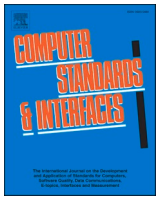
Computer Standards & Interfaces 97 (2026) 104098

Contents lists available at ScienceDirect

Computer Standards & Interfaces

journal homepage: www.elsevier.com/locate/csi



An autonomous deep reinforcement learning-based approach for memory configuration in serverless computing

Zahra Shojaee Rad, Mostafa Ghobaei-Arani *, Reza Ahsan

Department of Computer Engineering, Qo.C., Islamic Azad University, Qom, Iran

ARTICLE INFO

Keywords:
Serverless computing
Memory configuration
Deep reinforcement learning
Autonomous computing
Function-as-a-service

ABSTRACT

Serverless computing has become very popular in recent years due to its cost savings and flexibility. Serverless computing is a cloud computing model that allows developers to create and deploy code without having to manage the infrastructure. It has been embraced due to its scalability, cost savings, and ease of use. However, memory configuration is one of the important challenges in serverless computing due to the transient nature of serverless functions, which are stateless and ephemeral. In this paper, we propose an autonomous approach using deep reinforcement learning and a reward mechanism for memory configuration called Auto Opt Mem. In the Auto Opt Mem mechanism, the system learns to allocate memory resources to serverless functions in a way that balances overall performance and minimizes wastage of resources. Finally, we validate the effectiveness of our solution, the findings revealed that Auto Opt Mem mechanism enhances resource utilization, reduces operation cost and latency, and improves quality of service (QoS). Our experiments demonstrate that Auto Opt Mem mechanism achieves 16.8 % lower latency compared to static allocation, 11.8 % cost reduction, and 6.8 % improve in QoS, resource utilization, and efficient memory allocation compared with base-line methods.

1. Introduction

Serverless computing has emerged as an extended cloud computing model that offers many advantages in flexibility, scalability and cost efficiency [1]. By separating the management of the underlying infrastructure, developers can focus on writing and deploying code without worrying about server provisioning or maintenance. There has been a lot of progress in various areas related to serverless computing [2]. One aspect is Function as a Service (FaaS) that increasingly been associated with a variety of applications, including video streaming platforms [3], multimedia processing [4], Continuous Integration/Continuous Deployment (CI/CD) pipelines [5], Artificial Intelligence/Machine Learning (AI/ML) inference tasks [6], and query processing for Large Language Models (LLMs) [7]. FaaS is a serverless cloud computing model that allows developers to run small, manageable services in isolated environments called function instances [8].

Despite these advantages, memory configuration in serverless environments is a complex challenge due to the transient and stateless nature of serverless functions. Choosing the right amount of memory or resource size is important and a challenge because it can result in faster execution times and lower costs. A recent survey found that 47 % of serverless functions in production use the default memory size, indicating that developers often overlook the importance of resource size [9]. Traditional memory configuration methods are often manual settings or static allocation, which may lead to inefficiencies such as overprovisioning or underutilization. These inefficiencies can lead to increased costs or decreased performance and affect the effectiveness of the serverless function. By employing deep learning models, memory configuration can be automated, leading to an efficient solution. Deep learning models analyze historical data to dynamically predict optimal memory settings, adapt to varying workloads, and minimize latency and cost. This approach uses the ability of deep learning to identify complex patterns and relationships in data, enabling more accurate and efficient resource management.

Recent research shows the importance of intelligent and autonomous systems in different domains. For instance, Arduino-based IoT automation systems have shown how lightweight and adaptive architectures can improve efficiency and minimize manual intervention in constrained environments [10]. Likewise, autonomous AI frameworks for fraud detection in the Dark Web demonstrate the ability of self-learning mechanisms to adapt to dynamic and unpredictable conditions [11]. These advances further motivate the need for AI-driven autonomic

* Corresponding author.
E-mail address: mo.ghobaei@iau.ac.ir (M. Ghobaei-Arani).

https://doi.org/10.1016/j.csi.2025.104098
Received 2 May 2025; Received in revised form 16 November 2025; Accepted 17 November 2025
Available online 19 November 2025

solutions, such as our proposed Auto Opt Mem framework for serverless memory configuration.

1.1. Research gap and motivation

Previous approaches are often static or limited to a specific platform. The need for real-time adaptability to changing workloads is not met, and they have relied solely on statistical modeling and lacked the ability to adapt in real time. The direct relationship between cost, latency, and QoS has rarely been considered in a comprehensive framework. Our work addresses these research gaps by introducing Auto Opt Mem based on MAPE loop and DRL.

The motivation for this study is that many functions still use default memory values, resulting in wasted resources, increased costs, and reduced quality of service (QoS). For example, 47 % of users rely on default memory settings, which emphasizes the importance of intelligent and adaptive optimization [9]. Addressing this gap is essential to improve performance and cost-effectiveness in serverless environments.

1.2. Our approach

In this paper, we propose an autonomous memory configuration with deep learning model to predict memory setting based on a combination of the concept of the autonomic computing and the deep reinforcement learning (DRL) with the aim of increasing performance and cost-effectiveness. To realize autonomic computing, IBM has introduced a reference framework for autonomic control loops known as the MAPE (Monitor, Analyze, Plan, Execute) loop [12,13]. This control MAPE loop resembles the general agent model put forth by Russell and Norvig [14], where an intelligent agent observes its surroundings through sensors and utilizes these observations to decide on actions to take within that environment. The proposed approach follows the control MAPE loop, which consists of four phases: monitoring (M), analysis (A), planning (P), and execution (E). First in the monitoring phase, the system observes and collects the current state of the environment (memory in serverless functions). In the analysis phase, the agent which is a deep neural network analyzes the observed state and updates its policy based on it and the reward received. In the planning phase, the agent schedules an action (i.e., memory configuration) based on the learned policy. In the execution phase, the scheduled action is applied to the environment. We utilize Deep Reinforcement Learning (DRL) [15,16] as a decision-making tool that leverages the predicted outcomes from the analysis phase to determine the best memory configuration during the planning phase. Reinforcement Learning (RL) is a self-learning system that enhances its effectiveness by continuously interacting with the cloud environment.

1.3. Main contributions

The main contributions of this research can be summarized as follows:

- We propose an autonomic method using deep reinforcement learning method to predict memory configuration, and this method operates based on a reward mechanism.
- We designed a multi-objective reward normalization mechanism that simultaneously balances latency, cost, utilization, and QoS.
- We integrated the MAPE-K control loop with deep reinforcement learning (DRL) to enable closed-loop online adaptation.
- Auto Opt Mem supports real-time continuous learning across varying workloads, which clearly differentiates it from static or offline ML-based predictors.
- Experiments validate the effectiveness of the proposed method and demonstrate performance improvements in metrics such as latency and cost.

1.4. Paper organization

This paper is organized into several sections: Section 2 reviews related memory configuration methods. Section 3 offers background information. Section 4 presents a comprehensive explanation of the proposed solution. Section 5 assesses and discusses the experimental results. Section 6 presents the discussion. Section 7 presents the conclusions with our findings and outlines future research directions.

2. Related works

This section discusses memory configuration approaches in serverless computing. These approaches are categorized into three main groups: machine learning-based approaches, heuristic-based approaches, and framework-based approaches.

2.1. Machine learning-based approaches

Simon Eismann et al. [17] have presented an approach called "Sizeless" for predicting the optimal resource size for serverless functions in cloud computing, based on monitoring data from a single memory size. It highlights the challenges developers face in selecting resource sizes and shows that the method can achieve an average prediction error of 15.3 %, optimizing memory allocation for 79 % of functions, resulting in a 39.7 % speedup and a 2.6 % cost reduction. Anshul Jindal et al. [18] have presented a tool called FnCapacitor for estimating the Function Capacity (FC) of Function-as-a-Service (FaaS) functions in serverless computing environments. It overcomes performance challenges due to system abstractions and dependencies between functions. Through load testing and modeling, FnCapacitor provides accurate FC predictions using statistical methods and deep learning, demonstrating effectiveness on platforms like AWS Lambda and Google Cloud Functions. Djob Mvondo et al. [19] have presented OFC, an in-memory caching system designed for Functions as a Service (FaaS) platforms to improve performance by reducing latency during data access. It leverages machine learning to predict memory requirements for function invocations, utilizing otherwise wasted memory from over-provisioning and idle sandboxes. OFC demonstrates significant execution time improvements for both single-stage and pipelined functions, enhancing efficiency without requiring changes to existing application code. Myung-Hyun Kim et al. [20] have introduced ShmFaas, a serverless platform designed to improve memory utilization for deep neural network (DNN) inference by sharing models in-memory across containers. They address data duplication and cold start issues, particularly in resource-constrained edge cloud environments. Experimental results show that ShmFaas reduces memory usage by over 29.4 % compared to common systems, while maintaining negligible latency overhead and enhancing throughput. Siddharth Agarwal et al. [21] have presented MemFigLess, an input-aware memory allocation framework for serverless computing functions, designed to resource usage and reduce costs. Using a multi-output Random Forest Regression model, it correlates the input features of the function with memory requirements, leading to accurate memory configuration. The evaluation shows that MemFigLess can significantly decrease resource allocation and save on runtime costs. Finally, Table 1 shows a comparison of machine learning-based approaches from related studies.

2.2. Heuristic-based approaches

Goor-Safarian et al. [22], have present SLAM, a tool for optimizing memory settings for serverless applications consisting of multiple Function-as-a-Service (FaaS) functions. It addresses the issues in balancing cost and performance while meeting Service Level Objectives (SLOs). By utilizing distributed tracing, SLAM estimates execution times under various memory settings and identifies optimal configurations. Robert Cordingly et al. [23] presented a method called CPU Time

Table 1
Comparison of machine learning-based approaches.

Ref	Metric	Method	Advantage	Disadvantage	Tools
[17]	• Execution time • Resource consumption • Performance overhead	• Multi-target regression model • Monitoring data • Node.js	• Reduces execution time • Decrease cost • Without requiring performance test	• Limited to specific cloud providers	• AWS Lambda
[18]	• Function Capacity (FC)	• Statistical and deep learning approaches	• Accurate FC predictions	• Limited to specific FaaS platforms	• Python • FnCapacitor • Google Cloud Functions (GCF) • AWS Lambda
[19]	• Function Capacity (FC)	• OFC tool uses machine learning • Utilizes idle memory	• Reduction in execution time Cost-effective • Transparent	• Overhead from cache management	• Python • Java • OFC • AWS
[20]	• Efficiency of memory usage	• OFC system shares DNN models	• Reduces memory usage • Minimizes cold start delays • Minimal code changes	• Complexity	• Python • ShmFaas • Kubernetes
[21]	• Memory utilization	• Uses input-aware Random Forest Regression	• Reduce memory allocation • Reduce costs	• Limited to specific platforms • Overhead from monitoring	• AWS Lambda • Python • AWS CloudWatch

Accounting Memory Selection (CPU-TAMS) to optimize memory configurations for serverless Function-as-a-Service (FaaS) platforms. CPU-TAMS uses CPU time accounting and regression modeling methods to provide recommendations that reduce execution time and costs. Tetiana Zubko et al. [24] presented MAFF (Memory Allocation Framework for FaaS functions), which is a framework to optimize memory allocation for serverless functions automatically. MAFF adapts memory settings based on function requirements and employs various algorithms to minimize costs and execution duration. The framework was tested on AWS Lambda, demonstrating improved performance compared to existing memory optimization tools. Josef Spillner [25] discussed resource management for serverless functions, focusing on memory tracking, profiling, and automatic tuning. The author outlines the issues that developers face in determining memory allocation due to coarse-grained configurations from cloud providers. Also proposes tools to measure memory consumption and dynamically adjust allocations to reduce waste and costs, and improve performance in Function-as-a-Service (FaaS) environments.

Zengpeng Li et al. [26] have presented algorithms for optimizing memory configuration in serverless workflow applications, specifically a heuristic urgency-based algorithm (UWC) and a meta-heuristic hybrid algorithm (BPSO). These algorithms aim to balance execution time and cost for serverless applications, to solve the challenges posed by memory allocation and performance modeling. Andrea Sabioni et al. [27] have introduced a shared memory approach for function chaining on serverless platforms and proposed a container-based architecture that increases the efficiency of function composition on the same host. By using a message-oriented middleware that operates over shared memory, this approach reduces response latency and improves resource utilization. The results show performance improvements in request completion rates and reduced latency during function execution. Aakanksha Saha et al. [28] have presented EMARS, an efficient resource management system designed for serverless cloud computing, focusing on optimizing memory allocation for containers. Built on the OpenLambda platform, EMARS uses predictive models based on application workloads to adjust memory limits dynamically, enhancing resource utilization and reducing latency. Experiments demonstrate that tailored memory settings improve performance in serverless functions. Amit Samanta, et al. [29] have discussed the issues and opportunities of integrating persistent memory (PM) into serverless computing. It shows how PM's unique characteristics, such as its direct load/store access, can increase performance but also lead to bottlenecks when multiple threads concurrently write to it. They propose a PM-aware scheduling system for serverless workloads that optimizes job completion time by managing concurrent access and improving efficiency while ensuring fairness among applications. Meenakshi Sethunath et al. [30] have proposed a joint function warm-up and request routing scheme for serverless computing that optimally utilizes both edge and cloud resources. It addresses like high latency and cold-start delays by maximizing the hit ratio of requests. It reduced latency, by considering memory and budget constraints. Anisha Kumari, et al. [31] have proposed a performance model for optimizing resource allocation in serverless applications, addressing like cost estimation and performance evaluation. It introduces a greedy optimization algorithm to improve end-to-end response time while considering budget constraints. They utilize serverless applications on AWS to analyze the trade-offs between performance and cost, demonstrating the model's effectiveness in optimal resource configurations. Finally, Table 2 shows a comparison of heuristic-based approaches from related studies.

