RadSched: A Latency Optimizing Scheduler for Stateful Serverless Edge Computing

Jonathan Mindel

Advisor: Professor Wyatt Lloyd

Submitted in partial fulfillment

of the requirements for the degree of

Bachelor of Science in Engineering

Department of Computer Science

Princeton University

April 2025

I hereby declare that I am the sole author of this thesis.

I authorize Princeton University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Jonathan Mindel

I further authorize Princeton University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Jonathan Mindel

Abstract

This thesis presents and evaluates RadSched, a latency-optimizing scheduler designed for stateful, per function execution in a serverless edge computing environment. In distributed systems, where data consistency and latency vary by time and location, selecting the optimal edge location for function execution becomes a complex decision. Built on top of the Radical framework, RadSched maintains records of network conditions and learns from past data consistency outcomes to automate routing function requests to the optimal edge location. The system employs an ϵ-greedy exploration strategy to adapt to shifting network conditions and data availability, thereby ensuring responsiveness. Through empirical evaluation across multiple AWS regions, this thesis demonstrates that RadSched maintains comparable median latency to baseline systems in a stable environment, though with higher tail latency – a tradeoff that allows the system to route functions to a shifting optimal edge in volatile environments. Ultimately, by abstracting edge selection away from the client, RadSched both improves performance and simplifies developer interaction with stateful serverless functions on the edge.

Acknowledgements

I want to acknowledge all those that supported me throughout the development and writing process of this senior thesis.

First, thank you very much to Professor Wyatt Lloyd for being highly available, meeting with me weekly, and engaging in thought provoking discussion. Thank you to Nick Kaashoek for helping me understand the ins and outs of the Radical system, attending each one of my weekly meetings with Professor Lloyd, and answering my Slack questions super fast! I really appreciate the time and guidance that you both have provided over the course of these two semesters.

Second, thank you to my parents — Mom and Dad. You have both been role models in demonstrating the value of hard work and goal setting. I am doubtful that I would have succeeded in writing thousands of lines of code and a fifty-page paper for this thesis without the lessons that you taught me.

Lastly, to those who added some fun to the late night work sessions — Estelle, Ellie, Eden, Jacob, Andrew, Brandon, Matt, Eliana, Leah, Danielle, Kayla, and Kayla — thank you for the good laughs, nightly snacks, and steady sense of friendship that I hope made the process more enjoyable for us all. This place would seriously not be the same without friends like you.

Contents

List of Figures

Chapter 1

Introduction

Imagine a global financial institution that stores client's transactions and balances in a few centralized datacenters. On each debit card swipe, wire transfer, or cash deposit, the request must travel through the network to the closest datacenter in order to update its contents and return a confirmation. When used by millions of users across the world, the storage and computational power required to support replicas of such large datacenters becomes excessive and the delays in transactions result in frustrated clients. Alternatively, consider a multiplayer video game where players from all of the world partake in a fast-paced racing or combat game together. If every move a player made had to be sent to a central server before the game state could be reflected on all players' screens, the delays would be unacceptable for competitive gameplay.

In both of these situations, the cloud computing model involving a few centralized datacenters falls short in terms of latency performance, particularly for users located a considerable distance from a datacenter. That is where stateful edge computing comes in, enabling systems to store and update state at locations closer to individual clients.

Radical is one such edge computing system. It allows stateful serverless applications to be run on the edge while maintaining strong consistency (linearizability [3]) and durability. In the Radical system, clients are able to send requests to edge nodes, which return immediately after function completion. This is made possible by a consistency check and a write intent that is sent from the edge to the datacenter in parallel with the function execution. Upon function completion and consistency check validation, the edge can respond to the client because it knows that the datacenter will write the same values eventually.

However, Radical currently has no mechanism in place for scheduling and routing functions to run at specific times and edge locations. As such, there is limited guarantee that running on the chosen edge provides better latency than running in the datacenter directly. In the current system, each user is required to determine the best edge location to run the function at themselves. RadSched aims to fill this gap by scheduling and routing functions on the basis of decreased latency in order to achieve runtime improvements across as many function invocations as possible.

The RadSched system is a command-line interface that sits atop the Radical system and allows clients to specify a function to register or run. RadSched then runs an algorithm to select the optimal edge node to run the function at. This optimal edge node is determined on the basis of estimated round-trip-time latencies and the probability of a successful consistency check for that particular function. Once the optimal edge node has been determined, RadSched sends an execution request to Radical on behalf of the client specifying which edge node to run the function on.

The goal of RadSched is to enable the Radical system to achieve faster execution times and round-trip-times, and to abstract the act of choosing an edge location away from the user. Working towards improving end-to-end latency can position Radical as a leader in serverless computing, delivering faster processing times while maintaining

strongly consistent guarantees and therefore increased value for both individual users and businesses.

Chapter 2

Related Work

2.1 Distributed Scheduling

The core idea behind RadSched is to efficiently schedule the execution of functions in a distributed computing environment. There exists much research in this field, particularly regarding the proponents of having a decentralized, rather than monolithic architecture. Sparrow is one such system, a stateless, distributed scheduler designed for low-latency task execution in large-scale clusters. It utilizes randomized sampling in batches and late binding to achieve close-to-optimal latency [6]. Omega functions in a similar highly scalable manner, but uses a shared-state scheduling model, where multiple scheduler instances operate in parallel and maintain a flexible global state of the resource availability [8]. These ideas are directly relevant to RadSched, which also implements a dynamic, decentralized architecture in its approach to scheduling.

However, the RadSched system differs from these systems in that it maintains a consistent state by periodically pulling data from the abstracted Radical system, a necessity that arises due to the stateful properties of the serverless functions it sched-

ules. Unlike Sparrow and Omega, which are designed for relatively stable cluster environments, RadSched must account for fluctuating wide-area latencies and consistency check success rates when selecting an edge location for Radical to invoke a function at.

2.2 Latency Optimization

Reducing latency is a primary concern and responsibility of many distributed schedulers, and is a particularly useful feature when it comes to the latency-sensitive edge computing systems. Calder et al. demonstrate that while Anycast CDNs typically route clients to nearby servers, around 20% of clients experience subpar latency due to the limitations of BGP, the internet routing protocol that selects transit paths based on policy rather than prior performance. This performance weakness is shown to be mitigable through a history-based DNS transit path selection protocol [1].

