國 立 清 華 大 學

National Tsing Hua University

碩 士 論 文

Master's Thesis

DOME：面向人工智慧代理的動態MCP 邊緣協同調度

DOME: Dynamic On-Premise MCP Edge Orchestration for AI Agents

系所別：資訊工程學系

Dept. / Grad. Inst.: Computer Science

學號 Student ID: 113062544

研究生 Author： 賴政嘉 Cheng-Chia Lai

指導教授 Advisor： 徐正炘 Cheng-Hsin Hsu

中 華 民 國 一一五 年 六 月

June 2026

國 立 清 華 大 學

National Tsing Hua University

碩 士 論 文

Master's Thesis

DOME：面向人工智慧代理的動態MCP 邊緣協同調度

DOME: Dynamic On-Premise MCP Edge Orchestration for AI Agents

系所別：資訊工程學系

Dept. / Grad. Inst.: Computer Science

學號 Student ID: 113062544

研究生 Author： 賴政嘉 Cheng-Chia Lai

指導教授 Advisor： 徐正炘 Cheng-Hsin Hsu

中 華 民 國 一一五 年 六 月

June 2026

國立清華大學學位論文授權書

Authorized Agreement for Thesis/Dissertation

(裝訂於紙本論文書名頁之次頁 For author to bind after the title page)

● 立書人（即論文作者）Author:　賴政嘉　（以下稱本人 hereinafter referred to as "I ")

● 授權標的 Authorized subject：本人於　國立清華大學　（以下稱學校）電機資訊學院　資訊工程學系（研究所、學位學程）114學年度　第2學期之 ■碩士 □博士 學位論文
I hereby authorize that my ■ Master’s □ Doctoral thesis submitted in the 2 semester of the 114 academic year at National Tsing Hua University (hereinafter referred to as "the University"), College of Electrical Engineering and Computer Science College, Computer Science Department / Graduate Institute / Degree Program,
論文題目 Title：DOME：面向人工智慧代理的動態 MCP 邊緣協同調度
指導教授 Advisor：徐正炘

（以下稱**本著作**，本著作並包含論文全部、摘要、目錄、圖檔、影音以及相關書面報告、技術報告或專業實務報告等，以下同
Hereinafter referred to as "the publication", which contains all thesis/dissertation, abstracts, catalogues, graphic documents, audiovisual reports, technical reports or professional practice reports, etc.）

緣依據**學位授予法**等相關法令，對於本著作及其電子檔，學校圖書館得依法進行保存等利用，而國家圖書館則得依法進行保存、以紙本或讀取設備於館內提供公眾閱覽等利用。此外，為促進學術研究及傳播，本人在此並進一步同意授權(1)**國立清華大學**與(2)**國家圖書館**對本著作進行以下各點所定之利用：

In accordance with the Degree Conferral Act and other relevant laws and regulations, for this publication and its electronic file, the school library can be preserved and used according to the law. The National Central Library must preserve it in accordance with the law and permit public access in the library with paper or reading equipment. In addition, in order to promote academic research and scholarly communication, I hereby agree to authorize (1)**National Tsing Hua University** and the (2)**National Central Library** to use this publication for the following purposes

一、授權部分：

本人**同意**授權國立清華大學與國家圖書館，無償、不限期間與次數重製本著作並得為教育、科學及研究等**非營利用途之利用**，其包括得將本著作之電子檔收錄於數位資料庫，並透過自有或委託代管之伺服器、網路系統或網際網路向位於全球之使用者(包含但不限於國立清華大學校園內、校園外及國家圖書館館內、館外)公開傳輸，以供使用者為非營利目的之檢索、閱覽、下載及/或列印。

I hereby agree to authorize National Tsing Hua University and the National Central Library to reproduce this publication free of charge, without limitation on time or the number of reproductions, and for education, science, research, and other non-profit use. This involves recording the electronic file of this publication into a digital database and the public transmission of them via self-owned or entrusted servers, network systems or the Internet to the user for the purpose of search, reading, downloading or printing for non-profit purposes.

二、本授權書第一點所定授權，均為非專屬且非獨家授權之約定，本人仍得自行或授權任何第三人利用本著作。

The authorized agreement is non-exclusive permission. I retain the right to use the publication myself or to authorize any third party to use it.

三、本授權書第一點所定授權對象，依各該點授權利用本著作時，均應尊重本人著作人格權及權利管理電子資訊等相關權利，不得以任何方式省略、增修或變更本人署名、本著作名稱、本著作內容及相關資料（包括本人原記載取得學位論文之學校全銜、書目等詮釋資料等）。

The authorized parties specified in this authorized agreement, when using this publication, shall respect my moral rights and the rights related to the management of electronic information. They shall not omit, alter, or modify my authorship, the title of the publication, the content of the publication, or related information (including the full name of the school where I obtained my degree, bibliographic details, and other interpretative materials) in any way.

四、依本授權書第一點將本著作及其電子檔對外公開之時間 Public date of the publication：

□ 於完成審核後立即對外公開 Public immediately after completion of the review

■ 於完成審核之六個月後對外公開 Public six months after completion of the review

□ 本人要求本著作應自 Public access after 民國　　年　　月　　日起始得對外公開，故因本授權書第一點所定授權亦應自該日起始生效力。

五、本授權書第一點所定各該授權對象，均應各自遵守其授權範圍及相關約定。如有違反，由該違反之行為人自行承擔一切法律責任
The authorized party of this authorized agreement shall comply with the scope of authorization and relevant agreements. In case of any violation, the person responsible for the violation shall bear all legal responsibilities."

六、本人擔保本著作為本人創作而無侵害他人著作權或其他權利。如有違反，本人願意自行承擔一切法律責任
I guarantee that this publication was created by me without infringing copyright or other rights of others. In case of any violation, I am willing to assume full legal responsibility.

七、個資利用同意條款 Consent for the Use of Personal Data：
本人同意，學校及國家圖書館為本授權書所定各授權事項目的範圍內得蒐集、處理及利用本人所提供之個人資料，學校並可將該等個人資料提供給包括國家圖書館在內之相關第三人在同一目的範圍內處理及利用
I agree that National Tsing Hua University and the National Central Library may collect, process, and use the personal data I provide within the scope of the authorized matters specified in this authorized agreement. National Tsing Hua University may also provide such personal data to relevant third parties, including the National Central Library, for processing and use within the same purpose scope.

立授權書人 Signature of the author： 賴政嘉

(Date in ROC :Year/MM/DD) 2026 / 6 / 24

國立清華大學碩士學位論文
指導教授推薦書

National Tsing Hua University Master Thesis
Advisor Approval Form

資訊工程學系
賴政嘉 君（學號113062544）所提之論文

DOME：面向人工智慧代理的動態 MCP 邊緣協同調度

經由本人指導撰述，同意提付審查。

Department of Computer Science
Mr.Lai, Cheng-Chia（Student ID: 113062544）who has submitted the Thesis/Dissertation

DOME: Dynamic On-Premise MCP Edge Orchestration for AI Agents

under my guidance, I approve for the submission to the oral defense committee.

☑論文題目與內容符合本系(所、班、學位學程)專業領域
The subject and content of the thesis/dissertation are conformed to the professional field of the department (institute, class or program).

☑符合本系(所、班、學位學程)論文相似度比對標準
The thesis/dissertation meets the standard for the thesis/dissertation originality check set by the department (institute, class or program).

指導教授 (Advisor) 賴明昕 (簽章) (Signature)

中華民國 115 年 5 月 21 日
2026 / / (YYYY/MM/DD)

國立清華大學碩士學位論文
考試委員審定書

National Tsing Hua University
Final Thesis/Dissertation Review Form

資訊工程學系
賴政嘉 君（學號113062544）所提之論文

DOME：面向人工智慧代理的動態 MCP 邊緣協同調度

經本委員會審查，符合碩士資格標準。

Department of Computer Science
Mr.Lai, Cheng-Chia（Student ID: 113062544）who has submitted the Thesis/Dissertation

DOME: Dynamic On-Premise MCP Edge Orchestration for AI Agents

has passed the oral defense and has met the qualifications to be awarded the degree of Master of Science.

學位考試委員會 (Oral defense Committee)

主持人 (Chair) 黃敬群 (簽章)(Signature)

委　員 (Members)

中華民國 115 年 6 月 18 日
2026 / / (YYYY/MM/DD)

摘要

隨著大型語言模型（LLM）與人工智慧代理（AI agent）的快速發展，透過模型上下文協定（MCP）動態呼叫外部工具日益普及。相較於雲端部署，將MCP工具部署於本地端邊緣伺服器能同時兼顧低延遲與資料隱私。然而，現有的協同調度框架，如基於容器的微服務或函式即服務（FaaS），對於MCP工具細粒度且短生命週期的執行特性而言仍過於笨重。此外，在多變且不可預測的人工智慧代理工作負載下，如何同時協調高頻率工具呼叫路由與工具實例部署，也成爲邊緣環境中的重要挑戰。在本論文中，我們提出一個專爲MCP設計的邊緣協同調度框架DOME，用於動態協調人工智慧代理的工具路由與擴展管理。DOME包含三項核心設計：(i) 一種專爲MCP工具設計的輕量級程序層級隔離機制；(ii) 軟性最短延遲（SSD）路由演算法，基於細粒度的網路與處理延遲估測，以機率式方式分派工具呼叫；以及 (iii) 貪婪擴展（GS）部署演算法，在資源限制下週期性調整工具實例配置。我們在眞實異質邊緣測試平台上實作DOME，平台涵蓋工作站等級伺服器至資源受限的單板電腦（SBC），包括NVIDIA Jetson與Raspberry Pi，並使用公開資料集中的人工智慧代理工作流程進行大量實驗。實驗結果顯示，與基線演算法相比，SSD將P50與P99延遲分別降低至多73%與82.4%，並完成多達4.10倍的工具呼叫；GS則在維持CPU使用率低於預設閾值的同時，將請求完成數量提升18.7%，共同實現了可擴展且具隱私保護能力的本地端邊緣協同調度。最終，DOME爲人工智慧代理驅動的邊緣運算開創了新的系統設計典範，爲邊緣原生工具共享、預測式資源配置與去中心化協同調度奠定基礎。

關鍵字：邊緣運算、人工智慧代理、模型上下文協定、分散式協同調度、負載平衡、動態擴展、資源配置

Abstract

The rapid advancement of Large Language Models (LLMs) has popularized AI agents that rely on the Model Context Protocol (MCP) to dynamically interact with external tools. While hosting these MCP tools on on-premise edge servers offers optimal trade-offs between responsiveness and data privacy, existing orchestration frameworks, such as container-based microservices or Function-as-a-Service (FaaS), are too heavyweight for the fine-grained and short-lived nature of MCP executions. Furthermore, orchestrating distributed tools requires tackling the complex interplay between high-frequency tool call routing and periodic tool instance deployment under unpredictable agent workloads. In this thesis, we propose DOME, the first MCP-native edge orchestration framework that dynamically coordinates MCP tool routing and scaling for AI agents. DOME incorporates: (i) a lightweight process-level isolation mechanism tailored for MCP tools, (ii) a Soft Shortest Delay (SSD) routing algorithm that probabilistically dispatches tool calls based on fine-grained network and processing delay estimations, and (iii) a Greedy Scaling (GS) deployment algorithm that periodically adjusts tool instance placement under resource constraints. We implement DOME on a real-world heterogeneous edge testbed comprising devices from a workstation-class root server to resource-constrained Single-Board Computers (SBCs) like NVIDIA Jetson and Raspberry Pis, and conduct extensive evaluations using agent-driven workflows from a public dataset. Experiments demonstrate that SSD reduces P50 and P99 delays by up to 73% and 82.4% while completing 4.10× more tool calls than baselines, and GS improves request completion by 18.7% while maintaining CPU utilization below the predefined threshold, collectively enabling scalable, privacy-preserving on-premise edge orchestration for AI agents. Ultimately, DOME pioneers a new paradigm for agent-driven edge computing, paving the way for edge-native tool sharing, predictive resource provisioning, and decentralized orchestration.

Keywords: Edge Computing, AI Agents, Model Context Protocol (MCP), Distributed Orchestration, Load Balancing, Dynamic Scaling, Resource Provisioning

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisor, Professor Cheng-Hsin Hsu, for the invaluable guidance, encouragement, and continuous support throughout my research. His insightful advice and academic expertise greatly influenced the completion of this work. I would also like to extend my appreciation to Tun-Yuan Chang for the helpful discussions, suggestions, and support during this research. Her contributions and encouragement were truly valuable to my academic journey. Finally, I would like to thank all my labmates for their assistance, discussions, and companionship throughout my two years of master's degree, which made this research journey both productive and enjoyable.

Contents

Chapter 1 Introduction

Recent advances in Large Language Models (LLMs) have accelerated the development of AI agents capable of understanding user intents, reasoning over complex tasks, and interacting with external tools [1]. Rather than relying solely on static pre-trained knowledge, modern AI agents increasingly invoke external tools to access real-time information, perform structured computations, and interact with various services [2], [3], [4], [5], [6]. To support standardized interactions between agents and tools, several communication protocols have recently emerged [7], [8], [9], [10]. Among them, the Model Context Protocol (MCP) [7] has become one of the most widely adopted solutions for tool invocation and context exchange [11]. Meanwhile, with the widespread availability of Internet connectivity and the rapid development of heterogeneous computing devices ranging from resource-constrained edge devices to high-performance servers, MCP tools can now be deployed across distributed environments, including client devices, on-premise edge servers, and cloud infrastructures, enabling increasingly pervasive and interconnected agentic systems.

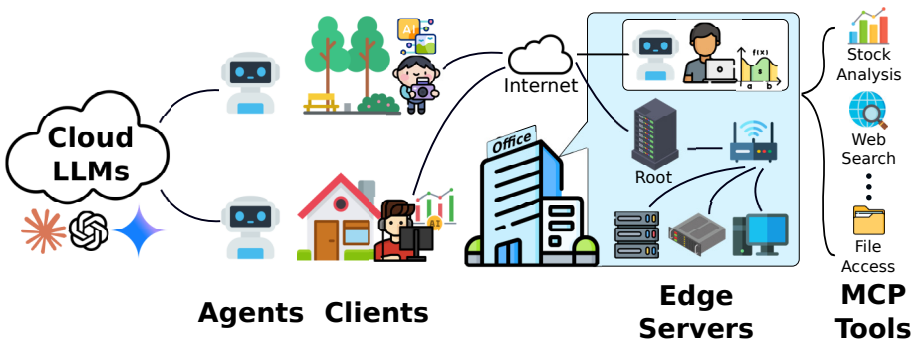

Fig. 1.1: The considered usage scenario for on-premise edge servers supporting multiple AI agents.

Fig. 1.1 illustrates the considered usage scenario in this thesis. Users interact with AI agents running on client devices, while cloud-based LLMs assist the agents in reasoning and generating MCP tool calls according to user requests. Instead of executing all MCP tools locally or forwarding every request to remote cloud servers, MCP tool calls

are offloaded to multiple heterogeneous on-premise edge servers for execution. These edge servers host multiple MCP tool instances, such as web search, file access, and stock analysis tools, enabling AI agents to complete complex tasks through multiple dependent tool calls with: (i) improved availability, (ii) lower execution delay, and (iii) better privacy protection, for sensitive user data. To coordinate the overall system, a root edge server continuously manages two major orchestration operations: *routing* and *deployment*. For routing, the root edge server determines which edge server should execute each incoming MCP tool call according to current runtime conditions. For deployment, the root edge server dynamically initializes or terminates MCP tool instances across edge servers to adapt to evolving workloads and resource conditions. Through such coordinated orchestration, distributed edge servers collectively provide scalable MCP tool execution while avoiding excessive dependence on remote cloud infrastructures.

Solving the abovementioned *routing* and *deployment* problems is quite challenging. For the routing problem, we need to take: (i) heterogeneous edge server capabilities, (ii) time-varying network conditions, (iii) rapidly changing queue dynamics, and (iv) diverse MCP tool call arrival rates into consideration, so as to minimize the total delay of each tool call. For the deployment problem, the system must consider: (i) demand-capability mismatches among edge servers, (ii) evolving workload and tool demand dynamics, and (iii) non-negligible scaling overhead introduced by instance initialization and termination. We notice that the routing problem is (frequently) solved whenever an MCP tool call arrives, while the deployment problem is solved periodically (less frequently). With that said, there exists a complex interplay between these two problems, which in turn further increases the difficulty of optimizing the edge orchestration, calling for our coordinated DOME orchestration framework.

1.1 Contributions

The main contributions of this thesis are summarized as follows:

- We design and implement DOME, a Dynamic On-premise MCP Edge orchestration framework for coordinating MCP tool executions across heterogeneous edge

servers. Unlike existing container-based microservice and Function-as-a-Service (FaaS) frameworks, DOME adopts lightweight process-level isolation tailored for the fine-grained and short-lived characteristics of MCP tools. The framework integrates AI agents, a logically centralized root edge server, and distributed edge servers to support scalable, privacy-preserving, and low-latency MCP tool orchestration in on-premise environments.

- A major challenge in MCP-native edge orchestration is efficiently routing MCP tool calls under heterogeneous computing capabilities, dynamic network conditions, and rapidly changing queue states. We address this challenge by formulating the routing problem as a delay-aware optimization problem and proposing the Soft Shortest Delay (SSD) routing algorithm. SSD leverages real-time execution feedback to probabilistically dispatch MCP tool calls while avoiding persistent load concentration on specific edge servers. Our design enables robust and adaptive routing under time-varying workloads.

- Another challenge in MCP orchestration is dynamically adapting deployment decisions to workload fluctuations while accounting for the overhead of scaling operations. We formulate the deployment problem as a utility-driven optimization problem and develop the Greedy Scaling (GS) deployment algorithm. GS periodically adjusts MCP tool instance scaling actions according to workload dynamics, resource constraints, and scaling overhead. By jointly considering tool congestion and server capacity, GS improves resource utilization and system scalability while maintaining stable service responsiveness.

- We implement the proposed DOME framework on a real heterogeneous edge testbed consisting of workstation-class servers and resource-constrained Single-Board Computers (SBCs), including NVIDIA Jetson and Raspberry Pi devices. Using realistic AI-agent workloads with multi-turn MCP tool-call traces, we conduct extensive experiments to evaluate the effectiveness and scalability of DOME. Experimental results demonstrate that our proposed SSD routing algorithm significantly reduces mean and tail delays while completing substantially more MCP tool calls than state-of-the-art baselines. In addition, the GS deployment algorithm effectively im-

proves resource utilization while maintaining low delay, validating the practicality of DOME for agent-driven edge computing environments.

1.2 Limitation

In this thesis, we make the following assumptions:

- We do not verify the semantic correctness or factual accuracy of MCP tool outputs, and instead consider a tool call successful as long as the MCP tool executes and returns a valid response. This assumption follows practical MCP-agent workflows, where output validation is typically performed by the agent through subsequent reasoning and decision-making processes.

- We only include executable, functionally available MCP tools during the experiments, where deprecated, non-functional, incompatible, or inaccessible tools are filtered out before evaluations. This limitation reflects the current instability and rapid evolution of the MCP ecosystem, where many publicly available MCP tools may not consistently operate in real-world deployments.

- We do not model concurrent execution within a single MCP tool instance and instead assume that tool calls are processed sequentially in isolated process-level execution environments. This simplification avoids additional complexities introduced by OS-level concurrency mechanisms, allowing us to focus on analyzing and modeling the performance characteristics of MCP-based tool execution.

1.3 Organization

The remainder of this thesis is structured as follows: Chapter 2 introduces the background of LLMs, AI agents, MCP, and distributed edge orchestration. Chapter 3 reviews related work on MCP tool routing, deployment, and orchestration optimization. Chapter 4 presents the proposed DOME framework and its overall architecture. Chapter 5 describes the routing problem formulation and the proposed SSD routing algorithm. Chapter 6 presents the deployment problem formulation and the proposed GS deployment algo-

rithm. Chapter 7 describes the implementation details and experimental environment. Chapter 8 evaluates the proposed framework through extensive experiments and performance analysis. Finally, Chapter 9 concludes this thesis and discusses future research directions.

Chapter 2 Background

This chapter introduces the fundamental concepts behind LLM-powered AI agents, including LLM evolution, tool-augmented agent execution, modern agent communication protocols, distributed MCP orchestration across heterogeneous edge environments, and the limitations of existing cloud-native orchestration paradigms for emerging MCP-native AI-agent ecosystems.