2.3. Framework-based approaches

Anjo Vahldiek-Oberwagner et al. [32] have proposed a Memory-Safe Software and Hardware Architecture (MeSHwA) to enhance serverless computing and microservices by leveraging memory-safe languages like Rust and WebAssembly. It aims to reduce infrastructure overheads associated with cloud architectures while improving performance and security through a unified runtime environment that isolates services effectively. Divyanshu Saxena, et al. [33] have presented Medes, a serverless computing framework that improves performance and resource efficiency by introducing a deduplicated sandbox state. This state reduces memory usage by removing redundant memory chunks across sandboxes, allowing for faster function startups and improved management of warm and cold states. Experiments show that Medes under memory pressure can reduce end-to-end latency and cold start. Ji Li, et al. [34] designed TETRIS, a memory-efficient serverless platform for deep learning inference. It addresses the memory overconsumption in serverless systems by implementing tensor sharing and runtime optimization. It reduces memory usage while increasing function density. TETRIS automates memory sharing and instance scheduling,

Table 2
Comparison of heuristic-based approaches.

Ref	Metric	Method	Advantage	Disadvantage	Tools
[22]	• Memory configuration effectiveness based on Service Level Objectives (SLOs)	• Distributed tracing • Max-heap-based optimization algorithm	• Balances cost and performance	• Limited to specific platforms	• Python • AWS • SLAM
[23]	• Efficiency	• CPU time accounting • Regression modeling	• Reduces runtime • Reduces cost	• Limited to specific FaaS platforms	• Python • AWS • GCF
[24]	• Effectiveness for FaaS functions	• Algorithms Linear, Binary, Gradient Descent for self-adaptive memory optimization	• Lower cost • Faster execution	• Require specific function profiling	• Python • AWS
[25]	• Efficiency • Cost optimization	• Utilizes memory tracing, profiling, and autotuning tools	• Reduce cost • Improve performance • Autoscaling resource	• Requires extensive profiling data	• AWS • Functracer • Autotuner • costcalculator
[26]	• Efficiency	• Heuristic (UWC) and meta-heuristic (BPSO) algorithms	• Time-cost tradeoff • Optimal workflow	• Complexity • Computational overhead	• AWS • Python • UWC • BPSO
[27]	• Response latency • Resource usage	• Shared-memory, using a message-oriented middleware	• Improve request completion • Improve rates response time • Optimized resource usage	• Limited to co-located functions • Complexity in managing shared memory	• Message-oriented middleware
[28]	• Memory allocation efficiency • Latency	• Workload-based model • Memory-based model	• Optimizes resource usage • Reduce latency	• Complexity	• Open Lambda • Python
[29]	• Throughput • Job completion time (JCT)	• Using performance modeling and admission control for concurrent access	• Improve throughput • Reduce latency	• Complexity	• Open FaaS • Intel Optane DCPMM
[30]	• Latency reduction • Request hit ratio	• Joint function warm-up • Routing for edge and cloud collaboration	• Reduces cold-start latency • Optimize performance	• Complexity • Dependent on accurate profiling	• AWS • Azure Function
[31]	• End to end response time • Cost	• Greedy optimization algorithm	• Reduce latency • Reduce cost • handles cold start delay	• Complexity • Dependent on accurate profiling	• Amazon Web Service • AWS

providing efficient resource utilization without compromising performance.

Dmitrii Ustiugov, et al. [35] have discussed cold-start latency in serverless computing and introduces vHive, an open-source framework for experimentation. It shows the inefficiencies of snapshot-based function invocations, which can result in high execution times due to frequent page faults. They propose REAP, that prefetches memory pages to reduce cold-start delays by 3.7 times, improving performance of serverless functions. Ao Wang, et al. [36] have presented INFINICACHE, an innovative in-memory object caching system leveraging ephemeral serverless functions to provide a cost-effective solution for large object caching in web applications. It shows the system's ability to achieve cost savings while maintaining high data availability and performance through techniques like erasure coding and intelligent resource management. Anurag Khandelwal et al. [37] have presented Jiffy, an elastic far-memory system for serverless analytics. It overcomes the limitations of existing memory allocation systems by enabling fine-grained, block-level resource allocation, allowing jobs to meet their real-time memory needs. Jiffy minimizes performance degradation and resource underutilization by dynamically managing memory for individual tasks. Orestis Lagkas Nikolos, et al. [38] have introduced HotMem, a mechanism designed to enhance memory reclamation in serverless computing environments, for Function-as-a-Service (FaaS) models using microVMs. It addresses the issues of memory elasticity, during scaling down, by segregating memory allocations for individual function instances, thereby enabling rapid and efficient reclamation without the overhead of page migrations. HotMem improves memory management performance, maintaining low latency for function execution. Finally, Table 3 shows a comparison of framework-based approaches from related studies.

3. Background

In this section, we explain the concepts of serverless computing and then provide explanations about memory configuration in serverless computing.

3.1. Serverless computing

Serverless computing enables developers to concentrate on coding without the necessity of managing or provisioning servers, which is why it's termed "serverless." Serverless computing provides an efficient and scalable solution for running programs. Its ease of management and lightweight features have made it popular as an implementation model

Table 3
Comparison of framework-based approaches.

Ref	Metric	Method	Advantage	Disadvantage	Tools
[32]	• Performance and resource efficiency	• MeSHwA, a memory-safe software and hardware architecture	• Increase security and performance • Resource sharing	• Complexity	• Python • Rust • Wasm
[33]	• End-to-end latency • Memory usage	• Medes, a framework utilizing memory deduplication to create a new deduplicated sandbox	• Reduce cold start • Increase flexibility • Optimal memory	• Complexity • Overhead from deduplication	• AWS Lambda • Python • Open Whisk • Open FaaS • Function Bench • CRIU
[34]	• Function startup latency	• Runtime sharing to reduce memory consumption	• Memory savings • Reduce cold start	• Complexity • Tensor management overhead	• Open FaaS • TensorFlow
[35]	• Cold-start latency • Overhead the first invocation • Memory efficiency	• REAP, a mechanism that records and prefetches a function's working set • Python s	• Reduce cold-start latency		• Kubernete
[36]	• Latency of function invocation	• INFINICACHE, caching through tensor sharing and erasure coding	• Cost saving • Data availability	• Limited to large object caching • Relying on the limitations of serverless architecture	• AWS Lambda
[37]	• Execution time • Memory utilization	• Jiffy, an elastic far-memory	• Improvement execution time • Increase resource utilization	• Complexity • Relies on specific serverless architecture	• Amazon EC2 • Python • C++ • Java
[38]	• Memory speed • Tail latency	• HotMem, a memory management framework with rapid reclamation of hotplugged memory	• Faster reclaim memory	• Challenges in memory management	• Open whisk • Azure

in cloud computing [39]. Serverless computing is an abstraction of cloud computing infrastructure. Serverless computing is a cloud computing model in which a cloud provider or a third-party vendor manages the company's servers. The company does not need to purchase, install, host, and manage the servers. Instead, the cloud administrator provides all of these services.

Serverless computing, also known as Function-as-a-Service (FaaS), ensures that the code exposed to the developer consists of simple, event-driven functions. As a result, developers can focus more on writing code and delivering innovative solutions, without the hassle of creating test environments and provisioning and managing servers for web-based applications [40]. FaaS and the term "serverless" can be used interchangeably, with serverless computing being FaaS. This is because the FaaS platform automatically configures and maintains the context of the functions and connects them to cloud services without the need for developers to provide a server [41,42].

Features of Serverless Computing:

- Pay-per-use: In serverless computing, users pay only for the time their code uses dedicated CPU and storage. Pay-per-use pricing models reduce costs. But in cloud services, users pay for over-provisioning of resources like storage and CPU time, which often sit idle [39,43].
- Speed: Teams can move from idea to market more quickly because they can focus entirely on coding, testing, and iterating without the operational costs and server management. There is no need to update fundamental infrastructure like operating systems or other software patches, allowing teams to concentrate on building the best possible features without worrying about the underlying infrastructure and resources.
- Scalability and elasticity: The cloud vendor is responsible for automatically scaling up capabilities and technologies to meet customer demands. Serverless functions should automatically scale down when there are fewer concurrent users. This elasticity transforms serverless computing models into a pay-as-value billing model [39].
- Efficiency and performance: Developers do not need to perform complex tasks like multi-threading, HTTP requests, etc. FaaS forces developers to focus on building the application rather than configuring it.
- Programming languages: Serverless computing supports many programming languages, including JavaScript, Java, Python, Go, C#, and Swift [44]. This versatility allows developers to choose the language that best suits their project needs and expertise, increasing productivity and enabling rapid application development.

Challenges of Serverless Computing:

- Vendor lock-in: Vendors typically use proprietary technologies to enable their serverless services. This can create problems for users who want to migrate their workloads to another platform. When migrating to another provider, changes to the code and application architecture are inevitable [45].
- Cold start latency: Serverless services can experience a latency known as "cold start." When the service is first started, it takes some time for the service to respond. The reason for this is the initial configuration of the cloud service provider and resource allocation and initialization of the infrastructure. This initial delay can be a concern in systems that respond to many requests per second. Methods and techniques are important for mitigating the cold start problem [46–48].
- Debugging complexity: Debugging serverless functions is difficult due to their transient nature. The reason for this problem is that Serverless functions typically do not maintain the state of previous calls (stateless). Also, serverless functions are stateless by design, which can complicate application state management. Also, reports on serverless function calls should be sent to the developer. These

reports should include detailed stack traces so that developers can identify the cause of an error. Stack tracing is currently not available for serverless computing, meaning developers cannot easily identify the cause of an error [49–51].

- Architectural complexity: A serverless application may consist of multiple functions. The more functions there are, the more complex the architecture becomes. This is because each function must be properly linked to other functions and services. Also, managing this large number of functions and services can be difficult [52].
- Long-running: Serverless computing with long-running functions executes the function in a limited and short execution time, while some tasks may require a long execution time. These functions do not support long-running execution because these functions are stateless, meaning that if the function is stopped, it cannot be resumed [53].

3.2. Memory configuration

Memory configuration in serverless computing is an important aspect that influences application performance, resource efficiency, and cost management. Understanding how to optimally allocate memory for serverless functions is important for developers who want to maximize the benefits of serverless computing [40]. In serverless environments, developers deploy small units of code, known as functions, which are executed in response to specific events. Each function operates in a stateless manner and is allocated a certain amount of memory at runtime. The memory configuration affects several factors:

- **Performance:** The memory allocated for a function will determine execution time. The more the memory, the faster, as it allows the CPU to perform better and cold start latency to reduce. But too less memory can make it very slow and cause downtime.
- **Cost efficiency:** The pricing of serverless computing is pay-per-use; the costs are determined by the amount of resources utilized in execution. Memory configuration can help prevent over-allocation of memory, which leads to high costs. However, a lack of memory will lead to performance issues that need to be balanced [40].
- **Scalability:** Memory configuration is performed in such a way that serverless applications will be able to scale seamlessly according to variable workloads. Since the demand fluctuates, dynamically allocating memory helps achieve good performance without incurring excessive costs.

Memory configuration optimization and automated, data-driven approaches to memory management in serverless computing increase performance and scalability, and help save costs. These methods can allow developers to minimize the challenges of manual memory configuration.

3.3. Deep reinforcement learning

Neural networks were combined with reinforcement learning for the first time in 1991 when Gerald Tsauro used reinforcement learning to train a neural network to play backgammon at the master level [54]. Deep reinforcement learning (DRL) is a combination of reinforcement learning and deep neural networks. In reinforcement learning, an agent, while interacting with its environment learns a policy incrementally that enables it to maximize long-term rewards. This approach enables the learning of much more complicated policies that are suitable for high-dimensional problems by combining with deep neural networks. Fig. 1 shows the elements in a reinforcement learning model.

Deep reinforcement learning is used in various fields such as computer games, robotics, resource management in distributed systems, and performance optimization of cloud computing systems. Deep learning techniques can be used in memory configuration in serverless computing. And predicted the amount of memory required to run a serverless application.

Serverless computing is a computing model in which the cloud service provider manages the infrastructure and server resources. A developer just needs to write his code and upload it onto the serverless platform. The advantages of serverless computing include many things: automatic scalability, cost per resource, ease of management. Deep learning can be used to improve memory configuration in serverless computing. Deep learning can automatically identify memory usage patterns in applications and allocate the required memory based on that. Benefits of using deep learning for automatic memory configuration in serverless computing applications:

- **Automation:** With deep learning, the technology can easily identify patterns concerning memory usage in an application and automate the memory needed for allocation. Consequently, this will reduce time spent on memory configuration.
- **Optimization:** Deep learning can learn how to optimize memory usage by managing how memory is allocated. It helps to decrease computing costs and enhances the performance of applications.
- **Flexibility:** Deep learning can learn from changes in memory usage patterns. It can help in the improvement of an application over time.

Deep learning to automate the configuration of memory in serverless computing faces some challenges. First, there is a need for training data. Deep learning requires enough training data in order to identify patterns in memory usage. Another challenge is the complexity of deep learning, which also requires a lot of time to learn. However, using deep learning for automating the configuration of memory in serverless computing offers numerous benefits. It will improve memory configuration techniques and reduce computational costs.

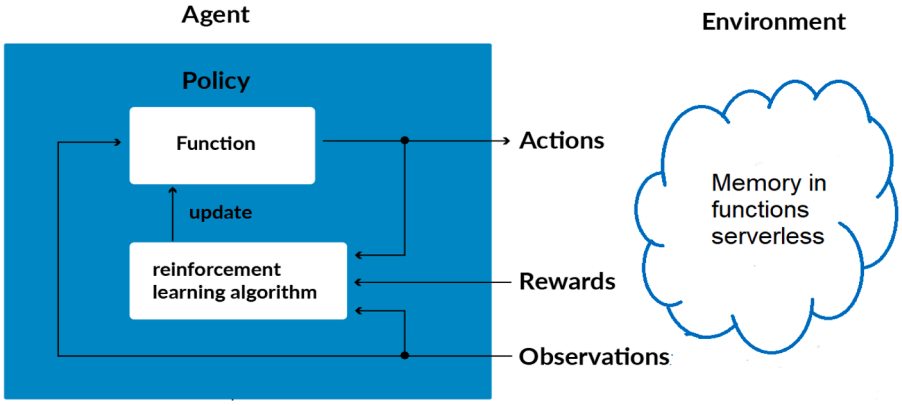


Fig. 1. Elements in a reinforcement learning model.