RadSched employs the idea of using past network and performance data in routing decisions, by putting systems in place that actively monitor round-trip times between clients, edge locations, and datacenters, and utilizing these metrics to guide scheduling decisions in close to real time. This approach aligns with research completed by Zhang et al. on Anycast, which emphasizes the advantage of combining passive and active measurements to adapt to network performance variability [10]. Further extending this idea, rather than just relying on latency, RadSched incorporates both latency observations and consistency check success rates when routing functions.

2.3 Scheduling Architecture

In designing the structure of a scheduler, there is existing research on how to minimize the overhead of scheduling in order to ensure that the system is worthwhile to run. Systems such as Dirigent eliminate much of overhead time common in serverless computing environments by executing functions without precise knowledge of the state of every possible computing location [2]. Radical already enables function execution at any location, regardless of its state, and RadSched builds upon Dirigent's principles with further abstraction – eliminating the need for the client itself to select the execution location.

Abstraction in scheduling is further explored by Lasa et al., who analyzes how different API designs affect the scheduling effectiveness in datacenter environments [4]. Specifically, the research concludes that overly complex APIs can prevent widescale adoption and present unnecessary work for developers. By applying these ideas to stateful serverless edge computing, RadSched aims to minimize developer error due to lack of knowledge regarding location choice.

The decision of when to offload computational tasks from a client device to the edge or a datacenter is well researched, with strategies primarily relying on measurements related to network conditions, workload intensity, and computational complexity. One such paper by F. Saeik et al. explored how to dynamically assess these three factors in the context of distributing communication technologies and IoT [7]. RadSched is unique in this field because it schedules stateful functions and therefore data consistency must be assessed as well; however, a similar multi-factor assessment strategy — considering network and workload conditions — is applied when determining whether to execute a function at the edge or simply send it to the datacenter.

2.4 Adaptive Scheduling with Reinforcement Learning

To adapt to varying network conditions, systems often employ machine learning-based scheduling strategies. In particular, reinforcement learning has been researched and tested in the context of heterogeneous distributed systems in order to balance exploration and exploitation in scheduling decisions [5]. In RadSched, reinforcement learning can be used in a similar fashion to shift function placement based on observed success and changes in network latencies.

Another key scheduling challenge is handling situations where execution conditions are not optimal, but may become optimal later on. The paper Delay Scheduling by M. Zaharia et al. introduces the idea of waiting until a node has the required input data, rather than moving onto the next node immediately, in order to preserve fairness in scheduling [9]. RadSched may not need to address this issue, as if clients are scattered throughout the globe, a varying geographic distribution of edges will naturally arise.

Chapter 3

System Design

3.1 Problem Visualization

Let us illustrate the edge computing network as a bi-directional graph, where there exists clients, edge nodes, and datacenters. In the simplified example below, Figure 3.1, there is exactly one client, two edge nodes, and one datacenter.

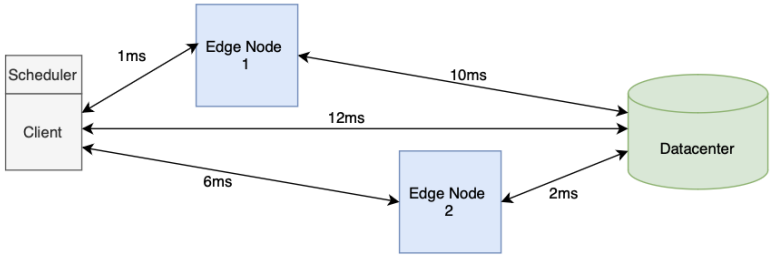

Figure 3.1: Graph Representation of the System Environment

The client can choose to run a given function at any edge node or datacenter, but the rate of consistency check failure is unknown. If the consistency check passes, the edge node has the most up-to-date data and the function does not need to be

executed in the datacenter. However, if the consistency check fails, the edge node does not have the most up-to-date data and the function does need to be executed in the datacenter. An efficient protocol must be designed such that the time to execute a function at the edge is faster than executing the same function in the datacenter.

3.2 A Naive Design

Graph Example

Edge node execution outperforms direct datacenter execution only when the consistency check time is shorter than the function runtime. Therefore, to deduce the optimal edge node, the datacenter-to-edge distance must be compared to the function runtime. Under this key observation, the optimal edge node can then be calculated as that which yields the lowest round-trip-time latency.

For example, in the illustration below, Figure 3.2, (where the times represent the RTTs between locations), Edge Node 2 has a total RTT of 8ms, while Edge Node 1 has a total RTT of 11ms. Based purely on these RTTs, Edge Node 2 would appear optimal. However, let us consider a function runtime of 9ms. Edge Node 2 delivers a response after 15ms (6ms + max(2ms, 9ms)), while Edge Node 1 delivers a response in 11ms (1ms + max(10ms, 9ms)). In this case, Edge Node 1 becomes the optimal choice when the function runtime is 9ms.

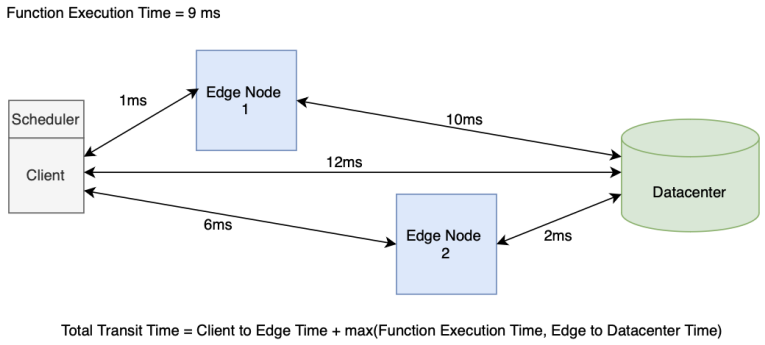

Figure 3.2: Graph Representation of the System Environment with Latency Formula

Formalization of Algorithm

Let us formalize this approach by defining a System N that is given as input:

- A function execution time $T_{\text{exec}} \in \mathbb{R}_{\geq 0}$:
- A client/scheduler node c, a datacenter node d, and a set of edge nodes E
- Network latencies:
  - $T_{c \to e}$: latency from client to edge node $e \in E$
  - $T_{e \to d}$: latency from edge node $e \in E$ to datacenter d
  - $T_{c \to d}$: latency from client to datacenter

Define the end-to-end latency of executing the function at edge $e \in E$ as:

$$L(e) = T_{c \to e} + \max\left(T_{\text{exec}}, T_{e \to d}\right)$$

Define the datacenter runtime as:

$$L(d) = T_{c \to d} + T_{\text{exec}}$$

Then, the optimal execution location is chosen as:

$$\text{OptLocation} = \begin{cases} d, & \text{if } \min\limits_{e \in E} L(e) > L(d) \\ \arg\min\limits_{e \in E} L(e), & \text{otherwise} \end{cases}$$

3.3 A Weighted, Epsilon-Greedy Design

Graph Example

In order to account for the probability that the consistency check fails, the system must maintain a record of the consistency success rate of each function at each edge node. By doing so, the likelihood that a function consistency check will fail at a given edge node can be factored into the algorithm. In particular, the true end-to-end function latency can be multiplied by the expected probability of failure, and the edge node with the minimum expectation is chosen.