2.1 Large Language Models

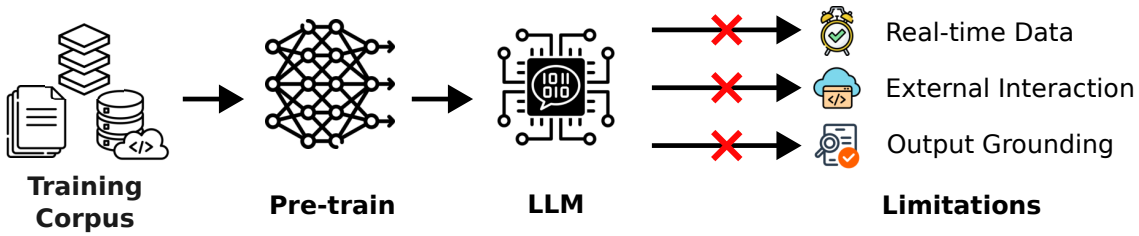

Fig. 2.1: LLM capabilities are bounded by pre-trained parameters.

Recent advances in Large Language Models (LLMs) have significantly improved the capabilities of natural language understanding, reasoning, programming, and content generation [1]. Through large-scale pre-training and instruction tuning, modern LLMs are capable of solving diverse tasks through conversational interactions, making them the core foundation of many emerging AI systems and intelligent services [12]. These developments have accelerated the transition from traditional rule-based systems toward generative AI systems that can flexibly interpret user intents and produce context-aware responses.

The rapid adoption of LLMs has also expanded their role beyond text generation into diverse real-world, domain-specific applications that require complex reasoning and decision support. Sanchez and Hitaj [13] explored the use of LLMs in chemistry-oriented recommendation and estimation tasks. Wang et al. [14] developed an LLM-powered educational system capable of generating personalized learning plans and adaptive student

support. Similarly, Ogdu et al. [15] investigated LLM-assisted clinical decision support systems for healthcare environments. These examples collectively demonstrate that LLMs are evolving into general-purpose reasoning engines capable of supporting increasingly sophisticated applications.

Despite these advances, standalone LLMs exhibit three inherent limitations. As illustrated in Fig. 2.1, pre-training compresses a large training corpus into fixed model parameters, leaving LLMs without access to real-time data, the ability for external interaction, and output grounding [4], [5], [6], [16]. First, since model parameters are fixed after pre-training, LLMs cannot access up-to-date information or perceive dynamically changing system states. Second, without additional execution mechanisms, they are further unable to invoke external programs or interact with live environments. As a result, purely generative reasoning risks producing hallucinated or inconsistent responses when alignment with real-world states is required. These limitations collectively motivate the integration of external tools and execution frameworks.

2.2 Tool Augmentation

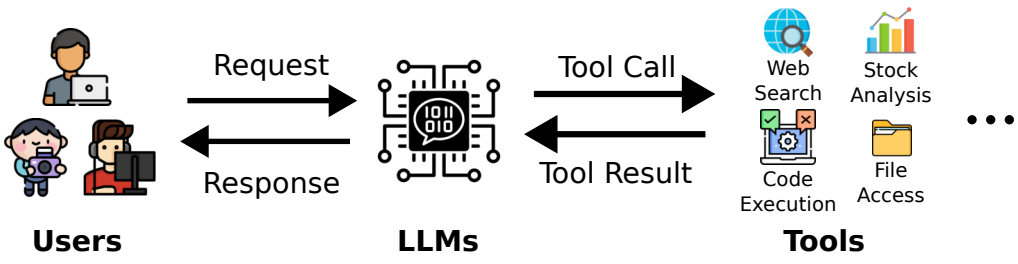

Fig. 2.2: Workflow of tool-augmented LLMs integrating external tools.

To overcome the limitations of standalone LLMs, recent research explores integrating LLMs with external tools and services [17]. As illustrated in Fig. 2.2, tool-augmented LLMs first interpret and reason about user requests, then determine whether external tools are required to complete the request. Instead of relying solely on internal model knowledge, LLMs can invoke external tools such as web search, stock analysis, code execution, and file access to obtain real-time information. After receiving the invocation results from these tools, the LLM further processes the returned information and generates the

final response for the user. This paradigm significantly expands the operational capability of LLMs by enabling them to combine language reasoning with real-world applications.

Recent studies have demonstrated the practicality of tool augmentation with LLMs. Luo et al. [18] proposed a self-training framework that enables LLMs to interact with search and calculation tools without requiring manually annotated demonstrations. Dong et al. [19] developed a reinforcement-learning-based framework for LLMs to dynamically invoke multiple external tools during reasoning. Wölflein et al. [5] introduced a framework that allows LLM systems to automatically transform scientific code repositories into callable tools, enabling LLMs to interact with external tools for domain-specific tasks. These developments transform LLMs from passive text-generation models into interactive systems capable of performing executable actions in external environments.

However, tool-augmented LLMs still face limitations in handling complex user objectives [20]. Most existing systems focus on individual tool invocations, where the LLM generates a single tool call and directly returns the result to users. Such interaction models become insufficient for complicated tasks that require multi-step reasoning, iterative invocations, or coordination across multiple tools. Moreover, tool usage decisions are often tightly coupled with intermediate tool results, requiring dynamic runtime adjustments rather than static one-shot interactions. These limitations motivate the emergence of AI agents.

2.3 AI Agents

Recent AI systems increasingly extend tool-augmented LLMs into AI agents capable of solving complex user tasks [21], [22]. As shown in Fig. 2.3, AI agents integrate *planning*, *execution*, *analysis*, and *decision* into a continuous decision-making loop, where user requests are autonomously decomposed into subtasks and coordinated through iterative tool interactions. This agentic workflow substantially reduces the need for explicit human coordination between tool calls and enables more sophisticated task automation across interconnected systems and services. Recent frameworks such as OpenClaw [23] and NemoClaw [24] further demonstrate the growing trend toward building practical in-

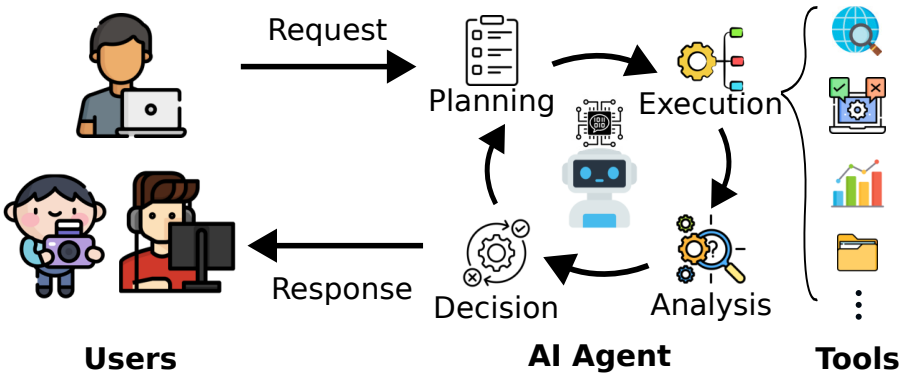

Fig. 2.3: Workflow among users, AI agents, and external tools.

frastructures for large-scale agentic workflows, where multiple interconnected tools are coordinated through iterative reasoning and invocation processes.

Recent studies further demonstrate the applicability of AI agents across diverse domains. For example, Zheng et al. [25] proposed an AI-agent framework that autonomously performs iterative web retrieval, evidence collection, and reasoning to support complex research-oriented tasks. Yang et al. [26] developed a software-engineering AI agent capable of interacting with external code repositories, execution environments, and testing tools to automatically resolve real-world programming issues. Banerjie et al. [27] presented a comprehensive survey on agentic AI in healthcare, highlighting how AI agents coordinate iterative reasoning, external medical tools, and domain-specific knowledge sources to support complex clinical analysis and decision-making workflows. These studies demonstrate how AI agents can combine iterative reasoning and coordinated tool usage to address increasingly sophisticated real-world tasks.

However, increasingly sophisticated AI-agent workflows also introduce substantial orchestration challenges. A single user request may involve multiple dependent tool invocations across heterogeneous execution environments. The responsiveness of external tool invocations directly affects the overall quality of user interactions, as delays accumulate throughout iterative reasoning processes. Meanwhile, tool invocations generated by AI agents are highly dynamic, and workload patterns may fluctuate rapidly in response to user behavior and intermediate tool results. Therefore, efficient coordination among agents, tools, and distributed services is becoming increasingly critical. To support interoperability across such heterogeneous ecosystems, several standardized communication

protocols have recently emerged.

2.4 Agent Communication Protocols

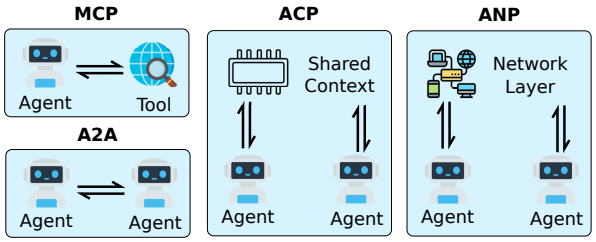

Fig. 2.4: Overview of agent communication protocols.

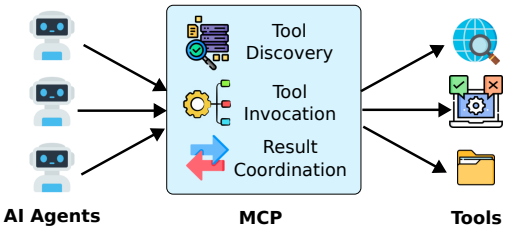

Fig. 2.5: Core functionalities of MCP for agent-tool interactions.

As AI agents increasingly interact with external services and collaborate across distributed environments, several agent communication protocols have recently emerged, including Model Context Protocol (MCP) [7], Agent-to-Agent (A2A) [8], Agent Communication Protocol (ACP) [9], and Agent Network Protocol (ANP) [10]. As illustrated in Fig. 2.4, MCP focuses on agent-tool interoperability, A2A enables direct communication between agents, ACP provides mechanisms for context and workflow coordination, and ANP extends communication across distributed agent networks. These protocols target different interaction scopes, ranging from agent-tool interoperability and inter-agent collaboration to context coordination and distributed agent networking. By standardizing communication interfaces and context exchange mechanisms, these protocols improve interoperability and integration flexibility across heterogeneous AI-agent ecosystems.

Among these protocols, MCP has rapidly emerged as one of the most widely adopted standards for agent-tool interoperability at the time of writing [11], [28], [29], [30]. As illustrated in Fig. 2.5, MCP introduces a standardized interaction layer between AI agents and external tools through three key functionalities: (i) *tool discovery*, which enables agents to identify available tools and their capabilities, (ii) *tool invocation*, which allows agents to execute tool functions through standardized interfaces, and (iii) *result coordination*, which returns execution results to agents for subsequent reasoning and action generation. Recent studies further demonstrate MCP adoption across diverse application domains. For example, Shehab [31] introduced a healthcare framework that leverages AI

agents to coordinate multilingual patient interaction, medical reasoning, and clinical service integration through MCP. Bhandari [32] applied MCP to financial markets through a multi-agent communication framework that models systemic risk and information flow among heterogeneous trading agents. Kumar et al. [33] introduced an MCP-driven infrastructure management system that applies policy-aware guardrails for enterprise automation, tool validation, and secure operational orchestration.

Although these communication protocols improve interoperability and integration flexibility, they primarily focus on standardizing interactions between agents and tools rather than managing how tools are executed across distributed infrastructures. As MCP ecosystems continue to expand, AI-agent systems increasingly rely on dynamically orchestrating large numbers of distributed tool invocations across heterogeneous environments. This trend introduces new operational challenges involving deployment coordination, runtime scheduling, resource management, and scalability, particularly when tools are deployed across edge servers.

2.5 Existing Orchestration Paradigms

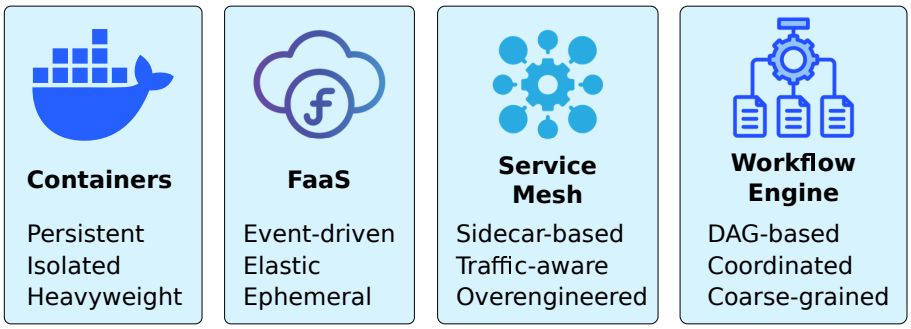

Fig. 2.6: Existing orchestration paradigms compared with MCP tools.

To support large-scale distributed execution and resource management, modern computing systems commonly employ four orchestration paradigms. As illustrated in Fig. 2.6, each orchestration paradigm exhibits distinct design characteristics: (i) *containerized microservices*, which emphasize persistent, isolated, and heavyweight service deployments; (ii) *Function-as-a-Service (FaaS)*, which provides event-driven, elastic, and ephemeral function execution, (iii) *service mesh*, which relies on sidecar-based and traffic-aware con-

trol mechanisms, often at the cost of additional operational complexity, and (iv) *workflow engines*, which coordinate coarse-grained tasks through DAG-based execution graphs. Together, these orchestration mechanisms have become the foundation of modern cloud-native computing systems.

Existing studies have demonstrated the effectiveness of these orchestration paradigms across cloud and edge environments. For example, Giannakopoulos et al. [34] investigated performance-aware scheduling and load balancing strategies for Kubernetes-based edge/cloud systems to improve response stability and workload distribution. Yerra and Middae [35] explored dynamic workload distribution for serverless functions to mitigate latency degradation under fluctuating demand. Calavaro et al. [36] examined serverless function execution across heterogeneous servers, highlighting that elastic function provisioning enables efficient resource management and adaptive execution for latency-sensitive edge applications. Li et al. [37] demonstrated adaptive microservices provisioning in heterogeneous IoT settings, further highlighting the mismatch between conventional orchestration mechanisms and the fine-grained execution demands of emerging edge workloads.

Despite their success in conventional distributed systems, directly applying these paradigms to MCP-native AI-agent ecosystems remains challenging. MCP tool calls exhibit a workload profile that differs fundamentally from the assumptions underlying each of these paradigms. Specifically, the MCP tool calls are (i) *lightweight*, involving small payloads and minimal per-call computation, (ii) *fine-grained*, with each tool encapsulating a single narrowly scoped capability rather than aggregating multiple responsibilities, (iii) *bursty*, as a single agent reasoning step may trigger dense, concurrent sequences of tool calls, and (iv) *iterative*, with agents repeatedly invoking tools across multiple rounds to progressively refine task outcomes. Rather than representing incremental variants of existing cloud-native workloads, MCP-native agent systems introduce a fundamentally different execution model. Containers introduce substantial deployment overhead due to their heavyweight runtime environments. Using Microsoft's Playwright MCP server [38] as an example, our measurements show that container-based deployment requires 936 ms to become operational under warm-start conditions, compared to 535 ms for native process-

based deployment. Under cold-start conditions, startup latency further increases to 62.0 s for containers, whereas process-based deployment remains around 0.5 s. FaaS platforms assume relatively self-contained function executions and provide limited support for maintaining cross-invocation execution states. Service meshes assume stable service topologies and focus primarily on traffic management among long-lived services, while workflow engines assume predefined execution graphs with explicitly specified task dependencies. Consequently, these paradigms address different optimization objectives and are not directly comparable to MCP-specific orchestration mechanisms. Instead, MCP-native environments require orchestration designs centered on tool-level routing, dynamic sharing, and fine-grained resource adaptation.

2.6 Distributed MCP Tool Execution

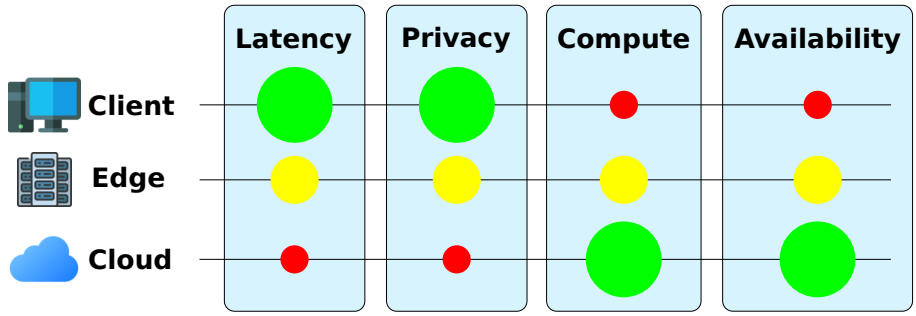

Fig. 2.7: Deployment tradeoffs for MCP tools across client, edge, and cloud.

Effective orchestration in MCP-native AI-agent systems requires efficient orchestration capable of managing MCP tools across heterogeneous servers. Unlike traditional monolithic applications, MCP ecosystems consist of loosely coupled tools that can be independently deployed across client devices, edge servers, and cloud infrastructures. In practice, MCP tools may run on personal computers and mobile devices, on-premise edge platforms such as NAS systems and smart-home hubs, or cloud platforms operated by providers such as Amazon, Microsoft, and Google. As shown in Fig. 2.7, each deployment location introduces different tradeoffs. Client-side execution benefits from low latency and strong data privacy through local execution, but suffers from limited compute capacity and low tool availability as tools remain confined to individual devices.

Cloud execution offers strong compute resources and high availability through globally distributed infrastructure, but incurs additional network latency and introduces privacy risks due to remote data transmission. Compared to these approaches, deploying MCP tools on edge servers provides a balanced solution for MCP-native AI-agent systems.

Prior works have examined edge infrastructure planning across diverse deployment contexts—including smart city Digital Twins [39], urban IoT environments [40], underground wireless networks [41], [42], and large-scale Digital Twin platforms [43], [44]—demonstrating the practical heterogeneity of edge deployments that demands adaptive runtime orchestration capable of handling diverse and dynamic network behaviors. Specifically, distributed edge environments exhibit heterogeneous processing capabilities, resource availability, and network conditions, making runtime coordination substantially more challenging. However, MCP workloads are inherently dynamic because AI agents may continuously generate iterative, unpredictable tool-invocation patterns during reasoning processes. Existing MCP research primarily focuses on connectivity, interoperability, and application integration, while runtime orchestration, adaptive scheduling, and resource management for large-scale MCP-native systems remain open problems. These challenges motivate the need to examine whether existing distributed-system orchestration paradigms can effectively support emerging MCP-native AI-agent ecosystems.

Chapter 3 Related Work

This chapter reviews prior studies on MCP tool routing, MCP tool deployment, and distributed orchestration across edge environments, followed by a discussion of their limitations in supporting dynamic MCP-native orchestration for emerging AI-agent workloads.

3.1 MCP Tool Routing

Yang et al. [45] utilized edge servers to bridge AI agents with MCP tools running on IoT devices, coping with the issues of hardware heterogeneity and control complexity. Ahmadi et al. [46] implemented a proxy for the MCP tool that enables resource-constrained mobile and edge devices to access remote resources through MCP. Barros [47] extended MCP for persistent sessions between AI agents and MCP tools, allowing agents to adapt to network conditions and subscribe to real-time state changes. Li et al. [48] built an experimental platform to emulate various network conditions, including delay fluctuations and network disconnections. They routed MCP tool calls by jointly considering the semantic relevance and real-time network conditions. Zeng and Du [49] developed a framework to coordinate distributed agents across wireless networks through MCP.

3.2 MCP Tool Deployment

Lumer et al. [50] introduced a synchronization mechanism that maintains a storage index by automatically reflecting updates from the original MCP tool developers. By doing so, they reduced the maintenance burdens and avoided version conflicts. Qiu et al. [51] developed an agent architecture to dynamically generate and package MCP tools from open-source projects. Their solution mitigated the inherited scalability and adaptability issues of existing MCP tools. Narajala et al. [52] established a registry system to manage MCP tool discovery using administrative oversight and dynamic trust assessments. Their system mitigated security threats, such as unauthorized MCP tool registration and decep-

tive representation among multiple agents. Guo et al. [53] proposed a dynamic MCP tool generator for AI agents to monitor animal behaviors and surrounding environments. They implemented their solution using a real-world zoo dataset, demonstrating the feasibility of generating MCP tools on demand from sensor data.

