FIRST LINE OF THE TITLE SECOND LINE OF THE TITLE

by

Author

A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Master of Science
Discipline

Committee:

________________________________	Dr. First Last, Thesis Director
________________________________	Dr. First Last, Committee Member
________________________________	Dr. First Last, Committee Member
________________________________	Dr. First Last, Department Head
________________________________	Dr. First Last, Dean
Date: ____________________________	X Semester Year George Mason University Fairfax, VA

The Complete Title is to be Repeated Here without any Line Breaks for the Second Page and for the Abstract Page

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Author
Bachelor of Science
My Other Former School, Year of first degree

Director: Dr. First Last, Professor
Department of Name of Department

X Semester Year
George Mason University
Fairfax, VA

Dedication

I dedicate this dissertation to ...

Acknowledgments

I would like to thank the following people who made this possible ...

Table of Contents

List of Tables

List of Figures

Abstract

THE COMPLETE TITLE IS TO BE REPEATED HERE WITHOUT ANY LINE BREAKS FOR THE SECOND PAGE AND FOR THE ABSTRACT PAGE

Author, MS

George Mason University, Year

Thesis Director: Dr. First Last

Enter abstract text.

Chapter 1: Introduction

1.1 Motivation

Serverless computing is an increasingly popular cloud execution model that liberates application developers from the burden of traditional infrastructure management. With serverless platforms (e.g., AWS Lambda, Google Cloud Functions, Azure Functions), developers solely focus on writing their code as event-driven functions that will execute on-demand in response to events or triggers. Cloud providers are responsible for dynamically allocating and scaling resources to meet demands as the event triggers occur. With a pay-as-you-go pricing model, users only pay for the resource consumed during their function invocations, making serverless computing a cost-effective solution.

Cloud providers designed serverless functions to be stateless, meaning that they do not retain state between function invocations. This intentional statelessness is a fundamental aspect for achieving high elasticity. By eliminating the need to store state within the function invocation, serverless platforms promote scalability and ease of deployment. Cloud providers can execute functions in parallel, allowing for efficient resource utilization. Any data needed between function invocations must be stored in remote storage.

Although the stateless nature of serverless computing is key to achieve high elasticity, it limits the type of applications that can run efficiently on serverless platforms. Previous studies [1] have found that data-intensive applications running in serverless platforms (i.e., data analytics, ML workflows, databases) are limited by the capacity and performance gaps that exist among the existing storage services. Object storage services, such as AWS S3, provide cheap long-term storage, but exhibits high access latencies. On the other hand, in-memory clusters, such as AWS ElastiCache, exhibit low access latencies and high throughput, but they are expensive and are not transparently provisioned. In between, key-value databases, such as AWS DynamoDB, provide high throughput, but are expensive and can take a long time to scale.

Given the limitations of existing storage solutions, previous works motivate the development of a serverless storage service capable of handling the wide variety of workloads running on serverless platforms. These studies mention three requirements that such service must meet. First, it should provide low latency and high throughput for a wide range of object size and data access patterns. Second, it should be transparently provisioned and should scale to meet workload demands. Third, it must ensure isolation and predictable performance across applications and tenants.

To meet the first requirement, cloud providers must first close the capacity and performance gap between main memory and persistent storage media. As mentioned above, existing storage service have fixed tradeoffs that reflect the traditional memory hierarchy built from RAM, flash memory, and magnetic disk drives. Leveraging Non-volatile memory is a promising approach to bridge the gap between the memory and storage tiers. Non-volatile memory combines the persistence and capacity of traditional storage with the low latency and byte addressability of main memory. This technology experienced a breakthrough with the release of Intel Optane DC Persistent Memory.

Non-volatile memory technology experienced a breakthrough with the release of Intel Optane DC Persistent Memory Module (PMM). Optane PMM is an emerging technology where non-volatile media is placed in a Dual In-Line Memory Module (DIMM) and installed on the memory bus, alongside traditional DRAM (Dynamic Random Access Memory) [2]. Similar to DRAM, this technology presents a byte-addressable interface and achieves speeds comparable to DRAM (2x-3x lower). The main difference between the two is that Optane PMM has higher capacities and can retain data when the system is shutdown or loses power. This allows Optane PMM to be used as a form of persistent storage with memory-like speeds.

The unique combination of persistence and low access latency makes Optane PMM an ideal candidate to speed up data-intensive workloads running in serverless platforms. Thus, thesis presents an analysis on how to make efficient use of Optane PMM to build a serverless storage service.

1.2 Research Questions

With the release of Intel Optane DIMM, researchers have started to understand its characteristics, capabilities, and limitations [3–5]. The initial expectation was that Intel Optane DC PMM would behave similar to DRAM, but with a lower performance (higher latency and lower bandwidth). However, recent studies suggest that it should not be treated as a "slower, persistent DRAM". Compared to DRAM, Optane DC PMM exhibits complicated behaviors and its performance changes based on multiple factors, such as the access size, access type (read vs. write), and degree of concurrency.

Intel Optane DC PMM differs from DRAM in two ways. First, there is a mismatch between the CPU cacheline access granularity (64-byte) and the 3D-XPoint media access granularity (256-byte) in Intel Optane DC PMM. This difference can lead to write or read amplification if the data access is smaller than 256 bytes. Second, to balance the gap in access granularity, the Intel Optane DC PMM implements a small (16KB) write-combining buffer to merge small writes and reduce write amplification. However, the buffer's limited capacity (16 KB) can cause contention within the device, limiting its ability to handle access from multiple threads simultaneously.

The complex behavior of Intel Optane DC PMM introduces interesting challenges for building a serverless storage service using this technology. Previous works have found that serverless functions vary considerable in multiple ways, including the way they access and process data, and their quality-of-service (QoS) demands. Furthermore, these workloads can spike by orders of magnitude and change dramatically over time. Knowing how these large-scale variations affect the system's performance and QoS for applications can assist in building an efficient serverless storage service.