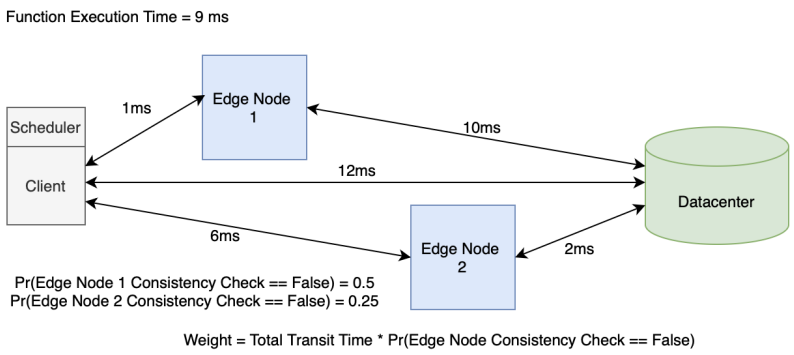

Figure 3.3: Graph Representation of the System Environment with Weight Formula

Formalization of Algorithm

Let us formalize this approach by defining a System W that extends System N, given all of System N's input in addition to:

- A consistency weighting function[1] $w : e \to [0, 1]$, where

$$w(e) = \frac{\text{num\_failure}(e)}{\text{num\_attempts}(e).}$$

- An exploration parameter[2] $\varepsilon \in [0.1, 0.9]$

Let the set of eligible edge nodes be:

$$\mathcal{E} = \{e \in E \mid L(e) < L(d)\}$$

[1]There is consistency weighting $w(e)$ for every edge e and for every function f. Thus, a less simplified representation is $w_f(e)$

[2]The ε-Greedy Exploration algorithm is expanded upon in Section 3.5.

The scheduling policy is:

- If $\mathcal{E} = \emptyset$, then OptLocation $= d$
- Otherwise:
  - With probability ε (exploration):

$$\text{OptLocation} \sim \text{Uniform}(\mathcal{E})$$

  - With probability $1 - \varepsilon$ (exploitation):

$$\text{OptLocation} = \arg\min_{e \in \mathcal{E}} \left(L(e) \cdot w(e) \right)$$

3.4 Data Retrieval

To make decisions that best reflect the state of the network and edges, RadSched requires up-to-date information about network latencies and consistency check performance on a function level granularity. The system collects and maintains this information both continuously and at set intervals of time. This retrieval process provides the input assumed to be given in the algorithms described for System N and System W in Section 3.3.

Function Registration & Metadata

A client registers a function with RadSched by providing a function name, the estimated runtime (T_{exec}), and the primary datacenter. That same function name is then used as a unique identifier for various metadata associated with each function. Specifically, for each function, RadSched must track the function's tolerance for in-

consistency (expressed as the epsilon value) and consistency check success rate at each edge location (described in following subsection). This metadata enables RadSched to make more informed decisions during scheduling, such as selecting edge locations with higher consistency success ratios and exploring new nodes to account for network changes. All registration data is stored locally by RadSched and propagated to the Radical function store to ensure consistency across RadSched instances.

Consistency Statistics

Each time a function is executed on an edge node, the RadSched add-on for Radical records whether the consistency check succeeded. These results are aggregated over time to compute $1 - w_f(e)$, the consistency success rate of each edge node for each function. These statistics are updated immediately after function execution and allow the approach in Section 3.3 to be actualized. The consistency data is stored on a remote server dedicated to processing updates and serving requests. Because the consistency data is vital to RadSched's location selection process and written to on each Radical function invocation, having a dedicated server is beneficial for maintaining a large store without overloading the Radical or RadSched systems. To ensure that RadSched does not need to query this remote server on each run, the data is cached locally and updated in the background at set intervals of time.

Each function registered with RadSched is also mapped to an epsilon value ϵ, which represents how much inconsistency the function should tolerate — how much RadSched should explore and how much it should exploit. Over time, the ϵ value for a function is updated based on the aggregation of consistency success rates across edge locations for a given function. These values can range from 0.1 to 0.9 and are updated based on the algorithm described in Section 3.5. The ϵ values are stored locally within the client's Radsched instance. Because the data is updated often and

unique to each client, it would be unnecessary to store an additional copy within Radical.

Latency Measurements

RadSched performs network latency calculations at set intervals of time to update latency values for client-to-edge ($T_{c \to e}$), edge-to-datacenter ($T_{e \to d}$), and client-to-datacenter ($T_{c \to d}$). These latency values are vital for calculating the eligible edge nodes when even using the naive, System N (Section 3.2). The latency data is stored locally within the client's Radsched instance and updated in the background at set intervals of time. Because the data is updated often and unique to each client, it would be unnecessary to store an additional copy within Radical.

3.5 Epsilon-Greedy Exploration

Network speeds fluctuate with changing conditions and edge nodes are constantly updated and replaced with new data from client interactions; hence the optimal execution location for a function is not static — it changes over time. Thus, the system must sometimes avoid selecting the supposed optimal edge node in order to observe whether network performance has shifted.

To balance the trade-off between exploring new edge locations and exploiting the currently optimal one, RadSched employs an ϵ-greedy exploration strategy. This approach introduces stochasticity into the selection process: with probability ϵ the scheduler selects a random eligible edge node, and with probability 1 - ϵ it selects the edge node that minimizes the weighted latency cost. As described in Section 3.3, an eligible edge node is defined as the set of nodes for which the naively calculated total

transit time is less than that of the datacenter.

The ϵ parameter represents the system's tolerance for inconsistency and controls the degree of exploration, it is dependent on the function's consistency success rate. If a function's success rate is high, ϵ will continue to decrease; if a function's success rate is low, ϵ will continue to increase. A higher ϵ encourages more frequent exploration, which is particularly useful in highly dynamic network environments. In periods of network stability, the system should gain confidence in its optimal choice based on consistency success and latency, ultimately decreasing exploration to a minimum.

