambda functions. Our approach focuses on easy-to-reproduce scenarios using common runtimes and simple metrics.

3.1 Environment Setup

3.1.1 OpenWhisk

We initially started our project on OpenWhisk, an open source, distributed Serverless platform that executes functions (fx) in response to events at any scale [11]. OpenWhisk offers a rich programming model for creating serverless APIs from functions, composing functions into serverless workflows, and connecting events to functions using rules and triggers [12]. Meanwhile, we referenced the paper studies we had earlier this semester. We believe that OpenWhisk would be a great platform for our implementation.

Furthermore, we were honored to invite Wang, an PhD student who focused on Serverless Infrastructure and Resource Management from Northeastern University, to join in our discussion of our further implementation. From the figure attached below (Fig. 1), it indicated how the process flows of action invocation in Openwhisk with a local deployment [13]. Notably, (5) The invoker pool, which is the key component of the Openwhisk, each of the innovations would execute separately at individual Docker Container [13]. The containers would be determined after the completion of functions.

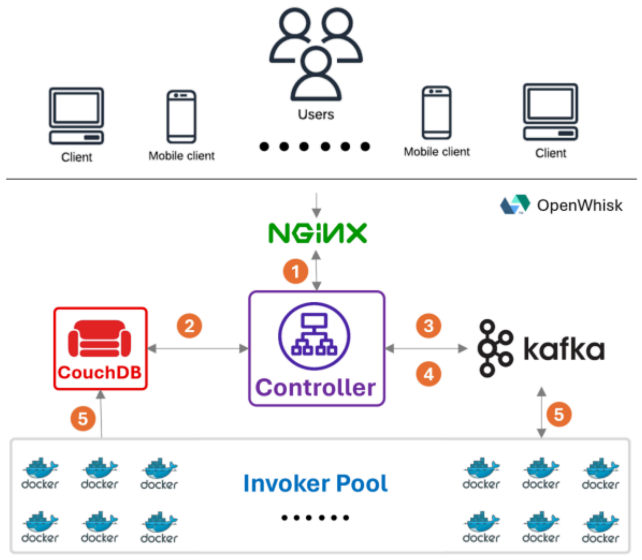
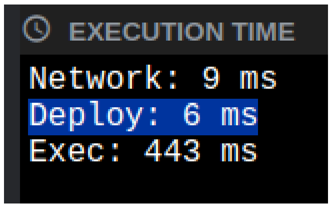


Fig. 1 The flow chart of Openwhisk operating with CouchDB and Kafka

In majority, the serverless providers were challenged at determining how many containers were kept alive for the next invocation. It would result in less start delay “warm starts” compared to the full cycle of operations “cold starts”. Therefore, we were initially inspired to deploy our Openwhisk with a single node Kubernetes cluster with a limited number of pods and containers, which would run two functions synchronously. One would start in “warm”, since there is an available container, and the other would be designated to start in cold, since there is no available container to execute directly. We would observe and compare the delays from our serverless model including leveling up the complexity and workload of the task functions. The snapshot attached below is a demonstration of our experiment view. It was a warm start with only 6ms deployment delay (Fig. 2).



```
Network: 9 ms
Deploy: 6 ms
Exec: 443 ms
```

Fig. 2 The execution in a warm start with a very short delay.

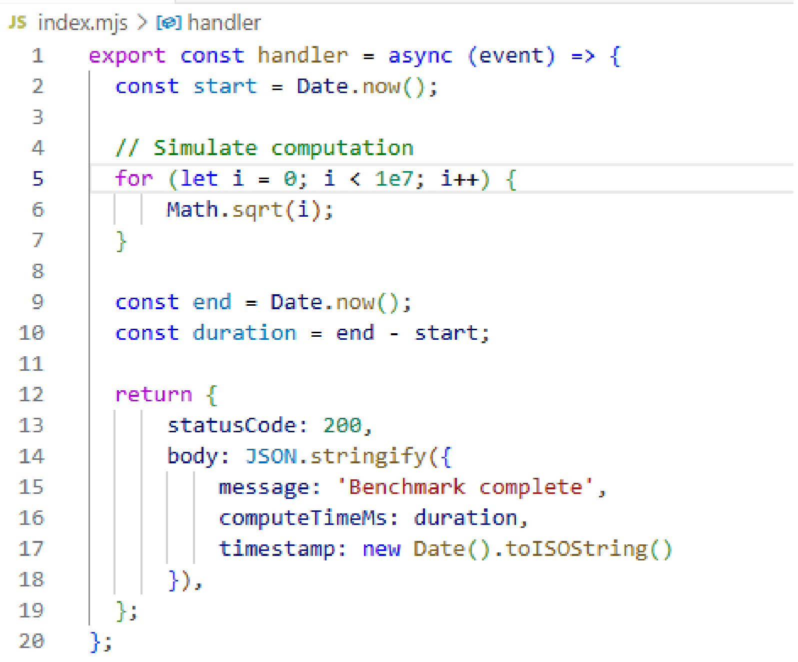
Furthermore, we were facing the limitations and accessibility of Openwhisk for moving forward with our implementation, especially configuring ***wsk*** CLI with our demands. Even though Openwhisk is considered as one of the biggest serverless models available, the community is less maintained (since 2022) in reality, and there were less people how had experience deploying OpenWhisk locally on a Windows machine with WSL, Windows Subsystem for Linux. That took us more than a month just to configure the local environment (including prerequisites, system privileges and more problems). These challenges further emphasized the simplicity and convenience of using AWS Lambda.

3.1.2 AWS Lambda

We use the AWS Lambda platform to deploy and test serverless functions. The setup involves:

- Creating functions directly through the AWS Management Console.
- Using default settings for memory and timeout, unless otherwise specified.
- Testing Python and Node.js runtimes, as they are the most common and lightweight.