Consequently, this thesis addresses the following research questions:

- How does Optane PMM affect the system's performance when used as persistent storage for serverless functions?
- How does Optane PMM performance under serverless workloads affect the (QoS) for applications?
- How can we overcome the limitations of Optane PMM to make efficient use of the device in a serverless scenario?
- How do we keep the system optimized and compliant with QoS requirements over time as workload shifts occur?

1.3 Contributions

The experiments described in Section 3 provide various helpful insights on the Optane PMM behavior when used as persistent storage for serverless workloads. First, we discover that sharing the Optane PMM among hundreds of serverless functions lead to performance loss (higher latency and lower bandwidth) in the device. This fact was expected given the contention issues experienced by Optane PMM with higher thread counts. Second, we discover that, depending on the workloads, the performance degradation in Optane PMM affects one performance metric more than the other (latency vs. bandwidth). This suggests that QoS of some applications might be affected more than others. Therefore, we conclude the success of Optane PMM should be measured by its capability of meeting the QoS requirements of the current workload.

To help alleviate the limitations of Intel Optane PMM, we introduce a control layer that runs on top of Optane and guides the efficient use of the device under dynamic workloads. Our control layer, called NVM Middleware, is designed to limit the access to persistent memory to reduce its contention. While doing so, the NVM Middleware keeps track of the type of applications running in the system and applies different optimization policies for each one to ensure that their QoS requirements are met. Using machine learning, the

NVM Middleware learns how to scale resources to meet the current demand and dynamically adapts them to changing workloads. We propose using online reinforcement learning algorithms, given that data access patterns in serverless workloads can change over time.

- We present an experimental study that describes the capabilities and limitations of Intel Optane PMM when used as persistent storage for serverless workloads. To our knowledge, Optane PMM has not been tested yet in this scenario.
- We present the NVM Middleware, a control layer promotes the efficient use of Optane PMM, while ensuring that QoS requirements for different type of applications are met.
- We propose a Reinforcement Learning model and framework that allows the NVM Middleware to learn from historical data and adapt resources to changing workloads.
- Finally, we present empirical results that demonstrate the benefits of our solution.

Chapter 2: Background and Methodology

2.1 Intel Optane DC Persistent Memory

Persistent memory, also known as Non-volatile Memory (NVM), is a new addition to the memory/storage hierarchy shown in Figure 2 that fills the performance/capacity gap between DRAM and storage by combining traits of both worlds. Like DRAM, persistent memory comes in the form of Dual In-line Memory Modules (DIMMs) that reside on the memory bus. Therefore, applications can access persistent memory like they do with traditional DRAM, eliminating the need to page blocks of data back and forth between memory and storage. However, unlike DRAM DIMMs, persistent memory DIMMs offer greater capacity and can retain data when the system is shutdown or loses power. Thus, persistent memory can dramatically increase system performance and enable a fundamental change in computing architecture.

Intel Optane DC Persistent Memory Module (PMM) is the first commercially available persistent memory technology. This technology comes in DIMM form factor and embeds capacities up to 512GiB. Intel Cascade Lake processors are the first CPUs to support Intel Optane PMM. Like traditional DRAM, the Optane DIMM sits on the memory bus and connects to the processor's integrated memory controller (iMC). Figure 1 shows a typical system configuration of a hybrid node with DRAM and PMM. A user can have up to one Intel Optane DIMM per channel and up to six on a single socket providing capacities up to 3TiB per socket. Thus, an 8-socket system could access up to 24TB of persistent memory.

To ensure persistence, Intel Optane PMM sits within Intel's asynchronous DRAM refresh (ADR) domain. Intel's ADR domain ensures that CPU stores that reach the ADR domain will survive a power failure. The iMC maintains read and write pending queues (RPQs and WPQs) for each Optane DIMM and the ADR domain includes WPQs. Once the data reaches the WPQs, the ADR domain ensures that the iMC will flush the updates to persistent memory media on power failure.

The iMC communicates with the Optane DIMM using the DDR-T protocol in cache line access granularity (64B) (Figure 2). The memory access to NVDIMM arrive first at an Apache Pass Controller which coordinates access to the Optane Media. Similar to SSDs, the Optane DIMM perforsms address translation for wear-leveling and bad block management. Thus, it keeps an address indirection table (AIT) for this translation.

The actual access to storage media occurs after address translation. Intel Optane DIMM physical media access granularity is 256 bytes. Thus, the Controller translates smaller requests into largest 256-byte accesses, causing write amplification as small stores become read-modify-write operations. The controller has a small write-combining buffer to merge adjacent writes.

Intel Optane PMem can operate in two modes: memory and App Direct. Memory mode uses Optane PMem as a large capacity main memory without persistence. DRAM is not visible to the users, and instead it serves as a cache for Optane PMem that is transparently managed by the operating system. In App Direct mode, Optane PMem DIMMs appear as independent, non-volatile storage devices. This allows Optane PMem to be used as a byte-addressable persistent memory that is mapped into the system physical address space and directly accessible by applications [].

2.2 Serverless Computing

Serverless computing is a cloud computing execution mode that enables developers to deploy their code without provisioning or managing server infrastructure. The term “serverless” is misleading, as servers are still being used by cloud providers to run the code for developers. However, instead of requesting and managing resources, developers simply provide their code, and the cloud providers handle the servers on behalf of their customers. Cloud providers are responsible for provisioning resources, scaling, fault tolerance, monitoring, security patches, and so on. Finally, developers simply pay by the execution time and resources used on their code invocations.

Function-as-a-service (FaaS) is the core compute engine for serverless computing. It was first introduced on 2015 by AWS Lambda, and since then, other commercial and open-source offerings have appeared, i.e., Google Cloud Functions, Azure Functions, Apache OpenWhisk, and others. With FaaS, a developer implements the application logic as stateless functions in a high-level language, such as Java, Python, C, C++, and so on. The code is then packaged together with its dependencies and submitted to the serverless platform. Finally, the developer associates an event to each function, i.e., HTTP requests, file uploads, and more. Once a trigger is fired, the cloud provider executes the code associated with that trigger.