3.3 Joint Routing and Deployment Optimization

While not tailored for MCP tools, orchestration of executions among multiple edge servers has been done in the literature. For example, Ullah et al. [54] and Pashaeehir et al. [55] proposed orchestration systems for managing executions in an edge-cloud environment. Their systems coordinated heterogeneous edge resources through decentralized or clustered control planes, enabling dynamic execution placement and migration under changing workloads. Song et al. [56] studied adaptive scheduling mechanisms for allocating edge-cloud resources under time-varying execution demands and network conditions. Their work focused on dynamically adjusting scheduling decisions to reduce and stabilize delay in distributed environments. Liu et al. [57] formulated the resource scheduling problem of an edge network as a multi-objective optimization problem considering network delay, energy consumption, and resource utilization. Their study evaluated the trade-offs among competing objectives under heterogeneous edge resources, providing insights into their implications on scheduling outcomes. Giannakopoulos et al. [58] and Michaelis et al. [59] proposed delay-aware load balancing systems for executions in distributed and multi-cluster environments. Their systems utilized lightweight round-trip time (RTT) predictors and Exponentially Weighted Moving Average (EWMA) to dynamically dispatch executions. Peng et al. [60] and Wang et al. [61] also investigated the joint optimization mechanisms for workload routing and execution deployment in edge environments using Reinforcement Learning (RL). By combining classic queuing models with adaptive agents, their work coordinated routing decisions and instance placement to minimize service delay.

3.4 Limitation of Existing Approaches

Existing studies on MCP tool systems and distributed orchestration still exhibit several important limitations. First, prior works on MCP tool routing frameworks [45], [46], [47], [48], [49] primarily focused on enabling MCP connectivity, remote accessibility, or network-aware routing under varying execution contexts. However, these works assumed statically deployed MCP tool instances and did not address dynamic orchestration or adaptive scaling across heterogeneous edge servers.

Second, research on MCP tool deployment [50], [51], [52], [53] mainly focused on MCP tool generation, synchronization, registry management, and deployment automation. While these approaches improved scalability, maintainability, and security of MCP ecosystems, they did not consider the efficiency of runtime MCP tool calls and the interaction between deployment decisions and execution performance.

Third, existing orchestration studies in edge-cloud environments [54], [55], [56], [57], [58], [59], [60], [61] investigated workload routing, scheduling, load balancing, and joint deployment optimization under distributed workloads. However, these systems were not designed for MCP-native AI-agent environments. Unlike conventional service requests, MCP tool calls generated by AI agents are highly dynamic, non-stationary, and interdependent due to intermediate reasoning processes and multi-step tool invocation chains [62]. Such characteristics introduce unique orchestration challenges that are not sufficiently captured by existing edge orchestration frameworks.

To the best of our knowledge, the current work is the first to investigate the joint orchestration of MCP tool routing and deployment for on-premise edge environments serving AI agents.

Chapter 4 Dynamic On-Premise MCP Edge (DOME) Framework

This chapter presents the detailed design of our proposed DOME framework for orchestrating MCP tool executions across heterogeneous edge environments. We first introduce the overall system architecture and the interactions among agents, the root edge server, and distributed edge servers. We then describe the internal components and workflows of each entity, highlighting how DOME coordinates MCP tool routing, deployment, and execution to support scalable and adaptive AI-agent orchestration.

4.1 Overview

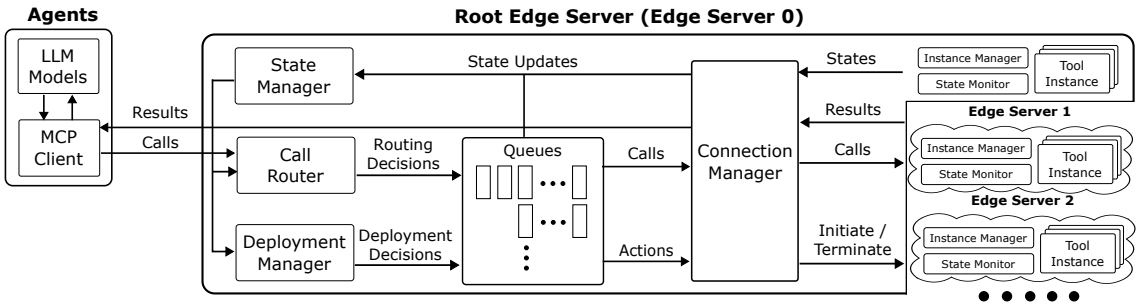

Fig. 4.1: Detailed architecture of the proposed DOME framework.

Fig. 4.1 illustrates the overall architecture of DOME, which is designed to orchestrate MCP tool executions across heterogeneous on-premise edge environments for AI agents. The framework consists of three primary entities: (i) *agents*, which receive user requests and generate MCP tool calls with assistance from LLMs, (ii) a *root edge server*, which globally orchestrates multiple edge servers, and (iii) *edge servers*, which host MCP tool instances and execute assigned MCP tool calls.

Within DOME, user requests are first processed by agents and translated into one or multiple MCP tool calls. These tool calls are then submitted to the root edge server, which determines suitable MCP tool instances running on itself or on other edge servers

based on current runtime conditions. The selected tool instance subsequently executes the requested tool call and returns the execution result to the agent. Based on the returned result, the agent may further generate additional MCP tool calls until the overall user request is completed, forming an iterative reasoning-and-execution workflow.

This centralized orchestration architecture enables DOME to maintain a global view of distributed runtime conditions across heterogeneous edge servers. By continuously collecting system states, the root edge server can make fine-grained *routing* decisions for individual MCP tool calls according to current queue dynamics, service capabilities, and resource availability. Furthermore, the maintained runtime statistics also support periodic *deployment* adaptation, dynamically initializing or terminating tool instances based on evolving workload conditions.

4.2 Agents

Agents serve as the entry point between users and the DOME framework. They are responsible for interpreting user requests and generating MCP tool calls that can be executed within the distributed edge environment. Each agent comprises two major components: *LLM models* and *MCP clients*.

4.2.1 Components

LLM Models. LLM models are responsible for resolving user intent and determining whether external MCP tools are required to complete a request. To bridge the capability gap between static pre-trained knowledge and dynamic external tools, LLM models maintain available MCP tools through tool descriptions or registries, including their functionalities, invocation formats, and input parameter requirements. Based on the user request and intermediate reasoning context, the LLM model determines whether a response can be directly generated or whether additional MCP tool executions are needed. When external capabilities are required, the LLM model performs tool discovery, selects suitable MCP tools, and transforms into MCP tool calls containing tool types and execution arguments, enabling deterministic MCP tool executions within DOME This design

enables LLMs to dynamically coordinate the use of external tools while preserving their reasoning capabilities.

MCP Clients. MCP clients provide the communication interface between agents and the DOME framework. They abstract the underlying protocol interactions required for submitting MCP tool calls and receiving execution results from distributed edge servers. By encapsulating request formatting, session management, and result handling, MCP clients allow LLM models to interact with heterogeneous distributed MCP tools through a unified and consistent interface. This abstraction decouples agent-side reasoning from low-level distributed system operations, improving modularity and extensibility within DOME.

4.2.2 Workflow

The interaction between LLM models and MCP clients forms a continuous reasoning-and-execution loop. When a user submits a request, the LLM model first analyzes the request and determines whether MCP tools are needed. If external tools are required, MCP tool calls are generated and submitted through the MCP client to the root edge server. After execution, the returned results are delivered back to the LLM model, which may further generate additional MCP tool calls according to intermediate outputs until the overall user request is completed. This iterative process continues until the overall user request is completed. By separating reasoning and execution into different modules, agents can dynamically adapt their MCP tool usage according to intermediate execution results and evolving task requirements.

4.3 Root Edge Server

The root edge server serves as the global orchestration entity in DOME, which is responsible for three major functionalities: (i) maintain system-wide runtime states collected from distributed edge servers, (ii) route incoming MCP tool calls to a suitable execution queue according to system conditions, and (iii) adjust MCP tool instance deployments according to workload conditions and resource availability. To support adap-

tive orchestration under heterogeneous edge environments, the root edge server consists of the following components: (i) *state manager*, (ii) *call router*, (iii) *deployment manager*, (iv) *queues*, and (v) *connection manager*. Together, these modules enable the root edge server to continuously coordinate routing and deployment across distributed edge servers under dynamic runtime conditions.

4.3.1 Components

State Manager. The state manager maintains runtime statistics collected from distributed edge servers, including queue conditions, service rates, active tool instances, network conditions, and resource utilization. It employs a look-up table mapped to each edge server and continuously updates its corresponding runtime statistics through both periodic measurements and event-driven updates. For example, networking statistics and resource utilization are periodically measured over time intervals, while execution-related statistics, such as service rates and queue states, are updated according to execution events generated by MCP tool calls and tool instances. This design allows the root edge server to maintain a consistent and continuously updated global view of the distributed system while reducing stale runtime information caused by rapidly changing execution conditions. These continuously updated system states provide the foundation for routing and deployment decisions, enabling the root edge server to adapt its strategies according to evolving workload conditions and execution behaviors.

Call Router. The call router determines the execution queue for each incoming MCP tool call by selecting a suitable execution queue among candidate edge servers hosting the required MCP tool type. It estimates the expected total delay of candidate queues using runtime statistics, including networking conditions, queue states, service rates, and active tool instances. More specifically, the routing process jointly considers networking and processing delay to approximate the expected end-to-end delay experienced by MCP tool calls. This design allows the root edge server to make fine-grained routing decisions based on current system conditions, rather than relying solely on static assignment policies or queue occupancy. As a result, MCP tool calls can be dynamically distributed across heterogeneous edge servers, reducing excessive execution delays and avoiding persistent

workload concentration on specific servers.

Deployment Manager. The deployment manager controls the placement and scaling of MCP tool instances across distributed edge servers. It periodically evaluates runtime statistics and determines whether MCP tool instances should be initialized or terminated according to current workload conditions, service capacities, and resource utilization, without requiring predefined tool-to-server placement policies. More specifically, scaling up actions increases service capacity for heavily used MCP tools, while scaling down actions reduces unnecessary resource consumption from underutilized tool instances. This design allows the root edge server to continuously adjust service capacities according to changing workload demands while coordinating resource allocation across heterogeneous edge servers. These deployment decisions complement routing decisions by adapting available service capacity over time, enabling DOME to maintain stable execution behaviors under dynamic workloads.

Queues. Queues temporarily buffer incoming MCP tool calls before execution according to their assigned edge servers and MCP tool types. Each queue is associated with one or multiple tool instances serving the same MCP tool type on an edge server, where MCP tool calls are processed in a First-In-First-Out (FIFO) manner. More specifically, DOME supports two queueing modes: (i) *dedicated*, where each tool instance maintains an independent queue, and (ii) *shared*, where multiple tool instances of the same MCP tool type hosted on the same edge server share a common queue. This design allows DOME to absorb temporary mismatches between incoming workload rates and available service capacities while supporting different workload distribution behaviors across tool instances. The resulting queue states also provide observable runtime signals for orchestration decisions, enabling the root edge server to estimate queue waiting conditions and continuously adapt routing and deployment behaviors.

Connection Manager. The connection manager handles communications between the root edge server and distributed edge servers. It is responsible for transmitting control messages, synchronizing runtime statistics, dispatching MCP tool calls, and collecting returned execution results throughout the DOME framework. More specifically, the connection manager maintains communication channels required for different orchestration

operations, including runtime monitoring, routing coordination, deployment control, and distributed MCP tool execution. This design separates orchestration logic from low-level communication operations while providing a unified communication abstraction across heterogeneous edge servers. As a result, the root edge server can continuously coordinate distributed monitoring and execution across the entire system using synchronized runtime information.

4.3.2 Workflow

When an MCP tool call arrives from an agent, the call router first retrieves runtime states maintained by the state manager and estimates the expected total delay of candidate edge servers hosting the required MCP tool type. The selected execution queue then buffers the MCP tool call until an available tool instance begins execution. While MCP tool calls are continuously processed across edge servers, the state manager simultaneously updates runtime statistics collected through periodic measurements and execution events. Based on these updated runtime states, the deployment manager periodically evaluates whether current service capacities remain aligned with workload demands and issues corresponding scaling actions when necessary. Throughout this orchestration process, the connection manager continuously exchanges runtime statistics, control messages, execution requests, and returned results between the root and edge servers, enabling the root edge server to continuously coordinate routing, deployment, and execution behaviors across the DOME framework.

4.4 Edge Servers

Edge servers provide the actual execution environment for MCP tool calls in DOME, which is responsible for three major functionalities: (i) executing MCP tool calls through distributed tool instances, (ii) managing the lifecycle and execution behaviors of local tool instances, and (iii) continuously monitoring and reporting local runtime states to the root edge server. To support these functionalities, each edge server consists of: (i) *tool instances*, (ii) an *instance manager*, and (iii) a *state monitor*. Together, these components enable distributed MCP tool execution while continuously synchronizing local execution

states with the root edge server for global orchestration.

4.4.1 Components

Tool Instances. Tool instances execute incoming MCP tool calls, each associated with a specific MCP tool type on an edge server. Each tool instance processes MCP tool calls sequentially, while multiple instances of the same MCP tool type can run as separate processes on the same edge server to support parallel execution. More specifically, DOME adopts lightweight process-level isolation for tool instances, allowing instances to be independently initialized, terminated, and monitored without relying on heavier virtualization mechanisms such as containers. This design enables edge servers to support fine-grained, controllable execution of MCP tools while reducing scaling overhead and preserving execution isolation across different tool instances. As a result, DOME can efficiently support dynamic MCP workloads across heterogeneous edge devices, including resource-constrained Single-Board Computers (SBCs) such as Raspberry Pi, ODROID N2+, and Rock Pi.

Instance Manager. The instance manager controls the lifecycle and execution coordination of tool instances on each edge server. It is responsible for initializing and terminating tool instances, and for coordinating local execution scheduling and results collection for each tool call. This design centralizes local instance management into a unified coordination module while separating high-level orchestration decisions from low-level operations. As a result, edge servers can support scalable and controllable tool management while remaining synchronized with system-wide orchestration policies issued by the root edge server.

State Monitor. The state monitor continuously collects runtime statistics reflecting local execution behavior on each edge server, including queue states, service rates, active tool instances, execution delays, and resource utilization. It updates these runtime statistics through both periodic measurements and execution-driven events. For example, CPU utilization and memory usage are periodically monitored over time intervals, while execution-related statistics, such as service completion rates and queue occupancy, are updated based on MCP tool execution events generated by tool instances. This de-

sign allows edge servers to maintain continuously updated local runtime visibility while capturing both instantaneous execution conditions and longer-term system trends.

4.4.2 Workflow

When an MCP tool call is dispatched from the root edge server, the instance manager first receives the request and forwards it to the selected tool instance for execution. During execution, the state monitor continuously updates local runtime statistics through periodic measurements and execution-driven events generated by MCP tool calls and tool instances. After execution completes, the instance manager collects the returned result and transmits it back to the root edge server through the connection manager. This workflow enables continuous execution control, runtime monitoring, and result collection throughout MCP tool execution while maintaining synchronized runtime visibility between edge servers and the root edge server. The resulting runtime states are subsequently used by the root edge server to support future routing and deployment decisions across the DOME framework.

Chapter 5 Routing Problem and Algorithm

Table 5.1: Notations for Routing Problem and Algorithm

Symbol	Description
c	Call
q	Tool queue
τ	Tool type
$\mathcal{Q}_\tau$	The set of all queues serving tool τ
L	Call size
$\tilde{D}(c,q)$	Delay modeling of a call c in queue q
$\tilde{D}_n(c,q)$	Network delay modeling of a call c in queue q
$\tilde{D}_p(c,q)$	Processing delay modeling of a call c in queue q
V_q	Backward transmission delay in queue q
B_s	Forward bandwidth to server s
B'_s	Backward bandwdith from server s
l_q	Number of waiting calls in q
N_s	Round-trip propagation delay to server s
k_q	Number of active instances serving queue q
δ_q	Remaining execution time in q
μ_q	Average service rate of queue q
γ	Parameter control between routing decision to total delay

This chapter investigates the routing problem in the DOME framework and presents the proposed routing algorithm. Since MCP tool calls must be dispatched to edge servers with varying computing capabilities and workloads, routing decisions directly affect overall service delay. To minimize this delay, we formulate the routing problem as a delay-aware optimization problem and design an efficient routing algorithm accordingly. Table 5.1 summarizes the notations used in this chapter.

5.1 Problem Formulation

Given an MCP tool call c of tool type τ, the call router selects the most suitable $q^* \in \mathcal{Q}_\tau$, where $\mathcal{Q}_\tau$ denotes the set of all queues among all edge servers capable of serving tool τ. The objective of the routing problem is to minimize the total delay $D(c, q)$ between the call's arrival and the result's return. With these notations, we formulate the routing problem as:

$$q^* = \arg\min_{q \in \mathcal{Q}_\tau} D(c, q). \tag{5.1}$$

Since the actual delay $D(c, q)$ is unknown at decision time, we approximate it using a delay model $\tilde{D}(c, q)$ based on the system states, i.e., $D(c, q) \approx \tilde{D}(c, q)$. However, directly selecting the queue with the minimum estimated delay may lead to persistent load concentration. Therefore, we apply a routing algorithm $f(\cdot)$ to select the optimal queue $q^* = f(\tilde{D}(c, q))$. To do so, $\tilde{D}(c, q)$ must capture the key delay components induced by network and system conditions, as we detail next.

5.2 Delay Model

The total delay can be written as $\tilde{D}(c, q) = \tilde{D}_n(c, q) + \tilde{D}_p(c, q)$, capturing the networking and computing delays, respectively.

5.2.1 Network Delay

It can be further decomposed into three components: (i) forward transmission delay, (ii) reverse transmission delay, and (iii) round-trip propagation delay. Specifically, we write the network delay as:

$$\tilde{D}_n(c, q) = \frac{L}{B_s} + V_q + N_s, \tag{5.2}$$

where the forward transmission delay is given by the fraction between the call size L and the available network bandwidth B_s. The call size L can be measured on-the-fly, while network bandwidth B_s is estimated via periodic bandwidth probing and maintained as a server-level statistic for each server s. In contrast, the reverse transmission delay

V_q cannot be directly calculated, as the result size is unknown to the call router. We therefore approximate it using historical observations and update it upon receiving each result, maintaining a per-queue estimate V_q for each q. The round-trip propagation delay N_s is estimated from periodic Internet Control Message Protocol (ICMP) probing packets and maintained as a server-level statistic for each edge server s.

Since the abovementioned measurements are inherently dynamic and noisy, we apply EWMA to smooth them out. The EWMA updates for a metric x using $x \leftarrow (1-\alpha)x+\alpha\hat{x}$, where $\hat{x}$ denotes the latest measurement, and $\alpha \in (0, 1]$ controls the trade-off between stability and responsiveness. A larger α value increases weights on recent measurements with faster adaptation to changes but potentially introduces higher variance, while a smaller α yields smoother estimates at the cost of slower responsiveness. Per-server measurements, specifically the forward network bandwidth B_s and propagation delay N_s, are updated every T_s using a shared α_s. Furthermore, per-queue measurements, i.e., the reverse transmission delay V_q, are updated with another α_q. If not otherwise specified, we empirically set $T_s = 30\ s$, $\alpha_s = 0.2$, and $\alpha_q = 0.3$, based on some pilot tests on our testbed.

5.2.2 Processing Delay

It can be decomposed into: (i) queueing delay (waiting time) and (ii) service delay (execution time). We adopt stateful online estimations and write the processing delay as:

$$\tilde{D}_p(c, q) = (\frac{l_q}{k_q\mu_q} + \delta_q) + \frac{1}{\mu_q}, \tag{5.3}$$

where the parentheses capture the queueing delay, and the last term represents the service delay. Here, the queueing delay is estimated from the queue length l_q, the number of tool instances k_q, and the average service rate μ_q by approximating the time required to clear the queued calls assuming a constant service rate of $k_q\mu_q$. The term δ_q captures the remaining execution time of ongoing executions, which is approximated as the average remaining time across active tool instances under the assumption of uniform assignments. In particular, l_q and k_q are measured, and μ_q and δ_q are calculated whenever they are needed by the call router.