This reactive growth and decay of ϵ ensures that RadSched remains responsive to changes in network conditions while converging toward optimal performance. By storing function ϵ values directly in its metadata, RadSched enables per-function granularity in scheduling behavior, ensuring that edge node selection is tailored to the availability of data that each function requires.

The ϵ-Greedy Algorithm

To accomplish the reactive behavior described above, RadSched uses a smooth update function that reacts to the observed success rate of a function's consistency checks. The formula is as follows:

$$\epsilon_{\text{new}} = \epsilon_0 + \alpha \cdot (1 - s_f - \epsilon_0)$$

where:

- ϵ_0: current ϵ value for function f
- $\alpha \in (0.0, 1.0)$: a constant floating point number, the smoothing factor that

controls adjustment sensitivity

- $s_f = \sum_{e \in \mathcal{E}_f} (1 - w_f(e))$: success rate for function f, computed as the sum of consistency success weights $1 - w_f(e)$ across all eligible edges $\mathcal{E}_f$

To ensure there is never no exploration or no exploitation, ϵ_{new} is bounded by minimum and maximum thresholds:

$$\epsilon_{\text{new}} = \min\left(\epsilon_{\max}, \max\left(\epsilon_{\min}, \epsilon_{\text{new}}\right)\right)$$

where: $\epsilon_{\max} = 0.9$ and $\epsilon_{\min} = 0.1$

This update formula increases ϵ when performance is poor (low success rate) and decreases ϵ when performance is good (high success rate), thereby accomplishing a reactive exploration/exploitation system.

3.6 Workflow & User Interaction Model

RadSched is designed as a lightweight command-line interface tool that is used in conjunction with the Radical system. Its interaction model is simple: allow users to register, update, and run serverless functions without knowledge of the optimal edge location to invoke them at.

The typical workflow consists of the following steps:

1. Function Registration: The user registers a function with RadSched using a CLI command, which sets up the functions associated metadata.

2. (Optional) Function Data Collection: The user manually triggers a CLI command to collect the most recent latency and consistency data. However, this

data is also automatically updated in the background every ten minutes.

3. Function Invocation: The user provides the function name to the RadSched CLI, which selects the optimal location for function invocation.

4. Step 3 is repeated, and RadSched improves its location selection over time.

Chapter 4

Implementation

The RadSched system is built as a Command Line Interface (CLI) tool that abstracts the Radical system from the client, choosing the optimal location for execution so that the client does not need to. CLI was selected as the implementation method because it is lightweight and follows standards that developers are accustomed to, enabling easier scripting and adoption into existing workflows.

RadSched follows the standard CLI format. To execute a command, a client types:

```
radsched [command] [args] [flags]
```

The scheduling system is built from three main components: the Radical system, the Consistency Storage Server, and the RadSched client program itself. The system was implemented using Go 1.23, Python 3.12, and Rust 1.85. These components work together to form the scheduling system, though only the RadSched CLI is exposed to users.

4.1 RadSched CLI

The client facing RadSched system was implemented as a command line interface using Go. There are three CLI commands that, when used in tandem, provide full access to RadSched's functionality: `prepare`, `bootsrap`, and `run`.

Prepare Command

The `prepare` command registers a function with the RadSched scheduler by adding it to both the local function registry and the Radical system. Only once a function is prepared and registered to the scheduling system is it available for invocation. The command registers the function name, along with function metadata that enable Radsched to make informed location selection decisions.

```
radsched prepare functionX 1000 us-east-1
```

This command takes three arguments:

- `function name` – the name used to identify the function (e.g., `functionX`)
- `execution time` – the estimated execution time in milliseconds (e.g., `1000`)
- `datacenter` – the name of the primary datacenter for the function (e.g., `us-east-1`)

The command performs the following logical execution steps:

1. *Local Registration*

   The function is stored in a local file named `function_registry` in JSON format. If the function already exists in the registry, it is updated instead of duplicated, thereby ensuring clients can update function metadata.

2. *Radical Registration*

   The function metadata is formatted into a JSON payload and sent via a `POST` request to the Radical system via the `/register` endpoint. The payload uses this Go struct:

```
type FunctionInfo struct {
    FunctionName string `json:"function_name"`
    ExecutionTime string `json:"execution_time"`
    FunctionURL string `json:"function_url"`
    Datacenter string `json:"datacenter"`
}
```

The `prepare` command is a required prerequisite for calling the `run` command.

Bootstrap Command

The `bootstrap` command retrieves the most recent function metadata, latency measurements, and consistency success rates necessary for the location selection algorithm to function optimally. This command can be run manually; however, it also runs at set intervals of time in the background (using cron) in order to ensure RadSched's local caches are up-to-date.

```
radsched bootstrap
```

The command does not take any arguments. The command performs the following logical execution steps:

1. *Client-to-Edge RTT Collection*

   The command invokes a Python script `ping_edges.py` that pings select AWS EC2 endpoints to retrieve RTTs from the client to those select edge locations.

```
run script python3 ping_edges.py:
    for each region in edges:
        rtt = ping "ec2.{region}.amazonaws.com"
        save map[region]rtt
    return map[region]rtt
store map[region]rtt in local JSON
```

   The client-to-edge RTTs are stored in a local file named `client_edge_rtts` in JSON format. The payload follows the structure below:

```
{
    "location_name": "us-east-1",
    "round_trip_time": "12.53 ms"
}
```

2. *Edge-to-Datacenter RTT Collection*

   The command then invokes a Lambda function named `PingDatacenters` from each region, which pings all other regions. This is done to retrieve the RTTs from each edge to each datacenter.

```
for each region in datacenters:
    invoke Lambda function "PingDatacenters" in that region
        for each other_region:
            rtt = ping "ec2.{other_region}.amazonaws.com"
```

```
            save map[other_region]rtt
        return map[other_region]rtt
    save map[region]map[other_region]rtt
store map[region]map[other_region]rtt in local JSON
```

The edge-to-datacenter RTTs are stored locally in a file named `edge_datacenter_rtts` in JSON format. The payload follows the structure below:

```
{
  "us-east-1": {
    "us-west-1": 43.6,
    "us-west-2": 48.3
    }
}
```

3. *Consistency Data Collection*

   The command sends a `GET` request to the Consistency Storage Server to retrieve the latest consistency statistics for all functions across all edge locations. The data is stored locally in a file named `edge_function_consistency` in JSON format. The payload follows the structure below.

```
{
  "us-east-1": {
    "functionX": {
      "num_attempts": 5,
      "num_success": 2,
      "num_failure": 3
    }
  }
```

```
}
```

4. *Function Metadata Collection*

   Lastly, the command sends a `GET` request to the Radical system's `/bootstrap` endpoint to retrieve all registered functions. The response is written to the local `function_registry` file — the same file written to by the `prepare` command. Existing entries are updated and new entries are created to ensure the scheduler maintains a global view of available functions, even across different clients or sessions.

