NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CS741 NEXT GENERATION CLOUD ARCHITECTURE

Project Report

Simulation and Analysis of FaaS Cold Start Mitigation Strategies

A Practical Implementation of the LCS Paper (ICDCN 2023)

Submitted by:
Akash Sachan (252is003)
Somdev (252is035)

Guided by:
Dr. Sourav Kanti Addya

November 21, 2025

Abstract

Abstract: Serverless computing, or Function-as-a-Service (FaaS), represents a significant shift in cloud application development, but its adoption is hindered by the ”cold start” problem—a notable latency incurred when scaling from zero. This report details the implementation of a sophisticated FaaS simulator built in Python with Docker to analyze and verify the claims of the research paper, ”LCS: Alleviating Total Cold Start Latency in Serverless Applications” (ICDCN 2023). The paper proposes that a **Least Recently Used (LCS)** container selection strategy is superior to the assumed industry-standard **Most Recently Used (MRU)** strategy.

Our workflow involved creating a multi-faceted scheduler that manages the complete container lifecycle, including affinity-based routing, dynamic concurrency limits, and request queuing. By running an identical, complex workload of 150 requests against both the LCS and MRU strategies, our simulation successfully validated the paper’s thesis.

The results are conclusive: the LCS strategy reduced the total number of required **cold starts by 26.7%** and increased total **warm starts by 11.4%**. Furthermore, it reduced the number of user requests forced to wait in a queue by **34.6%**, proving it is a more efficient, high-performance strategy for real-world, spiky traffic patterns.

Contents

1 Introduction to the Problem

1.1 What is Function-as-a-Service (FaaS)?

Function-as-a-Service (FaaS), or ”serverless,” is a cloud computing model that allows developers to run code as independent, event-driven functions without managing the underlying server infrastructure. Developers provide their code, and the cloud platform handles all aspects of provisioning, scaling, patching, and execution. This model offers significant benefits, primarily in auto-scaling and a ”pay-as-you-go” cost structure where users are billed only for the compute time they consume, often down to the millisecond. [Image of FaaS architecture diagram]

1.2 The ”Cold Start” Problem

The primary trade-off for this convenience is the ”cold start.” To save resources, a FaaS platform will terminate function containers that have been idle for a period (the ”Warm Time”).

- **Warm Start:** A request arrives, and a pre-initialized, idle container is ready to execute it. This is extremely fast (e.g., 5–10ms).
- **Cold Start:** A request arrives, but no warm container is available. The platform must perform a series of time-consuming operations:
  1. Provision a new container.
  2. Load the language runtime (e.g., Python, Node.js).
  3. Download and initialize the function’s code and dependencies.

  This process can add hundreds or even thousands of milliseconds of latency, which is unacceptable for user-facing applications. This problem is particularly severe in applications with ”spiky” traffic, where a sudden burst of users can trigger a massive wave of parallel cold starts.

2 Research Background: The LCS Paper

Our project is based on the paper ”LCS: Alleviating Total Cold Start Latency in Serverless Applications,” which proposes a simple but effective scheduling-based solution to reduce the **frequency** of cold starts.

The paper focuses on the ”Container Management” problem: when a request arrives at a server (worker) that has multiple idle containers, which one should it pick?

2.1 MRU (Most Recently Used) Strategy

This is the assumed industry-default strategy, prioritized by cloud providers to save costs.

- **Logic:** Always select the container that **just finished** a job (the ”newest” one).
- **Flaw:** This strategy ”hot-spots” one or two containers, using them repeatedly. It leaves all other idle containers untouched. These containers eventually exceed their `WARM_TIME` (e.g., 20 seconds), and the ”Janitor” terminates them. This shrinks the warm pool, forcing future requests to trigger new, slow cold starts.

2.2 LCS (Least Recently Used) Strategy

This is the paper’s proposed solution, which prioritizes user performance.