2.3 Reinforcement Learning

Reinforcement Learning considers a problem of a learning agent that actively learns from its own experience. Such agent interacts with its environment and periodically receives a reward signal. The agent's goal is to maximize the rewards in the long run. However, the agent is not told which actions to take. Instead, it must discover the actions that yield the highest rewards through trial-and-error.

Figure 1 illustrates a typical reinforcement learning scenario. The agent interacts with the environment in discrete time steps. At each time step t, the agent senses the environment's current state st E S, where S represent the full set of environment states. It then chooses an action at E A(st), where A(st) represents the set of all actions available in the current state. The environment moves to a new state st+1, and the agent receives a reward rt associated with the transition (st,at,st+1).

At any given time, the agent's behavior is defined by a policy. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states. The agent's purpose is to learn the optimal, or near-optimal, policy that maximizes total reward it receives in the long run.

Exploration vs Exploitation tradeoff One of the challenges that arises in Reinforcement Learning is the choice between exploiting a familiar action known for a reward and exploring unfamiliar actions for unknown rewards, known as the exploration-exploitation tradeoff. The dilemma is that the agent cannot pursue exploration nor exploitation exclusively without failing the task. Instead, it must find a balance between exploration and exploitation. The agent must try a variety of actions to gather enough information and progressively favor for those that appear to be the best.

QLearning Q-Learning is a model-free reinforcement learning algorithm, where the agent learns which is the best action to take given the current state. The agent assess the quality of an action by means of a quality-function (Q-function) Q(s,a), denoting the expected total discounted reward if the agent takes action a on state s and acts optimally thereafter. Given the Q-function, the agent's optimal policy is to choose the action that yields the highest reward.

Figure 4 illustrates the Q-Learning algorithm. The Q-function can be implemented using a simple lookup table. At each step, the agent selects an action a, and observes the reward r and the new state st+1. Then, the agent applies one-step Q-learning, given by: Qlearning formula Where 0¡alpha¡1 is the learning rate and determines to what extent new information overrides the old one. The learned Q-function directly approximates the optimal Q-function, independent of the policy being followed.

Chapter 3: A shim layer for persistent memory

As we have discussed, the release of Intel Optane PMM opens a major opportunity for serverless storage services. This memory technology provides a unique combination of affordable larger capacity, high-performance, and support for data persistence [6]. When configured in App-Direct mode, the Optane DIMM and DRAM DIMMs act as independent memory resources under direct load/store control of the applications. This allows the Optane PMM capacity to be used as byte-addressable persistent memory that is mapped into the system application space and directly accessible by applications. Together, these advantages enable Optane PMM to be used as persistent storage with memory-like speeds.

Unfortunately, the resource contention observed within Optane PMM can impose serious performance and contractual implications for a multi-tenant serverless storage service. Given the hallmark autoscaling features of serverless computing, the memory's limited ability to handle accesses from multiple threads can degrade the overall system's performance when workload spikes occur. Furthermore, these storage systems make efficient use of their infrastructure by allowing multiple users, or tenants, to share the physical resources. The performance degradation caused by Optane PMM can lead tenants to experience significant performance variations. The latter inhibits service providers from offering certain service level agreements.

To reduce the contention effect, previous studies recommend limiting the number of threads that access Optane PMM simultaneously. In [4], Yang et. al they improve the performance of an NVM-aware file system by limiting the number of writer threads that access each Optane DIMM. Similarly, Ribbon [5] controls the number of threads performing CLF and adjusts this number dynamically at runtime. While this approach provides a viable solution, it introduces management problems for a system administrator of a multi-tenant serverless storage.

Given the complexity of serverless computing workloads, implementing efficient concurrency control mechanisms for optimizing an Optane-based serverless storage service is a challenging task. These challenges are discussed in section 3.1, but in short, service providers have three crucial tasks when implementing these control mechanisms. First, they must provide predictable performance, ensuring that all the SLAs from different workloads are met. Second, they must scale resources transparently to meet the current workload demand. Finally, they must come up with policies that allow their system to adapt quickly to sudden workload shifts. To this end, we propose a solution that takes on these responsibilities from the service providers.

In this work, we present a shim layer that addresses the shortcomings of Intel Optane PMM highlighted above, while meeting the different service level agreements from multiple tenants under shifting workloads. Our shim layer, called NVM Middleware, distinguishes between latency-critical and throughput-oriented workloads and applies different concurrency control mechanisms for each one. This enables the system to reduce the contention on the memory device, as well as the interference among workloads with different service level agreements. In addition, we propose the development of a reinforcement learning agent to adapt the NVM Middleware quickly to changing workloads. The agent takes into account the characteristics and service level agreements and learns from past experiences to scale resources accordingly.

3.1 Motivation

In this section, we discuss the pain points of controlling the number of threads to optimize Optane PMM within a serverless storage service and explain the design goals of the NVM Middleware.

3.1.1 Concurrency Control Challenges in a serverless storage service

When building an Optane PMM based serverless storage service, optimizing the memory's performance is just the start. Early works in serverless computing have identified several tasks that a storage service must perform efficiently to meet the demands of serverless computing [1, 7–11]. As a result, service providers must ensure that their concurrency control policies do not interfere with these design goals. In this work, we focus on three challenges faced by service providers when designing a high-performance storage service based on Optane PMM.

Support for a wide heterogeneity of applications. In serverless computing, users typically deploy their applications as a collection of serverless functions that share data among them using remote storage. Previous studies suggest that these applications vary considerably in the way store, distribute, and process data. This diversity is reflected in multiple ways, such as data access size [9, 11], data access patterns [11], and their performance requirements [180275,jonas2019cloud]. Therefore, service providers face the challenge of tuning the concurrency level to support many types of applications. In this work, we argue that considering the workload characteristics is key for tuning the system efficiently. The allocation of resources can vary depending on the workload type.

Compliance with Service Level Agreements. The success of a storage service relies on its ability to comply with various service level agreements (SLAs). SLAs play a critical role in governing the relationship between the storage provider and its customers. They help establish clear expectations between both parties regarding the quality of storage service. Therefore, service providers face the challenge of staying in compliance with these SLAs while they seek to optimize Optane PMM.