5.3 Proposed Routing Algorithm: SSD

We propose a parameterized routing algorithm to solve the routing problem, called *Soft Shortest Delay* (*SSD*), which uses a parameter to probabilistically route each tool call without over-concentrating on a single edge server. SSD assigns a routing probability $P(q \mid c)$ to each candidate queue based on the normalized estimated delay $z(c, q)$. More specifically, we first compute the estimated delay $\tilde{D}(c, q)$ for each candidate queue using a look-up table, and then calculate the mean $\bar{D}(c)$ and standard deviation $\hat{D}(c)$ across candidates to obtain $z(c, q) = \big(\tilde{D}(c, q) - \bar{D}(c)\big)/\hat{D}(c)$. Based on the normalized estimated delay, we compute the routing probability using exponential weighting with:

$$P(q \mid c) = \frac{\exp(-\gamma \cdot z(c, q))}{\sum_{j \in \mathcal{Q}_\tau} \exp(-\gamma \cdot z(c, j))}. \tag{5.4}$$

Here, the parameter γ controls the sensitivity of the routing decision to total delay, where a larger γ value biases toward the lowest-delay queue and a smaller γ value leads to more balanced routing across queues. Finally, our SSD algorithm selects q^* for a tool τ using probabilities of individual candidate queues $P(q \mid c), \forall q \in \mathcal{Q}_\tau$.

5.4 Complexity Analysis

Lemma 1 (SSD Time Complexity)**.** *For a tool call of type τ, the time complexity of the SSD routing algorithm is bounded by $O(|K_\tau|)$, where K_τ denotes the set of tool instances supporting tool τ.*

Proof. Upon receiving a tool call, SSD evaluates all candidate queues in $\mathcal{Q}_\tau$. For each queue $q \in \mathcal{Q}_\tau$, both the network delay in Eq. (5.2) and the processing delay in Eq. (5.3) can be computed in constant time using maintained runtime statistics. SSD then performs delay normalization and routing probability computation over all candidate queues with an additional linear pass over $\mathcal{Q}_\tau$. Therefore, the total computational cost per routing decision is $O(|\mathcal{Q}_\tau|)$. Since each queue must be backed by at least one tool instance, we have $|\mathcal{Q}_\tau| \leq |K_\tau|$. Hence, the overall time complexity is bounded by $O(|K_\tau|)$. □

Lemma 2 (SSD Space Complexity)**.** *The space complexity of the SSD routing algorithm is bounded by* $O(|K_\tau|)$.

Proof. SSD maintains three categories of runtime statistics: (i) server-level bandwidth and propagation-delay statistics with complexity $O(|S|)$; (ii) queue-level transmission-delay and queue-length statistics with complexity $O(|\mathcal{Q}_\tau|)$; and (iii) per-instance service statistics with complexity $O(|K_\tau|)$. Since$|S| \leq |K_\tau|$ and $|\mathcal{Q}_\tau| \leq |K_\tau|$, the total memory requirement is bounded by $O(|S| + |\mathcal{Q}_\tau| + |K_\tau|) = O(|K_\tau|)$.

□

Chapter 6 Deployment Problem and Algorithm

Table 6.1: Notations for Deployment Problem and Algorithm

Symbol	Description
W	Time duration of a deployment window
$\mathcal{S}$	Set of all edge servers
$\mathcal{T}$	Set of all tool types
$\mathcal{U}$	All candidate scaling actions
$\mathcal{A}$	Selected scaling actions to be deployed
$u(\tau, s, a)$	A scaling action of tool τ in server s with adjustment a
$\tilde{G}(u)$	Utility increase model of a scaling action u
$\tilde{G}_\tau(u)$	Tool congestion reduction model of a scaling action u
$\tilde{G}_c(u)$	Server capacity increase model of a scaling action u
$\lambda(\tau)$	User demand of tool τ
$\lambda(u)$	User demand after scaling action u
$\Phi(u)$	Scaling action overhead of u
$\mu(\tau)$	Service rate of tool τ
$\mu'(\tau)$	Service rate after scaling action u
$k(\tau, s)$	Number of instances with tool τ in server s
$R(C)$	Saturation risk score of CPU utilization with C
C_s	CPU utilization of server s

This chapter studies the deployment problem in the DOME framework and presents the proposed deployment algorithm. Since workloads of different tool types vary over time, a static deployment strategy may become inefficient as workload conditions change. Moreover, the optimal placement of a tool depends not only on its type but also on current workload distributions and server resource availability. Therefore, rather than relying

on manually predefined tool placements, DOME continuously monitors up-to-date system states and dynamically determines both tool placement and instance provisioning decisions. This enables the deployment manager to adapt to changing demand patterns, balance workloads across edge servers, and improve resource utilization. Table 6.1 summarizes the notations used in this chapter.

6.1 Problem Formulation

Given a user-specified time duration of deployment window W, the deployment manager periodically aggregates system states collected over the disjoint previous deployment window and selects a set of scaling actions $\mathcal{A}$ from candidate actions $\mathcal{U}$ over tool types $\mathcal{T}$ and servers $\mathcal{S}$. We define each scaling action $u \in \mathcal{U}$ as a tuple $u = (\tau, s, a)$ to denote instance adjustment of tool τ on server s by a, where $a \in \{+1, -1\}$ corresponding to: (i) initializing a new tool instance, and (ii) terminating a tool instance. The optimal deployment decision is to select a subset of actions $\mathcal{A} \subseteq \mathcal{U}$ that matches user demands with service rates while balancing resource utilization.

With the above notations, we formulate the deployment problem as:

$$\mathcal{A}^* = \arg\max_{\mathcal{A} \subseteq \mathcal{U}} G(\mathcal{A}), \tag{6.1}$$

where the system utility change after applying action set $\mathcal{A}$ is denoted by $G(\mathcal{A})$, which is inherently challenging because scaling actions often compete for shared resources (like CPU cycles) to model, leading to non-trivial service rate changes in such a complex system. To avoid over-engineering the utility increase model $G(\mathcal{A})$, we add two practical constraints to our formulation:

- *Availability:* At least one active instance for each tool should be maintained across all edge servers:

$$\sum_{s' \in \mathcal{S}} k(\tau, s') + \sum_{u' \in \mathcal{A}} \{u'.a \mid u'.\tau = \tau\} \geq 1 \quad \forall \tau \in \mathcal{T}, \tag{6.2}$$

  where $k(\tau, s')$ denotes the number of instances of tool τ on server s'

- *Stability:* Each tool and each server can participate in at most one scaling action:

$$u.\tau \neq v.\tau \ \wedge \ u.s \neq v.s \quad \forall u, v \in \mathcal{A}, \ u \neq v. \tag{6.3}$$

- *Saturation:* Scaling-up actions should not overload edge servers beyond a predefined CPU utilization threshold:

$$C_{u.s} \leq \theta_c \quad \forall \{u \mid u.a = +1\} \in \mathcal{A}, \tag{6.4}$$

  where C_s denotes the current CPU utilization of server s and θ_c is the predefined CPU utilization threshold. If not otherwise specified, we set $\theta_c = 0.8$ in our experiments.

With the three constraints, the utility increase of an action set $\mathcal{A}$ can be decomposed into the sum of the utility increase $G(u)$ of each $u \in \mathcal{A}$, i.e., $G(\mathcal{A}) \approx \sum_{u' \in \mathcal{A}} G(u')$. However, exhaustively evaluating all feasible action combinations is impractical for online deployment control due to combinatorial complexity. Therefore, the deployment decision is obtained by applying a decision function $g(\cdot)$ over the estimated improvement of candidate scaling actions as

$$\mathcal{A}^* = g(\{\tilde{G}(u)\}_{u \in \mathcal{U}}). \tag{6.5}$$

Here, $G(u)$ is estimated with our developed utility increase model $\tilde{G}(u)$, i.e., $\tilde{G}(u) \approx G(u)$. We give $\tilde{G}(u)$ in the next section, which should capture the key factors affecting system performance under dynamic conditions.

6.2 Utility Increase Model

We model the utility increase of each scaling action u using system statistics in the previous deployment window. Each scaling action affects the system performance in multiple aspects, including service capacity, resource utilization, and action overhead. We break the utility increase into two components: (i) *tool congestion reduction* $\tilde{G}_\tau(u)$,

and (ii) *server capacity increase* $\tilde{G}_c(u)$. The tool congestion reduction term captures how effectively a scaling action improves the balance between tool demand and service capacity, while the server capacity adjustment term reflects whether the target edge server currently has sufficient available resources for additional workload. To jointly consider both tool-level and server-level conditions, we model the utility increase as:

$$\tilde{G}(u) = \tilde{G}_\tau(u)(1 + \tilde{G}_c(u)), \tag{6.6}$$

Under this design, scaling actions that effectively alleviate tool congestion remain favorable, while actions targeting heavily utilized servers receive lower utility gains to reduce excessive resource concentration. We model these two components in detail as follows.

6.2.1 Tool Congestion Reduction Model

It models how a scaling action u improves the balance of the tool $u.\tau$ between: (i) *user demand* changed from $\lambda(u.\tau)$ to $\lambda'(u)$, and (ii) *service rate* changed from $\mu(u.\tau)$ to $\mu'(u)$. Here, $\lambda(u.\tau)$ and $\mu(u.\tau)$ denote the current demand and service rate of tool $u.\tau$, while $\lambda'(u)$ and $\mu'(u)$ represent their estimated values after applying action u. In particular, we define the tool utilization before and after the action as $\rho(u.\tau) = \lambda(u.\tau)/\mu(u.\tau)$ and $\rho'(u) = \lambda'(u)/\mu'(u)$, respectively. We employ a user-specified parameter θ_τ for the tool utilization as the target balanced point, where $\rho(u.\tau) > \theta_\tau$ indicates under-provisioning, and $\rho(u.\tau) < \theta_\tau$ suggests over-provisioning. If not otherwise specified, we set $\theta_\tau = 0.5$ on our testbed.

With the above notations, the reduction of tool utilization deviation from θ_τ with scaling action u is written as $|\rho(u.\tau) - \theta_\tau| - |\rho'(u) - \theta_\tau|$. To aggregate the deviations across all tools, we define a demand-aware weight for each tool as:

$$w_u = \frac{\lambda(u.\tau)}{\sum_{\tau \in \mathcal{T}} \lambda(\tau)} \tag{6.7}$$

and write the tool congestion reduction as:

$$\tilde{G}_\tau(u) = w_u \cdot (|\rho(u.\tau) - \theta_\tau| - |\rho'(u) - \theta_\tau|). \tag{6.8}$$

User Demand Model

The pre-action user demand is estimated as:

$$\lambda(u.\tau) = \frac{N(u.\tau) + l(u.\tau)}{W}, \tag{6.9}$$

where $N(u.\tau)$ denotes the number of newly arrived calls in the previous deployment window, and $l(u.\tau)$ denotes the number of queued calls at the beginning of the window. These states are normalized to the deployment window duration W to capture the user demands.

The post-action demand $\lambda'(u)$ is estimated by adding the scaling action overhead $\Phi(u)$, which can be measured in advance, to the pre-action user demand. We write the post-action user demand as:

$$\lambda'(u) = \lambda(u.\tau) + \frac{\Phi(u)\mu(u.\tau, u.s)}{W}. \tag{6.10}$$

Here, the first term $\lambda(u.\tau)$ is the pre-action user demand. The second term incorporates the additional demand induced by the scaling action overhead. Specifically, the action overhead $\Phi(u)$ represents the service capacity drop. That is, we write the number of additional tool calls as $\Phi(u)\mu(u.\tau, u.s)$, where $\mu(u.\tau, u.s)$ denotes the per-instance service rate of tool $u.\tau$ on server $u.s$.

Service Rate

The pre-action service rate is aggregated across all edge servers into:

$$\mu(u.\tau) = \sum_{s' \in \mathcal{S}} k(u.\tau, s')\mu(u.\tau, s'), \tag{6.11}$$

where $k(u.\tau, s)$ and $\mu(u.\tau, u.s)$ denote the number of tool instances and average service rate per tool instance on server s, respectively. They are measured in the previous deployment window.

The post-action service rate is approximated with the current service rate and in-

stance changes with scaling action u as:

$$\mu'(u) = \mu(u.\tau) + u.a \cdot \mu(u.\tau, u.s), \tag{6.12}$$

where $\mu(u.\tau)$ is the pre-action service rate, $u.a$ denotes the tool instance change due to u, and $\mu(u.\tau, u.s)$ indicates the per-instance service rate of tool $u.\tau$ on the target server $u.s$.

6.2.2 Server Capacity Increase Model

It captures the impact of a scaling action u with respect to resource availability on server $u.s$, where CPU utilization $C_{u.s}$ serves as an inverse indicator of available server capacity. Directly modeling post-action CPU utilization is challenging due to complex interactions between workloads and resources. Instead, we design a saturation risk function $R(C)$ to map CPU utilization into a saturation risk score:

$$R(C) = \frac{1}{1 + e^{-(C-\theta_c)}}. \tag{6.13}$$

Here, we use the sigmoid transformation to model the nonlinear saturation effect of CPU utilization as server load approaches the predefined threshold. Additional workloads typically introduce only limited performance degradation when server utilization remains low, whereas the risks of queue buildup, resource contention, and delay grow significantly faster as utilization approaches the threshold.

Using the saturation risk function, the server capacity term for a scaling action operated on server $u.s$ can be written as:

$$\tilde{G}_c(u) = -u.a \cdot R(C_{u.s}), \tag{6.14}$$

where $u.a \in \{+1, -1\}$ corresponds to scaling-up and scaling-down actions, respectively. For scaling-up actions ($u.a = +1$), servers with lower saturation risk yield higher capacity scores, reflecting their larger remaining server capacity. In contrast, for scaling-down actions ($u.a = -1$), servers with higher saturation risk yield higher capacity scores, reflecting stronger benefits from workload reduction.

Algorithm 1: Greedy Scaling (GS)

Input: Candidate action set $\mathcal{U}$
Output: Selected deployment action set $\mathcal{A}^*$
Initialize: $\mathcal{A}^* \leftarrow \emptyset$, $\mathcal{T}' \leftarrow \emptyset$, $\mathcal{S}' \leftarrow \emptyset$
foreach $u \in \mathcal{U}$ **do**
  Compute $\tilde{G}(u)$
Sort all actions in descending order of $\tilde{G}(u)$ into $\mathcal{U}'$
while $\mathcal{U}' \neq \emptyset$ **do**
  Select and remove the top-ranked action u' from $\mathcal{U}'$
  if $u'.\tau \notin \mathcal{T}' \wedge u'.s \notin \mathcal{S}'$ **then**
    if $C_{u'.s} \geq \theta_c \wedge u'.a = +1$ **then**
      continue
    $\mathcal{A}^* \leftarrow \mathcal{A}^* \cup \{u'\}$ $\mathcal{T}' \leftarrow \mathcal{T}' \cup \{u'.\tau\}$ $\mathcal{S}' \leftarrow \mathcal{S}' \cup \{u'.s\}$
  if $|\mathcal{T}'| = |\mathcal{T}| \vee |\mathcal{S}'| = |\mathcal{S}|$ **then**
    break
return $\mathcal{A}^*$

6.3 Deployment Algorithm: GS

Eqs. (6.1)–(6.4) formulate an optimization problem, whereas exhaustive search over all feasible action sets remains impractical for online deployment control due to the combinatorial number of candidate scaling decisions. However, the stability and availability constraints in Eqs. (6.2)–(6.3) restrict each tool and each server to participate in at most one scaling action during a deployment window, thereby significantly reducing interactions among actions. Motivated by this observation, we design a greedy deployment algorithm, called *Greedy Scaling (GS)*, that ranks candidate scaling actions by their estimated utility increases and iteratively selects non-conflicting actions until no feasible action remains.

Algorithm 1 presents the pseudocode of GS. To cope with the constraints in Eqs. (6.2) and (6.3), GS maintains two sets $\mathcal{S}' = \emptyset, \mathcal{T}' = \emptyset$ to record the servers and tools that have been involved in selected actions. GS first enumerates all candidate actions $u \in \mathcal{U}$, computes their utility increases $\tilde{G}(u)$, and sorts them in descending order of the utility function values into a ranked list $\mathcal{U}'$. We then iteratively select the top-ranked action u' from $\mathcal{U}'$ and remove it from the list. The selected action is added to $\mathcal{A}^*$ if it satisfies:$u'.\tau \notin \mathcal{T}' \wedge u'.s \notin \mathcal{S}'$. If the above condition holds, GS updates:

$$\mathcal{A}^* \leftarrow \mathcal{A}^* \cup \{u'\}, \mathcal{T}' \leftarrow \mathcal{T}' \cup \{u'.\tau\}, \mathcal{S}' \leftarrow \mathcal{S}' \cup \{u'.s\}. \tag{6.15}$$

after adding each action to $\mathcal{A}^*$. The procedure continues until: (i) $|\mathcal{S}'| = |\mathcal{S}|$ (ii) $|\mathcal{T}'| = |\mathcal{T}|$, or (iii) $|\mathcal{U}'| = 0$.

6.4 Complexity Analysis

Lemma 3 (GS Time Complexity)**.** *For each deployment window, the GS deployment algorithm incurs a time complexity of* $O(|\mathcal{T}||\mathcal{S}| \log(|\mathcal{T}||\mathcal{S}|))$.

Proof. GS first enumerates all candidate scaling actions in $\mathcal{U}$ and evaluates the utility increase $\tilde{G}(u)$ for each action $u \in \mathcal{U}$. Since all required system statistics are maintained online, each utility evaluation can be computed in constant time. Therefore, the total cost of utility evaluation is $O(|\mathcal{U}|)$. GS then sorts all candidate actions according to their utility values, which requires $O(|\mathcal{U}| \log |\mathcal{U}|)$. After sorting, GS performs a linear scan over the ranked action list for greedy action selection. Each feasibility check and state update can be performed in constant time using the maintained sets $\mathcal{S}'$ and $\mathcal{T}'$. Thus, the greedy selection phase incurs $O(|\mathcal{U}|)$. Combining the above steps yields a total complexity of $O(|\mathcal{U}| \log |\mathcal{U}|)$. Since each candidate action corresponds to a tool-server pair with either a scaling-up or scaling-down operation, the action space size satisfies $|\mathcal{U}| = O(|\mathcal{T}||\mathcal{S}|)$. Therefore, the overall time complexity becomes $O(|\mathcal{T}||\mathcal{S}| \log(|\mathcal{T}||\mathcal{S}|))$. □

Lemma 4 (GS Space Complexity)**.** *The space complexity of the GS deployment algorithm is bounded by* $O(|\mathcal{T}||\mathcal{S}|)$.

Proof. GS maintains aggregated runtime statistics for each tool-server pair, including instance counts, service rates, and utilization statistics. Since these statistics are stored once for every tool-server pair, the total memory requirement grows linearly with the number of tool-server pairs. Therefore, the space complexity is bounded by $O(|\mathcal{T}||\mathcal{S}|)$. □

Chapter 7 Implementations

This chapter presents the implementation details of the proposed DOME framework. We introduce the software libraries adopted to support MCP-native orchestration across heterogeneous edge servers, describe the baseline algorithms implemented for comparison with our proposed SSD routing and GS deployment algorithms, and discuss the selection and processing of public MCP-related datasets used to emulate realistic workloads in our experiments.

7.1 Software Libraries

This section introduces the major software libraries adopted in DOME. These libraries provide the communication, messaging, and remote access mechanisms required to coordinate AI agents, edge servers, and distributed MCP tool instances. By leveraging existing mature frameworks and protocols, DOME reduces implementation complexity while maintaining compatibility with emerging MCP ecosystems and heterogeneous deployment environments.

7.1.1 MCP Python SDK

The MCP Python SDK [63] provides the official Python implementation of the MCP, offering standardized abstractions for MCP session management, tool definition, call handling, and bidirectional communication between MCP clients and servers. The SDK supports multiple communication transports, including `stdio` and `streamable HTTP`, allowing MCP tools to be integrated under a unified programming interface without manually implementing low-level protocol operations.

In DOME, the MCP Python SDK serves as the core communication foundation between AI agents, the root edge server, and distributed MCP tool instances. Specifically, the connection manager utilizes the SDK to establish and maintain persistent MCP sessions with both local and remote tool instances. For tool instances hosted on the root

edge server, the SDK directly launches tool processes and communicates through `stdio`, minimizing unnecessary networking overhead. For remote edge servers, the SDK interoperates with MCP Proxy through `streamable HTTP` sessions, enabling transparent access to distributed MCP tools under the same MCP abstraction. We use the MCP Python SDK to ensure compatibility with existing MCP ecosystems while reducing session management complexity. Its persistent-session design also helps DOME reduce connection overhead for lightweight and short-lived MCP tool executions.

7.1.2 MCP Proxy

MCP Proxy [64] is a lightweight proxy framework that exposes local MCP tools for remote access via HTTP. It bridges local MCP tools to network-accessible MCP endpoints, enabling distributed MCP deployment without modifying the original MCP tool implementations. By encapsulating MCP sessions behind streamable HTTP interfaces, MCP Proxy simplifies remote MCP tool access and deployment across heterogeneous environments.