3.4. Autonomic computing

The MAPE-K model provides a framework for the management of autonomous and self-adaptive systems [55–57]. The model comprises five major components: Monitoring, Analysis, Planning, Execution, and Knowledge Management, illustrated in Fig. 2.

Monitoring is where data is gathered on an ongoing basis to measure the system's current state and functionality relative to predetermined goals. The subsequent Analysis stage examines this data to bring to light any differences between the present condition and desired outcomes, giving the information necessary for adjustment. After problem detection, the Planning stage creates schemes to amend these disparities, specifying how the system needs to modify its behavior. The Execution stage then enacts these strategies, modifying the system's behavior to attain the desired results. Knowledge Management serves as a repository of knowledge for the important information shared among the various stages. Overall, the MAPE-K model offers a feedback loop that allows autonomous systems to adapt to new situations to ensure optimum performance by constantly monitoring and adjusting. This model is required to build resilient and efficient autonomous systems. In addition, autonomous control systems greatly improve the efficiency and reliability of industrial processes by minimizing human error and ensuring consistent performance. Autonomous control systems are closed-loop controls that are preprogrammed to operate autonomously without the intervention of an operator, and they ensure predictable results in complex industrial settings. Therefore, the integration of automated control systems and sophisticated monitoring techniques makes operations simpler and more reliable, and industrial processes become a necessity in contemporary manufacturing plants [58].

4. Proposed approach

In this section, we will explain our proposed approach in more detail. First, we will introduce the framework that utilizes machine learning for memory configuration in serverless computing, followed by the presentation of a formula and algorithm. Finally, we describe the autonomous memory configuration using deep learning to predict memory setting for serverless computing.

4.1. Proposed framework

In the proposed solution, Auto Opt Mem introduces an autonomous memory configuration approach based on learning for serverless computing. The goal of Auto Opt Mem is to address the challenge of efficiently distributing serverless functions (SF) in a serverless environment while considering a number of various real-world parameters. One of the key parameters to consider within Auto Opt Mem is memory configuration, a term that shows how much of the available memory resources are to be allocated for the serverless function. Memory configuration has a direct impact on the performance and resource utilization of these functions. Auto Opt Mem utilizes Deep Reinforcement Learning (DRL), a branch of artificial intelligence, to learn an optimal memory configuration policy. DRL enables the system to learn from experience and make decisions based on a reward mechanism [15, 16]. In this regard, Auto Opt Mem learns the memory resource allocation to SFs for maximum performance while minimizing waste of resources.

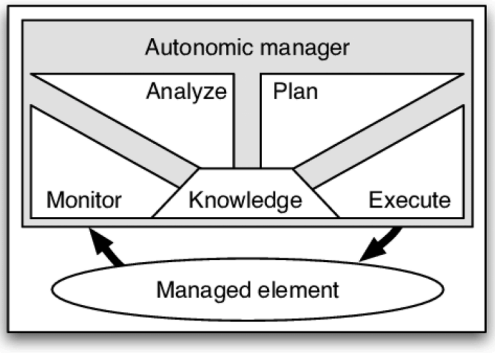


Fig. 2. Autonomic computing (MAPE-K loop) [57].

By employing DRL, Auto Opt Mem takes into account resource constraints in the serverless environment, resource requirements of functions, energy consumption, latency, and deployment costs. The system learns the optimal memory configuration policy through a training process that involves interacting with the environment and receiving feedback in the form of rewards. The automatic memory configuration learning approach in Auto Opt Mem enables the system to dynamically adapt to changing conditions and optimize memory resource allocation based on the specific needs of each SF. This approach contributes to efficient resource utilization and improved performance in serverless computing environments. This work proposes a framework called Auto Opt Mem for automatic optimal memory that uses a deep learning agent to automatically optimize the memory allocated to each serverless function with respect to computational resources. The primary goal of Auto Opt Mem, utilizing Deep Reinforcement Learning (DRL), is to optimize memory allocation for serverless functions. In other words, the main goal is to dynamically adjust the memory configurations to become highly performant, cost-effective, and efficient regarding resource utilization. Indeed, it seeks to optimize memory for serverless functions to minimize costs and reduce latency while maintaining or improving Quality of Service (QoS).

4.1.1. Key components

This section describes the elements of the Auto Opt Mem framework that are essential to the automating memory configuration process.

Environment

The environment is the platform of serverless, such as AWS Lambda or Google Cloud Functions where a set of functions are deployed and executed within. It reflects resource usage information, performance metrics, and all other relevant status information.

Agent

The agent does the job of making decisions about the memory size to be assigned to each function. It possesses one DRL model that tries to learn through interactions with the environment.

State

The state S_t at time t includes:

- Current memory allocation for each function.
- Performance metrics (e.g., execution time, latency).
- Number of requests received.
- Cost metrics.
- Specific performance requirements.

Action

The action A_t at time t is the choice of memory size from some predefined set of configurations for that particular function.

Reward

The reward R_t at time t is the signal that informs feedback and therefore controls learning. It needs to be defined with the aim of:

- Encourage efficient memory usage.
- Improve performance metrics.
- Maintain or increase Quality of Service (QoS).
- Minimize operational costs.

Policy

Policy is a strategy by which the agent makes decisions to select actions according to the current state. The DRL model implements the policy.

Fig. 3 shows the iterative loop diagram in deep reinforcement

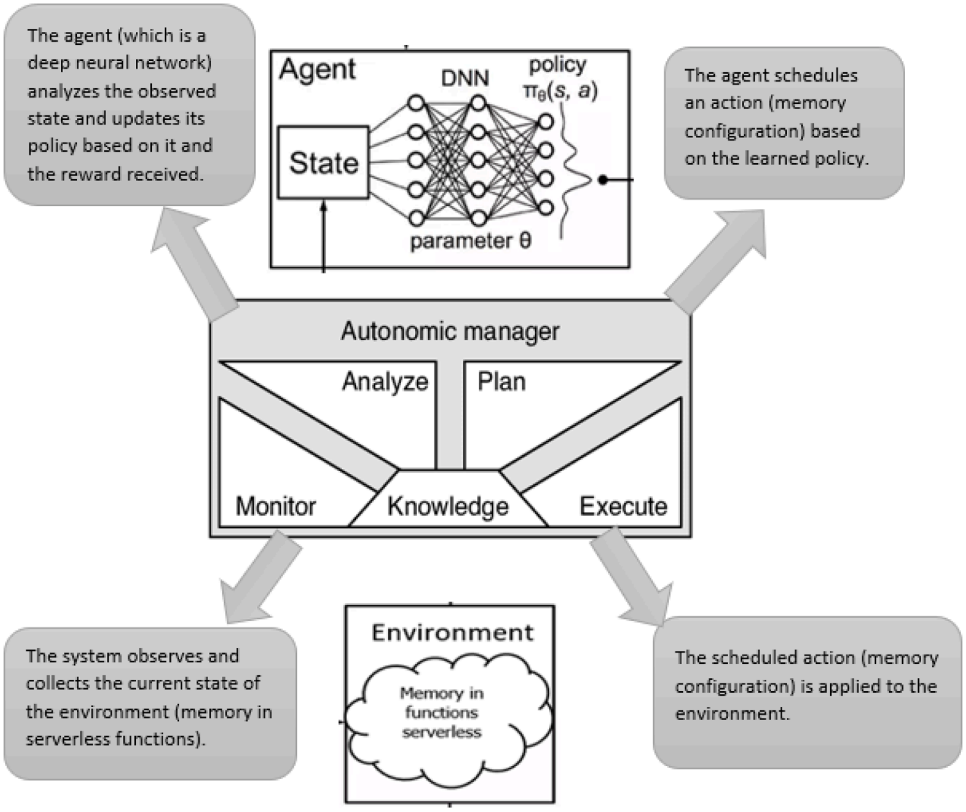


Fig. 3. Iterative loop diagram in deep reinforcement learning algorithms.

learning algorithms.

4.2. Problem formulation

This section presents a mathematical model for the memory configuration problem in serverless computing. In the following, we describe the used symbol in more details.

4.2.1. Notation

- F: Set of serverless functions
- Mi: Memory allocated to function F in fi
- Li (Mi): Latency of function fi with memory Mi
- Ci (Mi): Cost of executing function fi with memory Mi
- Ui (Mi): Memory utilization for function fi with memory Mi
- Qi (Mi): Quality of Service metric for function fi with memory Mi
- Rt: Reward at time t

The list of mathematical symbols is summarized in Table 4.

The framework and automated approach for learning-based memory configuration in serverless computing is as follows:

State space (S)

The state S_t at time t includes:

- Current memory allocation Mi,t for each function fi.
- Performance metrics such as latency Li,t , cost Ci,t, and Quality of Service Qi,t.
- The number of requests received.

Action space (A)

The action A_t at time t involves selecting the memory size $M_{i,t+1}$ for each function f_i.

Reward function (R)

The reward function is an essential component of reinforcement learning, guiding the agent toward desired behaviors by associating rewards or penalties based on the outcome of actions. The reward function R_t is designed to:

- Reduce latency and cost.

Table 4
List of Mathematical Symbols.

Definition	Notation
A set of serverless functions	F_i
Memory allocated to function F ϵ fi	M_i
Delay of function f_i with memory M_i	L_i (M_i)
Cost of executing function f_i with memory M_i	C_i (M_i)
Memory usage for function f_i with memory M_i	U_i (M_i)
Quality of service criterion for function f_i with memory M_i	Q_i (M_i)
Reward at time t	R_t
State space	S
State space at time t	S_t
Space of action	A
Action A at time t	A_t
Current memory allocation for each function f_i	$M_{i,t}$
Latency	$L_{i,t}$
Cost	$C_{i,t}$
Quality of service	$Q_{i,t}$
Weighting factor for delay	α
Weighting factor for cost	β
Weighting factor for usage (utilization)	γ
Weighting factor for service quality	δ
Minimum memory size that can be allocated to a function	$M_{i,min}$
The maximum amount of memory that can be allocated to a function	$M_{i,max}$
Minimum acceptable QoS for the function F_i	Q_i^{min}
Expected value	E
Policy	π
Value function	V^π
Policy network parameters	θ
Value network parameters	ϕ
Action-value function	Q^π (s,a)

- Decrease memory utilization and increase QoS.

Which is expressed by Eq. (1):

$$R_t = -\left(\sum_{i\in f}\left(\alpha L_i(M_{i,t}) + \beta C_i(M_{i,t}) - \gamma U(M_{i,t})\right)\right) + \delta Q_i(M_{i,t}) \quad (1)$$

where (α, β, γ, δ) are the weighting factors for latency, cost, utilization, and QoS, respectively. That weights can adjust the relative importance of each component in the reward function. For example, if reducing latency is more important than cost, α can be increased relative to β.

Lower latency is desirable; therefore, -αL$_i$ (M$_{i,t}$) penalizes higher latencies.

Lower costs are also preferable; thus, -βC$_i$ (M$_{i,t}$) penalizes higher expenses.

Memory utilization is desired to be efficient. Over-allocation of memory leads to resource waste, while under-allocation can degrade performance. γU$_i$ (M$_{i,t}$) rewards the agent for optimal memory usage.

Higher QoS is preferred; therefore, +δQ$_i$ (M$_{i,t}$) rewards better service quality.

To normalize the reward function for optimizing memory configuration in serverless computing, all components (cost, latency, utilization, and QoS) must be on a common scale. Because they have different units (for example, costs in dollars, latency in milliseconds). Below is an approach to achieve this normalization. We will use min-max normalization for each component. The min-max normalization formula is as follows, and is defined by Eq. (2):

$$x' = \frac{x - x_{\min}}{x_{\max} - x_{\min}} \quad (2)$$

Where:

- X is the original value,
- Xmin is the minimum value for that metric,
- Xmax is the maximum value for that metric.

After normalization, the metrics can be combined into the reward function without unit conflict, and the final reward formula becomes, which is expressed by Eq. (3):

$$R_t = -\left(\sum_{i\in f}\left(\alpha\frac{L_i(M_{i,t}) - L_{\min}}{L_{\max} - L_{\min}} + \beta\frac{C_i(M_{i,t}) - C_{\min}}{C_{\max} - C_{\min}} - \gamma\frac{U_i(M_{i,t}) - U_{\min}}{U_{\max} - U_{\min}}\right)\right) + \delta\frac{Q_i(M_{i,t}) - Q_{\min}}{Q_{\max} - Q_{\min}} \quad (3)$$

Normalization, in effect, normalizes each component into the range of [0, 1], thus making them comparable even if they are from different units. Normalizing the reward function components makes them comparable and hence they can be combined meaningfully during the optimization process. This method increases the robustness of the model and enhances convergence in reinforcement learning algorithms.

4.2.2. Optimization problem

First, it is necessary to mathematically formulate the problem in terms of optimization and reinforcement learning for the Auto Opt Mem algorithm and its solution. The problem of optimization can be done as follows, and is expressed by Eq. (4):

$$\min_M \sum_{t=0}^{T} R_t \quad (4)$$

With the following constraints, and is expressed by Eq. (5):

$$M_{i,\min} \le M_{i,t} \le M_{i,\max} \text{ for all } i \in F \quad (5)$$

Quality of Service constraints, and is expressed by Eq. (6):

$$Q_i(M_{i,t}) \ge Q_i^{\min} \text{ for all } i \in F \quad (6)$$

M$_{i,min}$ and M$_{i,max}$ are the minimum and maximum memory sizes that can be allocated to the function F$_i$.

The minimum acceptable QoS for the function F$_i$ is. This ensures that the QoS is not less than a certain threshold.

4.2.3. Reinforcement learning formula

This section explains the mathematical formulas used in the reinforcement learning component of the Auto Opt Mem framework.

Policy (π)

The policy π(a|s) is the probability distribution over actions given the current state. It is the strategy that the decision-maker considers for its next action in response to the current state.

Value function (V$^{\pi}$)

Value function refers to the expected long-term discounted reward, which contrasts with short-term rewards (R) as it focuses on the long term. It represents the expected return in the long-term resulting from the current state *s* under policy π. The value function is an important concept in reinforcement learning and represents the expected return (cumulative reward) starting from state *s* and following policy π, and is defined by Eq. (7):

$$V^{\pi}(S) = E\left[\sum_{t=0}^{\infty}\gamma^t R_t | S_0 = s, \pi\right] \quad (7)$$

Where γ is the discount factor.