Automatic and transparent scaling. Serverless workloads are extremely unpredictable. These workloads can launch hundreds of functions instantaneously to meet application demands [8]. Furthermore, the data access patterns of the applications can change dramatically over time [10, 11]. Service providers face the challenge of scaling the resources, such as number of threads, automatically to meet the demands of changing workloads. In addition, they must ensure that the system adapts quickly enough to avoid missing SLAs.

3.1.2 NVM Middleware Design Overview

We design NVM Middleware with three main design goals.

Workload-aware Contention Management. We focus our work on two main types of workloads: interactive and batch applications. Interactive applications, such as web-based platforms, enable real-time interactions between the user and the application. Low latency is critical to ensure that the user input is processed quickly, and feedback is delivered in real-time. On the other hand, batch applications, such as data analytics jobs, facilitate efficient processing of large-scale data. These workloads prioritize high throughput to process large volumes of data efficiently.

The NVM Middleware leverage insights about the workload characteristics, resource demands, and performance requirements of applications to make informed decisions about resource allocation and contention resolution. By dynamically adjusting resource allocation and contention resolution mechanisms based on the workload characteristics, the NVM Middleware mitigates contention-induced performance degradation and ensures efficient resource sharing among co-located applications. This adaptive approach enables the NVM Middleware to allocate resources judiciously to maximize overall system efficiency and meet diverse performance requirements of both interactive and batch applications. By using the content-aware contention management offered by the NVM Middleware, a storage system using Optane PMM can effectively balance the needs of different workload types, ensuring optimal performance and resources utilizing in multi-tenant environments.

SLA-driven autoscaling policies. The NVM Middleware leverages SLAs, which define the quality-of-service parameters agreed up between the service provider and their customers, to dynamically adjust contention resolution mechanisms in response to changes in service level agreement metrics. It continuously monitors SLA metrics, such as 99th latency and throughput, and evaluates its own performance against predefined SLA targets. This real-time monitoring allows the NVM Middleware to detect deviations from SLA requirements and triggers scaling actions to dynamically adjust resource allocation. By aligning resource provisioning with SLA requirements, the NVM Middleware can ensure a consistent and reliable performance from Optane PMM, even under dynamic workload changes.

RL-driven autoscaling policies. Besides leveraging SLAs to dynamically provision resources and adjust contention resolution mechanisms, our solution proposes the use of Reinforcement Learning to learn from past experiences and predict future behaviors. These RL-driven policies enable the NVM Middleware to adapt to changing workload patterns over time and meet SLAs objectives more effectively than traditional threshold-based approaches []. Moreover, given the dynamic and unpredictable of serverless workloads, we propose a model-free algorithm, Q-Learning, to continuously learn the optimal policy based on observed experiences, allowing the NVM Middleware to adapt to new scenarios without needing to explicitly model them.

3.2 Architecture

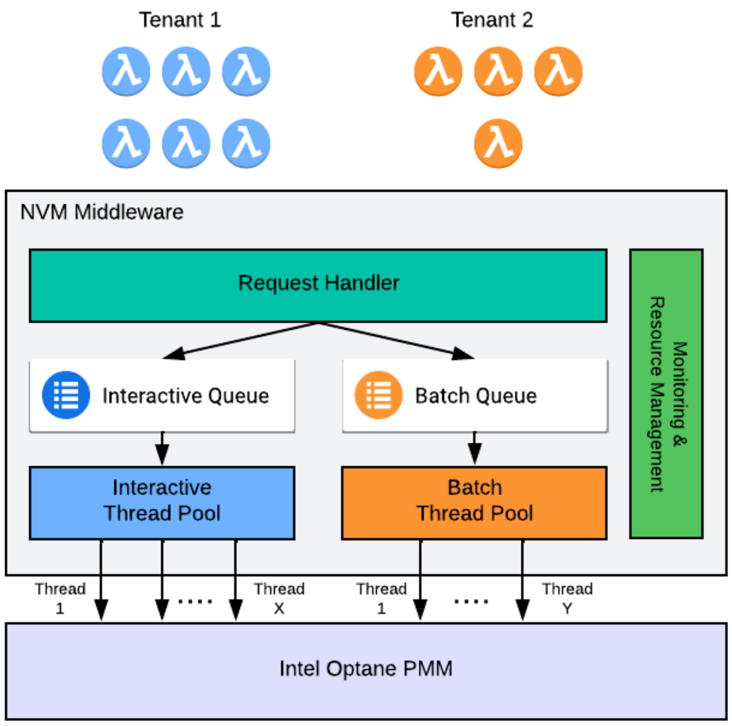

Figure 3.1: NVM Middleware Architecture

Figure 3.1 provides an overview of the NVM Middleware architecture. Positioned as a middle layer connecting user applications with Optane PMM, its design is tailored for seamless integration within a storage service, serving as an optimization layer specifically targeting Optane PMM. It comprises a request handler, two concurrency thread pools, and a monitoring and resource management module.

The request handler serves as the primary interface for handling user I/O requests. Upon receipt, it segregates requests into two distinct non-blocking First-In-First-Out (FIFO) queues: one tailed for latency-sensitive requests and the other for throughput-centric ones. Leveraging insights into workload characteristics, the handler intelligently allocates requests to the appropriate queue. Moreover, each queue is assigned a dedicated pool of worker threads tasked with dispatching I/O requests to Optane PMM using PMEMKV. Notably, these thread pools operate independently and are dynamically managed and scaled by the Reinforcement Learning agent to meet predetermined latency and throughput goals.

The Monitoring and Resource Management module offers an interface to monitor system load and SLA performance metrics. This module initiates a separate control thread tasked with gathering data on key parameters within the NVM Middleware, such as 99th latency, throughput, and system load. Utilizing this information, the RL agent makes data-driven decisions regarding optimal thread pool scaling. Subsequently, these decisions are communicated to the Monitoring and Resource Management module, which executes the required actions within the NVM Middleware.