Our benchmark Lambda function is implemented in JavaScript and performs a lightweight CPU-bound task to simulate a measurable workload. The code is shown below:

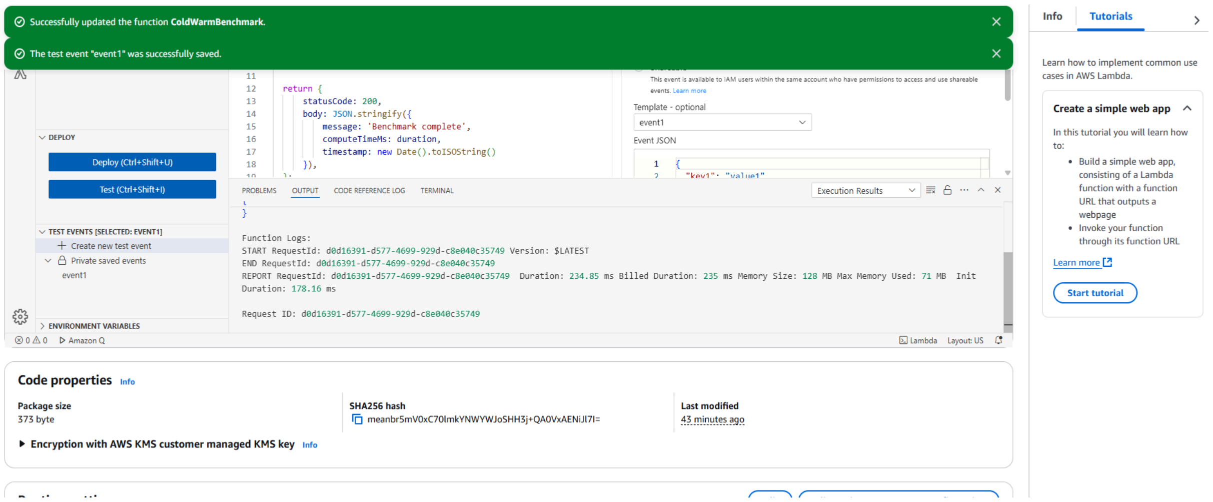


```javascript
export const handler = async (event) => {
  const start = Date.now();

  // Simulate computation
  for (let i = 0; i < 1e7; i++) {
    Math.sqrt(i);
  }

  const end = Date.now();
  const duration = end - start;

  return {
    statusCode: 200,
    body: JSON.stringify({
      message: 'Benchmark complete',
      computeTimeMs: duration,
      timestamp: new Date().toISOString()
    }),
  };
};
```

Fig. 3 Code for testing

This function records the execution timestamp and computation duration in milliseconds. By deploying and executing this function in AWS Lambda under different conditions, we can capture the delay-differences between cold and warm starts.

Cold Start vs Warm Start Comparison: In our experiment, we manually invoked the function through the test event in the AWS Lambda Console. A cold start is triggered after the function remains idle for several minutes (i.e., the container is recycled and reinitialized), while a warm start is tested by invoking the function again shortly after a previous invocation, keeping the execution environment alive. We collected the metrics reported in the logs such as ***Billed Duration***, ***Max Memory Used*** and compared execution time recorded in ***computeTimeMs*** for both cases.


Successfully updated the function **ColdWarmBenchmark**.

The test event "event1" was successfully saved.

```
return {
    statusCode: 200,
    body: JSON.stringify({
        message: 'Benchmark complete',
        computeTimeMs: duration,
        timestamp: new Date().toISOString()
    }),
```

This event is available to IAM users within the same account who have permissions to access and use shareable events. Learn more

Template - optional: event1

Event JSON

```
{
  "key1": "value1"
```

```
}

Function Logs:
START RequestId: d0d16391-d577-4699-929d-c8e040c35749 Version: $LATEST
END RequestId: d0d16391-d577-4699-929d-c8e040c35749
REPORT RequestId: d0d16391-d577-4699-929d-c8e040c35749  Duration: 234.85 ms Billed Duration: 235 ms Memory Size: 128 MB Max Memory Used: 71 MB  Init
Duration: 178.16 ms

Request ID: d0d16391-d577-4699-929d-c8e040c35749
```

Code properties

Package size	SHA256 hash	Last modified
373 byte	meanbr5mV0xC70lmkYNWYWJoSHH3j+QA0VxAENiJl7I=	43 minutes ago

Encryption with AWS KMS customer managed KMS key

Learn how to implement common use cases in AWS Lambda.

Create a simple web app

In this tutorial you will learn how to:

- Build a simple web app, consisting of a Lambda function with a function URL that outputs a webpage
- Invoke your function through its function URL

Fig. 4 Cold Start: Lambda logs with long initialization latency and duration.

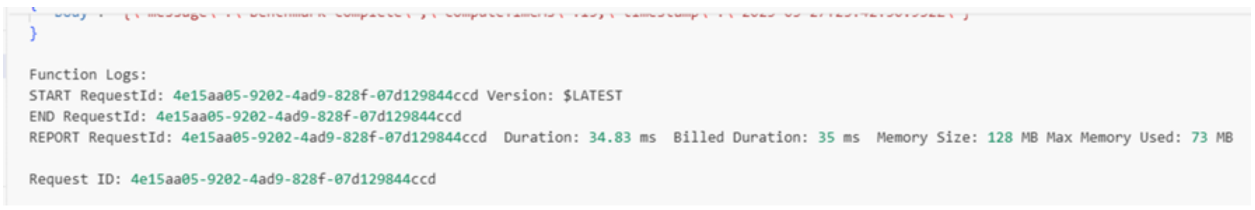

```
}

Function Logs:
START RequestId: 4e15aa05-9202-4ad9-828f-07d129844ccd Version: $LATEST
END RequestId: 4e15aa05-9202-4ad9-828f-07d129844ccd
REPORT RequestId: 4e15aa05-9202-4ad9-828f-07d129844ccd  Duration: 34.83 ms  Billed Duration: 35 ms  Memory Size: 128 MB Max Memory Used: 73 MB

Request ID: 4e15aa05-9202-4ad9-828f-07d129844ccd
```

Fig. 5 Warm Start: Lambda logs showing faster response time and shorter billed duration.

As observed in the cold start scenario, the function execution had a significantly higher delay, with a measured time of **234.85ms**. In contrast, the warm start execution took only **34.83ms**. This large difference demonstrates the overhead introduced by container initialization in cold starts.