- **Expected value (E):** Due to the random nature of the environment, it is the average or expected return in all possible future states.
- **Sum of reward:** $\sum_{t=0}^{\infty}\gamma^t R_t$ It represents the Sum of rewards over time. Rewards are discounted by a factor of γ*t* to prioritize immediate rewards over distant future rewards.
- **Discount factor (γ):** The discount factor γ (where $0 < \gamma < 1$) determines the present value of future rewards. A higher γ makes future rewards more significant, while a lower γ emphasizes immediate rewards.
- **Policy (π):** The policy π is the decision rule that gives the probability of executing action *a* from a state *s*.

Value function is important since it is utilized to measure how good a given state is under a given policy. It is also a basis for policy comparison and for determining optimal actions to maximize the long-term reward.

Bellman equation

The Bellman equation provides a recursive decomposition for the value function, making it a powerful tool for solving reinforcement learning problems. The Bellman equation for the value function is as follows, and is defined by Eq. (8):

$$V^{\pi}(S) = E_{a\sim\pi}[R_t + \gamma V^{\pi}(S+1) | S_t = s, A_t = a] \quad (8)$$

Reasons for using the Bellman equation:

- **Recursive nature:** The Bellman equation breaks down the value of a state into the immediate reward *Rt* plus the discounted value of the next state γVπ(St+1). The recursion makes computation efficient and forms the foundation for dynamic programming techniques.
- **Policy evaluation:** By repeatedly updating the value function using the recursive relationship, it aids in evaluating the expected return of a policy π.
- **Optimality principle:** For finding the optimum policy, the Bellman optimality equation expresses the relationship between the value of a state and the values of subsequent states. This is used in algorithms like value iteration and Q-learning for finding the optimal value function.
- **Simplification of complexity:** The use of a recursive approach allows the Bellman equation to simplify the calculation of the value

function for all states, which otherwise would be computationally infeasible in large state spaces.

Policy gradient

The policy gradient method is used to optimize policy π, and is defined by Eq. (9):

$$\nabla_\theta J(\pi_\theta) = E_{s-d^\pi, a-\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s,a)] \tag{9}$$

where θ are the parameters of the policy, and Q^π (s,a) is the action-value function.

The value-action function represents the expected return of taking action a in state s and then following policy π, providing a basis for policy improvement.

The Bellman equation and value function are the basis of studying and quantifying long-term impact from these decisions and guide the policy to optimal function.

4.3. Proposed algorithm

The proposed Auto Opt Mem algorithm uses deep reinforcement learning to learn how to assign serverless functions to compute resources efficiently. Incorporating the MAPE loop, the Auto Opt Mem algorithm continuously monitors, analyzes, plans, and executes actions to optimize memory allocation in an automous and adaptive manner. The process of the algorithm is as follows:

4.3.1. Initialization

This algorithm is the first step in the deep reinforcement learning (DRL) process for memory configuration. First, two main networks are prepared: the Policy Network, which is responsible for choosing the action (e.g., allocating the amount of memory) in each state. This network initially starts with random parameters because it has no knowledge at the beginning and gradually learns to make optimal decisions. The Value Network estimates the long-term value of a state based on the sum of future rewards. This network also starts with random parameters. Then, the initial state of the environment (S_0) is defined, which includes the memory configuration of each function, performance indicators (latency, cost, QoS, and resource utilization), and the rate of incoming requests. This step is the basis of training and provides the basis for the agent's interaction with the environment.

This initialization is very important, as it sets the starting point for training the networks to learn optimal memory allocation for serverless functions, with the aim of minimizing cost and latency and maintaining high QoS. This is shown in Algorithm 1.

4.3.2. MAPE loop

In this section, we describe an autonomous memory configuration includes four phases: monitoring, analysis, planning, and execution,

4.3.2.1. Monitor. In the monitoring phase, the system monitors the current state of the environment at all times, and the system is always aware of the state of the environment. This includes monitoring the memory allocated to each serverless function, obtaining performance data such as latency, cost, utilization, and quality of service (QoS), and monitoring the number of incoming requests for each function. The monitoring phase is important because it records real-time information that indicates system performance and resource utilization. In short:

- Continuously observe the current state of *St*, including memory allocation, performance metrics (latency *Li,t* , cost *Ci,t* , utilization *Ui,t*, QoS *Qi,t*), and incoming request rate.
- Collect data from serverless functions and the environment.

4.3.2.2. Analyze. In the analysis phase, the system uses the performance metrics collected during the monitoring phase, and the system accepts the performance measurements collected during the monitoring phase. This is achieved by examining the current performance of the serverless functions based on the collected data and calculating a reward value that controls the learning. The reward function is usually based on latency, cost, utilization, and QoS, thus allowing for any inefficiencies or issues that need to be addressed in subsequent phases. In short:

- Evaluate performance metrics *Li*(*Mi*), *Qi*(*Mi*), *Ci*(*Mi*), *Ui*(*Mi*).
- Calculate reward *Rt* based on the current state.

Which is expressed by Eq. (10):

$$R_t = -\left(\sum_{i\in f}\left(\alpha\frac{L_i(M_{i,t}) - L_{\min}}{L_{\max} - L_{\min}} + \beta\frac{C_i(M_{i,t}) - C_{\min}}{C_{\max} - C_{\min}} - \gamma\frac{U_i(M_{i,t}) - U_{\min}}{U_{\max} - U_{\min}}\right)\right) + \delta\frac{Q_i(M_{i,t}) - Q_{\min}}{Q_{\max} - Q_{\min}} \tag{10}$$

4.3.2.3. Plan. In the planning phase, the system decides the next actions based on the analysis observations. In this stage, best memory allocation decisions are selected based on the policies learned from the deep reinforcement learning model and updates in policy network parameters for enhanced future decision-making. This planning assists the system in adjusting its memory allocation policies effectively based on the current situation and performance analysis. It can be said that:

- Use policy network πθ to determine the optimal action *At* (memory allocation for the next time step).
- Update policy and value networks using reinforcement learning techniques. Typically involves:
  - Calculating the policy gradient using the policy gradient method.
  - Updating the policy network parameters θ.
  - Using the Bellman equation to update the network parameters ϕ.

4.3.2.4. Execution. The execution phase is responsible for carrying out the actions that have been planned. It allocates memory to every function according to the decisions of the planning phase and changes to the next state, ready to trigger the MAPE cycle once more. Its role is to realize the change from planning and analysis, optimize resource utilization, and improve the system as a whole. In summary:

- Apply the selected action to adjust the memory allocation for each function.
- Go to the next state *St*+1 and repeat the loop.

The MAPE loop enables a dynamic and automated way of memory management in serverless computing systems, where the system can learn and change with new situations on a continuous basis, eventually resulting in enhanced performance and resource efficiency, as shown in Algorithm 2.

4.3.3. Training loop

The training loop is an important part of the Auto Opt Mem algorithm, which uses deep reinforcement learning to improve how memory

Algorithm 1
Pseudo code for initialization phase ( ).

```
Input: Set of serverless functions F
Output: Initialized policy network π_θ, value network V_ϕ, and initial state S_0
Initialize policy network π_θ with random parameters θ
Initialize value network V_ϕ with random parameters ϕ
Define initial system state S_0 including:
  - Memory allocation for each f_i ∈ F
  - Performance metrics: Latency Li(Mi), Cost Ci(Mi), Utilization Ui(Mi), QoS Qi(Mi)
  - Incoming request rate
Return (π_θ, V_ϕ, S_0)
```

is allocated for serverless functions. The process starts by resetting the environment to its initial state, called S_0. For each time step t, the algorithm goes through the MAPE loop. It begins by checking the current state S_t, gathering data about memory usage, performance metrics, and other relevant information. After collecting this data, it analyzes it to see how well the system is performing and calculates a reward that helps guide future learning. Next, the algorithm plans the next action by choosing the best way to allocate memory using its policy network. This decision is based on the insights gained from the analysis. Once it decides on an action, the system carries it out, adjusting the memory for each function as needed. Finally, the algorithm moves to the next state S_{t+1} and prepares to start the loop again. This ongoing process allows the system to learn and adapt continuously, refining its memory allocation strategies based on real-time feedback. Each cycle helps improve the overall performance and efficiency of memory management in the serverless environment. In summary, it can be said:

1. Environment reset: Reset the environment to its initial state S_0.
2. MAPE execution: For each time step *t*:
   - Monitor: Observe the current state *St*.
   - Analyze: Evaluate performance and calculate reward.
   - Plan: Select an action using the policy network.
   - Execute: Execute the action and move to the next state *St*+1.

Algorithm 3 shows the execution phase of the MAPE loop during the training process.

5. Performance evaluation

In this section, we present the performance evaluation of the novel automatic deep learning-based approach (Auto Opt Mem) for memory setting in serverless computing. We describe the experimental setup, performance metrics, and the result of the experiments.

5.1. Experimental setup

We carried out experimental analysis in this study on a Windows 11–64-bit computer with an Intel Core i7 processor. The evaluation uses a serverless simulation environment, where memory sizes between 128 MB and 2048 MB are modeled to study their impact on latency, cost, and quality of service (QoS). A virtual CPU (vCPU) model with burst behavior and dynamic workload scaling is incorporated into the simulation. Python was utilized to implement deep reinforcement learning algorithms in machine learning. Proximal Policy Optimization algorithm was utilized to train the deep reinforcement learning agent as it is known to be stable and efficient in policy optimization procedures. Also, the reason for choosing the Proximal Policy Optimization (PPO) algorithm is that it is widely suitable for continuous control and policy optimization problems in environments with large dynamic states such as Serverless environments. And it has higher stability in training than algorithms such as REINFORCE or Vanilla Policy Gradient. It has the ability to control the trade-off between exploration and exploitation in dynamic environments where the workload changes randomly. It is suitable and scalable in environments with high-dimensional state spaces where multiple parameters such as latency, cost, utilization, and quality of service (QoS) need to be optimized simultaneously. The workloads used in our evaluation consisted of four modeled serverless scenarios: ML inference, API aggregation, data preprocessing (ETL), and video processing. These workloads are designed to emulate the performance behavior of serverless applications while being executed in a fully simulated environment. Furthermore, all experiments are implemented in Python language, and the source code of simulation can be downloaded at the GitHub repository.[1]

[1] https://github.com/zahrashj-rad/Auto-Opt-Mem

Although AWS Lambda and CloudWatch were used conceptually to structure the resource and metric model, all experiments were executed in a simulated environment. As shown in Table 5.

The experiments involved tuning a set of hyperparameters. The parameters were:

Learning rate: Different learning rates like 0.01, 0.001 and 0.0001 were attempted during training.

Discount Factor (γ): Discounting coefficient of 0.99 was chosen to give higher importance to long-term rewards for the reinforcement learning.

Batch size: A batch size of 32 was utilized in training the deep learning models, a trade-off between training time and model accuracy.

Number of episodes: Training was carried out over 100 episodes, where the agent learns through experience interacting with the environment and tunes its memory allocation policies.

Both the policy and value networks are implemented as multilayer perceptrons (MLPs). The input layer receives the state vector, which includes memory allocation, latency, cost, utilization, QoS, and request rate. Each network has two fully connected hidden layers with 128 and 64 neurons, respectively, and ReLU activation functions. The policy network ends with a softmax output layer producing a probability distribution over possible memory allocations, while the value network has a single linear output estimating the state value. The generated experience data was split into 70 % for training and 30 % for evaluation, which is standard in DRL-based optimization studies.

5.2. Performance metrics

To evaluate the effectiveness of the proposed approach, we utilize several performance metrics, including:

Latency: The time taken for a function to execute, measured in milliseconds. Lower latency indicates better performance. Which is expressed by Eq. (11):

$$L = \frac{T_{end} - T_{start}}{N} \tag{11}$$

Where and are the T_{start} and T_{end} times of execution, and N is the number of function invocations.

Cost: The total cost incurred during function execution, measured in US dollars. Our goal is to minimize this cost. Which is expressed by Eq. (12):

$$C = \sum_{i=1}^{N} (M_i \times P_{mem} + T_i \times P_{exec}) \tag{12}$$

Where M_i is the allocated memory, P_{mem} is the price per MB, T_i is the execution time, and P_{exec} is the execution price per second.

Quality of Service (QoS): A composite score reflecting the reliability and user satisfaction of the service. A composite metric based on latency and availability, defined as, and is expressed by Eq. (13):

$$QoS = \frac{1}{L + \delta(1 - A)} \tag{13}$$

Where A is the availability factor (percentage of successful execution) and helps the quality of service to take into account the impact of availability on the overall system performance. δ is a weighting parameter.

Utilization: The efficiency of memory usage during function execution, aiming for optimal allocation without wastage. Which is expressed by Eq. (14):

$$U = \frac{M_{used}}{M_{allocated}} \times 100 \tag{14}$$

Where M_{used} is the actual memory usage and $M_{allocated}$ is the assigned memory. A higher value indicates optimal management of memory resources.

Algorithm 2
Pseudo code for MAPE loop phase ( ).

1: Input: Current state St, networks (π_θ, V_ϕ)
2: Output: Updated state S_{t+1}
3: Monitor:
4: Collect metrics (Latency $L_{i,t}$, Cost $C_{i,t}$, Utilization $U_{i,t}$, QoS $Q_{i,t}$, Requests)
5: Analyze:
6: Compute reward R_t using the reward function
7: $R_t = -(\alpha L_i(M_{i,t}) + \beta C_i(M_{i,t}) - \gamma U_i(M_{i,t})) + \delta Q_i(M_{i,t})$
8: Normalize all components (cost, latency, utilization, QoS)
9: using:
10: $X' = (X - X_{min}) / (X_{max} - X_{min})$
11: Plan:
12: Select action $A_t = \pi_\theta (S_t)$
13: Update policy network π_θ using policy gradient
14: $\nabla_\theta J(\pi_\theta) = E_{(s\text{-}d^\pi,\ a\text{-}\pi_\theta)} [\ \nabla_\theta \log \pi_\theta(a|s)\ Q^\pi\theta(s,a)\ ]$
15: Update value network V_ϕ using Bellman equation
16: $V^\pi(S) = E_{\ a\text{-}\pi}\ [\ R_t + \gamma\ V^\pi\ (S+1)\ |\ S_t = s,\ A_t = a\ ]$
17: Execute:
18: Apply action A_t (update memory allocation $M_{i,t+1}$)
19: Transition to next state S_{t+1}
20: Return S_{t+1}

Algorithm 3
Pseudo code for MAPE Execution phase ( ).