3.3 Programming Interface

Table 3.1: Programming Interface

Category	API Name	Functionality
System	start(db, interactiveThreads, batchThreads)	Create PMEMKV database. Start interactive and batch thread pools. Initiate system monitoring in Monitoring and Resource Management Module.
System	stop()	Closes PMEMKV database. Stop thread pools. Stop system monitoring.
System	get(key, mode)	Retrieves key from persistent memory.
System	put(key, value, mode)	Writes key to persistent memory.
RL	get_stats()	Provides the 99th percentile and throughput observed by the NVM Middleware.
RL	get_state()	Provides the current state within the NVM Middleware.
RL	perform_action(action)	Triggers a scaling action.

Table 3.1 outlines the NVM Middleware's programming interface, presenting a set of functions designed to facilitate interaction with a storage system and the reinforcement learning agent.

The *start* function initializes the PMEMKV database, initializes the thread pools with an initial thread count, and triggers the system monitoring within the Monitoring and

Resource Management Module. In contrast, the *stop* function terminates the database connection, halts all threads in the thread pools, and stops system monitoring. Furthermore, the *get* and *put* functions facilitate key-value interactions with the persistent memory, allowing for read and write operations. These functions are designed to accommodate hints regarding the request type (e.g., latency-sensitive or throughput-oriented), aiding the request handler in queue allocation.

The *get_stats* function furnishes insights into the 99th percentile and throughput observed by the NVM Middleware at any given moment. Similarly, the *get_state* function provides real-time state information as outlined in Table 3.2. Finally, the $perform\_action$ function accepts scaling actions detailed in Table 3.3 and initiates the corresponding action within the NVM Middleware.

3.4 Reinforcement Learning Component

In this section, we discuss the Q-learning algorithm used by the Reinforcement Learning agent to dynamically adjust the number of threads assigned to each thread pool. The agent's goal is to find the best combination of threads that meets predetermined latency and throughput SLAs while minimizing contention and ensuring efficient utilization of Intel Optane PMM.

3.4.1 Integration with the NVM Middleware

Figure 3.2 offers a visual representation of the interaction between the reinforcement learning (RL) agent and the NVM Middleware. At each time step, the NVM Middleware receives a diverse influx of requests, spanning both latency-sensitive and throughput-oriented tasks. These requests necessitate translation into actionable I/O commands directed towards the Intel Optane Persistent Memory Module (PMM).

Concurrently, the RL agent adeptly captures the environment's current state, leveraging real-time workloads' characteristics and performance metrics provided by the monitoring module. Utilizing this information, the agent orchestrates the selection of an optimal action,

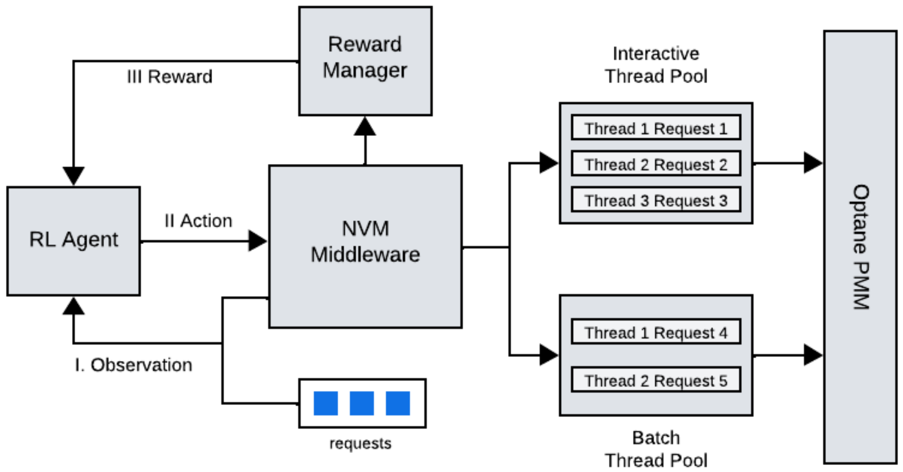

Figure 3.2: RL Workflow

guiding the dynamic adjustment of threads within the interactive and batch thread pools. This adaptive decision-making process is exemplified by actions like augmenting the count of interactive threads to address evolving workload demands.

Following action selection, the NVM Middleware's resource management module implements the chosen course of action, fine-tuning the NVM Middleware's interactive and batch threads to efficiently handle incoming user requests. Upon the completion of each time step, the action's effectiveness is rigorously assessed against predefined service level agreement (SLA) targets, yielding a reward signal generated by a reward manager.

This reward serves as invaluable feedback for the RL agent, empowering iterative policy updates aimed at refining decision-making strategies in subsequent time steps. Thus, the presented framework embodies a recursive learning cycle, wherein the RL agent continuously hones its behavior through real-world interactions, ensuring adaptive responsiveness to evolving workload dynamics.

3.4.2 Reinforcement Learning Model

State Space

Table 3.2: The State Representation

Name	Description	Values
interactiveThreads	Number of (interactive) threads assigned to the interactive thread pool.	$1 \leq$ interactiveThreads ≤ 32
batchThreads	Number of (batch) threads assigned to the batch thread pool.	$1 \leq$ batchThreads ≤ 32
interactiveQueueSize	Number of requests in the interactive queue.	interactiveQueueSize $\in \mathbb{R}^+$
batchQueueSize	Number of requests in the batch queue.	batchQueueSize $\in \mathbb{R}^+$
interactiveBlockSize	Average block size of interactive workload.	interactiveBlockSize $\in \mathbb{R}^+$
batchBlockSize	Average block size of batch workload.	batchBlockSize $\in \mathbb{R}^+$
interactiveWriteRatio	Proportion of write requests compared to read requests in the interactive workload.	interactiveRWRatio $\in \mathbb{R}^+$
batchWriteRatio	Proportion of write requests compared to read requests in the batch workload.	batchRWRatio $\in \mathbb{R}^+$

Table 3.2 presents the features of our state representation. At each time step t, we define the state s_t as a tuple:

$$s_t = (\text{interactiveThreads}_t, \text{batchThreads}_t, \text{InteractiveQueueSize}_t, \text{batchQueueSize}_t,$$
$$\text{interactiveBlockSize}_t, \text{batchBlockSize}_t, \text{interactiveRWRatio}_t, \text{batchRWRatio}_t)$$

where $s_t \in S$ represents the state space. The tuple encapsulates the key features characterizing the system's current state, including the number of interactive and batch threads, number of pending requests in the queues, individual workload block sizes, and write ratio for both interactive and batch workloads.