In DOME, MCP Proxy is deployed on edge servers to expose locally hosted MCP tool instances to the root edge server. Instead of requiring the root server to directly manage remote processes via custom networking logic, MCP Proxy provides a unified, transparent remote-access layer for distributed MCP sessions. This design allows the root edge server to interact with remote MCP tool instances using the same MCP abstractions as locally hosted tools. We adopt MCP Proxy to support transparent remote access to distributed MCP tool instances while preserving MCP-native communication semantics. Its lightweight proxy-based architecture aligns well with DOME's process-level deployment design on resource-constrained edge servers.

7.1.3 MQTT

MQTT [65] is a lightweight publish-subscribe messaging protocol designed for resource-constrained and unreliable network environments. Unlike request-response communication models, MQTT uses an asynchronous, event-driven architecture in which publishers publish messages through a broker, and subscribers selectively receive relevant updates.

This decoupled communication mechanism reduces synchronization overhead and improves scalability in distributed systems.

In DOME, MQTT is used by the state manager and state monitors to disseminate runtime statistics among edge servers and the root edge server. Each edge server periodically publishes local monitoring information, including CPU utilization, queue states, network bandwidth, and service rates. The root edge server subscribes to these updates and aggregates them into global system states for routing and deployment decisions. We use MQTT to support lightweight, continuous dissemination of runtime monitoring information across heterogeneous edge servers. Its asynchronous publish-subscribe architecture also enables DOME to continuously maintain distributed system monitoring.

7.1.4 gRPC

gRPC [66] is a high-performance Remote Procedure Call (RPC) framework that supports efficient communication between distributed applications through HTTP/2 and Protocol Buffers. It provides strongly typed service interfaces, bidirectional streaming capabilities, and low-latency request handling for distributed systems.

In DOME, gRPC is used as the communication interface between AI agents and the root edge server. AI agents submit user requests and MCP tool calls to the DOME orchestrator via gRPC-based RPC services, while execution results are returned via the same channel. This design provides a unified interface for handling concurrent agent requests in the DOME framework. We use gRPC to efficiently and scalably dispatch requests between AI agents and the DOME orchestrator. Its persistent HTTP-based communication model and compact serialization mechanism further help reduce communication overhead during frequent MCP tool invocations.

7.2 Baseline Algorithms

To evaluate the effectiveness of our proposed SSD routing algorithm and GS deployment algorithm, we implement several representative baseline algorithms for comparison. These baselines capture different routing and load-balancing strategies commonly

adopted in distributed systems, providing diverse reference points for evaluating DOME under varying orchestration behaviors.

7.2.1 Routing Algorithms

We introduce the routing baseline algorithms implemented for comparison with our proposed SSD routing algorithm, ranging from load-agnostic random assignment to queue-aware scheduling policies.

- **Random (RD).** RD is a load-agnostic routing policy that uniformly distributes requests across all candidate queues without considering runtime system conditions. It is commonly adopted as a simple baseline for evaluating whether state-aware routing mechanisms can effectively improve system performance. RD selects a queue with equal probability: $1/|\mathcal{Q}_\tau|.P(q \mid c) = 1/|\mathcal{Q}_\tau|,\ \forall q \in \mathcal{Q}_\tau$, and samples the decision as $q^* \sim P(\cdot \mid c)$.

- **Round Robin (RR) [67].** RR is a deterministic load-balancing policy that evenly distributes workloads across multiple candidate queues. It is particularly effective in environments where servers exhibit similar processing capacities and relatively homogeneous runtime conditions. RR selects the queue in a cyclic order: $q^* = q_{(g_\tau \bmod |\mathcal{Q}_\tau|)}$, where $\{q_1, \ldots, q_{|\mathcal{Q}_\tau|}\}$ denotes an ordered list of queues in $\mathcal{Q}_\tau$, and g_τ is a per-tool counter incremented after each assignment.

- **Power-of-Two Choices (PO2) [68].** PO2 is a lightweight probabilistic load-balancing algorithm that approximates globally balanced routing decisions using only partial system information. Instead of evaluating all candidate queues, it compares a small subset of randomly sampled queues to reduce routing overhead while still mitigating congestion imbalance. PO2 first samples two queues $q_1, q_2 \in \mathcal{Q}_\tau$ uniformly at random with $q_1 \neq q_2$ from $\mathcal{Q}_\tau$ and selects the one with the smaller total delay: $q^* = \arg\min_{q \in \{q_1, q_2\}} \tilde{D}(c, q)$, where $\tilde{D}(c, q)$ is the estimated delay defined in Chapter 5.

- **Join-the-shortest-queue (JSQ) [69].** JSQ is a queue-aware routing policy that dynamically routes requests toward the least congested queue. Its primary objective

is to reduce queueing delay by continuously reacting to instantaneous workload imbalance among candidate queues. JSQ selects the queue with the minimum queue length: $q^* = \arg\min_{q \in \mathcal{Q}_\tau} l_q$.

7.2.2 Deployment Algorithms

For the deployment problem, we use *Static*, which refers to static deployment, as the baseline. It is used to compare adaptive scaling methods. We also implement *Threshold-based Scaling (TS)* as a baseline deployment algorithm. TS monitors the utilization of each edge server and triggers scaling actions when the utilization exceeds the upper threshold θ_h or falls below the lower threshold θ_l. When $\rho_s > \theta_h$, TS considers server s under-provisioned and generates candidate scale-up actions $u = (+1, \tau, s)$ for the tools deployed on that server. Conversely, when $\rho_s \leq \theta_l$, TS considers server s over-provisioned and generates candidate scale-down actions $u = (-1, \tau, s)$ for tools whose remaining instance count would remain positive after scaling. Each candidate action is assigned a priority according to the severity of utilization threshold violation, i.e., $\rho_s - \theta_h$ for scale-up actions and $\theta_l - \rho_s$ for scale-down actions. TS then greedily selects actions in descending order of priority while enforcing the same feasibility constraints as our proposed system, where each server and each tool can participate in at most one scaling action. Unless otherwise specified, we set $\theta_h = 0.8$ and $\theta_l = 0.2$.

7.3 Dataset Selection

To emulate realistic AI agents for driving our experiments, we need public MCP workload datasets with three desired properties: (i) *tools*: with a list of executable MCP tools, (ii) *requests*: with user requests to AI agents, and (iii) *calls*: with each user request mapped into a sequence of MCP tool calls. Table 7.1 surveys existing datasets with qualitative comparisons in the abovementioned properties.

We observe that many datasets [70], [71], [72] focus on MCP tool collections, tool descriptions, or security analysis, but lack user requests and tool-call traces. Other datasets [73], [74] include real-world user requests, but do not provide tool-call mappings. Xu et al. [75]

Table 7.1: Public MCP-related Datasets

Dataset	Tools	Requests	Calls
Lin et al. [70]	✓	✗	✗
Wu et al. [71]	✓	✗	✗
Fei et al. [72]	✓	✗	✗
Wang et al. [73]	✓	✓	✗
Luo et al. [74]	✓	✓	✗
Xu et al. [75]	△	✓	✓
Bandi et al. [76]	✓	✓	✓
Mo et al. [77]	✓	✓	✓

provide both requests and tool-call traces; however, part of the tool execution depends on cloud-managed infrastructure, making the dataset less suitable for edge-deployment experiments. Only Bandi et al. [76] and Mo et al. [77] satisfy all three desired properties. Between these two datasets, we use the well-known Mo et al. [77] in our experiments, and will consider the very recent Bandi et al. [76] in our future work.

Mo et al. [77] classifies MCP tools into three categories: (i) visualization, (ii) file access, and (iii) discovery. We focus on discovery categories because visualization tool results are interactively consumed by local users, while file access tools are fairly lightweight. The workloads incurred by visualization and file access tools may not benefit from edge-server offloading. Among all discovery tools, we check whether they are deprecated, non-functional, or black-listed as malicious. After filtering out the problematic MCP tools, we have 14 MCP tools invoked by 41 user requests, with each user request having at least 1, at most 7, and an average of 3.01 tool calls.

Chapter 8 Experiments

This chapter evaluates the performance of our DOME framework and the effectiveness of the proposed routing and deployment algorithms. Using a heterogeneous edge testbed, we conduct a series of experiments to analyze how DOME adapts to dynamic workloads and heterogeneous computing environments. We evaluate the routing algorithm for delay reduction and request completion performance, and examine how the deployment algorithm improves resource utilization under varying user demands.

8.1 Experiment Setup

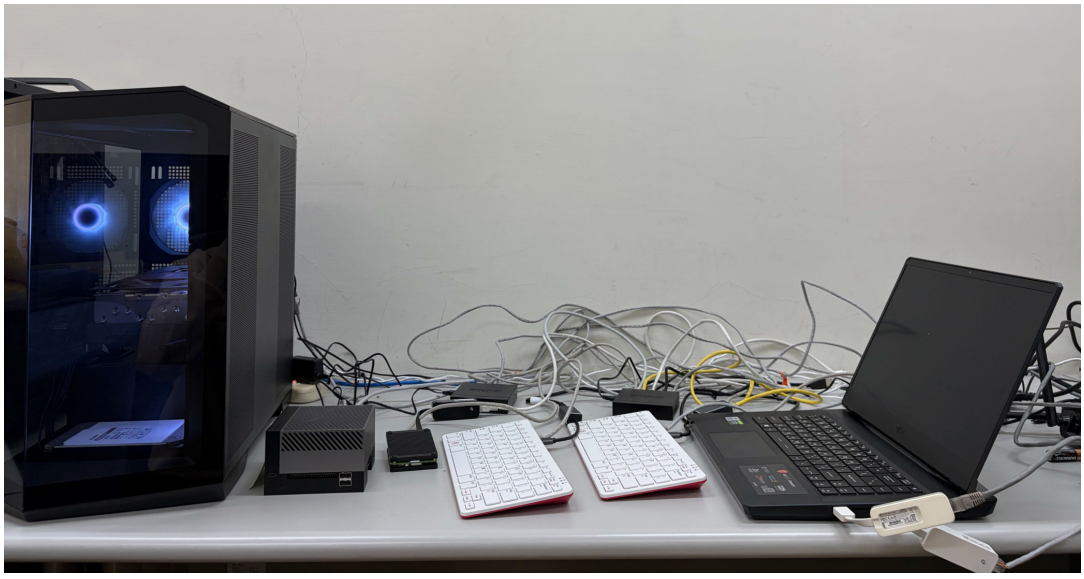

Fig. 8.1: The real DOME testbed with a client computer, a root edge server, and four edge servers.

We set up an on-premise edge testbed with heterogeneous edge servers, as shown in Fig. 8.1. Our testbed consists of an Intel i5 10-core PC @ 4.7 GHz serving as the root edge server. It also contains multiple edge servers, including an 8-core NVIDIA Jetson AGX Orin @ 2.2 GHz, one 4-core Raspberry Pi 5 @ 2.4 GHz, and two 4-core Raspberry Pi 400 @ 1.8 GHz. We use a laptop as the client computer to emulate AI agents. The

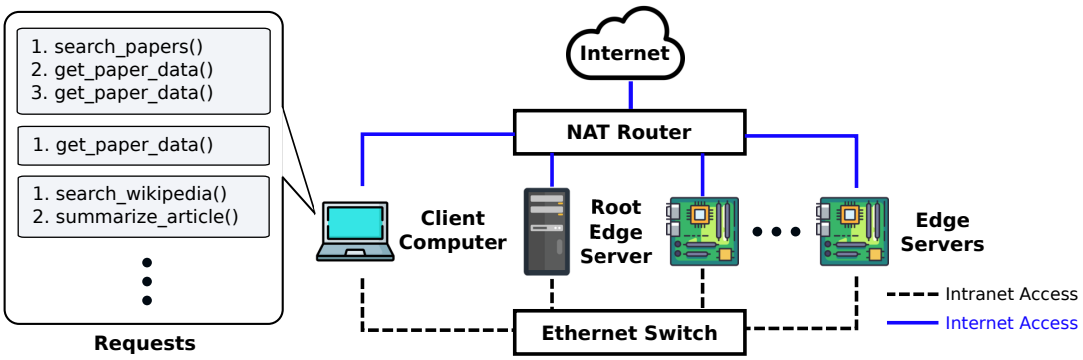

Fig. 8.2: The network topology of our testbed with sample user requests.

abovementioned edge servers and the client computer are connected via an internal switch and a NAT router, as illustrated in Fig. 8.2. Here, the Intranet is primarily used to facilitate edge orchestration, while the Internet is used for MCP tools that require online resources, such as Web content.

Using the processed dataset, we run AI agents on the client computer to emulate U users, each of whom randomly replays a user request. A tool call is considered successful or failed according to the outcome returned by the corresponding MCP tool. In addition, DOME enforces a timeout threshold of 45 s, and any tool call that does not complete within this period is treated as failed. Upon completing a user request, the AI agent replays another random user request. Each experiment lasts for (i) $T = 300$ s for evaluating routing algorithms, and (ii) $6 \cdot T = 1800$ s for evaluating deployment algorithms. In particular, to evaluate routing algorithms, we vary $U \in \{14, 28, 42\}$. with $U = 28$ as the default. Furthermore, to evaluate deployment algorithms, we vary the number of users once every $T = 300$ s, so that the numbers of users are U, $1.5U$, $2U$, $2U$, $2U$, $1.5U$, and U. We let deployment window $W = 60$ s and we vary the SSD control parameter $\gamma \in \{1.0, 1.5, 2.0, 2.5, 3.0\}$ with $\gamma = 2.0$ as the default. We measure and analyze the following performance metrics: (i) *total delay*, (ii) *bandwidth consumption*, (iii) *CPU utilization*, and (iv) *number of completed requests or calls*. Notably, we use downlink bandwidth consumption as the performance metric, as uplink bandwidth shows similar values. To be more specific, we compute the average value of each metric every 30 seconds and collect all samples from a single run to obtain the cumulative distribution function (CDF). Each experiment consists of five runs. We report the average performance from five runs and

provide the 95% confidence interval when applicable.

8.2 Results

This section presents the experimental results of the proposed DOME framework, evaluating the effectiveness of the SSD routing algorithm and the GS deployment algorithm under dynamic workloads on the real-world heterogeneous edge testbed. We organize the results into two parts: *routing* and *deployment*.

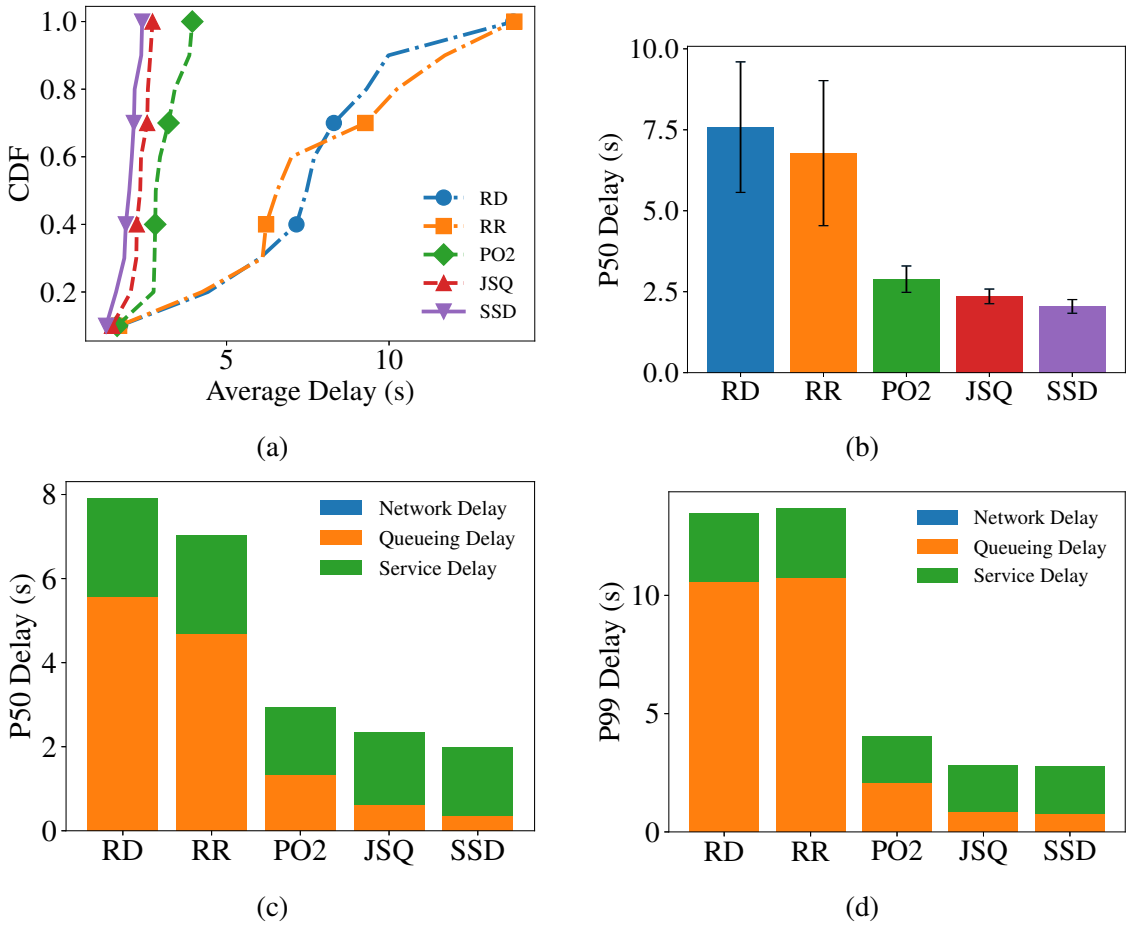

Fig. 8.3: Total delay per tool call across all five runs: (a) CDF, (b) P50 total delay, (c) P50 delay breakdown, and (d) P99 delay breakdown.

8.2.1 Routing

This section evaluates the routing performance of the proposed SSD algorithm against four baseline algorithms under varying workload conditions. We analyze the results

across four dimensions. First, we compare the total delay per tool call to assess system responsiveness. Second, we examine the number of completed user requests and tool calls to measure overall system throughput. Third, we evaluate CPU utilization and downlink bandwidth consumption to quantify computational and network overhead. Finally, we investigate the scalability of each algorithm under increasing numbers of concurrent users. Across all evaluated routing algorithms, the tool execution success rate remains above 99.5%, indicating that performance differences primarily arise from delay, throughput, and resource utilization rather than execution failures.

SSD achieves the lowest delay. Fig. 8.3 illustrates the comparison of total delay per tool call across different routing algorithms. As shown in Fig. 8.3(a), two distinct distributions can be observed: (i) RD and RR, and (ii) PO2, JSQ, and SSD. A steeper CDF indicates that most tool calls experience lower delays, reflecting better system responsiveness. Algorithms based on queuing theory, such as JSQ and SSD, exhibit steeper cumulative distributions, whereas others show a more gradual increase. Moreover, Fig. 8.3(b) shows the P50 delay for each tool call, which represents overall responsiveness. Our proposed SSD achieves the lowest P50 delay at 2.04 s, compared to the other routing algorithms. Meanwhile, RD and RR incur the highest P50 delay, which is approximately 3.7 and 3.2 times that of SSD. SSD also outperforms PO2 and JSQ with delay reductions of 29.2% and 13.1%, respectively.

To understand the source of this improvement, Figs. 8.3(c) and 8.3(d) decompose the delay into three components: network delay, queueing delay, and service delay. Network delay is negligible across all algorithms below 1 ms. Service delay also remains consistent across algorithms, ranging from 1.61 s to 2.35 s at P50, as it reflects the inherent execution time of each MCP tool rather than routing decisions. Queueing delay, by contrast, is the dominant differentiator: RD and RR incur P50 queueing delays of 5.56 s and 4.68 s, respectively, while SSD reduces this to only 0.34 s. The effect is even more pronounced at P99, where RD and RR accumulate queueing delays exceeding 10 s, compared to 0.76 s under SSD. This confirms that the large delay gap between algorithms stems primarily from load concentration on slower servers under load-oblivious routing, which SSD's delay-aware dispatching effectively avoids. *In summary, SSD consistently delivers supe-*

rior responsiveness compared to the other routing algorithms, primarily by eliminating excessive queueing delay through adaptive load distribution. In summary, SSD consistently delivers superior responsiveness compared to the other routing algorithms.