1: Input: Environment, networks (π_θ, V_ϕ)
2: Output: Optimized memory allocation policy
3: Initialize environment and obtain initial state S_0
4: For each episode do
5: For each time step t do
6: Run MAPE Loop with input S_t
7: Observe reward R_t and next state S_{t+1}
8: Store tuple (S_t, A_t, R_t, S_{t+1})
9: Update π_θ and V_ϕ based on (S_t, A_t, R_t, S_{t+1})
10: Policy network update
11: $\nabla_\theta J(\pi_\theta) = E_{(s\text{-}d^\pi,\ a\text{-}\pi_\theta)} [\ \nabla_\theta \log \pi_\theta(a|s)\ Q^\pi\theta(s,a)\ ]$
12: Value network update
13: $V^\pi(S) = E_{\ a\text{-}\pi}\ [\ R_t + \gamma\ V^\pi\ (S+1)\ |\ S_t = s,\ A_t = a\ ]$
14: Set $S_t \leftarrow S_{t+1}$
15: End For
16: End For
17: Return optimized policy π_θ

Table 5
Tools and technologies.

Component	Description
Cloud Provider	AWS Lambda (conceptual model), implemented as a simulated environment in code.
Programming Language	Python, for implementing the deep reinforcement learning algorithms.
Deep Learning Framework	TensorFlow/Keras, for building and training the deep learning models.
Reinforcement Learning Algorithm	Proximal Policy Optimization (PPO)
Monitoring Tools	Simulated monitoring module (MAPE-K), no real CloudWatch data used.
Data Management	Pandas, for data manipulation and analysis of the performance metrics.
Development Environment	Jupyter Notebook / Google Colab for interactive development and experimentation.

5.3. Experimental results

We evaluated our proposed approach with baseline methods, including machine learning-based approaches, the impact of learning rate and reward function.

5.3.1. First scenario: impact of learning rate on optimization

One of the important hyperparameters in deep reinforcement learning is the learning rate (LR), which controls the extent of updates to the model's weights. We tested different learning rates to examine their impact on Auto Opt Mem's efficiency.

In this experiment, different learning rates were implemented for the problem of prediction and memory regulation in serverless environments, and a reinforcement learning agent based on policy gradient was implemented. The goal was to show the impact of learning rate on memory allocation policy learning and ultimately on the performance metrics Latency, Cost, QoS, and Utilization. Three learning rate values were tested: 0.01, 0.001, and 0.0001; each experiment was performed on 100 episodes. The environment model and the relationships between memory and metrics were implemented by a noisy function to represent the overall system behavior and the impact of memory selection on metrics.

The learning rate can affect the performance metrics as follows:

Latency: A very high learning rate may destabilize training, preventing the model from converging to an optimal policy. This instability leads to fluctuations in memory allocation and, consequently, higher execution latency. Conversely, a moderate learning rate supports stable convergence, resulting in lower latency.

Cost: If memory allocation decisions are unstable due to an excessively high learning rate, resource usage can increase, raising the execution cost. However, in some cases, a higher learning rate can accelerate convergence to an efficient allocation, thus reducing costs. The impact on cost is therefore dual, depending on whether training stabilizes.

Quality of Service (QoS): High learning rates may cause instability in memory allocation policies, leading to inconsistent performance and degraded QoS. In contrast, a well-tuned moderate learning rate achieves more stable optimization and improved QoS.

Utilization: Rapid but unstable adjustments caused by a high learning rate can lead to inefficient memory allocations, resulting in either under-utilization or over-utilization of resources. A balanced learning rate is more likely to achieve efficient utilization. Fig. 4 shows the learning rates and results.

The cost and quality of service results obtained from training the agent with different learning rates showed:

The intermediate learning rate LR = 0.001 shows the lowest latency and highest QoS among the three values. In contrast, the cost increases significantly and the utilization decreases. In fact, the agent with LR = 0.001 learns faster and more significantly policies that favor higher memory (or policies that make allocations that reduce latency) this leads to improved QoS but increases the cost per execution unit. The intermediate rate allows the agent to accept weight changes strongly enough to reach regions with lower latency (but may be cost-ineffective).

The larger learning rate LR = 0.01 shows very low cost and very high utilization, but latency and QoS remain at moderate levels. In fact, high learning rates usually make large updates; In this simulation, the agent has arrived at a policy that keeps the cost low (e.g., choosing low or average memories) while maintaining high utilization. This could mean learning a cost-saving policy; however, this policy may cause fluctuations and not reach the optimal latency. It is also possible for the agent to get stuck in a local boundary (with low cost) under noisy gradients.

A smaller learning rate of LR = 0.0001 had intermediate results (latency and QoS close to LR=0.01 but average cost). In fact, a too small rate leads to slow and stable learning; the agent may not have fully converged yet and not have seen significant improvement by the end.

Table 6 shows impact of learning rate effects on performance metrics.

5.3.2. Second scenario: reward function formula based on MAPE loop

To further analyze the impact of Auto Opt Mem, we conducted eight experiments and calculated the reward function. As explained in the Section 4, the reward function is calculated from the following formula.

$$R_t = -\left(\sum_{i\in f}\left(\alpha\frac{L_i(M_{i,t}) - L_{\min}}{L_{\max} - L_{\min}} + \beta\frac{C_i(M_{i,t}) - C_{\min}}{C_{\max} - C_{\min}} - \gamma\frac{U_i(M_{i,t}) - U_{\min}}{U_{\max} - U_{\min}}\right)\right) + \delta\frac{Q_i(M_{i,t}) - Q_{\min}}{Q_{\max} - Q_{\min}}$$

Where α, β, γ, and δ are the weights for each variable. The negative at the beginning of the formula shows that we want to include the negative impact of cost and latency in the reward function. Quality of service (Q_i) is considered positive and its positive impact is included in the reward function. Adding up the metrics, we would like to consider the sum of the influence of all functions. To calculate the reward function, the minimum and maximum values are given below:

$L_{\min} = 80, L_{\max} = 100$

$C_{\min} = 0.15, C_{\max} = 0.50$

$U_{\min} = 60, U_{\max} = 80$

$Q_{\min} = 40, Q_{\max} = 70$

The reward function for different data in the experiment is shown in Table 7, (Latency in ms, Cost in USD, QoS %, Utilization %, Reward score).

We calculate the reward function for each experiment.
Experiment 1:

$$R_1 = -\left(\frac{90-80}{100-80} + \frac{0.17-0.15}{0.50-0.15} - \frac{75-60}{80-60}\right) + \frac{66-40}{70-40}$$

$$R_1 = -(0.5 + 0.1333 - 0.75) + 0.8667 = -(-0.1167) = 0.1167$$

The reward function for the remaining experiments is calculated similarly.

These findings strengthen the ability of Auto Opt Mem to optimize execution time while reducing costs, and it has better performance. The reward function is analyzed as follows.

- Increasing QoS and Utilization have increased the reward, because the system has better efficiency.
- Reducing latency and cost has a positive effect on the reward, which indicates more optimal performance.
- The highest reward value is observed in Experiment 8 (0.91), which indicates the optimal balance between QoS, Utilization, latency, and cost.
- The lowest reward value is recorded in Experiment 6 (0.03), which is due to the increase in latency and the decrease in system efficiency (Utilization).

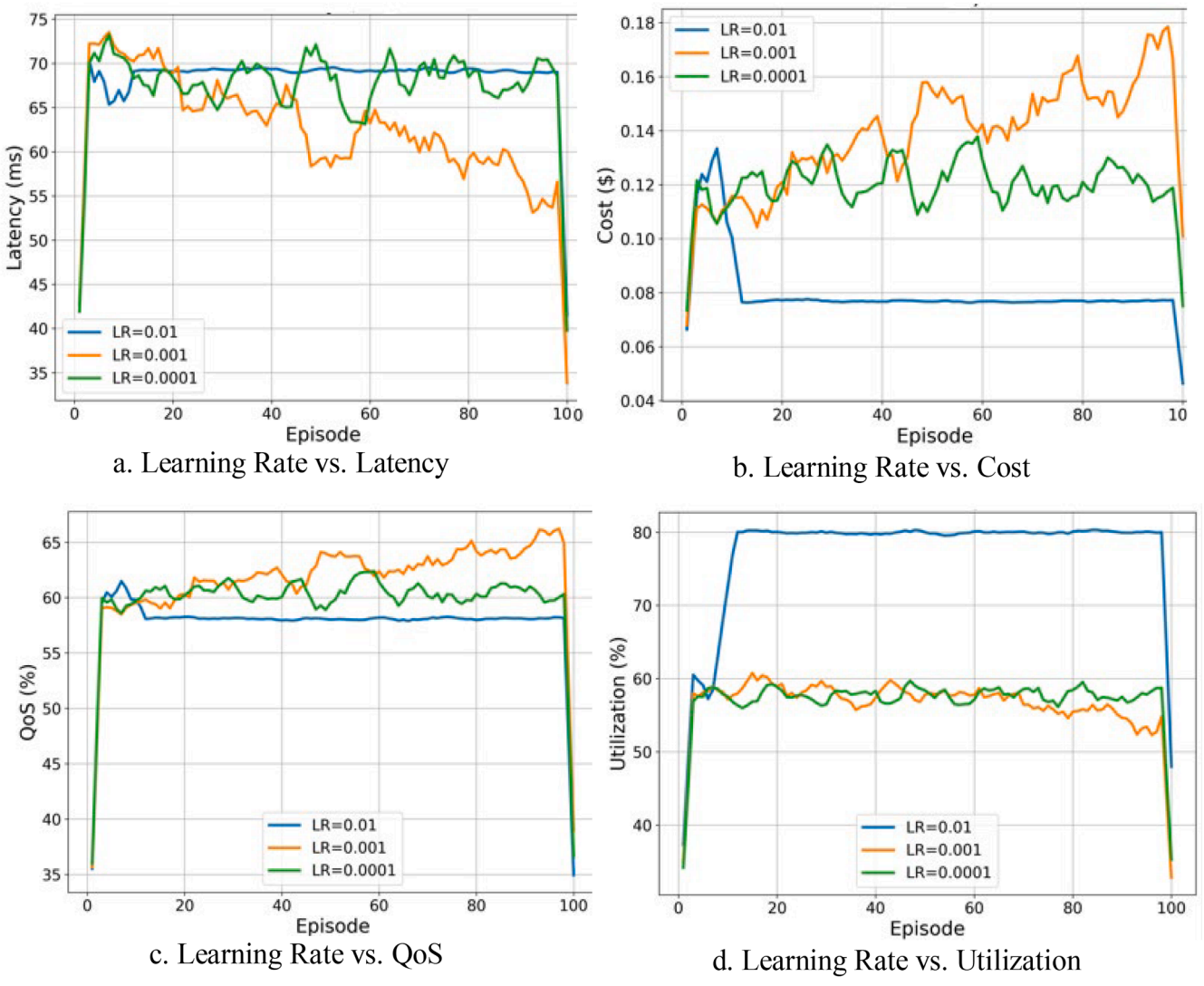


Fig. 4. Impact of learning rate on performance metrics.

Table 6
Comparison of learning rate effects on performance metrics.

Learning Rate (LR)	Latency	Cost	QoS	Utilization
• 0.01 (high)	• Relatively high, unstable, converges to moderate level	• Lowest cost	• Moderate QoS	• Highest utilization
• 0.001 (medium)	• Lowest latency (best)	• Highest cost	• Highest QoS	• Reduced utilization
• 0.0001 (low)	• Moderate, slow convergence	• Moderate cost	• Moderate QoS	• Moderate utilization

Fig. 5 shows the reward function graph. X-axis: Experiment ID (1–8). Y-axis: Reward value (normalized)

Fig. 6 shows Comparison of reward function and other metrics in different experiments.

- Comparison of latency and reward: Fig. 6a shows that decreasing latency generally leads to increasing reward. The reward value is higher at lower latencies (such as 84 and 85 milliseconds), but decreases at higher latencies.
- Comparison of cost and reward: Fig. 6b shows that while costs may be decreasing, the rewards are increasing, indicating a potentially favorable outcome for the experiments.
- Comparison of quality of service and reward: Fig. 6c shows that increasing QoS usually increases reward. Quality of service in the range of 70–72 % has the highest reward value.
- Comparison of utilization and reward: Fig. 6d shows that higher utilization usually leads to increased reward. The reward value increases sharply for values of 80 % and above.

These graphs show that the optimized system tends to reduce latency, reduce cost, increase quality of service, and improve efficiency or utilization to obtain maximum reward. The suggested reward function, which incorporates delay, cost, utilization, and QoS, can effectively balance the system's various goals. Experiment 8 with the highest reward of 0.91 depicts the optimal balance between all the objectives. Experiment 6 with the lowest reward of 0.03 is worse due to the high delay and low utilization. The comparison graphs in Fig. 6 clearly show that the improvement in each of the criteria, such as reducing delay and cost, or increasing QoS and utilization, leads to an increase in reward. This means that the reward function design is suitable and can be used as a criterion for multi-objective optimization.

5.3.3. Third scenario: comparison with previous studies

To validate our results, we compared Auto Opt Mem with two papers [17] and [18] that tackled memory optimization in serverless computing. When compared to the works of Eismann et al. [17] and Jindal et al. [18], Auto Opt Mem achieves a more substantial reduction in execution latency, a greater cost reduction, and a significant improvement in QoS. Unlike previous studies that primarily relied on static function profiling or statistical estimations, Auto Opt Mem continuously learns and adapts to varying workloads, and thus is more adaptive and scalable. Eismann et al. [17] developed a model for resource prediction based on monitoring a single memory size, achieving performance gains but with limited adaptability to dynamic workloads. Jindal et al. [18] introduced a statistical and deep learning approach to estimate function capacity, improving resource efficiency but lacking real-time optimization capabilities. Auto Opt Mem integrates reinforcement learning to dynamically adjust memory allocations, ensuring optimal performance across diverse execution environments without requiring manual intervention.