Table 3.3: Possible Actions in the Action Space

Action	Effect on Interactive Threads	Effect on Batch Threads
0	No change	No change
1	Increase by 1	No change
2	Decrease by 1	No change
3	No change	Increase by 1
4	No change	Decrease by 1
5	Increase by 1	Increase by 1
6	Increase by 1	Decrease by 1
7	Decrease by 1	Increase by 1
8	Decrease by 1	Decrease by 1

Action Space

Table 3.3 illustrates the feasible actions within the action space. Each action corresponds to a potential adjustment in the number of interactive and batch threads. The table enumerates nine distinct actions, each with its respective effect on the number of interactive threads and batch threads.

Mathematically, the set of actions A is defined as $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ for a given state $s_t \in S$.

Reward

To guide the optimization process of the reinforcement learning agent, we establish an algorithm (Algorithm 1) to calculate a reward value based on observed and target latency and throughput metrics. This algorithm, outlined below, serves as a crucial component in training the RL agent to make informed decisions.

1. Lines 1-5 define goals, scaling factors, and penalties. The observed and target latency (*lat*, *lat_goal*) and throughput (*tp*, *tp_goal*) metrics are scaled to a normalized range using scaling factors (*max_scale_lat*, *max_scale_tp*) and minimum scale (*min_scale*). This normalization process ensures that both metrics contribute proportionally to the reward calculation.

2. Lines 6-7 compare the scaled latency (*lat*) and throughput (*tp*) metrics against the scaled target values for latency (*lat_goal*) and throughput (*tp_goal*). The absolute differences between observed and target values are computed to quantify the error in latency (*error_lat*) and throughput (*error_tp*).

3. Lines 8-12 determine the reward based on three distinct scenarios. Firstly, if both latency and throughput goals are achieved, a high positive reward is assigned. Secondly, if both goals are not met, a low negative reward is assigned, taking into account both latency and throughput errors. The disparity in penalties, represented by *lat_penalty* and *tp_penalty*, ensures that both types of errors contribute proportionately to the overall reward. Thirdly, if only the latency goal remains unmet, a low negative reward is assigned, incorporating the latency penalty and error. Finally, if only the throughput goal is unmet, a similar low negative reward is assigned, encompassing the throughput penalty and error.

3.4.3 Training Methodology

Environment Design

The environment architecture designed for training and evaluating the RL agent is depicted in Figure 3.3. This architecture comprises several key components, including an interactive multi-threaded application, a batch multi-threaded application, the NVM Middleware, and Intel Optane PMM.

To simulate a multi-tenant serverless scenario, both applications are executed concurrently. Workload patterns for each application are derived from collected serverless traces. To emulate high concurrency levels typical in serverless environments, multiple threads within each application are employed to dispatch requests to the NVM Middleware via the API described in Section 3.3. Meanwhile, the NVM Middleware processes these requests in accordance with the workflow outlined in Section 3.2.

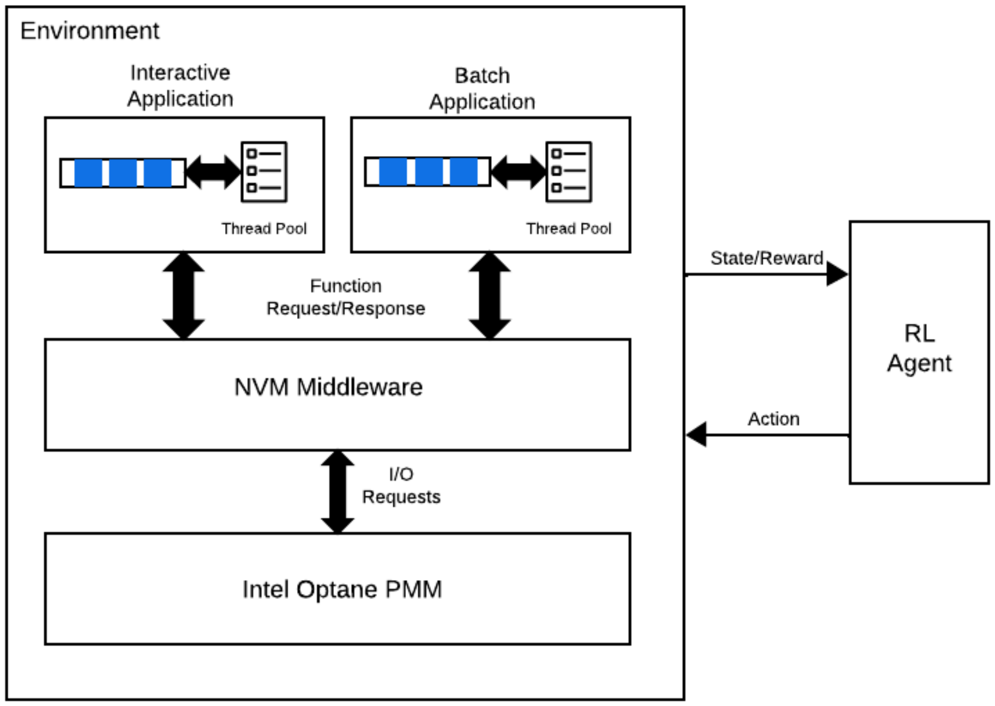

Figure 3.3: Overview of the Environment Architecture

In order to model the time steps inherent in an RL process, the environment organizes the applications' requests into 1-second windows, processing one window per time step. Figure 3.4 illustrates the interactions between the RL agent and the environment at each time step. Beginning with a state observation from the preceding step, the agent communicates the intended action to the environment. Subsequently, the environment relays this action to the NVM Middleware, which then allocates resources accordingly. Upon successful execution of the action, the environment initiates processing for the next window of requests. Once all requests within the window are handled, the environment gathers metrics from the NVM Middleware and furnishes a new state observation along with a reward signal to