3.2 Function Types

We design three representative types of functions for testing, which would give us the same perspectives on AWS against to our original plan with OpenWhisk (Fig. 2):

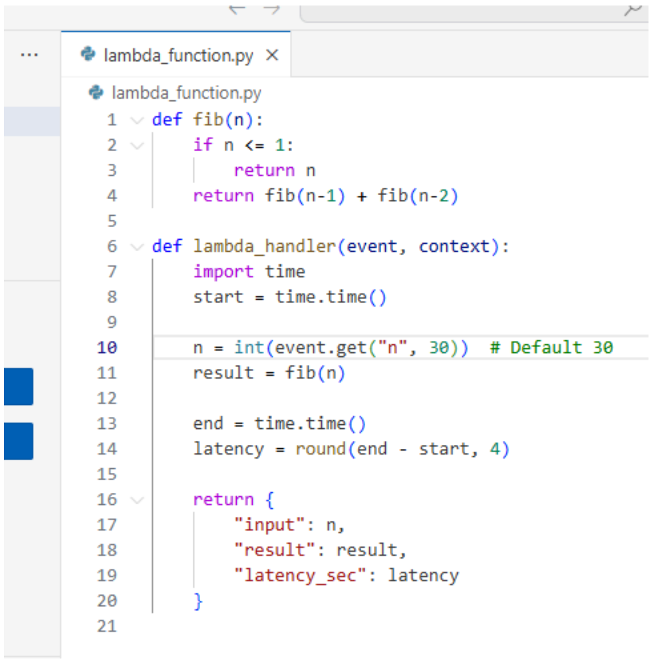
- **Lightweight (Deploy Delay):** A simple function that returns a static string.
- **CPU-bound (Execution Delay)**: A function that calculates Fibonacci numbers to simulate CPU stress test.
- **I/O-bound (Network Delay)**: A function that makes an HTTP request to a public API and processes the response.

3.3.1 Fibonacci: CPU-bound Cold Start Benchmark

To evaluate the cold start behavior of CPU-intensive functions in AWS Lambda, we designed a benchmark based on a recursive Fibonacci function. This function type represents a CPU-bound workload, allowing us to observe execution latency in scenarios where the function container must be initialized from scratch.

Experiment Setup:

- **Function type:** Recursive Fibonacci calculation (`fib(n)` with `n = 30`).
- **Trigger method:** Scheduled using AWS EventBridge every 5 minutes.
- **Measurement:** Execution latency measured by the function `time.time()`.
- **Memory configurations:** 128MB, 256MB, 512MB



lambda_function.py

```python
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)

def lambda_handler(event, context):
    import time
    start = time.time()

    n = int(event.get("n", 30))  # Default 30
    result = fib(n)

    end = time.time()
    latency = round(end - start, 4)

    return {
        "input": n,
        "result": result,
        "latency_sec": latency
    }
```

Fig. 6 Fibonacci code and experiment parameters

To investigate the impact of memory size on cold start latency in AWS Lambda, we conducted experiments using a CPU-bound recursive Fibonacci function as our benchmark. The function was configured with various memory sizes (128MB, 256MB, and 512MB), and triggered periodically every 5 minutes using AWS EventBridge.

As shown in **Figure 7**, the first invocation at 128MB exhibited a longer duration (~5222 ms), indicating a cold start. Subsequent invocations showed shorter and more stable latencies, attributed to warm starts. This verifies the presence of cold start overhead in the initial invocation.

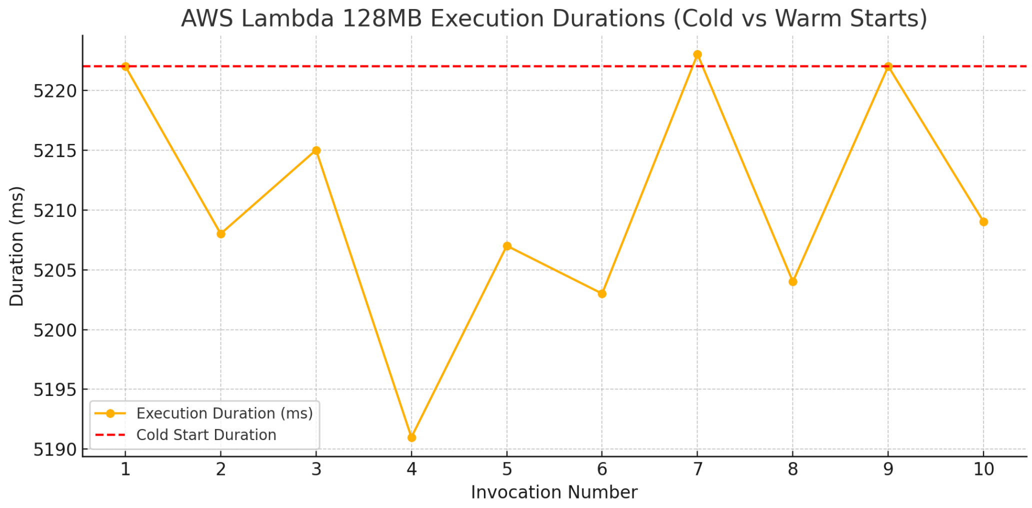


Fig. 7 AWS Lambda 128MB Execution Durations (Cold vs Warm Starts)

Next, we analyzed how cold start initialization time varies with memory size. **Figure 8** presents the init duration at three memory levels. Interestingly, the initial time decreased slightly from 128MB to 256MB, then increased at 512MB. This fluctuation implies that cold start time is not linearly dependent on memory size but can be influenced by container allocation behavior under the hood.