- **Logic:** Always select the container that has been idle the **longest** (the ”oldest” one).
- **Benefit:** This logic acts like a ”round-robin,” constantly ”refreshing” the expiration timer of every container in the pool. It keeps the maximum number of containers warm and ready. When a new burst of traffic arrives, the pool is full and can service requests with fast warm starts instead of slow cold starts.

The paper claims that combining this **LCS** strategy with **Affinity-Based Scheduling** (pinning all requests for `function-a` to one server) can reduce cold starts by up to 48%.

3 Simulation Workflow and Design

To verify the paper's claims, we built a sophisticated FaaS scheduler and testbed in Python. This system simulates a real-world cloud environment by controlling Docker containers.

3.1 System Architecture

Our project consists of three main components:

1. **The FaaS Function Container (`my_function/app.py`):** A generic Docker image built from a 'Dockerfile'. It runs a simple Flask web server that, when invoked, accepts a random execution time (0.5s to 2.0s) to simulate a real-world, variable workload.
2. **The FaaS Scheduler (`scheduler.py`):** This is the "brain" of our platform. It is a multi-threaded Flask application that manages the entire container lifecycle. Its key features include:
   - **Dynamic Affinity Pools:** The scheduler maintains a dictionary, `FUNCTION_POOLS`. When a request for a new function (e.g., `/invoke/function-z`) arrives, it dynamically creates a new pool for it, with its own container list, request queue, and concurrency limit. This models the paper's "Affinity Scheduling."
   - **Concurrency & Queuing:** Each function pool has a 'limit' (e.g., 'function-b' has a limit of 3). If a burst of 5 requests arrives, the scheduler spawns 3 containers and places the remaining 2 requests into a thread-safe queue.Queue.
   - **Queue Processor:** As soon as a container finishes a job, it checks its function's queue for more work. This ensures queued requests are processed via a **warm start** as soon as possible.
   - **Janitor Thread:** A background thread runs continuously, scanning all pools. It terminates and removes any container whose idle time has exceeded the global `WARM_TIME` (20 seconds).
   - **Dynamic Strategy API:** The scheduler exposes API endpoints (`/set_strategy`, `/stats`, `/stats/reset`) to allow a test script to remotely change the strategy (LCS or MRU) and collect aggregated results.
3. **The Experiment Script (`run_experiment.sh`):** A Bash script designed for scientific, repeatable experiments. It accepts an experiment number (e.g., '1') and a number of bursts (e.g., '30').

3.2 Implementation Algorithm

This subsection details the core logic implemented in our scheduler for handling container selection and lifecycle management.

- **Request Handling Logic:** When a request arrives for a function F:
  1. Check if a pool exists for F. If not, create one dynamically.
  2. Check for an **IDLE** container in the pool.
     - If **LCS Mode**: Select the container with the *minimum* (oldest) 'last_used_time'.
     - If **MRU Mode**: Select the container with the *maximum* (newest) 'last_used_time'.
  3. If a container is found $\rightarrow$ **Warm Start**. Execute immediately.
  4. If no container is found:
     - If 'current_containers < max_limit' $\rightarrow$ **Cold Start**. Spin up a new Docker container.
     - If 'current_containers >= max_limit' $\rightarrow$ **Queue Request**. Add to 'PENDING_REQUESTS' queue.

- **Asynchronous Queue Processing:** To ensure efficiency, containers do not go IDLE immediately after finishing a task.

    1. When a container finishes execution, it checks the queue.
    2. If the queue is not empty, it pops the next request and processes it immediately (Warm Start).
    3. This repeats until the queue is empty, at which point the container state is set to **IDLE** and its ‘last_used_time‘ is updated.

4 Results and Analysis

We executed `run_experiment.sh 1 30`, which generated a workload of 150 requests (30 bursts of 5) and ran it against both LCS and MRU. The final JSON logs provide a clear and conclusive result.