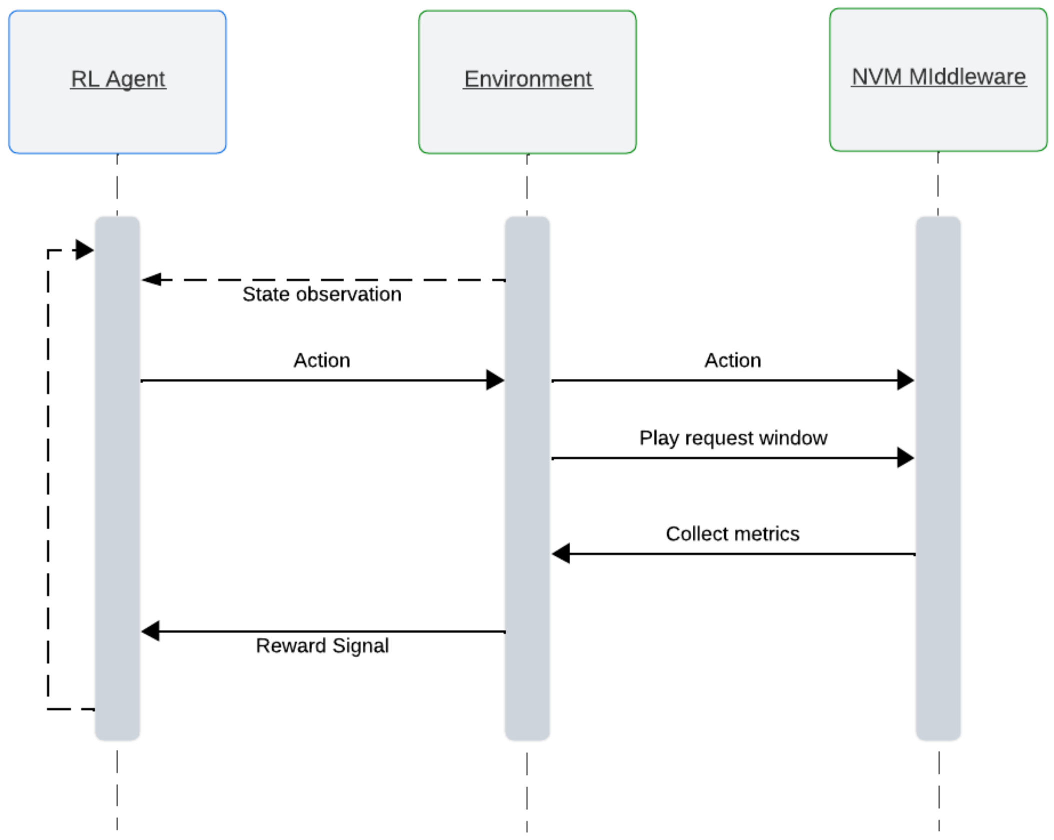

Figure 3.4: Agent Process flow

the agent. The agent utilizes this reward to update its policy, perpetuating the iterative learning process.

Function Approximation

To address the challenge posed by the continuous state space in our environment, traditional Q-learning approaches become impractical due to the vast number of states that cannot be feasibly mapped into a Q-table. Consequently, we employ function approximation techniques to estimate the value of each action based on the state.

Specifically, we train nine separate linear regression models, each corresponding to one of the available actions, using stochastic gradient descent. This approach allows us to capture the underlying patterns in the data and generalize across states not encountered during training, enabling our agent to make informed decisions even in novel situations.

However, selecting appropriate hyperparameters for our regression models presents a significant challenge. Online training alone is insufficient for accurately assessing model performance, as it can be time-consuming and computationally intensive. To overcome this limitation, we adopt a batch learning approach with offline historical data.

By leveraging historical data collected from the environment, we can tune our models' hyperparameters and incorporate prior knowledge into our RL agent. This approach accelerates the learning process by bootstrapping our models with valuable insights gained from past experiences [12, 13].

To construct our dataset, we deploy a non-optimal agent that performs random actions in the environment, capturing state-action-reward transitions. Following established machine learning practices, we split the dataset into training and testing sets and employ 5-fold cross-validation on the training set to evaluate model performance rigorously.

Additionally, we preprocess the features by standardizing them using the standard scaler and apply polynomial preprocessing to enhance the model's ability to capture nonlinear relationships within the data.

Proposed Q-Learning Algorithm

Algorithm 2 outlines the Q-Learning process for training an agent to make optimal decisions in an environment. It takes the bootstrapped Q-value models M_a for all actions a and outputs the new learned models after training.

The algorithm initializes the training parameters and then iterates over a specified number of episodes. Within each episode, the environment is reset, and the agent interacts with it until the episode is complete. At each step, the agent observes the current state s_t, selects an action a_t based on an ϵ-greedy policy, takes the action, and observes the resulting reward r and next state s_{t+1}.

The Q-value models are updated based on the observed reward and next state. If the episode is not done, the target Q-value is calculated using the reward and the maximum Q-value for the next state. If the episode is done, the target Q-value is simply set to the reward.

The model for the selected action a_t is updated using the target Q-value, and the state is updated to the next state. Additionally, the exploration rate ϵ is decreased according to an exploration schedule.

3.5 Implementation

The NVM Middleware, detailed in Section 3.3, is implemented using C++. We leverage PMEMKV from the Persistent Memory Development Kit [14] to facilitate reading and writing data into Intel Optane PMM. To manage concurrent operations efficiently, we utilize the non-locking, concurrent queue provided by the Intel Threading Building Blocks [15] library for both the interactive and batch queues.

For the RL Environment, as described in Section 3.4.3, we adopt a hybrid approach employing C++ and Python. The environment itself is constructed in C++, aligning with the specifications outlined in Section 3.4.3. Conversely, the RL agent and the Q-Learning algorithm, also discussed in the same section, are developed using Python. We leverage the SGDRegressor model from the Scikit-learn[16] library to facilitate the representation of our linear regression models for function approximation. Additionally, we employ Scikit-learn for hyperparameter tuning. To seamlessly integrate the C++ and Python components, we utilize pybind11[17].