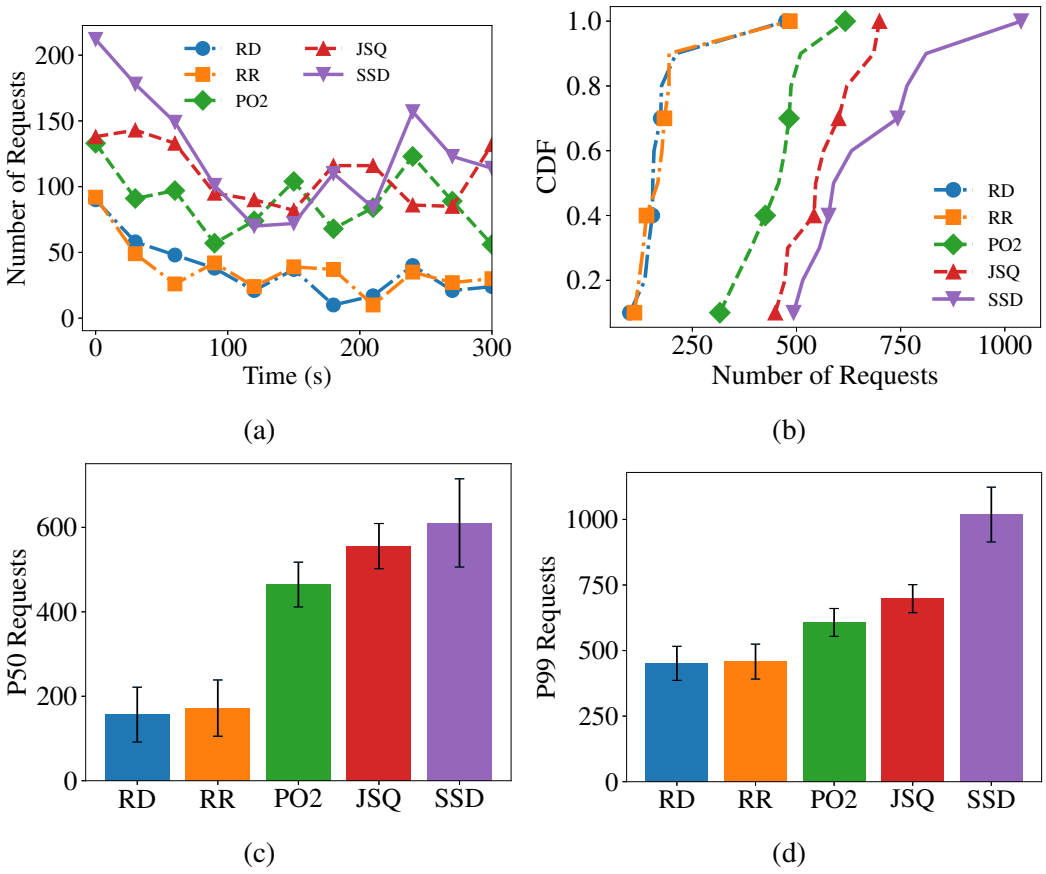

Fig. 8.4: Completed user requests by different routing algorithms. (a): sample run over time. (b): CDF from all five runs. (c): P50 results from all five runs. (d): P99 results from all five runs.

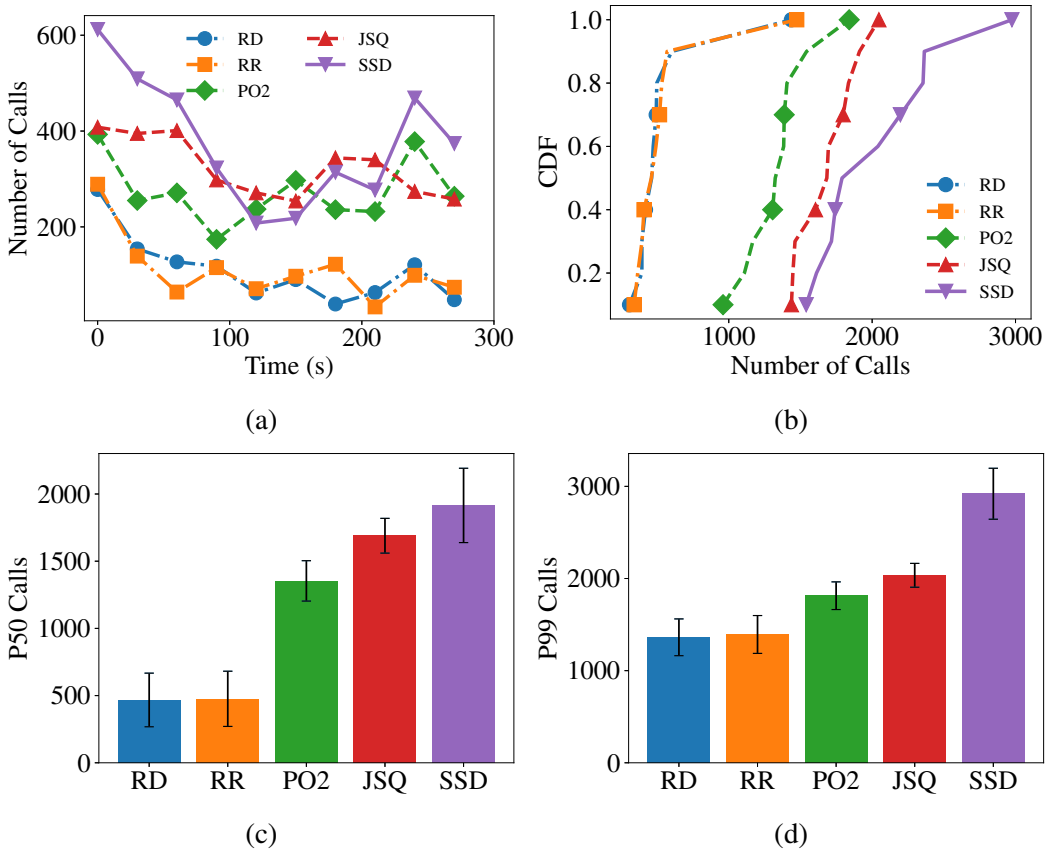

Fig. 8.5: Completed tool calls by different routing algorithms. (a): sample run over time. (b): CDF from all five runs. (c): P50 results from all five runs. (d): P99 results from all five runs.

SSD completes more user requests and tool calls. Fig. 8.4 and Fig. 8.5 compare the number of requests and calls completed by different routing algorithms. In Fig. 8.4(a) and Fig. 8.5(a), RD and RR generally complete fewer requests and calls than the other algorithms, whereas PO2, JSQ, and SSD complete more. Furthermore, Fig. 8.4(b) and Fig. 8.5(b) demonstrate that SSD completes substantially more requests and calls across all five runs. Meanwhile, Fig. 8.4(c) and Fig. 8.4(d), as well as Fig. 8.5(c) and Fig. 8.5(d) reveal that SSD consistently achieves the highest number of completed requests and calls across five runs. In particular, SSD completes over 4 times more tool calls than RD and RR in the P50 results. Even in the P99 results, which capture the higher-end performance, SSD still outperforms the stronger baselines with 43.5% and 61.0% more completed tool calls than JSQ and PO2, respectively. These results confirm that SSD provides effective and efficient routing. *In summary, SSD significantly outperforms other algorithms by*

completing more requests and calls.

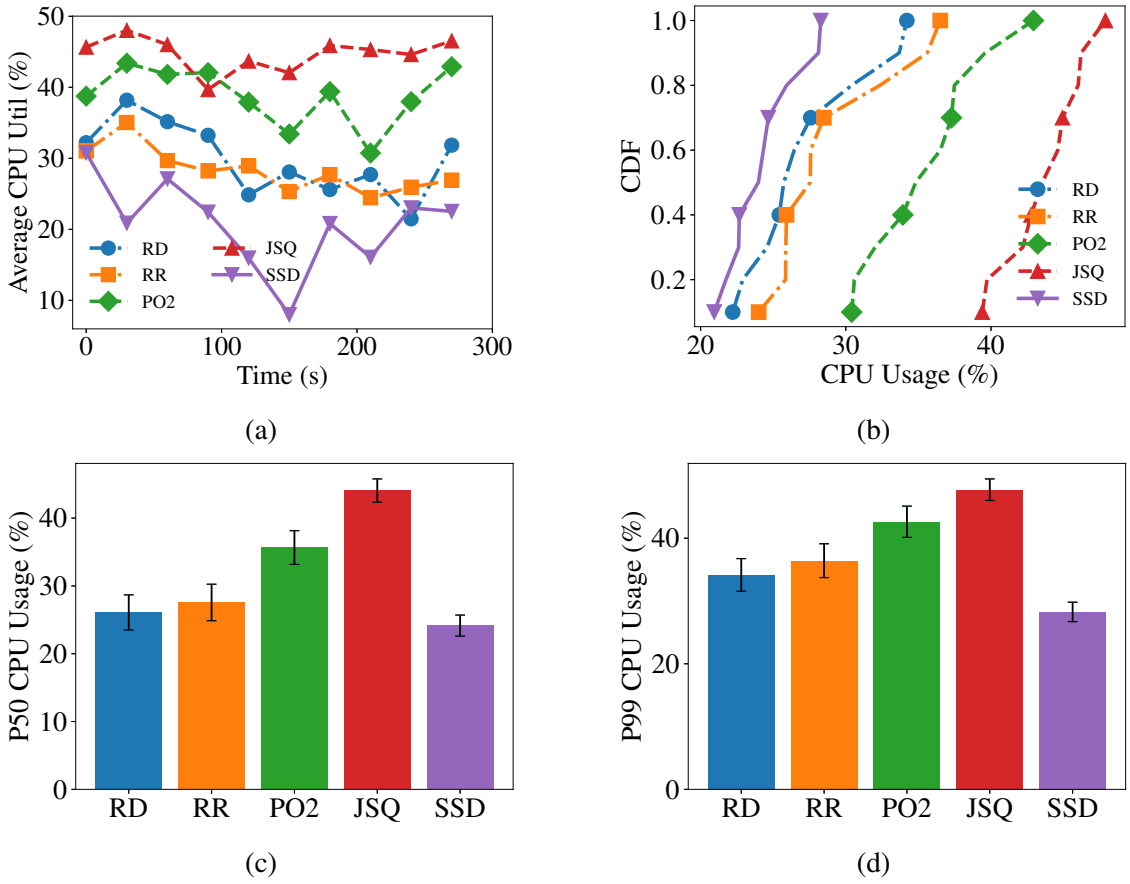

Fig. 8.6: CPU utilization incurred by different routing algorithms. (a): sample run over time. (b): CDF from all five runs. (c): P50 results from all five runs. (d): P99 results from all five runs.

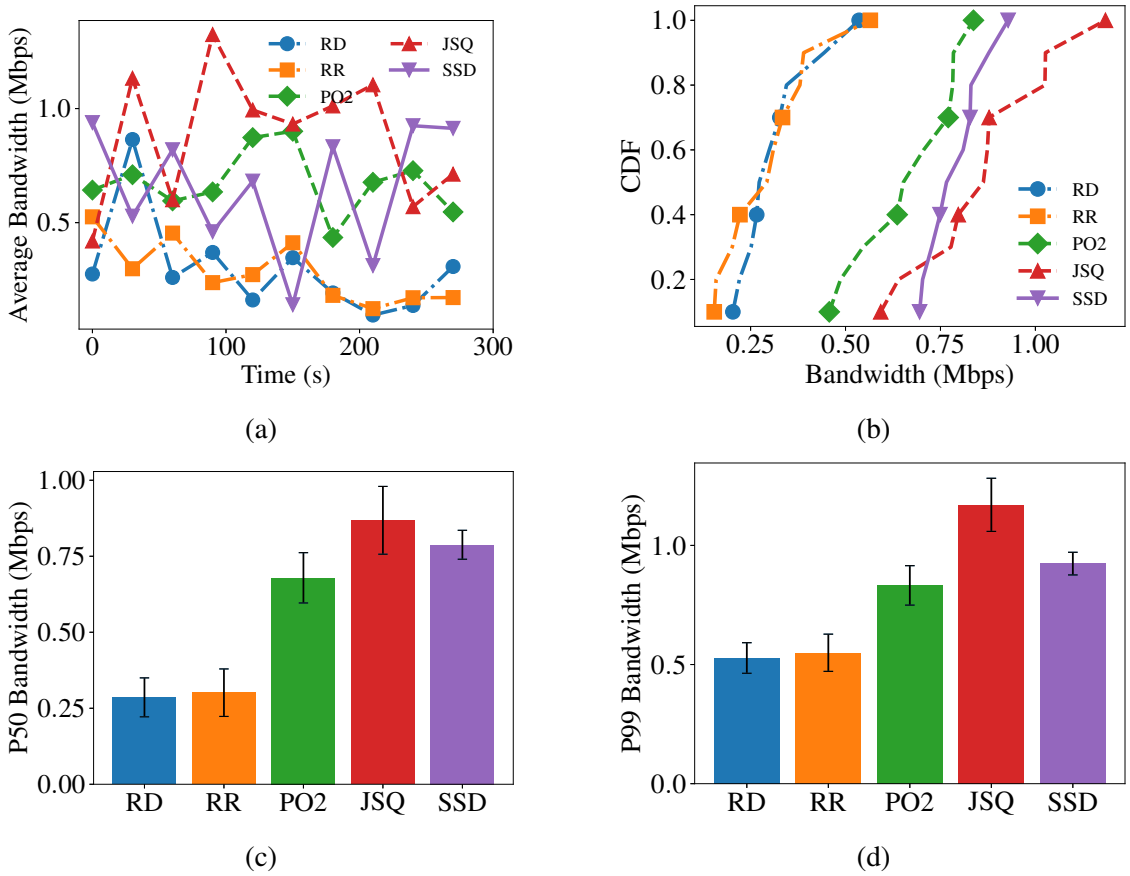

Fig. 8.7: Downlink bandwidth consumption incurred by different routing algorithms. (a): sample run over time. (b): CDF from all five runs. (c): P50 results from all five runs. (d): P99 results from all five runs.

SSD does not incur excessive computation and network overhead. Fig. 8.6 and Fig. 8.7 compare the CPU utilization and downlink bandwidth consumption across different routing algorithms. As shown in Fig. 8.6(a) and Fig. 8.7(a), SSD generally achieves performance comparable to that of other routing algorithms. The CDFs of CPU utilization and bandwidth consumption, presented in Fig. 8.6(b) and Fig. 8.7(b), further indicate that SSD introduces only marginal overhead in both computation and network usage. We report CPU utilization in Fig. 8.6(c) and Fig. 8.6(d). SSD uses 24.15% CPU on average and 28.26% at P99. In Fig. 8.7(c) and Fig. 8.7(d), we observe that SSD consumes 0.48 Mbps more bandwidth than RD and RR, and 0.21 Mbps more than PO2. These increases have a negligible impact on overall network usage. However, SSD reduces bandwidth consumption by 9.2% compared to JSQ, demonstrating its higher efficiency. *In summary, SSD incurs minimal overhead in both computation and downlink bandwidth consumption*

compared to other routing algorithms.

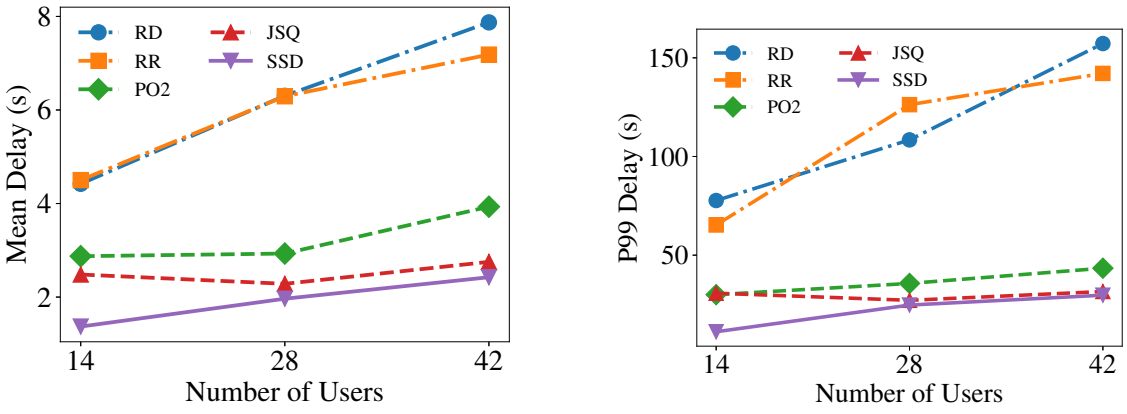

Fig. 8.8: Delay of different routing algorithms under increasing number of concurrent users: (a) Mean delay and (b) P99 delay results from 5 runs.

SSD scales well with increasing load. Fig. 8.8 compares the delays of the routing algorithms under different numbers of concurrent users. An increase in the number of concurrent users affects both the workload and the system responsiveness. Fig. 8.8(a) shows the P50 delay under different numbers of concurrent users for different routing algorithms. As the number of concurrent users increases, SSD, JSQ, and PO2 remain relatively stable across different workloads, with SSD consistently achieving the lowest P50 delay, reducing it by up to 44.8% compared to JSQ and 52.3% compared to PO2. On the other hand, RD and RR are significantly affected by the increasing workload. They suffer from a much higher P50 delay under heavy workload (42 concurrent users), approximately 3.3 times that of SSD. Fig. 8.8(b) presents the P99 delay, where tail delay characterizes worst-case system behavior. Among all routing algorithms, SSD maintains the lowest P99 delay even under a heavy workload, at 29.72 s. Specifically, PO2 and JSQ incur up to 63.1% additional P99 delay compared to SSD. Moreover, RD and RR experience severe degradation, with P99 delay increasing to around 142 s to 157 s, which is up to 428% higher than that of SSD.

Implications of γ values on SSD algorithm. Fig. 8.9 shows the impact of γ on our proposed SSD in terms of delay and CPU utilization. As γ increases, Fig. 8.9(a)

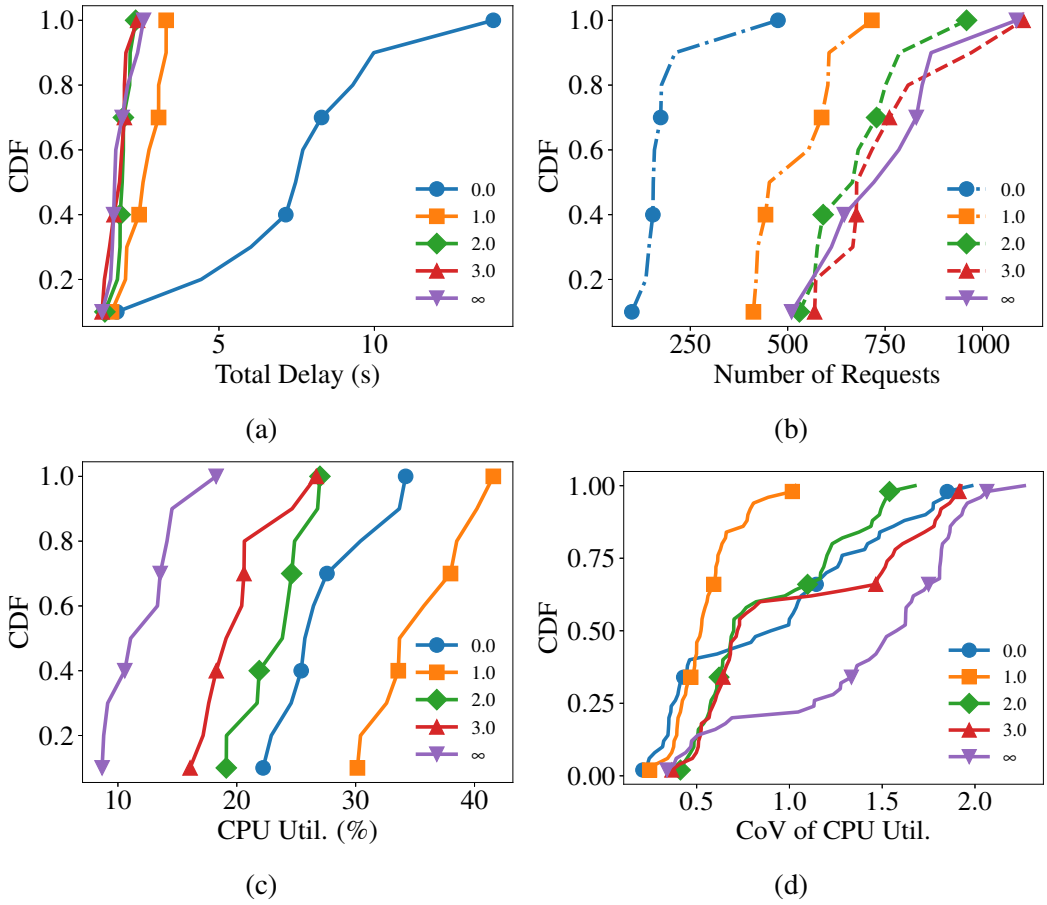

Fig. 8.9: Impact of γ values on SSD algorithm: (a) total delay, (b) number of completed requests, (c) CPU utilization, and (d) coefficient of variance in CPU utilization results from all five runs.

shows that the total delay decreases, Fig. 8.9(b) shows that the number of completed requests increases, and Fig. 8.9(c) shows that CPU utilization decreases. However, the coefficient of variation of CPU utilization, shown in Fig. 8.9(d), increases due to workload imbalance, where more powerful servers tend to handle a larger number of requests.