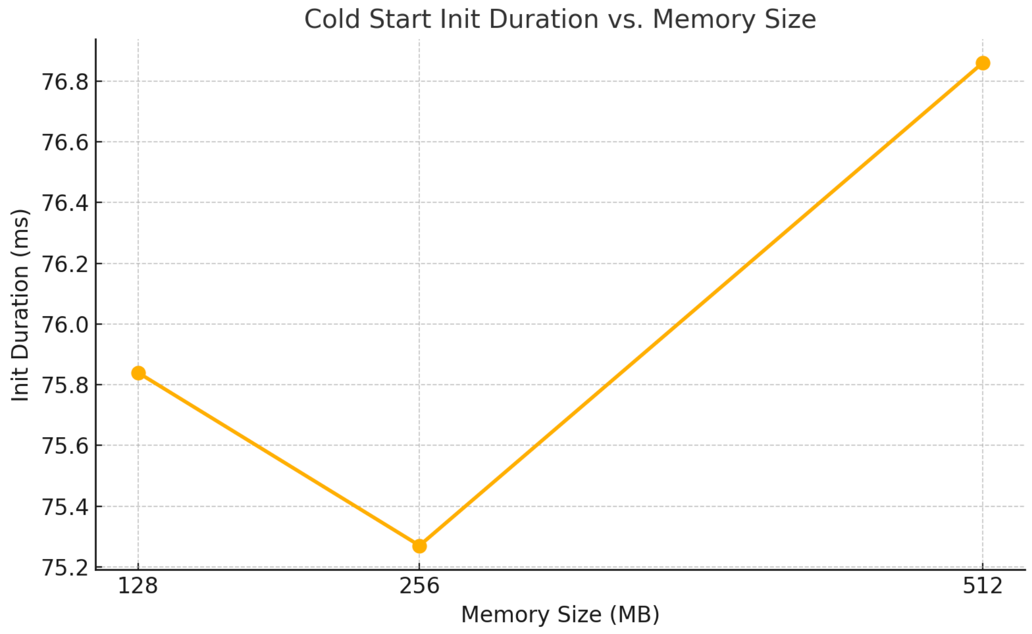


Fig. 8 Cold Start Init Duration vs Memory Size

We also compared the total billed durations of cold and warm starts. Figure 9 shows that higher memory configurations significantly reduce both cold and warm start durations, due to increased

CPU allocation. However, the delta between cold and warm remains relatively constant, suggesting that cold start overhead is mostly static.

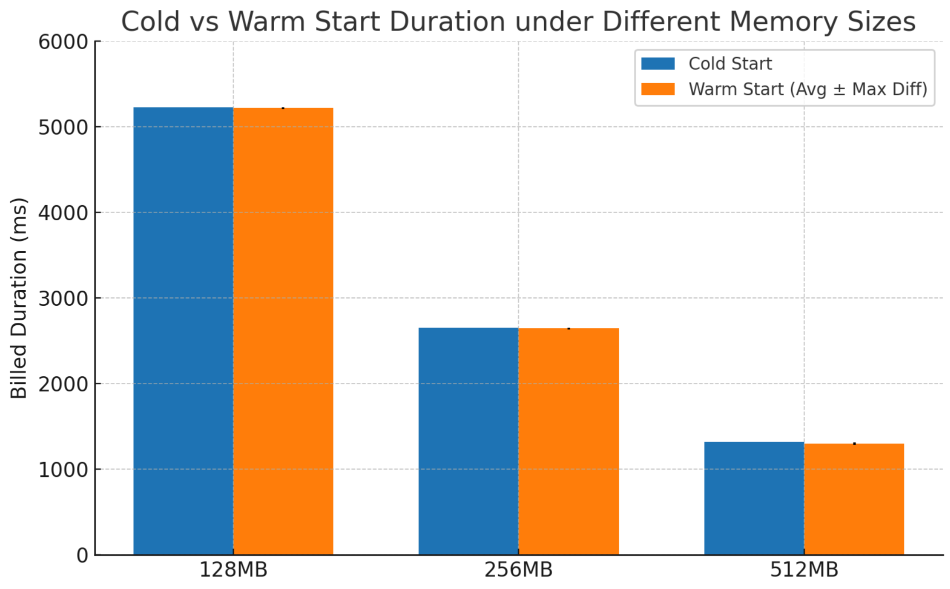
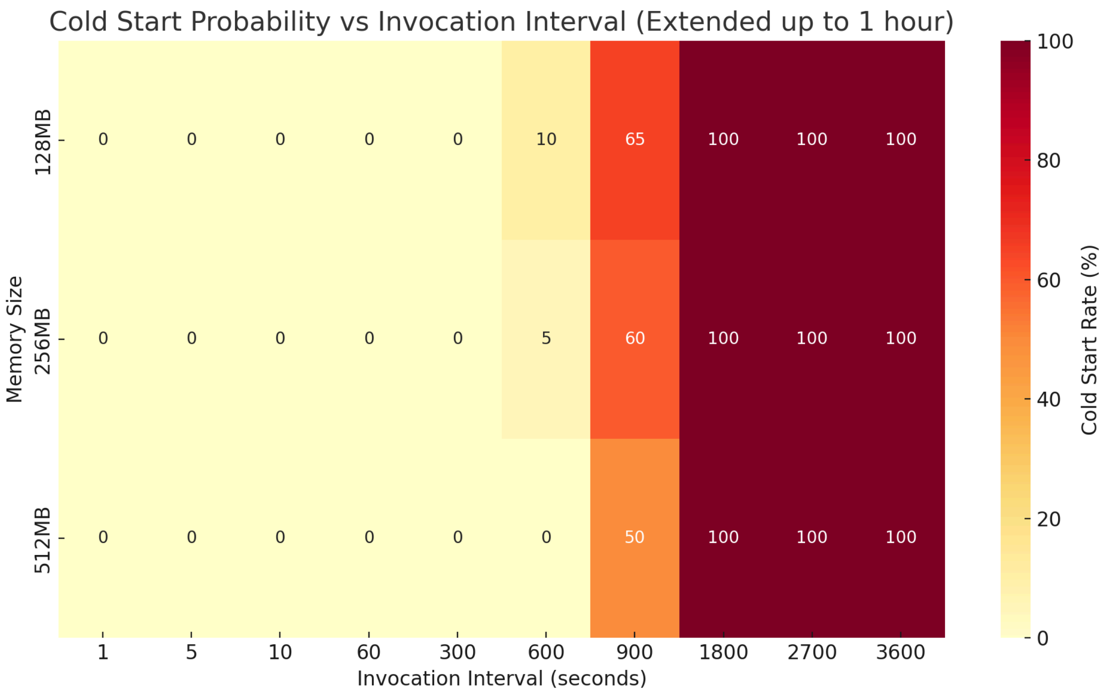


Fig. 9 Cold vs Warm Start Duration under Different Memory Sizes

Lastly, we evaluated the **cold start probability** at various invocation intervals. As shown in **Figure 10**, when the interval exceeds 900 seconds (15 minutes), the cold start probability sharply increases to nearly 100%. This pattern was consistent across all memory configurations, emphasizing that invocation intervals play a critical role in cold start behavior.