Table 7
Reward function values in different experiments.

Experiment	Latency (ms)	Cost ($)	QoS (%)	Utilization (%)	Reward
1	90	0.17	66	75	0.11
2	88	0.16	63	78	0.43
3	92	0.18	67	74	0.8
4	87	0.15	70	80	0.65
5	89	0.17	64	76	0.21
6	91	0.16	68	73	0.03
7	86	0.15	71	79	0.65
8	85	0.14	72	82	0.91

This section assesses the proposed Auto Opt Mem in a realistic serverless simulation environment. Four workloads were modeled as representatives, including ML inference, API gateway, data processing, and video processing, each independently defined by its memory range, baseline latency, and cost functions. A neural network with two hidden layers (32 and 16 neurons) was used to train the PPO agent for 80 episodes. The reward function incorporates normalized latency, cost, and QoS terms that drive the policy to achieve a balanced optimization. An autonomic MAPE-K (Monitor, Analyze, Plan, Execute – Knowledge) loop was implemented at runtime to make the system self-adaptive. MAPE continuously monitors the recent performance, analyzes QoS/cost trends, plans corrective actions such as tuning the memory, and executes them through the adjustment of PPO policy parameters. The results are summarized in Table 8.

Auto Opt Mem demonstrates superior improvements across all metrics compared to existing research, establishing its effectiveness. Fig. 7 shows the Comparison with previous studies.

Table 8 reports the average performance achieved by Auto Opt Mem across all benchmark functions in terms of latency reduction, cost savings, and QoS improvement. These values are directly compared with the results of Sizeless [17] and FnCapacitor [18]. Results demonstrate that, on average, the proposed Auto Opt Mem achieves 25–30 % less latency, 15–18 % less cost, and 10–12 % more QoS when compared to the Sizeless [17] baseline, while outperforming FnCapacitor [18] in all key metrics. This proves that PPO-based MAPE-K autonomic loop can dynamically adjust for the workload variability and provide optimized resource allocation in serverless environments.

The results of 8 experiments conducted with different metrics are presented in Table 9. Synthetic yet reproducible data were used, and the dataset is openly provided for replication.

Table 10 summarizes the statistical comparison of Auto Opt Mem with the baseline methods. For each metric, the mean, standard deviation, and 95 % confidence interval were calculated. For instance, Auto Opt Mem has a latency of 267.92 ms $\pm$ 1.65 with a 95 % CI of (266.74, 269.10), outperforming both Sizeless [17] and FnCapacitor [18].

For statistical significance, we use an independent two-sample *t*-test. Indeed, our results indicate that the improvements of Auto Opt Mem over Sizeless [17] are statistically significant for all metrics ($p < 0.02$). For FnCapacitor [18], the cost differences are significant with $p = 0.035$.

The statistical analysis confirms that Auto Opt Mem provides consistently better performance, with several improvements being significant at the 95 % confidence level.

The average latency, average cost, and average quality of service in different methods are shown in the graph in Fig. 8.

In Fig. 8a, Auto Opt Mem (green line) has the lowest latency in all tests, indicating better optimization in memory allocation and faster execution of serverless functions.

In Fig. 8b, right, Auto Opt Mem minimizes the cost of executing functions in all experiments. Jindal et al. [18] method reduces the cost compared to Eismann et al. [17] but is still higher than Auto Opt Mem.

In Fig. 8c, Auto Opt Mem (green line) has the highest Quality of Service (QoS), indicating increased reliability and optimal performance under different workload conditions. Jindal et al. [18] method provides better QoS than Eismann et al. [17] but still falls short of Auto Opt

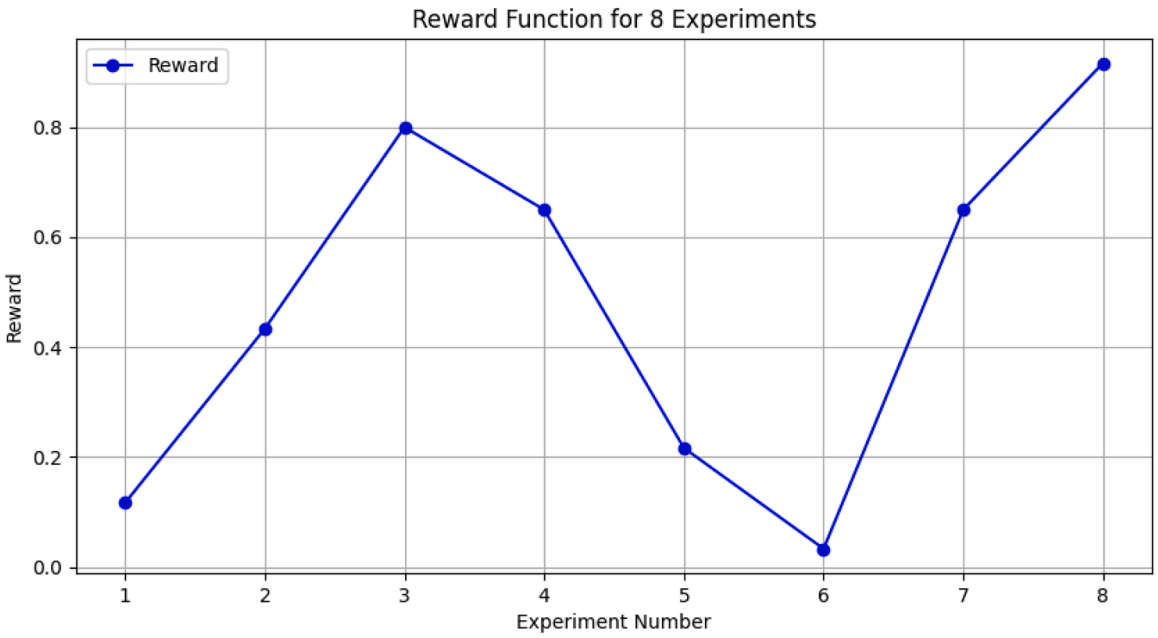


Fig. 5. Reward function values for eight experimental runs.

Mem's. Table 11 further illustrates the differences of our approach from previous methods.

As compared to static allocation strategies, Auto Opt Mem saves a lot of resource wastage by memory allocation according to actual function requirements rather than over-provisioning. Such dynamic allocation leads to cost saving in a large extent, especially in the case of varying workload or mixed kinds of functions. Moreover, the ability of Auto Opt Mem to minimize latency leads to improved application performance and user experience. By efficient memory management and reducing cold start time, Auto Opt Mem promises that functions will execute both quickly and reliably even in high-demand situations. In contrast to heuristic methods that typically operate on oversimplification assumptions and struggle with dynamic states, Auto Opt Mem employs deep reinforcement learning to discover decisions from real current states. Such responsiveness is important in numerous serverless computing applications. While machine learning-based methods are somewhat flexible, Auto Opt Mem is better at balancing conflicting objectives, e.g., cost minimization and optimal performance. By means of a reward function that considers several parameters, Auto Opt Mem gives an overall optimization of the system.

6. Discussion

In this section, we discuss about the AWS lambda platform, potential application scenarios, and then provide explanations about the comparative analysis on the memory configuration in serverless computing.

6.1. Use of AWS

The experiments were conducted in a simulated environment; the workload behavior, latency patterns, and cost models were derived from documented AWS Lambda configuration rules. This ensures that the characteristics of a real AWS serverless environment are captured while still allowing full reproducibility.

AWS Lambda serves as the conceptual reference platform due to its very high market share and representative resource allocation model with regards to, for example, memory-to-vCPU coupling and pricing scheme, which closely aligns with Azure Functions and Google Cloud Functions. Auto Opt Mem is provider-agnostic since platform-specific constraints, such as memory ranges and CPU scaling rules, are embedded in the state representation. Thus, even though execution was simulated, the framework is transferable to real AWS Lambda deployments and other cloud providers as well.

6.2. Potential application scenarios

Due to resource and space limitations, we used controlled micro-benchmarks. Still, Auto Opt Mem is not restricted to this setup and can also be used in real applications. For example, it helps with machine learning inference tasks, such as running image or text classification models where reducing cost and latency is important. It is also useful for video transcoding, since converting formats with tools such as ffmpeg usually takes a lot of CPU and memory. Another case is data preprocessing, where large datasets need to be read, compressed, or filtered. Auto Opt Mem also works well in API aggregation, when several external services are called at the same time and the process is mostly I/O-bound.

These examples illustrate that Auto Opt Mem is workload-agnostic and can be applied to real-world scenarios. A full-scale practical evaluation is considered for future research. Table 12 shows real-world scenarios where Auto Opt Mem can be used.

6.3. Comparative analysis

Table 13 summarizes recent research in intelligent cloud computing that focuses on automation, optimization, and deep learning. This table shows how the proposed Auto Opt Mem framework aligns with these studies and focuses on dynamic memory and performance optimization in serverless environments.

7. Conclusions

Memory configuration in serverless computing can be challenging

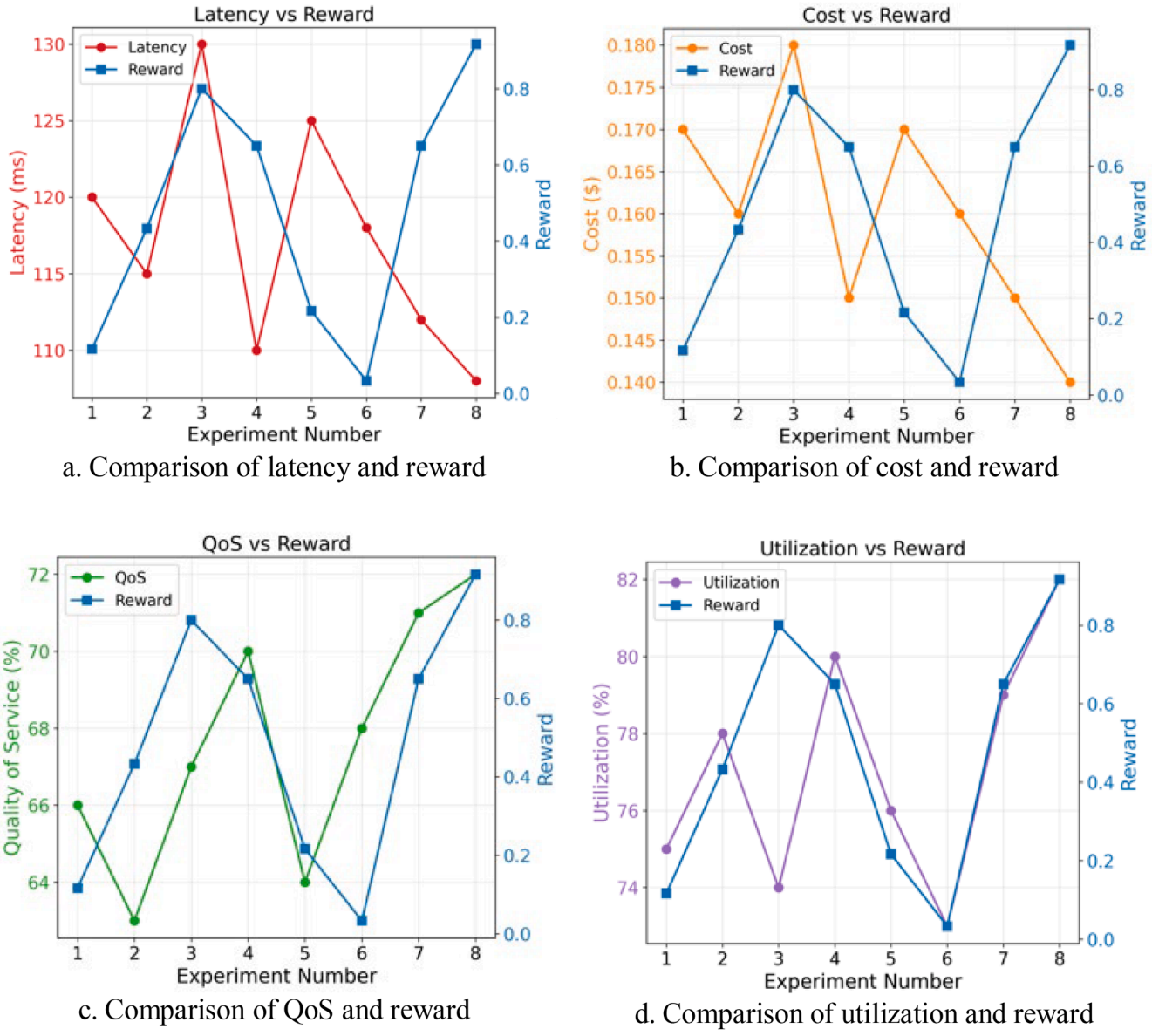


Fig. 6. Comparison of reward function and other metrics.

Table 8
Comparison with previous studies.

Approach	Latency Reduction (%)	Cost Reduction (%)	QoS Improvement (%)
Auto Opt Mem vs Sizeless [17]	25–30 %	15–18 %	10–12 %
Auto Opt Mem vs FnCapacitor [18]	5–7 %	6–8 %	2–3 %

due to the ephemeral nature of serverless functions, which are short-lived and stateless. This research examines memory configuration mechanisms and then classifies these mechanisms in serverless computing into three main approaches: machine learning-based, exploration-based, and framework-based approaches. The advantages and disadvantages of each mechanism, as well as the challenges and performance metrics affecting their effectiveness, are discussed. Memory configuration is one of the important challenges in serverless computing; In this paper, we propose an autonomous deep learning-based serverless computing memory optimization system, referred to as Auto Opt Mem. The results show that Auto Opt Mem optimizes resource utilization, decreases operation costs and latency, and enhances quality of service (QoS), and hence it can be a perfect fit for developers in serverless systems. Auto Opt Mem shows noticeable improvements over previous methods. Compared to Sizeless, it reduces latency by 25–30 %, lowers cost by 15–18 %, and improves QoS by 10–12 %. Against FnCapacitor, it achieves 5–7 % latency reduction, 6–8 % cost reduction, and 2–3 % QoS improvement. Our experiments demonstrate that Auto Opt Mem On average provides 16.8 % lower latency, 11.8 % cost reduction, and 6.8 % QoS improvement across both methods.