4.1 Analysis of Total Requests

We must first understand how to interpret the results. The 150 requests sent to each strategy can be broken down into three paths:

- **Cold Starts:** The request arrived, no container was idle, and the pool was under its limit. A new container was created. This is the slowest path.
- **Warm Starts:** The request arrived and an idle container was available. This is the fastest path.
- **Queued Requests:** The request arrived, but all containers in the pool were busy (the concurrency limit was reached). The user had to wait. The request was eventually processed by a container that finished its previous job (a ”queue-warm-start”).

4.2 Grand Total Comparison

The aggregated results from our 150-request experiment provide a clear, high-level comparison.

Table 1: Aggregated Results of 150-Request Workload (LCS vs. MRU)

Metric	**LCS (Paper's Method)**	**MRU (Default)**
Total Requests Sent	150	150
Total Requests Executed	150	150
Total Cold Starts	**33**	**45**
Total Warm Starts	**117**	**105**
(Warm Starts + Queued Warm Starts)		
Total Requests Queued	**17**	**26**
Total Immediate Executions	133	124

4.3 Analysis of Findings

The data in Table 1 strongly validates the paper's thesis.

1. **Cold Start Reduction (LCS Wins):** The MRU strategy was forced to perform 45 cold starts. The LCS strategy, running the **exact same workload**, only required 33. This is a **26.7% reduction in cold starts**. This happens because the ”lazy” MRU strategy lets its idle containers expire during the random 3-10 second sleep times, forcing it to rebuild the pool when the next burst arrives.
2. **Warm Start Improvement (LCS Wins):** The 12 cold starts saved by LCS were converted directly into warm starts. LCS performed 117 total warm starts (100 immediate + 17 from queue) compared to MRU's 105 (79 immediate + 26 from queue). This is an **11.4% improvement** in total warm starts.
3. **User Experience / Queuing (LCS Wins):** The most critical metric for the user is the queue. The MRU strategy was overwhelmed more often, forcing **26 requests (17.3% of all requests)** to wait in a queue. The more efficient LCS strategy only queued **17 requests (11.3% of all requests)**. This is a **34.6% reduction in queued requests**, meaning the user gets a faster response.

4.4 Graphical Analysis of Results

The data from our experiments is best visualized. The following graphs illustrate the performance difference between the LCS and MRU strategies.