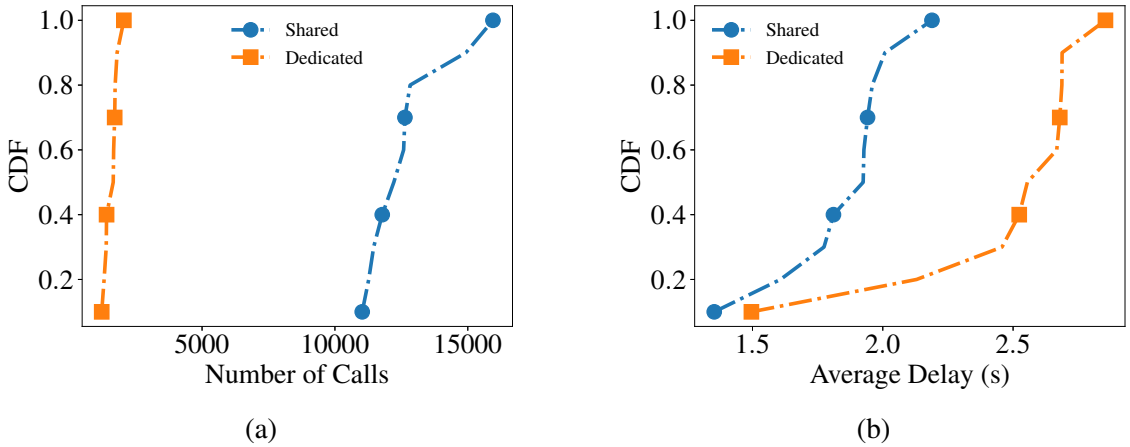

Fig. 8.10: Comparisons of different queue designs of our proposed SSD algorithm, CDFs from all five runs of: (a) number of tool calls and (b) total delay.

Queuing design. Fig. 8.10 compares two queuing designs on our SSD routing algorithm. Fig. 8.10(a) shows the CDF of the number of tool calls. SSD with a shared queue completes significantly more tool calls compared to a dedicated queue. Fig. 8.10(b) demonstrates the CDF of total delay. SSD with a shared queue also achieves shorter delay. *Therefore, we recommend using our proposed SSD with a shared queue, as it improves both the number of completed tool calls and delay performance.*

8.2.2 Deployment

This section evaluates the deployment performance of the proposed GS algorithm against static deployment and the TS baseline under a time-varying workload. We analyze the results across three dimensions. First, we examine the number of completed user requests and tool calls to assess throughput gains from adaptive scaling. Second, we compare the total delay to evaluate the responsiveness improvement under dynamic workloads. Third, we analyze CPU utilization to measure how effectively GS utilizes heterogeneous edge resources while respecting the predefined saturation threshold.

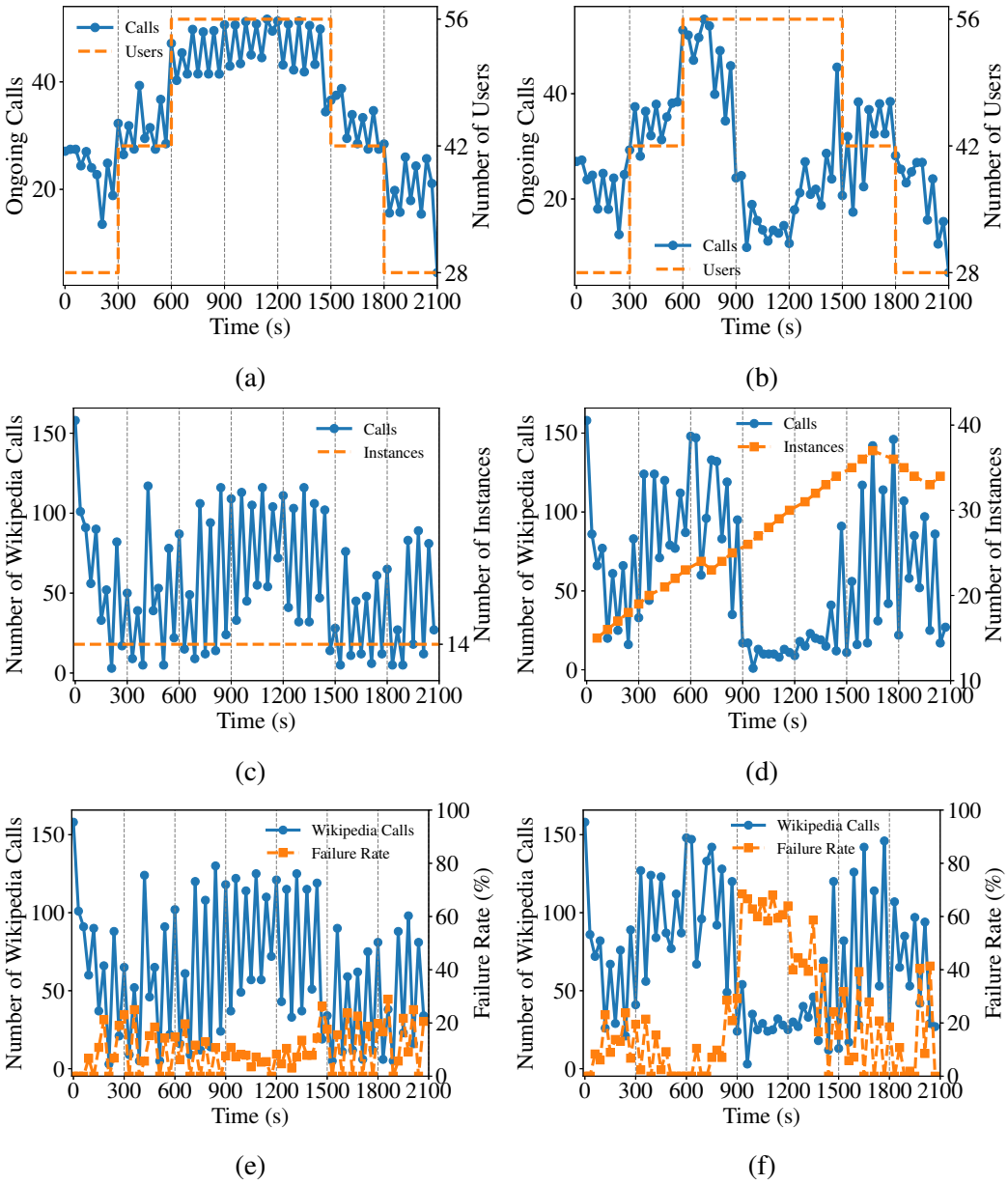

Fig. 8.11: Impact of GS and static deployment on tool calls and failure. (a), (b): relationship between the number of ongoing tool calls and concurrent users; (c), (d): relationship between completed Wikipedia calls and number of instances; (e), (f): relationship between completed Wikipedia calls and failure rate results from a sample run.

GS fails to improve efficiency due to increasing failure rate on specific tools. Fig. 8.11 compares the behavior of static and GS during a sample run. Fig. 8.11(a) shows that, under static deployment, the number of ongoing tool calls generally increases with the number of users. However, Fig. 8.11(b) reveals a different behavior under GS. Although the number of users continues increasing, the number of ongoing tool calls ex-

periences a noticeable drop during the execution period. This observation suggests that increasing deployment instances alone may not always translate into higher efficiency.

To further investigate this behavior, we analyze one of the major bottleneck tools, `Wikipedia`. Fig. 8.11(c) and Fig. 8.11(d) compare the number of completed calls from Wikipedia processed under static and GS. Under static deployment, the number of completed Wikipedia calls remains relatively stable during periods with the highest number of concurrent users, ranging from 34.4 to 51.7 calls. In contrast, GS exhibits a noticeable reduction in completed Wikipedia calls even when the number of instances increases from 23 to 37. Specifically, the number of completed Wikipedia calls decreases by 30.6% during the scaling period. These results indicate that the efficiency of GS can be constrained by external tool limitations despite allocating additional instances.

We further analyze the failure behavior of Wikipedia calls. Fig. 8.11(e) and Fig. 8.11(f) show the relationship between the number of Wikipedia calls and failure rate. Compared with static deployment, GS introduces noticeable spikes in failure rate during scaling periods. The failure rate reaches its highest point of 68.5%, while the number of completed Wikipedia calls decreases to only 1. This behavior can be resorted to bursts of concurrent calls generated by newly scaled instances within a short time interval, which trigger HTTP 429 rate-limit errors from the Wikipedia tool. Consequently, the temporary increase in request failures reduces the number of successfully processed calls during these periods. *In summary, these results demonstrate that external service bottlenecks can affect the effectiveness of GS under highly concurrent workloads.*

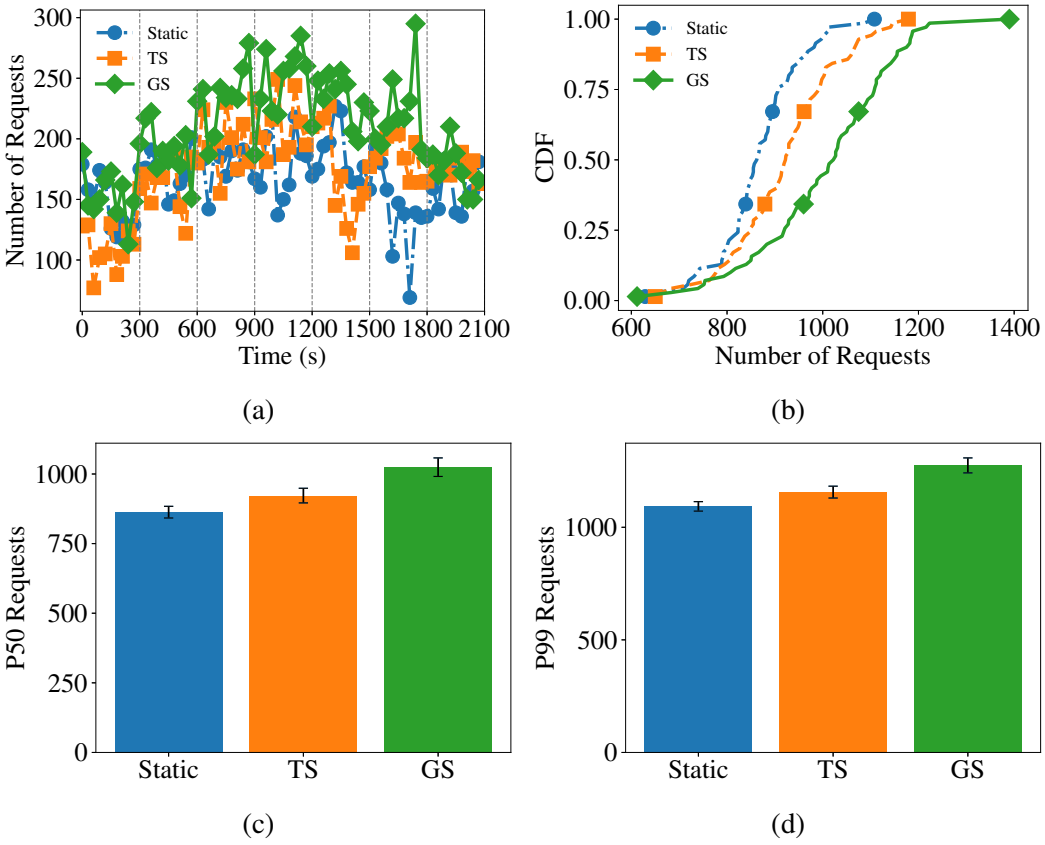

Fig. 8.12: Completed user requests under different deployment algorithms after removing the Wikipedia tool. (a): Sample run over time. (b): CDF from all five runs. (c): P50 results from all five runs. (d): P99 results from all five runs.

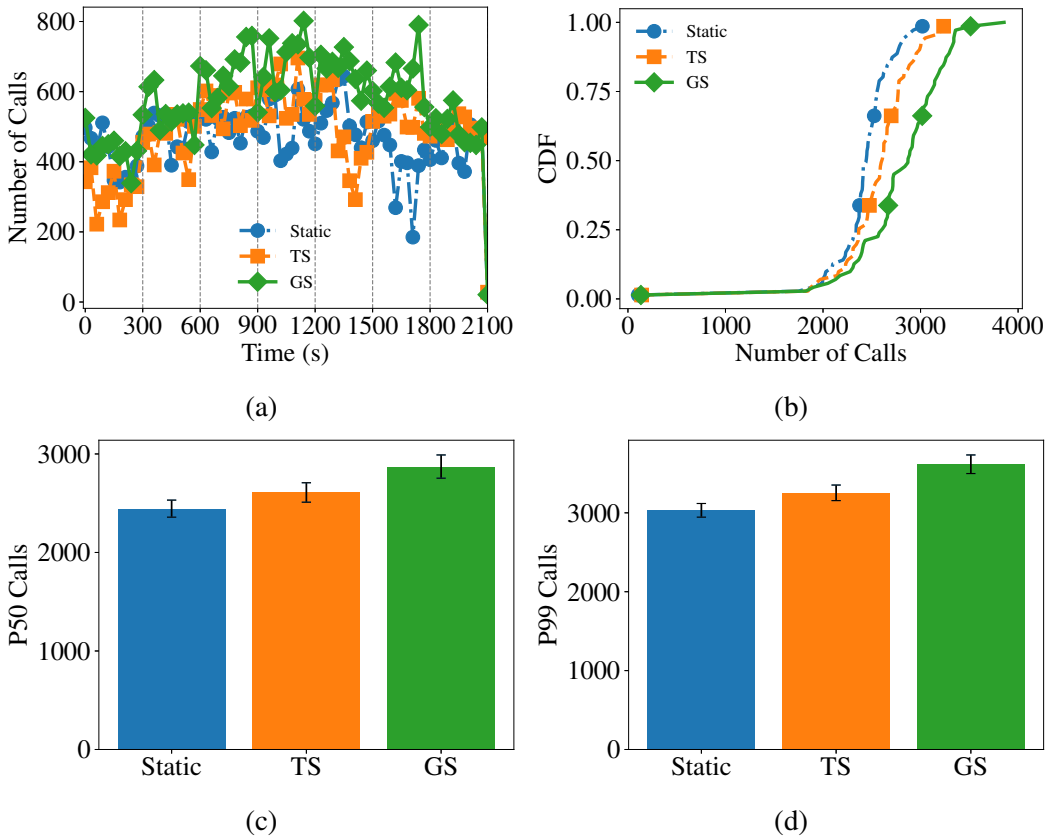

Fig. 8.13: Completed user calls under different deployment algorithms after removing the Wikipedia tool. (a): Sample run over time. (b): CDF from all five runs. (c): P50 results from all five runs. (d): P99 results from all five runs.

GS enables more requests and tool calls to be completed. Fig. 8.12 and Fig. 8.13 compare the number of completed requests and calls under different deployment algorithms after excluding the Wikipedia tool. As shown in Fig. 8.12(a) and Fig. 8.13(a), GS generally sustains a higher number of completed requests and calls during periods with increasing workload. The cumulative distributions in Fig. 8.12(b) and Fig. 8.13(b) demonstrate that GS consistently processes more requests across all five runs. Moreover, Fig. 8.12(c) and Fig. 8.12(d) show that GS completes the most requests at both P50 and P99 cases, while Fig. 8.13(c) and Fig. 8.13(d) demonstrate the same trend for tool calls. Specifically, GS increases the number of completed requests by 18.7% compared with static deployment and by 11% compared with TS at P50, while also increasing completed tool calls by 17.5% and 10.1%, respectively. At peak performance, GS further improves

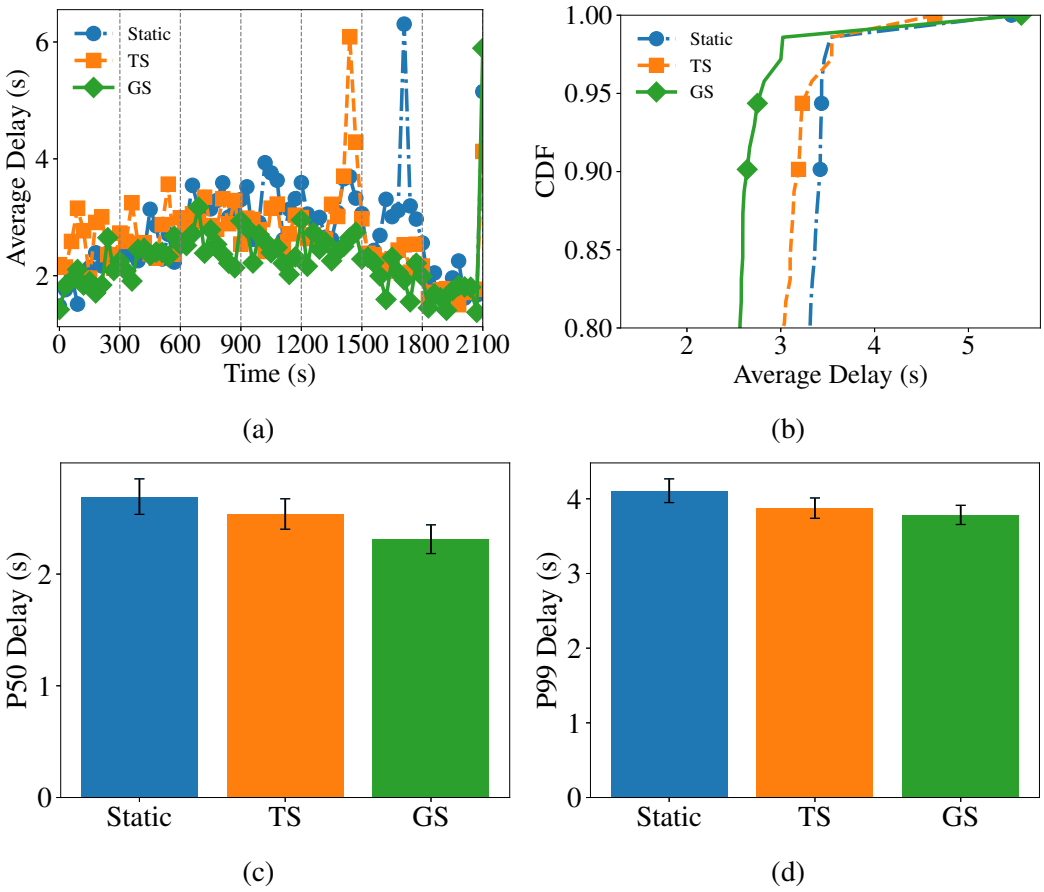

Fig. 8.14: Delay under different deployment algorithms after removing the Wikipedia tool. (a): Sample run of delay over time. (b): CDF from all five runs. (c): P50 results from all five runs. (d): P99 results from all five runs.

the P99 request completion count by up to 16.6%, accompanied by a corresponding increase of 19.3% in completed tool calls. These results suggest that adaptive deployment scaling enables GS to maintain higher processing capability under dynamic workloads. *In summary, GS consistently completes more user requests than static deployment and TS.*

GS achieves lower request and tool call delay. Fig. 8.14 presents the delay performance of different deployment algorithms after removing the Wikipedia tool from the workload. As shown in Fig. 8.14(a), GS maintains stable delay behavior throughout the execution period. The delay distribution in Fig. 8.14(b) further shows that GS shifts a larger portion of requests toward lower latency ranges, indicating improved responsiveness across different workload conditions. This improvement is also reflected in Fig. 8.14(c) and Fig. 8.14(d), where GS achieves the lowest P50 and P99 delays among all deployment algorithms. GS consistently outperforms both static deployment and TS, re-