Run Command

The `run` command selects the optimal location to run a function at, as determined using the algorithms described in Section 3.2 and Section 3.3. This abstraction frees developers from manually determining where a function should run.

```
radsched run functionX [--with-weight]
```

The command takes one argument:

- `function name` – the name of the function to execute (e.g., `functionX`)

and one optional flag:

- `--with-weight` – enables consistency-aware weighted scheduling and ϵ-greedy exploration

The command performs the following logical execution steps:

1. *Function Lookup*

   The function is queried in the local registry. If it is not found, the command fails with instructions to first run `prepare`.

```
functions = load function_registry as map
if function_name not in functions:
    error("Function unknown. Please prepare before running.")
```

2. *Latency-Based Scheduling (default)*

   Without the `--with-weight` flag, RadSched selects the optimal edge location based on end-to-end latency — the algorithm described in Section 3.2.

```
for each edge:
    latency = client_to_edge_rtt + max(function_time,
        edge_to_datacenter_rtt)
    save (edge, latency)
optimal_edge = edge with min latency
latency_datacenter = client_to_datacenter_rtt + function_time
if latency of optimal_edge < latency_datacenter:
    return optimal_edge
else:
    return datacenter
```

3. *Weighted Scheduling with Epsilon-Greedy Exploration (with `--with-weight`)*

   If the `--with-weight` flag is provided, RadSched selects the optimal edge location based on end-to-end latency, consistency success rates, and stochastic exploration — the algorithm described in Section 3.3

```
get calculated latencies from above (Step 2)
eligible_edges = edges where latency < latency_datacenter
epsilon = GetEpsilon(function_name, SMOOTH)
if rand() < epsilon:
    return random edge from eligible_edges
else:
    for each edge:
        consistency = num_failure / num_attempts
        weighted_latency = latency * (consistency)
    return edge with min weighted_latency
```

4. *Radical Execution and Output*

   Once the optimal location is selected, the pipeline parses the scheduler's output, invokes the function at that location via Radical, and reports results to the user.

This command represents the core functionality of RadSched: an adaptive, consistency-aware scheduler that minimizes latency.

4.2 Consistency Storage Server

The consistency check server supports two HTTP methods on a single endpoint that are used by Radical to log consistency data and by RadSched to pull consistency data. The HTTP server was implemented on an AWS EC2 instance and written in Python.

```
   GET: Returns all consistency data as a JSON object.
   POST: Creates or updates consistency data entries for a given function
      and edge location.
```

The server stores data in a file named `hit_ratio` in JSON format, using the following structure:

```
{
  "us-west-1": {
    "functionX": {
      "num_attempts": 5,
      "num_success": 4,
      "num_failure": 1
    },
    "functionY": {
      "num_attempts": 2,
      "num_success": 2,
      "num_failure": 0
    }
  }
}
```

Update Logic

The server handles each `POST` request as follows:

```
edge <- payload["edge"]
function <- payload["function"]
```

```
attempts <- payload["num_attempts"]
success <- payload["num_success"]
failure <- payload["num_failure"]

open or create data_file
lock data_file

data = load(data_file)
if edge or function not in data:
    create entry{edge, function, num_attempts, num
    _success, num_failure}
else:
    get entry
    increment entry.num_attempts, entry.num
    _success, and entry.num_failure

write(temp_file, data)
copy(temp_file, data_file)
unlock data_file
```

File locking ensures that only one process can read or write to the consistency data at a time, preventing race conditions under concurrent access. Atomic commits via a temporary file guarantee that the file is never left in a partially written or corrupted state, even if a write is interrupted. These protocols provide durability and consistency. When the system was tested without these safety measures, the JSON file was repeatedly corrupted due to invalid formatting from partial writes.

This server is queried by RadSched when the `bootstrap` command is called. It is also automatically queried in the background every 10 minutes. The data is vital to

calculating up-to-date consistency success ratios. After each function invocation, the Radical system posts consistency results to the server, ensuring that the data remains live.

4.3 Radical Additions

To integrate the RadSched system with the Radical system, several additions were made to Radical. First, a new HTTP endpoint was added to allow RadSched to register functions with their associated metadata — `/register`. Correspondingly, another endpoint was added to send function data from Radical to RadSched client instances — `/bootstrap`, allowing RadSched to maintain an up-to-date and global view of the system. Lastly, Radical was modified to log the result of each consistency check to the Consistency Storage Server after function execution.

Chapter 5

Evaluation

5.1 Baseline Evaluation

To evaluate the effectiveness of RadSched in selecting the optimal location for Radical function execution, its performance is compared against three other baseline location selection strategies. The goal of the experiment is to determine whether RadSched consistently selects edge locations that improve latency metrics compared to naive or uninformed selection methods. For the same function, the Radical end-to-end latency is evaluated across all four baselines after several rounds of trials.

Experiment Setup

The experiment environment consisted of three available AWS regions, which functioned as both edges and datacenters. These regions were us-east-1, us-east-2, and us-west-1. The client was located in Princeton, NJ.

The function (`fn_us-west-1_100`), with an estimated execution time of 100 ms and

primary datacenter of us-west-1, was executed 1000 times under each of the following four baselines:

- Baseline 0 (RadSched): RadSched selects the location using its scheduling algorithm.
- Baseline 1 (Random): The location is selected uniformly at random from the list of available edge regions.
- Baseline 2 (Geographic Closest): The location is selected based on closest physical proximity to the client.
- Baseline 3 (Fastest Ping): The location is selected based on the lowest round-trip time (RTT) to the client measured by ICMP ping.

For each strategy, the function was invoked 1000 times, and the following metrics were recorded:

- Radical Latency: Total time for the function to execute on the Radical system (ms).
- System Latency: Total time from request receipt to response receipt, including scheduling (ms).
- Consistency Check: Whether the Radical consistency check returned true or false (True/False).

Experiment Hypothesis

We hypothesize that RadSched may perform slightly worse in this experiment than the Geographic Closest or Lowest Ping strategies due to its use of ϵ-greedy exploration.

Specifically, because RadSched explores with probability ϵ, choosing what may be a suboptimal location in that case, its median latency may increase.