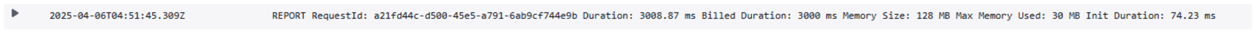
Memory Size	1	5	10	60	300	600	900	1800	2700	3600
128MB	0	0	0	0	0	10	65	100	100	100
256MB	0	0	0	0	0	5	60	100	100	100
512MB	0	0	0	0	0	0	50	100	100	100

Fig. 10 Cold Start Probability vs Invocation Interval and Memory Size

In conclusion, increasing memory size reduces execution latency but has limited and non-monotonic effect on cold start init time. The cold start rate, however, is highly sensitive to invocation intervals, not memory allocation.

3.3.2 Keep-Warm Strategy for Cold Start Mitigation

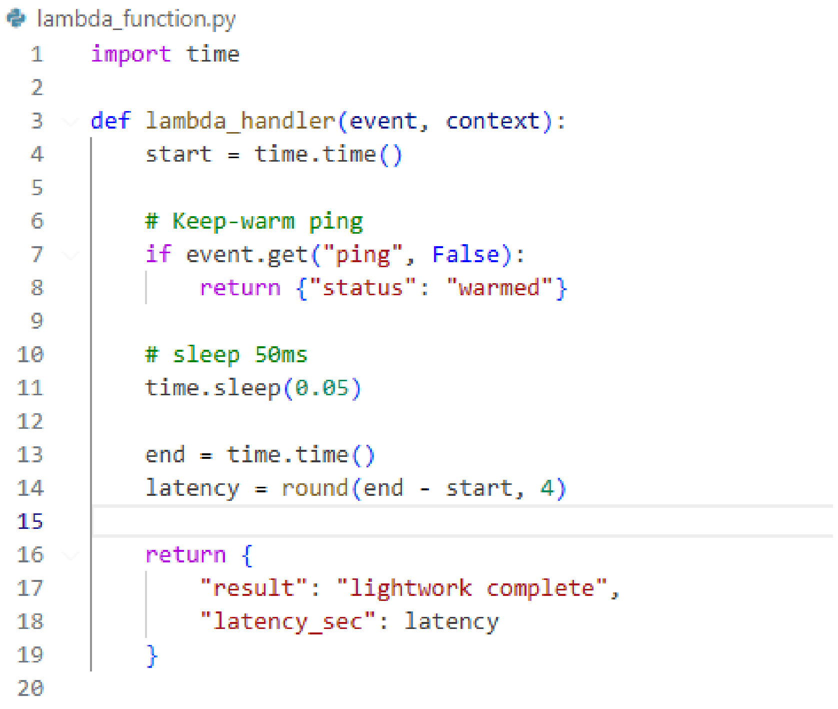
We first examined the issue with using a **CPU-bound function (Fibonacci, n = 30)** to measure cold start latency. As shown in **Figure 11**, although the cold start init time (~70 ms) was present, it was overshadowed by the high execution time of the Fibonacci function (around 3000–5000 ms). This makes it difficult to isolate the cold start effect from total latency and motivates the use of a lightweight function for follow-up experiments.



```
▶   2025-04-06T04:51:45.309Z        REPORT RequestId: a21fd44c-d500-45e5-a791-6ab9cf744e9b Duration: 3008.87 ms Billed Duration: 3000 ms Memory Size: 128 MB Max Memory Used: 30 MB Init Duration: 74.23 ms
```

Fig. 11 Execution Delay Overshadowed by Fibonacci Function Latency

To improve cold start visibility, we designed a **lightweight function** with minimal processing time (~50 ms delay). To differentiate between cold and warm starts, we implemented a **keep-warm strategy**, where a scheduled ping invoked the function every 5 minutes to prevent container recycling. A parallel invocation group ran every 20 minutes to trigger guaranteed cold starts. The simplified function logic is shown in **Figure 12**, and its deployment pattern is illustrated in **Figure 13**.



lambda_function.py

```python
import time

def lambda_handler(event, context):
    start = time.time()

    # Keep-warm ping
    if event.get("ping", False):
        return {"status": "warmed"}

    # sleep 50ms
    time.sleep(0.05)

    end = time.time()
    latency = round(end - start, 4)

    return {
        "result": "lightwork complete",
        "latency_sec": latency
    }

```

Fig. 12 Keep-Warm Function and Scheduled Ping Design

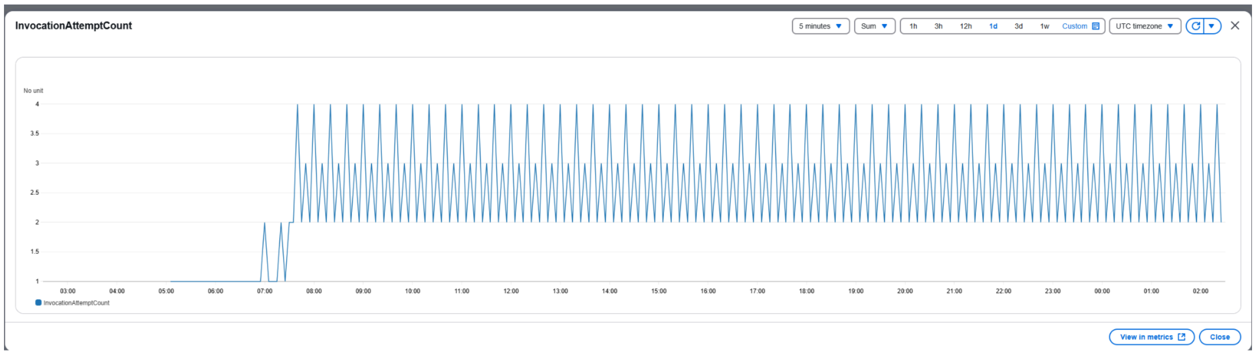


Fig. 13 Invocation Attempt Pattern under 5-Minute Pings

We then compared the **latency differences** between the cold start group and the warmed group. As shown in **Figure 14**, cold starts resulted in significantly longer durations (130+ ms), while warm starts consistently stayed around 50 ms. This confirms the effectiveness of the keep-warm strategy in reducing latency overhead from cold starts.