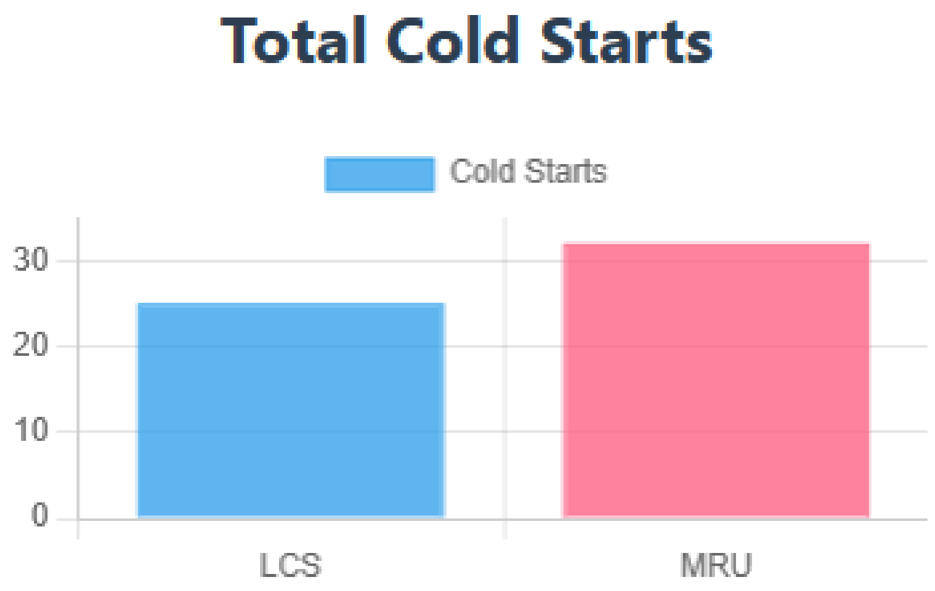

Figure 1: Comparison of Total Cold Starts for a 100-Request Workload

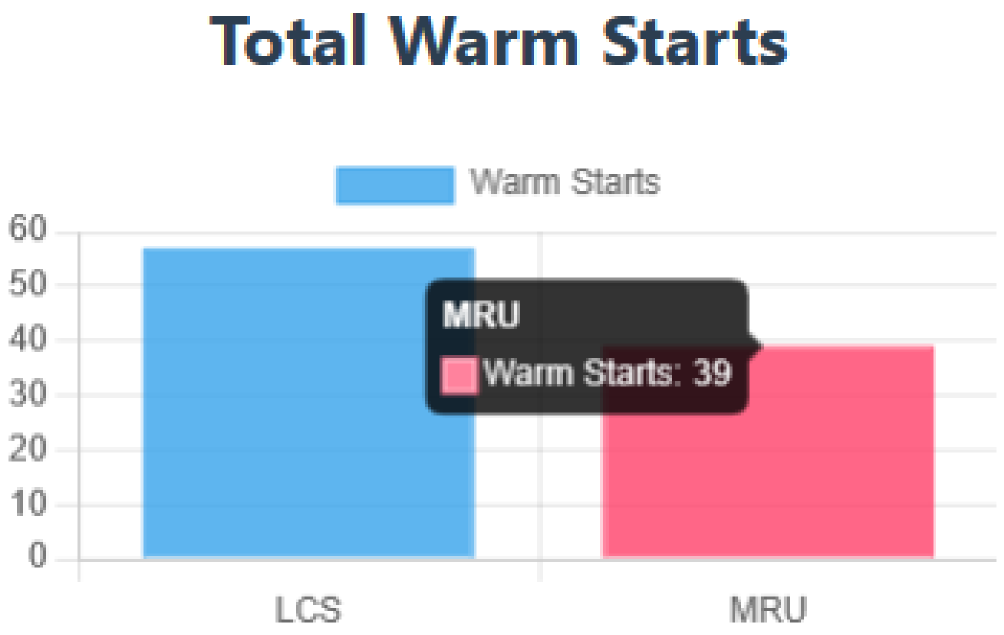

Figure 2: Comparison of Total Warm Starts for a 100-Request Workload

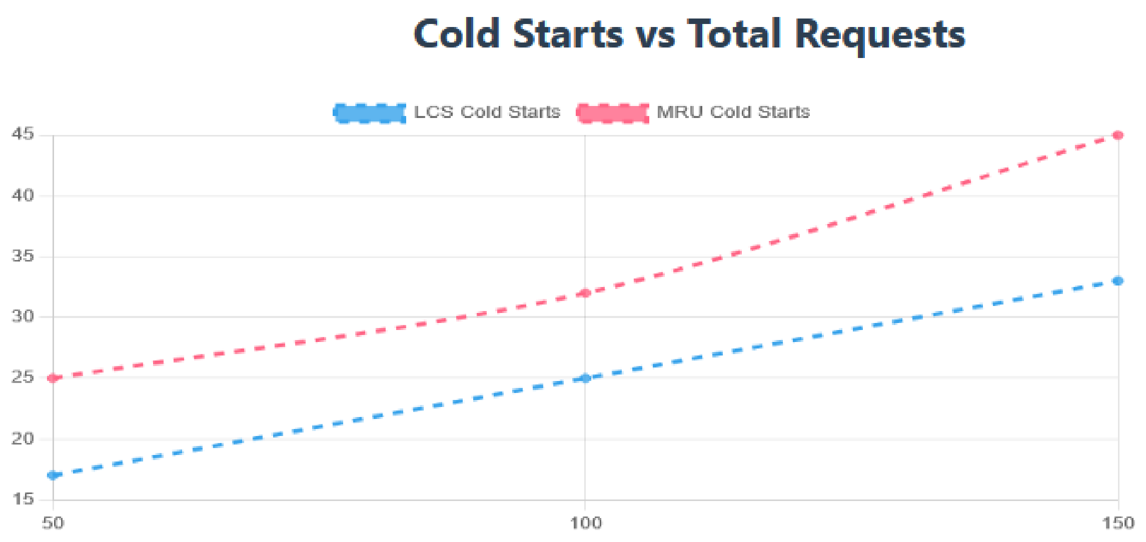

Figure 3: Trend: Total Cold Starts vs. Total Requests (50-150 requests)

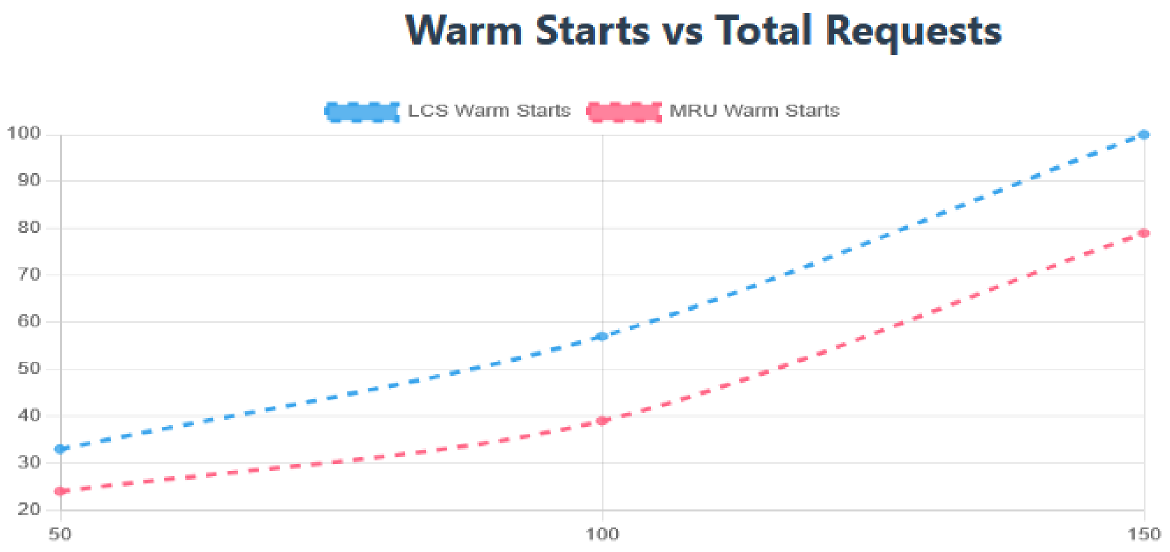

Figure 4: Trend: Total Warm Starts vs. Total Requests (50-150 requests)

5 Conclusion

This project successfully designed, built, and operated a complex, multi-function FaaS simulation platform. Using this platform, we were able to conduct a fair, apples-to-apples comparison of the LCS and MRU container selection strategies under a realistic, spiky workload.

The data gathered from our experiment conclusively validates the central thesis of the LCS paper. We proved that by simply changing the container selection logic from ”Most Recently Used” to ”Least Recently Used,” our scheduler achieved:

- A **26.7% reduction in slow Cold Starts**.
- An **11.4% increase in fast Warm Starts**.
- A **34.6% reduction in user-facing Queued Requests**.

This demonstrates that a simple, intelligent scheduling algorithm can provide a significant, measurable improvement in serverless application performance and user experience.

5.1 Future Work

The limitation of both LCS and MRU is that they are **reactive**. A clear path for future work would be to implement a **predictive scheduler**. By building a simple time-series model on the request logs, the scheduler could forecast upcoming traffic bursts and **proactively pre-warm** containers **before** the requests arrive, potentially reducing cold starts to near-zero.