Exploration is hypothesized to only be a performance concern in this stable environment experiment, where the consistency check success rate remains constant across all edge locations. In such a stable setting, the geographically closest or fastest ping time edge location is typically the optimal choice. In a more realistic environment where consistency success rates vary by location, the closest node may not always yield the lowest overall latency — such a setting is discussed in Section 5.3.

Yet, we expect RadSched to converge back to the optimal edge node even after bursts of exploration runs, and to thus maintain close to optimal median performance.

Experiment Results

The Radical end-to-end execution latency was recorded for 1000 function invocations across all four baseline strategies. The results are summarized in Figure 5.1 and Figure 5.2

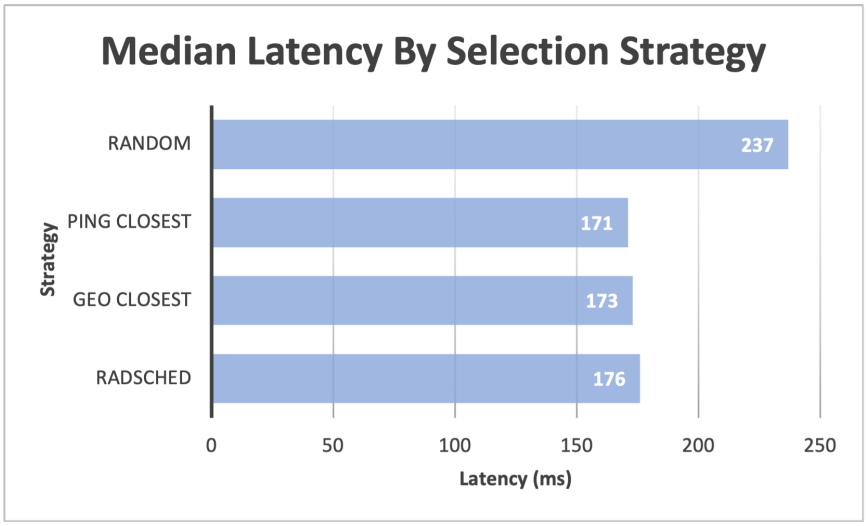

Figure 5.1: RadSched achieved median latency close to, but higher than, that of Geo Closest and Ping Closest

The results confirm that RadSched delivers median latency performance (176 ms), comparable to, but 2%-3% lower than, those of the Geographic Closest (171 ms) and Fastest Ping (173 ms) strategies. RadSched's median latency performance was significantly better than that of the Uniformly Random decision strategy (237 ms), indicating that RadSched ruled out low performing edge nodes.

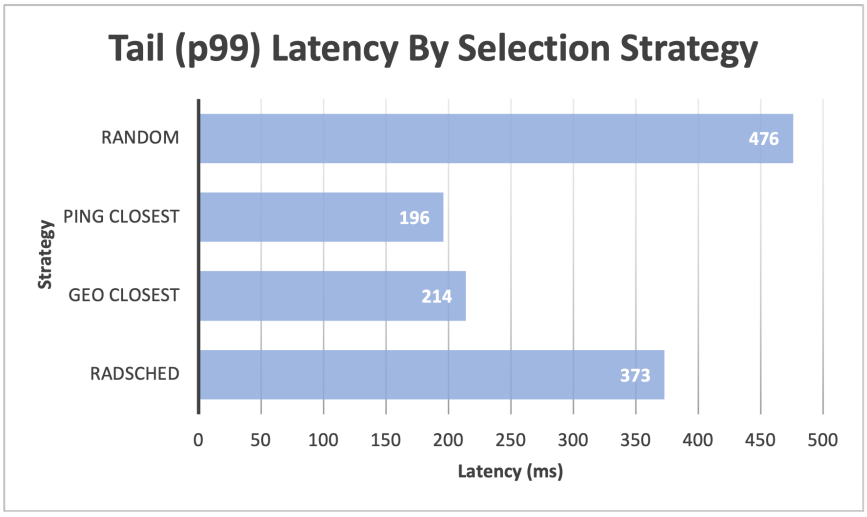

Figure 5.2: RadSched achieved tail latency lower than Random, but higher than Geo Closest and Ping Closest

However, RadSched exhibited significantly higher tail latencies. At the 99th percentile, its Radical latency was 350 ms — far higher than that of the Geo Closest baseline (214 ms) or Fastest Ping baseline (196 ms). This aligns with the hypothesis: because RadSched uses an ϵ-greedy exploration strategy, some function executions are routed to suboptimal nodes, impacting tail performance. In contrast, the Geo Closest and Fastest Ping strategies always select the same edge node and thus experience less variance. This is evidenced by Figure 3.3, in which the distribution of edge node selection under each strategy is presented.

	us-east-1	us-east-2	us-west-1
RadSched	634	366	0
Geo Closest	1000	0	0
Ping Closest	1000	0	0
Random	354	320	326

Figure 5.3: RadSched selects us-east-1 63.4% of the time us-east-2 36.6% of the time

For the RadSched selection strategy, a total of 63.4% of function executions were routed to us-east-1 and 36.6% to us-east-2, suggesting that the RadSched decision model learned to prefer us-east-1 as the optimal region, while still exploring occasionally. Initially, the ϵ value for the function (`fn_us-west-1_100`) was set to 0.5. However, the non-uniform distribution of location selections across the two eligible edge options over 1000 trials, shows that the ϵ value, and thereby exploration, must have decayed over the experiment. Indeed, inspecting the system after the experiment confirmed that the ϵ value had gradually decreased to 0.1. Notably, us-west-1 was never selected, indicating that the RadSched system deemed it suboptimal across every function invocation.

5.2 Adaptive Learning Evaluation: Network Volatility

This experiment evaluates the effectiveness of RadSched in dynamically adapting to changes in network latency conditions over time. While the baseline testing environment experienced almost static network performance, this experiment simulates a scenario where RTTs between the client and edge locations fluctuate, requiring the scheduler to detect the change and reevaluate the optimal node.

Experiment Setup

Just as in the baseline experiments (Section 5.1), the experiment environment consisted of three available AWS regions, us-east-1, us-east-2, and us-west-1 and a client located in Princeton, NJ.

The function (`fn_us-west-1_100`) with an estimated execution time of 100 ms and

primary datacenter of us-west-1, was invoked 200 total times.