In our approach one of the important hyperparameters in deep reinforcement learning is the learning rate, which controls the extent of updates to the model's weights. A high learning rate typically causes the model to update its weights more aggressively. A higher learning rate may cause inefficient memory usage because of unstable learning, leading to suboptimal allocation. But, a very small learning rate could lead to very slow training, Long-term training and slow convergence, leading to late optimization of memory. It can be said that a high learning rate can achieve convergence quickly, but it may pass the

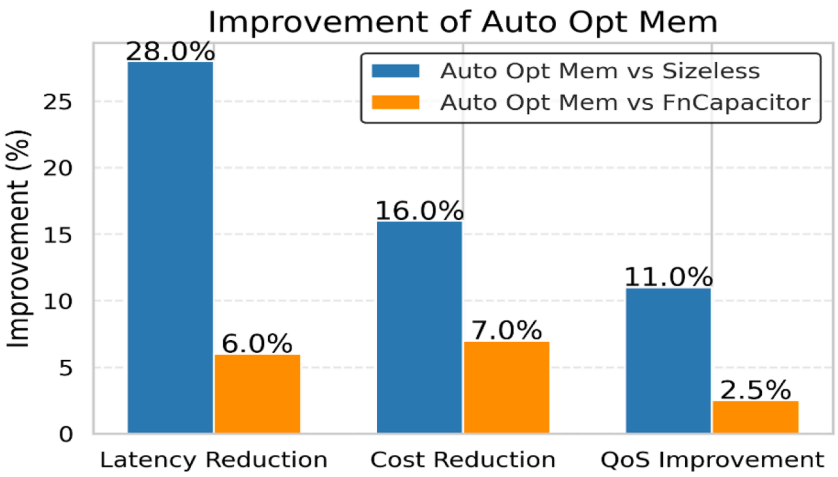


Fig. 7. Comparison with previous studies.

Table 9
Comparison of approaches across 8 experiments.

Approach	Latency (ms)	Cost ($)	QoS (%)
Sizeless [17]	362.85, 370.14, 355.92, 380.18, 364.77, 359.60, 372.31, 368.54	0.00243, 0.00236, 0.00248, 0.00244, 0.00241, 0.00237, 0.00246, 0.00242	80.15, 81.24, 79.87, 80.45, 81.06, 80.83, 79.92, 80.61
FnCapacitor [18]	278.42, 271.66, 282.10, 276.74, 274.89, 280.13, 273.80, 275.55	0.00256, 0.00252, 0.00257, 0.00255, 0.00251, 0.00253, 0.00259, 0.00254	89.44, 90.22, 88.97, 89.75, 90.13, 89.61, 89.88, 90.06
Auto Opt Mem	267.91, 270.34, 265.10, 268.05, 266.83, 269.14, 267.54, 266.41	0.00221, 0.00219, 0.00223, 0.00220, 0.00218, 0.00222, 0.00221, 0.00220	91.08, 91.46, 90.83, 91.21, 91.37, 90.97, 91.32, 91.15

Table 10
Statistical comparison of Auto Opt Mem with baseline methods.

Approach	Metric	Average	Std. Dev.	95 % CI	p-value
					vs. Auto Opt Mem
Sizeless [17]	Latency (ms)	366.29	8.18	(360.43, 372.15)	0.002
	Cost ($)	0.00243	0.00004	(0.00240, 0.00246)	0.018
	QoS (%)	80.52	0.43	(80.22, 80.82)	0.001
FnCapacitor [18]	Latency (ms)	276.29	3.29	(273.97, 278.61)	0.120
	Cost ($)	0.00255	0.00003	(0.00253, 0.00257)	0.035
	QoS (%)	89.88	0.41	(89.59, 90.17)	0.280
Auto Opt Mem	Latency (ms)	267.92	1.65	(266.74, 269.10)	–
	Cost ($)	0.00221	0.00002	(0.00220, 0.00222)	–
	QoS (%)	91.17	0.24	(91.00, 91.34)	–

optimal point. As a result, it will lead to irregular updates and require additional iterations to correct errors, which will increase memory consumption. And a low learning rate helps in more stable and gradual convergence, but may require more periods to reach convergence, which can increase the memory load due to long-term storage of intermediate states. Future research directions include extending Auto Opt Mem to multi-cloud environments, extending and stress-testing Auto Opt Mem's real-time adaptability under more diverse and large-scale workloads. Additionally, the scalability of Auto Opt Mem can also be researched on other serverless frameworks except AWS Lambda.

Funding

Funding was received for this work.

All of the sources of funding for the work described in this publication are acknowledged below:

[List funding sources and their role in study design, data analysis, and result interpretation]

No funding was received for this work.

Intellectual property

We confirm that we have given due consideration to the protection of intellectual property associated with this work and that there are no impediments to publication, including the timing of publication, with respect to intellectual property. In so doing we confirm that we have followed the regulations of our institutions concerning intellectual property.

Research ethics

We further confirm that any aspect of the work covered in this manuscript that has involved human patients has been conducted with the ethical approval of all relevant bodies and that such approvals are acknowledged within the manuscript.

IRB approval was obtained (required for studies and series of 3 or more cases)

Written consent to publish potentially identifying information, such as details or the case and photographs, was obtained from the patient(s) or their legal guardian(s).

Authorship

The International Committee of Medical Journal Editors (ICMJE) recommends that authorship be based on the following four criteria:

1. Substantial contributions to the conception or design of the work; or the acquisition, analysis, or interpretation of data for the work; AND
2. Drafting the work or revising it critically for important intellectual content; AND
3. Final approval of the version to be published; AND

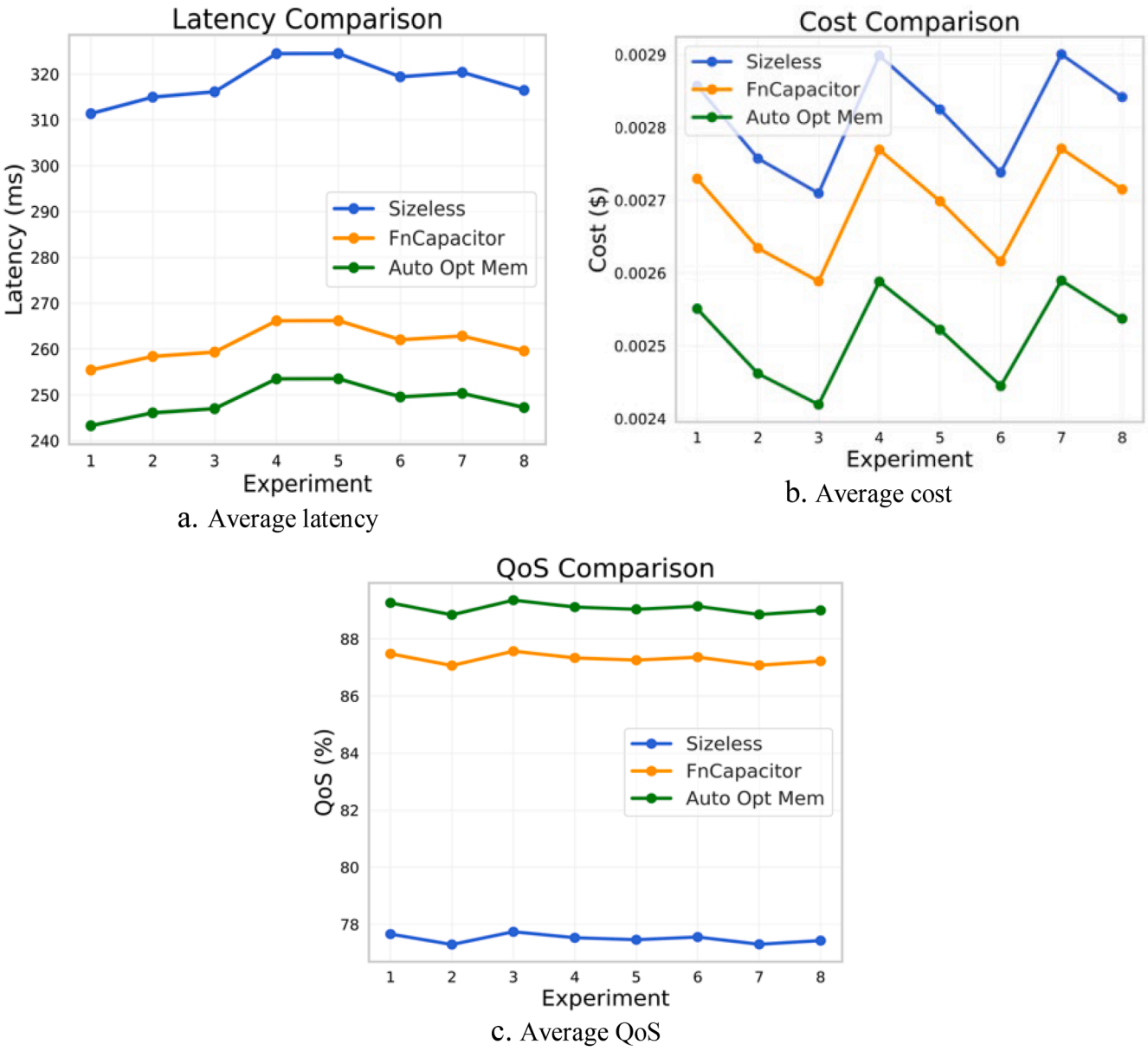


Fig. 8. Compare performance metrics.

4. Agreement to be accountable for all aspects of the work in ensuring that questions related to the accuracy or integrity of any part of the work are appropriately investigated and resolved.

All those designated as authors should meet all four criteria for authorship, and all who meet the four criteria should be identified as authors. For more information on authorship, please see https://www.icmje.org/recommendations/browse/roles-and-responsibilities/defining-the-role-of-authors-and-contributors.html#two.

All listed authors meet the ICMJE criteria. We attest that all authors contributed significantly to the creation of this manuscript, each having fulfilled criteria as established by the ICMJE.

One or more listed authors do(es) not meet the ICMJE criteria.

We believe these individuals should be listed as authors because:

[Please elaborate below]

We confirm that the manuscript has been read and approved by all named authors.

We confirm that the order of authors listed in the manuscript has been approved by all named authors.

Contact with the editorial office

The Corresponding Author declared on the title page of the manuscript is:

[Mostafa Ghobaei-Arani]

This author submitted this manuscript using his/her account in EVISE.

We understand that this Corresponding Author is the sole contact for the Editorial process (including EVISE and direct communications with the office). He/she is responsible for communicating with the other authors about progress, submissions of revisions and final approval of proofs.

We confirm that the email address shown below is accessible by the Corresponding Author, is the address to which Corresponding Author's EVISE account is linked, and has been configured to accept email from the editorial office of American Journal of Ophthalmology Case Reports:

Someone other than the Corresponding Author declared above

Table 11
Differentiation of our approach from previous studies.

Aspect	Sizeless [17]	FnCapacitor [18]	Our Work (Auto Opt Mem)
Methodology	• Multi-target regression using monitoring data from a single memory size	• Sandboxing, performance tests, and statistical/DNN modeling	• Deep Reinforcement Learning with MAPE control loop
Decision Type	• Predicts execution time and cost for other memory sizes	• Estimate's function capacity (max concurrency under SLO)	• Learns and selects optimal memory configuration
Adaptivity	• Static once trained, no continuous learning	• Requires offline profiling for changes	• Dynamic and continuous adaptation at runtime
Focus	• Memory-performance trade-offs	• Function capacity and concurrency	• Memory optimization balancing latency, cost, QoS, and utilization
Limitation	• No runtime adaptability	• *Re*-profiling needed for new workloads	—
Innovation	• Efficient prediction with limited input	• Accurate FC estimation for functions	• Self-adaptive and autonomous optimization

Table 12
Application Scenarios for Auto Opt Mem.

Scenario	Type of Workload	Role of Auto Opt Mem
• ML inference	• CPU-bound	• Optimizes latency and cost by adjusting memory/CPU
• Video transcoding	• CPU & memory-intensive	• Balances higher memory cost with faster execution time
• Data preprocessing (ETL)	• Mixed (CPU + *I*/O)	• Adapts memory allocation based on input size
• API aggregation	• I/O-bound	• Keeps memory low while ensuring QoS in parallel API calls

Table 13
Comparative analysis with recent studies in the field of intelligent cloud computing.

Ref	Focus Area	Technique	Key Contribution	Relation to Present Work
[59]	Secure data deduplication	Convergent encryption	Reduces redundancy and ensures secure cloud storage	Our DRL approach similarly targets resource efficiency but in serverless memory optimization
[60]	Cloud key management	Machine learning-based security framework	Intelligent key lifecycle management	Both emphasize intelligent automation for secure and efficient cloud operations
[61]	Deep learning for cloud/edge/fog/IoT	Deep learning models survey	Provides insight into intelligent distributed learning paradigms	Inspires our DL-driven resource optimization framework
[62]	Cloud security and privacy	Deep learning-based attack detection	Enhances privacy and adaptive threat response	Our work extends this intelligence toward performance optimization and QoS improvement in serverless systems

submitted this manuscript from his/her account in EVISE:

[Insert name below]

We understand that this author is the sole contact for the Editorial process (including EVISE and direct communications with the office). He/she is responsible for communicating with the other authors, including the Corresponding Author, about progress, submissions of revisions and final approval of proofs.

CRediT authorship contribution statement

Zahra Shojaee Rad: Resources, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Mostafa Ghobaei-Arani:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration. **Reza Ahsan:** Writing – review & editing, Writing – original draft.

Declaration of competing interest

Potential conflict of interest exists:

We wish to draw the attention of the Editor to the following facts, which may be considered as potential conflicts of interest, and to significant financial contributions to this work:

The nature of potential conflict of interest is described below:

No conflict of interest exists.