Algorithm 1: Reward Calculation Algorithm

```
   Input: System statistics: stat
   Output: Reward value: reward
   /* Initialize variables                                              */
1  max_scale_lat ← 1000, max_scale_tp ← 10, min_scale ← 1, lat_goal ← 200,
     tp_goal ← 250000, lat_penalty ← 500.0, tp_penalty ← 5000.0;
   /* Scale observed and target latency and throughput                  */
2  lat ← ((max_scale_lat − min_scale) × (stat.tailLatency − min_value)/(max_latency −
     min_value)) + min_scale;
3  tp ← ((max_scale_tp − min_scale) × (stat.throughput −
     min_value)/(max_throughput − min_value)) + min_scale;
4  lat_goal ← ((max_scale_lat − min_scale) × (lat_goal − min_value)/(max_latency −
     min_value)) + min_scale;
5  tp_goal ← ((max_scale_tp − min_scale) × (tp_goal − min_value)/(max_throughput −
     min_value)) + min_scale;
   /* Calculate errors                                                  */
6  error_lat ← |lat − lat_goal|;
7  error_tp ← |tp − tp_goal|;
   /* Calculate reward                                                  */
8  if lat ≤ lat_goal and tp ≥ tp_goal then
9      reward ← 10 × (error_lat + error_tp) ;     // High reward for meeting both
         latency and throughput goals
10 else
11     if lat > lat_goal and tp < tp_goal then
12         reward ← −1 × (lat_penalty × error_lat + tp_penalty × error_tp) ;
             // Penalize for high latency and low throughput
13     else
14         if lat > lat_goal then
15             reward ← −1 × lat_penalty × error_lat ; // Penalize for high latency
16         else
17             reward ← −1 × tp_penalty × error_tp ;          // Penalize for low
                 throughput
18         end
19     end
20 end
```

Algorithm 2: Q-Learning Algorithm

```
   Input: Pre-trained Q-value models $M_a$ for all actions $a$
   Output: Learned Q-value models $M_a$ for all actions $a$
1  Initialize the training parameters $\alpha$, $\gamma$, $\epsilon$;
2  for $episode \leftarrow 1$ to $E$ do
3      Reset the environment;
4      repeat
5          Observe the state $s_t$;
           // Choose action $a_t$ using the $\epsilon$-greedy policy
6          Generate random number $r$ from uniform distribution in [0, 1];
7          if $r < \epsilon$ then
8              Select a random action $a_t$ from the action space ;
9          end
10         else
11             for each action $a$ do
12                 Predict Q-value $Q_a(s_t)$ using model $M_a$: $Q_a(s_t) \leftarrow M_a.predict(s_t)$ ;
13             end
14             Select action $a_t \leftarrow \arg\max_a Q_a(s_t)$ ;
15         end
16         Take action $a_t$, observe reward $r$ and next state $s_{t+1}$;
           // Update the Q-value model using reward and next state
17         if not done then
18             for each action $a$ do
19                 Predict Q-value $Q_a(s_{t+1})$ using model $M_a$:
                     $Q_a(s_{t+1}) \leftarrow M_a.predict(s_{t+1})$ ;
20             end
21             Calculate target Q-value: $target \leftarrow r + \gamma \cdot \max_a Q_a(s_{t+1})$ ;
22         end
23         else
24             Set target Q-value to the reward: $target \leftarrow r$ ;
25         end
26         Update the model for action $a_t$ with the target Q-value:
             $M_{a_t}.partial\_fit(s_t, target)$ ;
27         Update state: $s_t \leftarrow s_{t+1}$;
28     until episode is done;
29     Decrease $\epsilon$ according to exploration schedule;
30 end
```

Chapter 4: Evaluation

Chapter 5: Related Work

Chapter 6: Conclusions and Future Work

Bibliography

[1] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[2] Boston. What is intel optane dc persistent memory? `https://www.boston.co.uk/blog/2019/07/10/intel-optane-dc-persistant-memory.aspx`. (Accessed on 02/02/2024).

[3] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[4] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, 2020.

[5] Kai Wu, Ivy Peng, Jie Ren, and Dong Li. Ribbon: High performance cache line flushing for persistent memory. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 427–439, 2020.

[6] Intel. Intel® optane™ persistent memory. `https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html`. (Accessed on 02/02/2024).

[7] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for Multi-Tenant cloud storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 349–362, Hollywood, CA, October 2012. USENIX Association.

[8] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. Understanding ephemeral storage for serverless analytics. In *2018 USENIX annual technical conference (USENIX ATC 18)*, pages 789–794, 2018.

[9] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, 2018.

[10] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. Autoscaling tiered cloud storage in anna. *Proceedings of the VLDB Endowment*, 12(6):624–638, 2019.

[11] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$t: A transparent auto-scaling cache for serverless applications, 2021.

[12] Ignacio Cano, Srinivas Aiyar, Varun Arora, Manosiz Bhattacharyya, Akhilesh Chaganti, Chern Cheah, Brent Chun, Karan Gupta, Vinayak Khot, and Arvind Krishnamurthy. Curator:{Self-Managing} storage for enterprise clusters. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 51–66, 2017.

[13] N Marivate Vukosi. *Improved Empirical Methods in Reinforcement Learning Evaluation*. PhD thesis, PhD thesis, Rutgers, New Brunswick, New Jersey, 2015.

[14] Steve Scargall. *Programming Persistent Memory: A Comprehensive Guide for Developers*. Apress, USA, 1st edition, 2020.

[15] Intel Corporation. Intel® oneapi threading building blocks. `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html#gs.4oc8fg`. (Accessed on 02/18/2024).

[16] scikit-learn: machine learning in python — scikit-learn 1.4.1 documentation. `https://scikit-learn.org/stable/`. (Accessed on 02/18/2024).

[17] Wenzel Jakob. pybind11 documentation. `https://pybind11.readthedocs.io/en/stable/index.html`. (Accessed on 02/18/2024).

Biography

Include your *biography* here detailing your background, education, and professional experience.