- 100 times during Phase 1: where the latency from the client to us-east-1 is lower than that from the client to us-east-2
- 100 times during Phase 2: where the latency from the client to us-east-2 is lower than that from the client to us-east-1

Experiment Hypothesis

We hypothesize that RadSched will initially favor us-east-1, but over time, as it receives updated RTT data via `bootstrap` and utilizes the new latency in calculations, preference will shift to us-east-2. If this occurs, the percentage of requests routed to us-east-2 should increase in Phase 2.

Experiment Results

The Radical end-to-end execution latency was recorded for all 200 function invocations, where the network latency of us-east-1 increased at around the 100th invocation. The results are summarized in Figure 5.4 and Figure 5.5.

	us-east-1	us-east-2	us-west-1
Phase 1	93	7	0
Phase 2	9	91	0

** Phase 1: before change in network speed ** Phase 2: after change in network speed

Figure 5.4: RadSched selects us-east-1 93% of the time in Phase 1 and 9% of the time in Phase 2

As evident in Figure 5.4, prior to the shift in network speed, 93% of function in-

vocations were routed to us-east-1, which mirrors the baseline evaluation using the same function — RadSched detected us-east-1 as the optimal edge location. After the network condition shifted, only 9% of invocations were routed to us-east-1, displaying that RadSched was able to account for the increased transit time between the client and us-east-1 and optimize its decision policy accordingly.

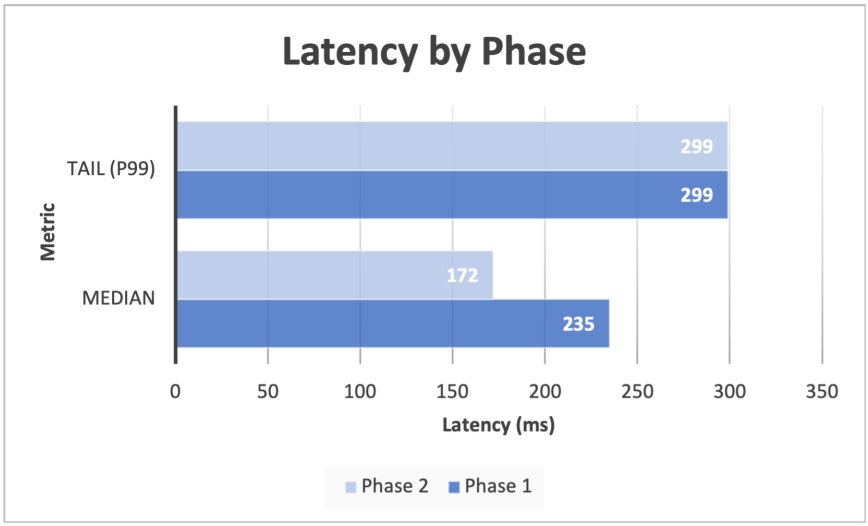

Figure 5.5: System median and tail latency before and after the network speed changed

The results in Figure 5.5 show a clear increase in median latency from Phase 1 to Phase 2. Specifically, 172 ms in Phase 1 to 235 ms in Phase 2 — a 36.6% increase. This increase can be attributed to RadSched's adaptive behavior: after the network change, the latency from the client to the previously optimal edge (us-east-1) became higher than that from the client to us-east-2. Consequently, RadSched identified us-east-2 as the new optimal edge location. However, the latency of us-east-2 was higher than that of us-east-1 before the network change, so the system latency increased from Phase 1 to Phase 2.

Assessing the system data before and after the experiment revealed that in Phase

1, the client to us-east-1 RTT was approximately 11 ms and the client to us-east-2 RTT was approximately 28 ms. During phase 2, the client to us-east-1 RTT was approximately 30 ms and the client to us-east-2 RTT remained at approximately 28 ms. Hence, the latency minimizing algorithm chose us-east-2 as the optimal edge, which accounts for the increase in median latency.

Interestingly, the tail (p99) latency during the two phases was identical — 299 ms. This suggests that although the median increased due to higher client-to-edge RTTs, the occasional slower-than-expected network responses remained similar. Evidence that the tail latency was expected to be similar is found from the fact that the maximum RTT among us-east-1 and us-east-2 remained nearly unchanged across phases — max(11 ms, 28 ms) compared to max(30 ms, 28 ms) — and thus it is fitting that the occasional high latency function invocation increased latency by about the same amount in both Phase 1 and Phase 2.

In summary, the RadSched selection algorithm successfully adjusted its optimal edge location choice in response to network changes, without any concerning increases in latency.

5.3 Adaptive Learning Evaluation: Data Volatility

This experiment evaluates the effectiveness of RadSched in adapting to changes in data consistency conditions across edge locations. Unlike the previous experiment on network volatility, this experiment maintains a stable network environment but instead introduces variability in the consistency check success rate. This is done to assess whether RadSched can dynamically learn to avoid edge nodes that consistently fail the Radical consistency check, even if their network latencies are low.

Experiment Setup

Aligned with previous experiments, the experiment environment consisted of three available AWS regions, us-east-1, us-east-2, and us-west-1 and a client located in Princeton, NJ.

The function (`fn_us-west-1_100`) with an estimated execution time of 100 ms and primary datacenter of us-west-1, was invoked 1100 total times.

- 100 times during Phase 1: where the consistency check at the us-east-1 location always returned true.
- 1000 times during Phase 2: where the consistency check at the us-east-1 location always returned false.

Experiment Hypothesis

We hypothesize that in Phase 1, when the system has perfect consistency, RadSched will choose us-east-1 a large majority of the time, as shown in the baseline experiments for a stable environment. In Phase 2, as us-east-1 begins to fail consistency checks, RadSched will learn to reduce its reliance on that region and route more requests to us-east-2. As such, over the 1000 runs, we anticipate that the percentage of function invocations routed to us-east-2 will increase, though the exact number is challenging to estimate.

Experiment Results

The Radical end-to-end execution latency was recorded for all 1100 function invocations, where the data stored in us-east-1 became inconsistent at around the 100th

invocation. The results are summarized in Figure 5.6 and in Figure 5.7

	us-east-1	us-east-2	us-west-1
Phase 1	96	4	0
Phase 2	382	618	0

Figure 5.6: RadSched selects us-east-1 96% of the time in Phase 1 and 38% of the time in Phase 2

Figure 5.6 illustrates that location selection decisions transitioned from favoring us-east-1 to favoring us-east-2. In Phase 1, 96% of function invocations were routed to us-east-1, consistent with previous experiments and confirming that RadSched initially identified it as the optimal edge. In Phase 2 however, the percentage of function invocations routed to us-east-1 dropped to 38.2%, with the remaining 61.8% routed to us-east-2. This indicates that in Phase 2, after the consistency success rate of us-east-1 began to diminish, RadSched adjusted its selection policy to favor us-east-2, redirecting a significant portion of request traffic to the now more reliable edge node.