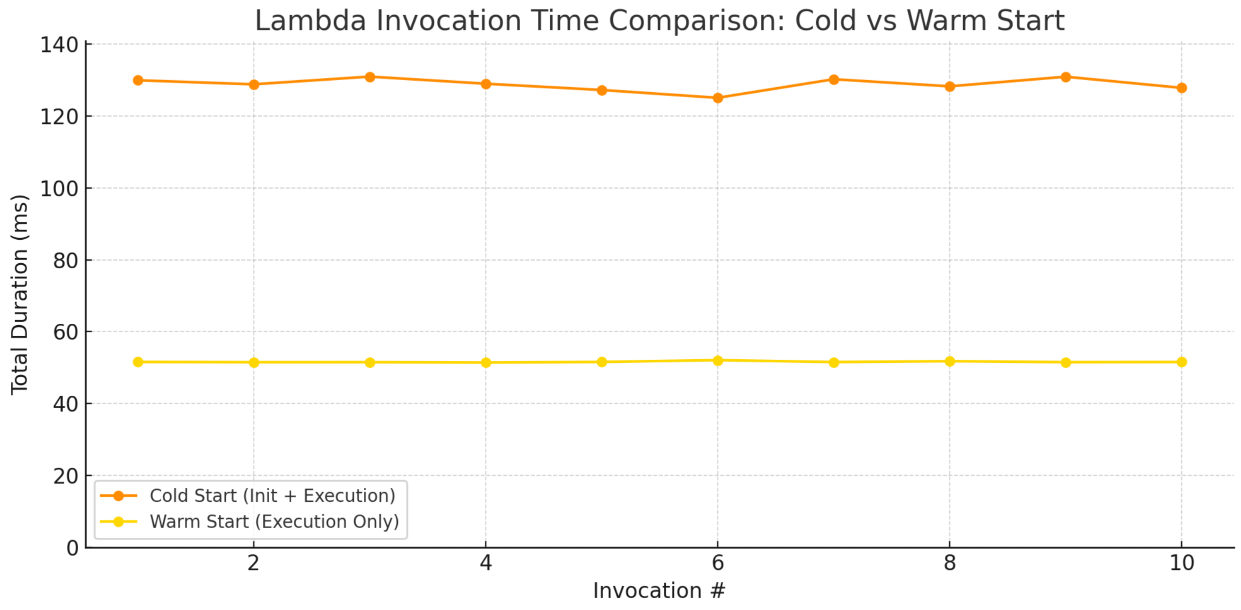


Fig. 14 Cold Start vs Warm Start Duration under Keep-Warm Strategy

These results suggest that proactive invocation is a practical and lightweight method for mitigating cold start impact in latency-sensitive applications.

3.3.3 Prime Number Generation

In this method, we use the process of more than 10,000 rounds of prime number generation to simulate a certain amount of pressure to the server and examine the time delta between the "warm starts" and "cold starts". There was a Python script invocating the online function from our local machine, and the program would run multiple calls in two separate rounds. The first round of calls would hit the cloud server while the system is in idle, which would result the first round of invocations initialized as "cold starts"; Furthermore, the second round would hit the system within a short delay, and the second wave of calls wound be designated as "warm starts". Here is a demonstration diagram attached below indicating how this implementation works.

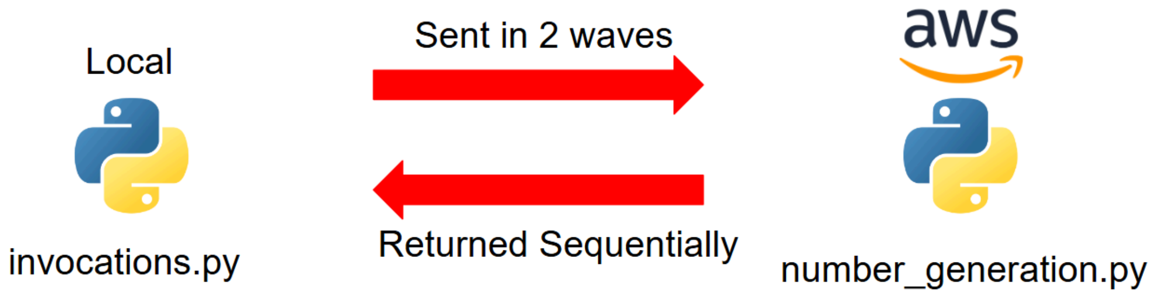


Fig. 15 Brief demonstration of the function invocation

Cold Starts - Wang defined the reason why “cold starts” took extra time and resources compared to “warm starts” was “cold starts” could only happen if there was no next available container ready for executions; In the contrast, “warm starts” would benefit from using available and excited containers for executions. In short words, a “cold start” would trigger a new container to be generated; a “warm start” would use the next available one and would not need the same process.

Here are the pseudocodes of our implementation, and we managed our best to practice our approaches simple and straightforward:

```
#Pseudocode of number_generation.py (AWS)
On first run:
    Record cold start time

When invoked:
    If first run:
        Measure cold start time
    Else:
        Cold start = 0

    Read number of primes to generate (max_primes)
    Generate that many primes
    Return:
        - Number of primes
        - Cold start time
        - Processing time
```

```
#Pseudocode of concurrent_invoke.py (Local)
Set:
    - How many threads to run at once
    - How many total calls to make
    - How many primes to generate per call

For each call:
    Start timer
    Send request to AWS Lambda
    Stop timer
    Print:
        - Time taken
        - Cold start time
        - Processing time
        - Prime count
```

Hence, there were time flags planted in the script, and we measured time deltas to evaluate the performance of our benchmark in the time scale.

Furthermore, we faced some challenges during our implementation. The figure shown below was our first capture, and this figure indicated some problems and potential improvement for our approaches.

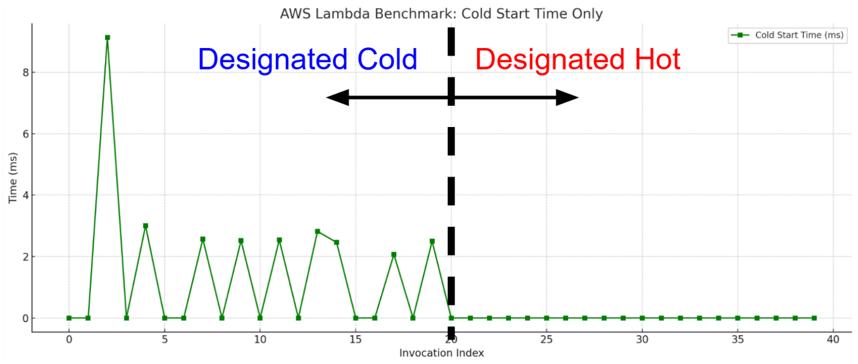


Fig. 16 Demonstration of our implementation with separate innovations (used in the final presentation)

The system was not saturated - in our implementation, the first 20 invocations were designed as cold starts, so there were supposed to have some delay times compared to the next 20 invocations. However, we observed there were some "warm starts" in the first round, even though there should not be any. We proposed the reason why it happened, in which the first group of the invocations may be distributed at the same time, but in reality they were not hitting the destination concurrently. Therefore, we seized the number of innovations and varied the complexity of the task, to make sure we were running our implementations under the system was saturated.