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, et al., Serverless computing: current trends and open problems, Res. Adv. Cloud Comput. (2017) 1–20.

[2] A. Ebrahimi, M. Ghobaei-Arani, H. Saboohi, Cold start latency mitigation mechanisms in serverless computing: taxonomy, review, and future directions, J. Syst. Archit. 151 (2024) 103115.

[3] AWS, "Serverlessvideo: connect with users around the world!." https://serverlessland.com/, 2023.

[4] AWS, "Serverless case study - netflix." https://dashbird.io/blog/serverless-case-study-netflix/, 2020.

[5] CapitalOne, "Capital one saves developer time and reduces costs by going serverless on aws." https://aws.amazon.com/solutions/case-studies/capital-one-lambda-ecs-case-study/, 2023.

[6] E. Johnson, "Deploying ml models with serverless templates." https://aws.amazon.com/blogs/compute/deploying-machine-learning-models-with-serverless-templates/, 2021.

[7] A. Sojasingarayar, "Build and deploy llm application in aws." https://medium.com/@abonia/build-and-deploy-llm-application-in-aws-cca46c662749, 2024.

[8] A. Gholami, M. Ghobaei-Arani, A trust model based on quality of service in cloud computing environment, Int. J. Database Theor. Appl. 8 (5) (2015) 161–170, https://doi.org/10.14257/ijdta.2015.8.5.13.

[9] DataDog. 2020. The State of Serverless. https://www.datadoghq.com/state-of-serverless/.

[10] M. Tari, M. Ghobaei-Arani, J. Pouramini, M. Ghorbian, Auto-scaling mechanisms in serverless computing: a comprehensive review, Comput. Sci. Rev. 53 (2024) 100650, https://doi.org/10.1016/j.cosrev.2024.100650.

[11] M. Ghorbian, M. Ghobaei-Arani, R. Asadolahpour-Karimi, Function placement approaches in serverless computing: A survey, J. Syst. Archit. 157 (2024) 103291.

[12] B. Jacob, R. Lanyon-Hogg, D.K. Nadgir, A.F. Yassin, A Practical Guide to the IBM Autonomic Computing toolkit. IBM, International Technical Support Organization, 2004.

[13] Michael Maurer, Ivan Breskovic, Vincent C. Emeakaroha, Ivona Brandic, Revealing the MAPE loop for the autonomic management of cloud infrastructures, in: 2011 IEEE Symposium on Computers and Communications (ISCC), IEEE, 2011, pp. 147–152.

[14] Russell, Stuart J., and Peter Norvig. Artificial intelligence: a modern approach. pearson, 2016.

[15] Leslie Pack Kaelbling, Michael L. Littman, Andrew W. Moore, Reinforcement learning: a survey, J. Artif. Intell. Res. 4 (1996) 237–285.

[16] Rajkumar Rajavel, Mala Thangarathanam, Adaptive probabilistic behavioural learning system for the effective behavioural decision in cloud trading negotiation market, Fut. Gener. Comput. Syst. 58 (2016) 29–41.
[17] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, Samuel Kounev, Sizeless: predicting the optimal size of serverless functions, in: Proceedings of the 22nd International Middleware Conference, 2021, pp. 248–259.
[18] Anshul Jindal, Mohak Chadha, Shajulin Benedict, Michael Gerndt, Estimating the capacities of function-as-a-service functions, in: In Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion, 2021, pp. 1–8.
[19] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, et al., OFC: an opportunistic caching system for FaaS platforms, in: Proceedings of the Sixteenth European Conference on Computer Systems, 2021, pp. 228–244.
[20] Myung-Hyun Kim, Jaehak Lee, Heonchang Yu, Eunyoung Lee, Improving memory utilization by sharing DNN models for serverless inference, in: 2023 IEEE International Conference on Consumer Electronics (ICCE), IEEE, 2023, pp. 1–6.
[21] Agarwal, Siddharth, Maria A. Rodriguez, and Rajkumar Buyya. "Input-based ensemble-learning method for dynamic memory configuration of serverless computing functions." arXiv preprint arXiv:2411.07444 (2024).
[22] Gor Safaryan, Anshul Jindal, Mohak Chadha, Michael Gerndt, SLAM: sLO-aware memory optimization for serverless applications, in: 2022 IEEE 15th International Conference on Cloud Computing (CLOUD), IEEE, 2022, pp. 30–39.
[23] Robert Cordingly, Sonia Xu, Wes Lloyd, Function memory optimization for heterogeneous serverless platforms with cpu time accounting, in: 2022 IEEE International Conference on Cloud Engineering (IC2E), IEEE, 2022, pp. 104–115.
[24] Tetiana Zubko, Anshul Jindal, Mohak Chadha, Michael Gerndt, Maff: self-adaptive memory optimization for serverless functions, in: European Conference on Service-Oriented and Cloud Computing, Cham: Springer International Publishing, 2022, pp. 137–154.
[25] Josef. Spillner, Resource management for cloud functions with memory tracing, profiling and autotuning, in: Proceedings of the 2020 Sixth International Workshop on Serverless Computing, 2020, pp. 13–18.
[26] Zengpeng Li, Huiqun Yu, Guisheng Fan, Time-cost efficient memory configuration for serverless workflow applications, Concurr. Comput.: Pract. Exp. 34 (27) (2022) e7308, no.
[27] Andrea Sabbioni, Lorenzo Rosa, Armir Bujari, Luca Foschini, Antonio Corradi, A shared memory approach for function chaining in serverless platforms, in: 2021 IEEE Symposium on Computers and Communications (ISCC), IEEE, 2021, pp. 1–6.
[28] Aakanksha Saha, Sonika Jindal, EMARS: efficient management and allocation of resources in serverless, in: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), IEEE, 2018, pp. 827–830.
[29] Amit Samanta, Faraz Ahmed, Lianjie Cao, Ryan Stutsman, Puneet Sharma, Persistent memory-aware scheduling for serverless workloads, in: 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, 2023, pp. 615–621.
[30] Meenakshi Sethunath, Yang Peng, A joint function warm-up and request routing scheme for performing confident serverless computing, High-Confidence Comput. 2 (3) (2022) 100071 no.
[31] Anisha Kumari, Manoj Kumar Patra, Bibhudatta Sahoo, Ranjan Kumar Behera, Resource optimization in performance modeling for serverless application, Int. J. Inf. Technol. 14 (6) (2022) 2867–2875, no.
[32] Vahldiek-Oberwagner, Anjo, and Mona Vij. "Meshwa: the case for a memory-safe software and hardware architecture for serverless computing." arXiv preprint arXiv:2211.08056 (2022).
[33] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, Aditya Akella, Memory deduplication for serverless computing with medes, in: Proceedings of the Seventeenth European Conference on Computer Systems, 2022, pp. 714–729.
[34] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, Keqiu Li, Tetris: memory-efficient serverless inference through tensor sharing, in: 2022 USENIX Annual Technical Conference (USENIX ATC 22), 2022.
[35] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, Boris Grot, Benchmarking, analysis, and optimization of serverless function snapshots, in: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021, pp. 559–572.
[36] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, Yue Cheng, {InfiniCache}: exploiting ephemeral serverless functions to build a {cost-effective} memory cache, in: 18th USENIX conference on file and storage technologies (FAST 20), 2020, pp. 267–281.
[37] Anurag Khandelwal, Yupeng Tang, Rachit Agarwal, Aditya Akella, Ion Stoica, Jiffy: elastic far-memory for stateful serverless analytics, in: Proceedings of the Seventeenth European Conference on Computer Systems, 2022, pp. 697–713.
[38] Nikolos, Orestis Lagkas, Chloe Alverti, Stratos Psomadakis, Georgios Goumas, and Nectarios Koziris. "Fast and efficient memory reclamation for serverless MicroVMs." arXiv preprint arXiv:2411.12893 (2024).
[39] Zahra Shojaee Rad, Mostafa Ghobaei-Arani, Data pipeline approaches in serverless computing: a taxonomy, review, and research trends, J. Big. Data 11 (1) (2024) 1–42, no.
[40] Zahra Shojaee rad, Mostafa Ghobaei-Arani, Reza Ahsan, Memory orchestration mechanisms in serverless computing: a taxonomy, review and future directions, Cluster. Comput. (2024) 1–27.
[41] R. Wolski, C. Krintz, F. Bakir, G. George, W.-T. Lin, Cspot: portable, multi-scale functions-as-a-service for iot, in: Proceedings of the 4th ACM/IEEE Symposium on Edge Computing (SEC '19). Association for Computing Machinery, New York, 2019, pp. 236–249, https://doi.org/10.1145/3318216.3363314.
[42] V. Yussupov, U. Breitenbucher, F. Leymann, M. Wurster, A systematic mapping study on engineering function-as-a-service platforms and tools, in: Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC'19). Association for Computing Machinery, New York, 2019, pp. 229–240, https://doi.org/10.1145/3344341.3368803.
[43] Zahra Shojaee Rad, Mostafa Ghobaei-Arani, Federated serverless cloud approaches: a comprehensive review, Comput. Electric. Eng. 124 (2025) 110372.
[44] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, et al., Serverless computing: current trends and open problems, Res. Adv. Cloud Comput. (2017) 1–20.
[45] M. Elsakhawy, M. Bauer, Faas2f: a framework for autoining execution-sla in serverless computing, in: 2020 IEEE Cloud Summit, 2020, pp. 58–65, https://doi.org/10.1109/IEEECloudSummit48914.2020.00015.
[46] A.U. Gias, G. Casale, Cocoa: cold start aware capacity planning for function-as-a-service platforms, in: 2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2020, pp. 1–8, https://doi.org/10.1109/MASCOTS50786.2020.9285966.
[47] C.K. Dehury, S.N. Srirama, T.R. Chhetri, Ccodamic: a framework for coherent coordination of data migration and computation platforms, Futur. Gener. Comput. Syst. 109 (2020) 1–16, https://doi.org/10.1016/j.future, 2020.03.029.
[48] A. Tariq, A. Pahl, S. Nimmagadda, E. Rozner, S. Lanka, Sequoia: enabling quality-of-service in serverless computing, in: Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20). Association for Computing Machinery, New York, 2020, pp. 311–327, https://doi.org/10.1145/3419111.3421306.
[49] J. Manner, S. Kolb, G. Wirtz, Troubleshooting serverless functions: a combined monitoring and debugging approach, SICS Softw.-Intensiv. Cyber-Phys. Syst. 34 (2) (2019) 99–104, https://doi.org/10.1007/s00450-019-00398-6.
[50] J. Nupponen, D. Taibi, Serverless: what it is, what to do and what not to do, in: 2020 IEEE International Conference on Software Architecture Companion (ICSA-C), 2020, pp. 49–50, https://doi.org/10.1109/ICSA-C50368.2020.00016.
[51] G. Cordasco, M. D'Auria, A. Negro, V. Scarano, C. Spagnuolo, Fly: a domain-specific language for scientific computing on faas, in: U. Schwardmann, C. Boehme, B. Heras D, V. Cardellini, E. Jeannot, A. Salis, C. Schifanella, R.R. Manumachu, D. Schwamborn, L. Ricci, O. Sangyoon, T. Gruber, L. Antonelli, S.L. Scott (Eds.), Euro-Par 2019: Parallel Processing Workshops, Springer, Cham, 2020, pp. 531–544.
[52] B. Jambunathan, K. Yoganathan, Architecture decision on using microservices or serverless functionswith containers, in: 2018 International Conference on Current Trends Towards Converging Technologies (ICCTCT), 2018, pp. 1–7, https://doi.org/10.1109/ICCTCT.2018.8551035.
[53] A. Keshavarzian, S. Sharifian, S. Seyedin, Modified deep residual network architecture deployed on serverless framework of iot platform based on human activity recognition application, Futur. Gener. Comput. Syst. 101 (2019) 14–28, https://doi.org/10.1016/j.future.2019.06.009.
[54] Gerald. Tesauro, Temporal difference learning and TD-gammon, Commun. ACM 38 (3) (1995) 58–68.
[55] Eric Rutten, Nicolas Marchand, Daniel Simon, Feedback control as MAPE-K loop in autonomic computing. Software Engineering for Self-Adaptive Systems III. Assurances: International Seminar, Dagstuhl Castle, Germany, December 15-19, 2013, Revised Selected and Invited Papers, Springer International Publishing, Cham, 2018, pp. 349–373.
[56] Evangelina Lara, Leocundo Aguilar, Mauricio A. Sanchez, Jesús A. García, Adaptive security based on mape-k: a survey. Applied Decision-Making: Applications in Computer Sciences and Engineering, Springer International Publishing, Cham, 2019, pp. 157–183.
[57] Jeffrey O. Kephart, David M. Chess, The vision of autonomic computing, Computer. (Long. Beach. Calif) 36 (1) (2003) 41–50.
[58] Alistair McLean, Roy Sterritt, Autonomic Computing in the Cloud: an overview of past, present and future trends, in: The 2023 IARIA Annual Congress on Frontiers in Science, Technology, Services, and Applications: Technical Advances and Human Consequences, 2023.
[59] Shahnawaz Ahmad, Mohd Arif, Javed Ahmad, Mohd Nazim, Shabana Mehfuz, Convergent encryption enabled secure data deduplication algorithm for cloud environment, Concurr. Computat.: Pract. Exp. 36 (21) (2024) e8205.
[60] Shahnawaz Ahmad, Shabana Mehfuz, Shabana Urooj, Najah Alsubaie, Machine learning-based intelligent security framework for secure cloud key management, Cluster. Comput. 27 (5) (2024) 5953–5979.
[61] Shahnawaz Ahmad, Iman Shakeel, Shabana Mehfuz, Javed Ahmad, Deep learning models for cloud, edge, fog, and IoT computing paradigms: survey, recent advances, and future directions, Comput. Sci. Rev. 49 (2023) 100568.
[62] Shahnawaz Ahmad, Mohd Arif, Shabana Mehfuz, Javed Ahmad, Mohd Nazim, Deep learning-based cloud security: innovative attack detection and privacy focused key management, IEEE Trans. Comput. (2025).