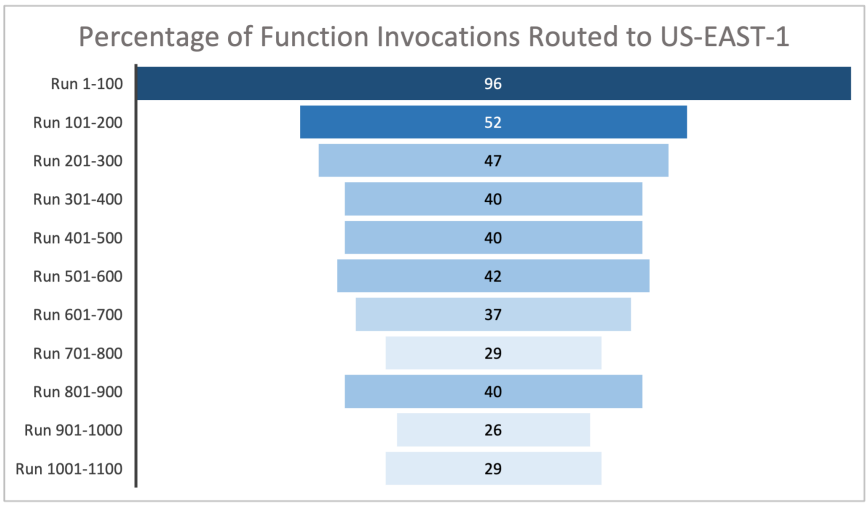

Figure 5.7: Percentage of function invocations routed to us-east-1 for every 100 consecutive RadSched runs

Figure 5.7 provides a closer view of how RadSched's location selection decisions evolved in response to increased consistency check failures at us-east-1. In the first 100 invocations following the shift into Phase 2 (Runs 101–200), the percentage of requests routed to us-east-1 dropped steeply from 96% to 52%, reflecting a clear change in the behavior of RadSched's edge selection algorithm. Although us-east-1 still offered faster latency, RadSched began exploring other locations more often as the consistency success rate (which inversely influences epsilon, the rate of exploration) began to decrease for the us-east-1 location.

Over subsequent runs, exploration continued and the percentage of invocations routed to us-east-1 steadily decreased, reaching as low as 26% by Runs 901–1000. This decline suggests that the increasing rate of consistency failures at us-east-1 began to significantly impact RadSched's location selection algorithm, which weighs consistency and latency in its decision making. The repeated failures at us-east-1 decreased the location's overall weighted latency, which prompted RadSched to favor the more consistent, but slower, us-east-2.

It is important to note that us-west-1 was never chosen by Radsched, indicating that the edge selection process continuously deemed it ineligible due to its higher latency. The continued variance in location selection percentages across each 100-run segment, even when several hundred runs into the experiment, can be attributed to RadSched's exploration feature. RadSched had not yet accumulated enough consistent success at us-east-2 to reduce its exploration rate to a minimum. This is supported by a post-experiment inspection, which showed that the function's ϵ value was still approximately 0.64. Given more RadSched runs under the same conditions (where us-east-2 continues to maintain consistent data), we would expect ϵ to decay further, leading RadSched to explore less and converge more confidently on us-east-2 as the optimal edge location.

In summary, the experiment confirms that RadSched can adapt to data volatility by evaluating the tradeoff between latency and consistency. As consistency check failures increase, RadSched transitions toward more reliable edge locations.

Chapter 6

Conclusions and Future Work

Overall, RadSched's performance in evaluation experiments shows that it is both feasible and practical to build an adaptive scheduler for stateful serverless edge computing that optimizes for latency. By combining live round-trip latency calculations with observed consistency check outcomes, RadSched was able to achieve reasonable location selection decisions in both stable and volatile environments. In a static setting, its location choices achieved a median latency close to that of choosing the nearest edge, though there was an increase in tail latency due to exploration. In a dynamic setting, RadSched was able to navigate to a new optimal edge location when network speed or data availability shifted. In doing so, the system abstracts the infrastructure complexity of Radical via a simple CLI, requiring minimal developer setup or input.

Next steps would include adding more advanced reinforcement learning techniques to enhance RadSched's decision-making process and minimizing system overhead by ensuring the optimal data structures and data requesting techniques are employed. In a latency optimizer scheduler, removing any amount of overhead would contribute to

tangible performance improvements. While the current version is focused on latency, the system can be extended by using a similar optimization framework for other factors such as cost, energy usage, or time of day. By doing so, RadSched would become a complete scheduler that both abstracts, and provides enhanced features for, the Radical system.

Appendix A

Code

The source code for the RadSched scheduler can be found here:

https://github.com/jm252/RadSched

Bibliography

[1] M. Calder, A. Flavel, E. Katz-Bassett, R. Mahajan, and J. Padhye. Analyzing the performance of an anycast cdn. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, pages 531–537, New York, NY, USA, 2015. Association for Computing Machinery.

[2] L. Cvetković, F. Costa, M. Djokic, M. Friedman, and A. Klimovic. Dirigent: Lightweight serverless orchestration. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2024.

[3] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, jul 1990.

[4] A. M. Lasa, S. Talluri, T. D. Matteis, and A. Iosup. The cost of simplicity: Understanding datacenter scheduler programming abstractions. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 2024.

[5] A. I. Orheana, F. Pop, and I. Raicu. New scheduling approach using reinforcement learning for heterogeneous distributed systems. *Journal of Parallel and Distributed Computing*, 2017.

[6] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low

latency scheduling. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2013.

[7] F. Saeik, M. Avgeris, D. Spatharakis, N. Santi, D. Dechouniotis, J. Violos, A. Leivadeas, N. Athanasopoulos, N. Mitton, and S. Papavassiliou. Task offloading in edge and cloud computing: A survey on mathematical, artificial intelligence and control theory solutions. *Computer Networks*, 195, 2021.

[8] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*. ACM, 2013.

[9] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*. ACM, 2010.

[10] X. Zhang, S. Lin, T. Huang, B. M. Maggs, K. Schomp, and X. Yang. Characterizing anycast flipping: Prevalence and impact. In C. Testart, R. van Rijswijk-Deij, and B. Stiller, editors, *Passive and Active Measurement*, pages 361–388, Cham, 2025. Springer Nature Switzerland.