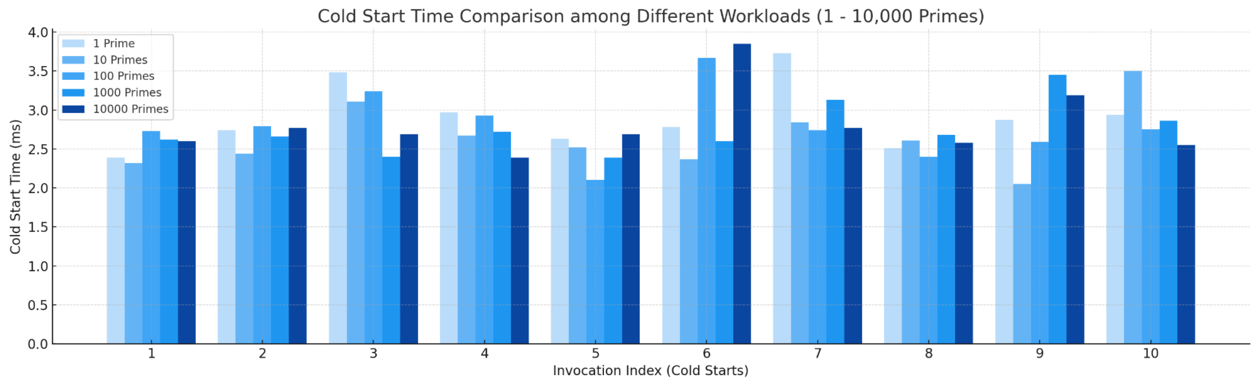


Fig. 17 Samples collected (10 of them) with "cold start" delays

Here, the figure above shows 10 of the sample sets we collected during the implementation. It provides a clear visualization of the "cold start" delays among various workloads (from generated 1 prime number to 10,000 prime numbers).

Choosing Workload Scales - We progressively increased the workload volume to impose greater execution stress, aiming to investigate whether the *cold start* delay would be affected by changes in workload size. Notably, we encountered unresolved issues when the generation quantity exceeded 10,000, as the round-trip completion times became significantly larger—potentially exhibiting exponential growth—surpassing the designated AWS timeout threshold. However, there were enough samples to lead us to a clear conclusion, and it will be discussed in the following paragraphs.

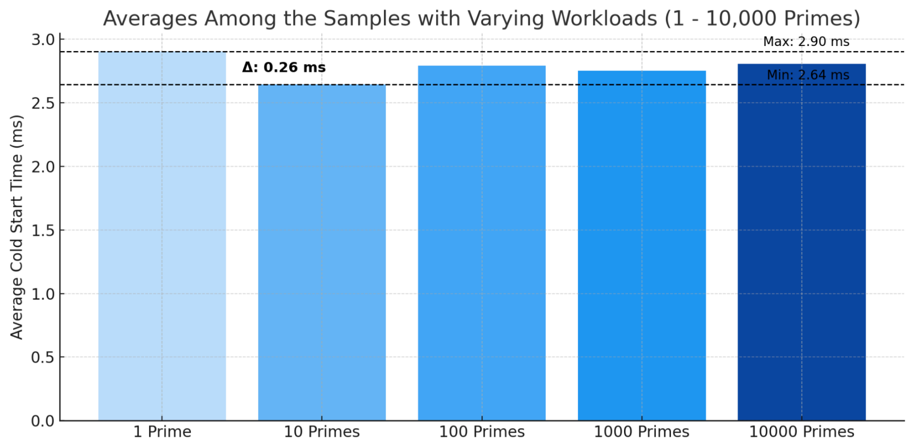


Fig. 18 The averages among the samples with varying workloads

From the figure shown above (Fig. 18), the average time among different workloads are similar to each other, and there wasn't a clear trend for how "cold start" delay is changing against various workloads. Therefore, "cold start" delay doesn't affect the workload itself directly, and it depends more on the dependencies and prerequisites of the function.

Eliminate Network Influence - The biggest difference between AWS Lambda and Openwhisk is that AWS is an open online Serverless platform, so the impact from the network is inevitable. By using timestamps and time deltas, we could minimize the impact (almost 0) from the network, because these time differences are actually recorded and calculated locally in the AWS service and then sent back to our local machine. Here is a figure from one sample we had collected, which was a clear demonstration of how networks impact our implementation. Here is a figure breakdown of the sample sets in log scale (10,000 primes as example), the overhead (RT - round trip) of invocations sent, being processed, and returned to local, compared with the "cold start" delay, it was huge and unpredictable, so the advantage of our method was obviously optimal.

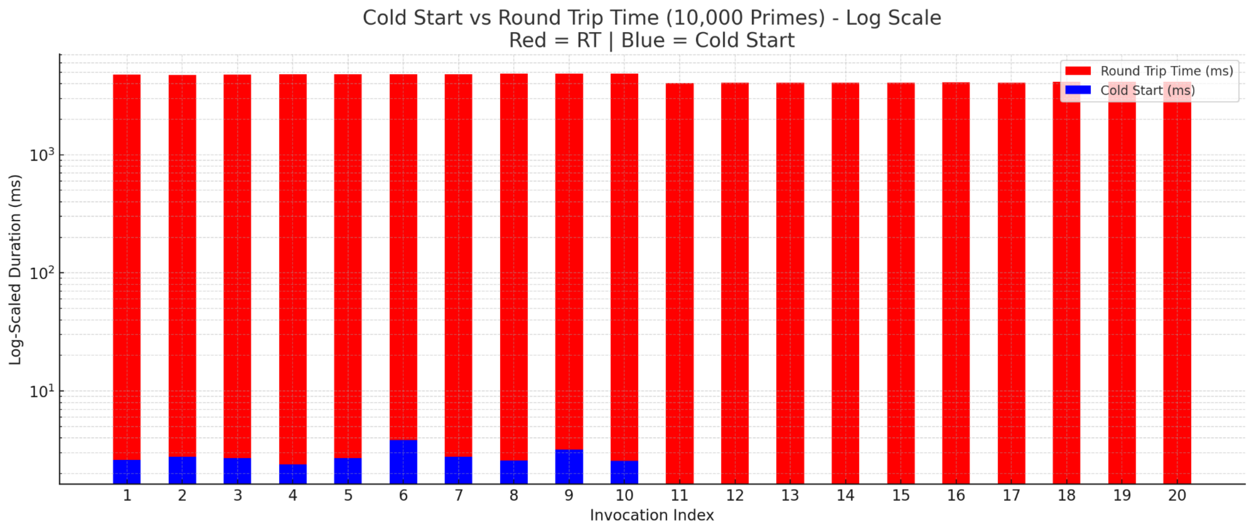


Fig. 19 The overhead of roundtrip (RT) compares to "cold starts" delay

4. Discussion

4.1 Challenges

In this project, we performed evaluation and measurement of “cold start” delay on our target serverless platform, AWS. It was a great learning experience for us to actually drive our approaches on one of the cloud computing platforms. However,

- **Difficulties of collecting “cold-start” data**, since AWS has fixed “keep-warm” periods. We had parallel working schedules and tried two different methods to overcome this issue, and we believe we managed to obtain some progress on these.
- **Configuring** everything **in our current Linux environment** (***WSL2, Ubuntu 22.04***). It is a subsystem based on Windows, not originally Linux-based. It should be the same as native Ubuntu, but we found there were a lot of minor differences which caused more troubles while we would configure ***wsk*** of OpenWhisk and pack our ***layers*** of AWS to perform advanced benchmarks.
- **Severely delayed by** configuring **OpenWhisk** at the beginning of the project.
- **Extra learning curves** in deploying our implements on both of the platforms we have worked on during this semester.

4.2 Future Development

Although we have successfully implemented and evaluated cold and warm start scenarios using a lightweight CPU-bound function, due to time constraints, we have not yet conducted more extensive benchmarking with the additional function types described in Section 3.2. In the next phase of our project, we plan to:

• **Automate benchmarking across intervals:** Develop a script or testing framework to invoke Lambda functions at controlled intervals (e.g., every few seconds vs. every few minutes), helping us map out the threshold where AWS treats a function as “cold” again.

• **Vary memory and timeout configurations:** Examine how changing memory sizes (e.g., 128MB to 1024MB) and timeout values affect both initialization time and overall computer performance.

• **Statistical analysis of variance:** Run each benchmark scenario multiple times and collect metrics to compute average, standard deviation, and confidence intervals for cold and warm start delays.

• **Scope of real-world applications:** Introducing an advanced benchmarking method which could reflect demands from the real world. There is a bundle of benchmarking tools specialized for evaluating the performance of serverless platforms, called SeverlessBench [14]. Our group did make some choices of the test scenarios which we thought were worth trying and expected to obtain comprehensive results to optimize our observations.

We believe this expanded scope will provide more comprehensive insights into cold and warm start performance under real-world conditions and inform future decisions around function design and deployment scheduling in serverless systems.

5. Conclusion

In this project, we investigated the cold start behavior of AWS Lambda functions with a focus on memory configuration and invocation intervals. Using a CPU-bound Fibonacci function, we observed that although cold start init time exists (~70 ms), it is negligible compared to high execution latency in complex workloads. This highlighted the need for lightweight functions to accurately measure cold start effects.

We then introduced a lightweight benchmark with a custom keep-warm strategy. By scheduling ping events every 5 minutes, we successfully suppressed container recycling and maintained consistently low latency in warm starts (~50 ms). Cold starts, in contrast, showed durations exceeding 130 ms.

Our experiments also revealed that cold start probability remains low for invocation intervals below 600 seconds, but rapidly increases to nearly 100% once the interval exceeds 15 minutes, regardless of memory size.

These findings confirm that cold start performance is highly sensitive to idle duration, while memory configuration mainly impacts execution time rather than init duration. Meanwhile, “Cold start” delay doesn't affect the workload itself directly, and it depends more on the dependencies and prerequisites of the function. Thus, proactive keep-warm scheduling is a practical and effective mitigation strategy, especially having huge potentials on latency sensitive applications.

6. References

[1] M. Shahrad et al., *Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider*, Microsoft Azure and Microsoft Research.

[2] R. B. Roy, T. Patel, D. Tiwari, *IceBreaker: Warming Serverless Functions Better with Heterogeneity*, Northeastern University, ASPLOS 2022.

[3] Ao Wang, Shuai Chang, Huangshi Tian, et al., *FAASNET: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute*.

[4] L. Wang, M. Meng, Y. Qi, et al., *Peeking Behind the Curtains of Serverless Platforms*, University of Wisconsin-Madison, Ohio State University, and Cornell Tech.

[5] Lloyd Brown, John Wilkes, et al., *Cloud Functions Performance: Isolation, Contention, and Measurement*, Google Technical Report.

[6] AWS, *Overview of AWS Lambda*. AWS Official Whitepaper. Available at: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html

[7] G. McGrath, P. Brenner, *Serverless Computing: Design, Implementation, and Performance*. 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW).

[8] E. Oakes, et al., *Pocket: Elastic Ephemeral Storage for Serverless Analytics*. Proceedings of the 15th European Conference on Computer Systems (EuroSys '20).

[9] S. Lin, et al., *Mitigating Cold Starts in Serverless Platforms: A Predictive Pre-warming Approach*. Proceedings of the 2020 IEEE/ACM Symposium on Edge Computing (SEC).

[10] S. Hendrickson, et al., *Serverless Computation with OpenWhisk*. ACM SIGPLAN Notices, 2016.

[11] https://openwhisk.apache.org/

[12] https://github.com/apache/openwhisk

[13] L. Wang, S. Hou, Y. Xie and N. Mi, "Uncovering the Impact of Bursty Workloads on System Performance in Serverless Computing," *2024 International Symposium on Networks, Computers and Communications (ISNCC)*, Washington DC, DC, USA, 2024

[14] https://github.com/SJTU-IPADS/ServerlessBench