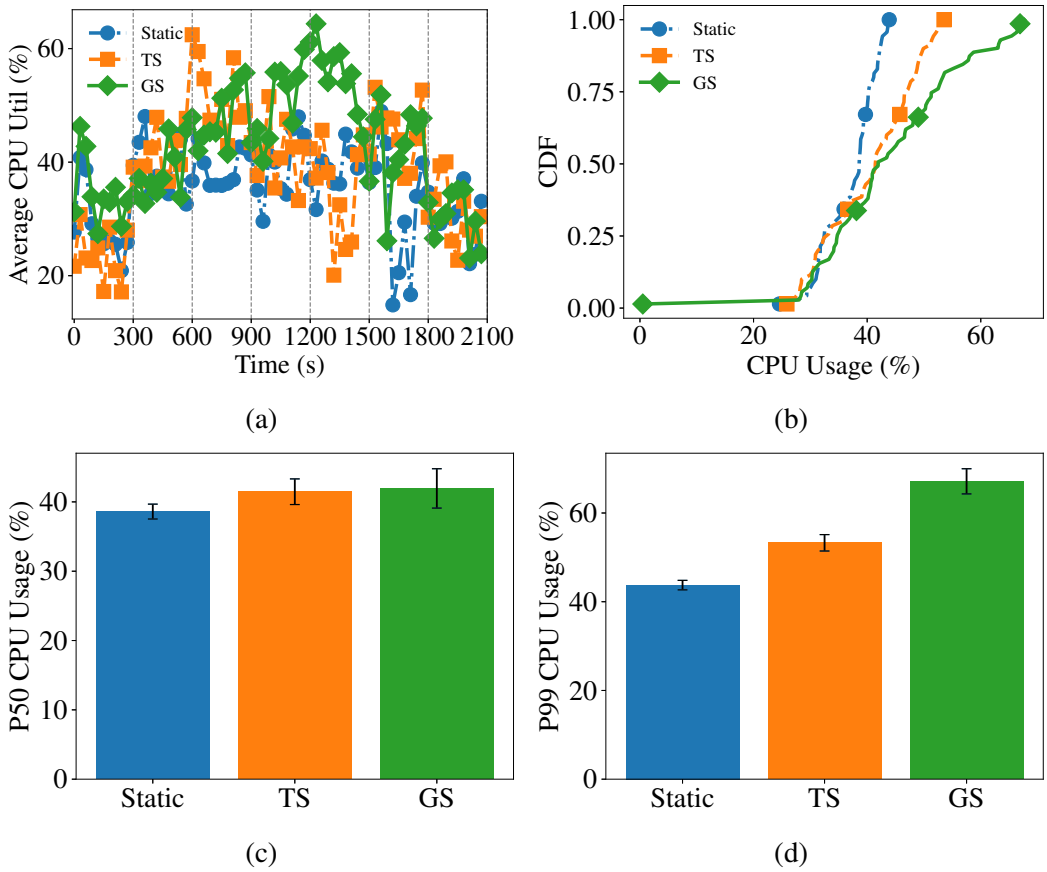

Fig. 8.15: CPU utilization under different deployment algorithms after removing the Wikipedia tool. (a): Sample run over time. (b): CDF from all five runs. (c): P50 results from all five runs. (d): P99 results from all five runs.

ducing the P50 delay by 14.2% and 8.9%, respectively, and lowering the P99 delay by up to 7.9%. These results indicate that dynamic scaling allows GS to react more effectively to workload variations, improving both overall responsiveness and worst-case request latency. *In summary, GS consistently achieves lower delay than static deployment and TS under dynamic workloads.*

GS increases CPU utilization while keeping CPU pressure under control. Fig. 8.15 illustrates the CPU utilization of different deployment algorithms after removing the Wikipedia tool from the workload. As shown in Fig. 8.15(a), GS exhibits higher CPU utilization than static deployment and TS at peak performance, while the CPU utilization generally remains around our predefined CPU utilization threshold θ_c rather than significantly exceeding it. The CPU utilization distributions shown in Fig. 8.15(b) further show that the CPU utilization patterns of GS remain consistent across all five runs.

Moreover, Fig. 8.15(c) and Fig. 8.15(d) show that GS consistently achieves higher CPU utilization than both static deployment and TS across P50 and P99 measurements, with P99 CPU utilization increasing by 53.5% and 26.0%, respectively. Meanwhile, the P99 CPU utilization of GS still remains 12.9% below the predefined CPU utilization threshold θ_c. These results suggest that GS effectively utilizes heterogeneous computing resources while preventing excessive CPU pressure. *In summary, GS improves the system utilization and maintains CPU pressure.*

8.3 Summary of Findings

The experimental results demonstrate the effectiveness of both the SSD routing algorithm and the GS deployment algorithm in improving agent system performance under dynamic workloads.

For routing, SSD consistently achieves the highest request and tool-call completion counts while introducing only marginal computation and network overhead. It also maintains the lowest delay across different workload levels, demonstrating strong scalability as system load increases. Furthermore, the evaluation of SSD design choices shows that larger γ values improve request completion and delay performance at the cost of increased workload imbalance, while the shared-queue design provides superior delay and tool-call completion performance compared to dedicated queues.

For deployment, the results reveal that external service bottlenecks, such as rate-limited tools, may temporarily reduce the effectiveness of adaptive scaling despite increasing the number of instances. Nevertheless, after excluding the impact of these external limitations, GS consistently completes more requests and tool calls, while achieving lower delay than both static deployment and TS. GS also achieves higher CPU utilization while keeping resource usage below the predefined CPU utilization threshold. Overall, the results indicate that SSD and GS complement each other by improving routing efficiency and resource utilization, enabling agent systems to maintain strong performance under dynamic workloads.

Chapter 9 Conclusion

In this chapter, we summarize the key contributions of this thesis toward efficient MCP-native orchestration for LLM-powered AI agents in heterogeneous edge environments. We also discuss several promising future research directions to further enhance the scalability and robustness of agent-driven edge computing systems.

9.1 Concluding Remarks

In this thesis, we addressed the emerging challenges of orchestrating lightweight, fine-grained MCP tools across heterogeneous on-premise edge servers for LLM-powered AI agents. To tackle the complex interplay between high-frequency request routing and periodic resource scaling, we proposed and developed DOME, the first MCP-native edge orchestration framework. Specifically, we designed a delay-aware routing algorithm, called SSD, to dynamically dispatch tool calls based on real-time execution feedback and fine-grained network-processing delay estimations. Furthermore, we formulated the tool deployment optimization problem and introduced a GS algorithm that periodically adapts tool instance placements to evolving user demands and resource constraints without incurring excessive overhead.

To validate our design, we implemented the DOME framework and deployed it on a real-world heterogeneous edge testbed—encompassing a workstation-class root server and resource-constrained SBCs, including NVIDIA Jetson and Raspberry Pis—leveraging carefully chosen libraries such as MCP Python SDK, MCP Proxy, MQTT, and gRPC. Driven by realistic agent workflows from a real-life dataset, our extensive evaluations demonstrated DOME's practicality and efficiency. Notably, under heavy concurrent workloads, the SSD algorithm reduced the P50 and P99 delays by up to 73% and 82.4%, respectively, while completing up to 4.10 times as many tool calls as baselines. Furthermore, the GS algorithm improved system responsiveness and resource utilization through adaptive deployment scaling. Compared with static deployment and TS, GS completed

up to 18.7% more requests and 19.3% more tool calls while reducing delay by up to 14.2% under dynamic workload. At the same time, GS maintained CPU utilization below the predefined utilization threshold, demonstrating effective utilization of heterogeneous edge resources under dynamic workloads.

9.2 Future Work

Our work in this thesis can be extended in the following directions:

Future Research Problem 1 (Large-Scale Heterogeneous Edge Orchestration)**.**

In this thesis, we evaluate DOME on a heterogeneous on-premise edge testbed connected through a local network environment. Although our framework already incorporates network-aware routing and delay estimation, future work can further investigate orchestration behavior under larger-scale and more diverse network topologies, including geographically distributed edge domains, unstable wide-area links, and cross-tier edge-cloud environments. Moreover, our current deployment decisions primarily consider CPU utilization as the main resource indicator and do not explicitly model tool-specific affinity toward particular hardware platforms. In practice, different MCP tools may exhibit diverse bottlenecks, such as memory consumption, storage I/O, GPU acceleration, or network bandwidth usage. Future work can therefore investigate resource-aware placement strategies that learn and exploit tool-specific execution characteristics while jointly considering heterogeneous resource capacities and workload dynamics across edge servers.

Future Research Problem 2 (Agent-Aware Orchestration for Real-World Reasoning Workflows)**.**

In this thesis, the evaluations replay workloads derived from a public MCP dataset to provide reproducible and controlled experimentation. However, practical AI agents often exhibit more complex and dynamic execution behaviors. For example, different LLMs may adopt distinct tool-selection preferences, retry policies, timeout handling strategies, or error-recovery mechanisms when tool executions fail or return incomplete results. Furthermore, emerging multi-agent systems may introduce long-horizon reasoning depen-

dencies, iterative planning loops, and chained tool interactions that significantly affect workload dynamics. In the future, DOME can be integrated with real-world agent frameworks and production-grade agent workflows to better understand how these agent behaviors influence orchestration efficiency, scalability, and system responsiveness.

Future Research Problem 3 (Predictive and Learning-Based Orchestration)**.**

The SSD routing algorithm and GS deployment algorithm proposed in this thesis primarily rely on reactive decisions based on recently observed system states. Although such approaches are effective under dynamic workloads, reactive scaling inevitably incurs adaptation delays when workload surges occur. Future orchestration systems may benefit from predictive mechanisms that anticipate future resource demands using historical execution traces, workload forecasting models, and learning-based optimization techniques. In particular, reinforcement learning and sequence prediction models may enable orchestration policies that jointly reason about agent behaviors, workload evolution, and infrastructure dynamics. Developing predictive orchestration frameworks that remain accurate, stable, and computationally efficient under highly dynamic workloads constitutes an important research challenge.

Future Research Problem 4 (Multi-Objective Orchestration for MCP-Native Systems)**.**

This thesis primarily focuses on improving service responsiveness and balancing resource utilization for MCP tool executions. However, future agent-driven edge infrastructures will likely require orchestration objectives that extend beyond latency optimization alone. For example, fault tolerance and service reliability become increasingly important as orchestration decisions span larger numbers of distributed servers. In practical deployments, servers may fail unexpectedly due to hardware faults, software crashes, or network disconnections, potentially disrupting ongoing tool executions and degrading overall system performance. Furthermore, as MCP tools are developed and deployed by multiple independent parties, concerns regarding execution isolation, trust management, access control, and security assurance become increasingly significant. These objectives may exhibit complex trade-offs and conflicting requirements, making single-objective optimization insufficient. Future research is therefore needed to develop multi-objective or-

chestration frameworks capable of balancing performance, efficiency, reliability, security, and trustworthiness in heterogeneous MCP-native ecosystems.

References

[1] Y. Xu et al., "Toward personalized LLM-powered agents: Foundations, evaluation, and future directions", *arXiv preprint arXiv:2602.22680*, 2026.

[2] T. Schick et al., "Toolformer: Language models can teach themselves to use tools", *Advances in Neural Information Processing Systems*, vol. 36, pp. 68 539–68 551, 2023.

[3] S. Patil, T. Zhang, X. Wang, and J. Gonzalez, "Gorilla: Large language model connected with massive APIs", *Advances in Neural Information Processing Systems*, vol. 37, pp. 126 544–126 565, 2024.

[4] W. Xu, C. Huang, S. Gao, and S. Shang, "LLM-based agents for tool learning: A survey", *Data Science and Engineering*, pp. 1–31, 2025.

[5] G. Wölflein, D. Ferber, D. Truhn, O. Arandjelovic, and J. Kather, "LLM agents making agent tools", in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2025, pp. 26 092–26 130.

[6] A. Yehudai et al., "Survey on evaluation of LLM-based agents", *arXiv preprint arXiv:2503.16416*, 2025.

[7] Anthropic. "Introducing the model context protocol". https://www.anthropic.com/news/model-context-protocol.

[8] R. Surapaneni, M. Jha, M. Vakoc, and T. Segal. "Announcing the Agent2Agent protocol (A2A)". https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability.

[9] S. Besen and A. Gutowska. "What is agent communication protocol (ACP)?" https://www.ibm.com/think/topics/agent-communication-protocol.

[10] AgentNetworkProtocol. "Agent network protocol: The HTTP of the agentic web era". https://www.agent-network-protocol.com/.

[11] A. Ehtesham, A. Singh, G. Gupta, and S. Kumar, "A survey of agent interoperability protocols: Model context protocol (MCP), agent communication protocol

(ACP), agent-to-agent protocol (A2A), and agent network protocol (ANP)", *arXiv preprint arXiv:2505.02279*, 2025.

[12] S. Guan, J. Wang, J. Bian, B. Zhu, J.-G. Lou, and H. Xiong, "Evaluating LLM-based agents for multi-turn conversations: A survey", *ACM Transactions on Intelligent Systems and Technology*, 2025.

[13] H. Sanchez and B. Hitaj, "LLM chemistry estimation for multi-LLM recommendation", *arXiv preprint arXiv:2510.03930*, 2025.

[14] X. J. Wang, C. P. Lee, and B. Mutlu, "LearnMate: Enhancing online education with LLM-powered personalized learning plans and support", in *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, 2025, pp. 1–10.

[15] C. U. Ogdu, S. Gurbuz, M. Karakose, and E. Hanoglu, "Medical implications of LLM based clinical decision support systems in healthcare", in *2025 29th International Conference on Information Technology (IT)*, IEEE, 2025, pp. 1–4.

[16] L. Huang et al., "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions", *ACM Transactions on Information Systems*, vol. 43, no. 2, pp. 1–55, 2025.

[17] J. Chen, H. Wu, J. Pang, Y. Wang, D. Zhang, and C. Sun, "Tool learning with language models: A comprehensive survey of methods, pipelines, and benchmarks", *Vicinagearth*, vol. 2, no. 1, p. 16, 2025.

[18] N. Luo, A. P. Gema, X. He, E. Van Krieken, P. Lesci, and P. Minervini, "Self-training large language models for tool-use without demonstrations", in *Findings of the Association for Computational Linguistics: NAACL 2025*, 2025, pp. 1253–1271.

[19] G. Dong et al., "Tool-Star: Empowering LLM-brained multi-tool reasoner via reinforcement learning", *arXiv preprint arXiv:2505.16410*, 2025.

[20] Z. Shen, "LLM with tools: A survey", *arXiv preprint arXiv:2409.18807*, 2024.

[21] A. Ferrag, N. Tihanyi, and M. Debbah, "From LLM reasoning to autonomous AI agents: A comprehensive review", *arXiv preprint arXiv:2504.19678*, 2025.

[22] H. Xu et al., "The evolution of tool use in LLM agents: From single-tool call to multi-tool orchestration", *arXiv preprint arXiv:2603.22862*, 2026.

[23] "OpenClaw". https://github.com/openclaw/openclaw.

[24] "NemoClaw". https://github.com/NVIDIA/NemoClaw.

[25] Y. Zheng et al., "DeepResearcher: Scaling deep research via reinforcement learning in real-world environments", in *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, 2025, pp. 414–431.

[26] J. Yang et al., "SWE-agent: Agent-computer interfaces enable automated software engineering", *Advances in Neural Information Processing Systems*, vol. 37, pp. 50 528–50 652, 2024.

[27] S. Banerjie et al., "Agentic AI in healthcare: A comprehensive survey of foundations, taxonomy, and applications", *Authorea Preprints*, 2025.

[28] P. Ray, "A survey on model context protocol: Architecture, state-of-the-art, challenges and future directions", *Authorea Preprints*, 2025.

[29] A. Sharma and S. Chaturvedi, "Agentic AI with model context protocol", *ICT Systems and Sustainability: Proceedings of ICT4SD 2025, Volume 2*, vol. 2, p. 433, 2025.

[30] A. Singh, A. Ehtesham, S. Kumar, and T. Khoei, "A survey of the model context protocol (MCP): Standardizing context to enhance large language models (LLMs)", *Preprints*, 2025.

[31] M. A. Shehab, "Agentic-AI healthcare: Multilingual, privacy-first framework with MCP agents", *arXiv preprint arXiv:2510.02325*, 2025.

[32] A. Bhandari, "Multi-scale network dynamics and systemic risk: A model context protocol approach to financial markets", *arXiv preprint arXiv:2507.08065*, 2025.

[33] T. Kumar et al., "Infrastructuresentinel: Policy enforced guardrails for secure MCP-driven infrastructure agents", in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 40, 2026, pp. 40 295–40 301.

[34] P. Giannakopoulos, B. van Knippenberg, K. C. Joshi, N. Calabretta, and G. Exarchakos, "Performance-aware scheduling and load balancing for edge/cloud systems", in *2025 IEEE International Conference on Cloud Engineering (IC2E)*, IEEE, 2025, pp. 290–298.

[35] S. Yerra and M. V. Lakshmi, "Intelligent workload readjustment of serverless functions in cloud to edge environment", in *2025 IEEE 4th World Conference on Applied Intelligence and Computing (AIC)*, IEEE, 2025, pp. 661–666.

[36] C. Calavaro, V. Cardellini, F. Lo Presti, and G. Russo Russo, "Beyond cloud: Serverless functions in the compute continuum", *SN Computer Science*, vol. 6, no. 3, p. 194, 2025.

[37] Y. Li, F. Liu, C.-H. Hsu, and N. Venkatasubramanian, "AMPHI: Adaptive mission-aware microservices provisioning in heterogeneous IoT settings", in *2024 IEEE International Conference on Smart Computing (SMARTCOMP)*, IEEE, 2024, pp. 63–70.

[38] "Playwright MCP server". https://github.com/microsoft/playwright-mcp.

[39] C.-C. Lai, A. Sarkar, P. A. Dinh, S. Gaikwad, and C.-H. Hsu, "Urban Digital-Twin planning for sustainable smart cities: System architecture, preliminary experiments, and open challenges", in *Proceedings of the 2nd International Workshop on Middleware for Digital Twins*, 2024, pp. 7–12.

[40] T.-C. Chang, C.-C. Wu, G. Bouloukakis, C.-H. Hsu, and N. Venkatasubramanian, "SmartParcels: Constructing smart communities through human-in-the-loop urban iot planning", *ACM Transactions on Internet of Things*, vol. 6, no. 2, pp. 1–31, 2025.

[41] C.-C. Wu, R. Rahman, C. Hsu, C.-C. Lai, N. Venkatasubramanian, and C.-H. Hsu, "Evaluating subsurface single-hop wifi and lora networks for the internet of underground things", in *2023 IEEE Globecom Workshops (GC Wkshps)*, IEEE, 2023, pp. 702–707.

[42] R. Rahman, C.-H. Lin, C.-H. Hsu, and N. Venkatasubramanian, "A measurement-informed approach to modeling underground iot communications", in *2024 IEEE 100th Vehicular Technology Conference (VTC2024-Fall)*, IEEE, 2024, pp. 1–7.

[43] Z. Zhou, C.-C. Lai, B. Han, C.-H. Hsu, S. Chen, and B. Li, "CARLA-Twin: A large-scale Digital Twin platform for advanced networking research", in *IEEE INFOCOM 2025-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, IEEE, 2025, pp. 1–6.

[44] C.-.-C. Wu, C.-.-C. Lai, N. Venkatasubramanian, and C.-H. Hsu, "A software-defined approach to enabling network controllers for smart environment Digital Twins", in *2024 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE, 2024, pp. 33–40.

[45] N. Yang et al., "IoT-MCP: Bridging LLMs and IoT systems through model context protocol", in *Proceedings of the ACM Workshop on Wireless Network Testbeds, Experimental evaluation & Characterization*, 2025, pp. 73–80.

[46] A. Ahmadi, S. Sharif, and Y. Banad, "MCP Bridge: A lightweight, LLM-agnostic RESTful proxy for model context protocol servers", *arXiv preprint arXiv:2504.08999*, 2025.

[47] S. Barros, "Extending the model context protocol (MCP) for telco networks", *Available at SSRN 5211843*, 2025.

[48] E. Li, H. Du, and K. Huang, "NetMCP: Network-aware model context protocol platform for LLM capability extension", *arXiv preprint arXiv:2510.13467*, 2025.

[49] Y. Zeng and H. Du, "M3LLM: Model context protocol-aided mixture of vision experts for multimodal LLMs in networks", *arXiv preprint arXiv:2508.01805*, 2025.

[50] E. Lumer, A. Gulati, V. Subbiah, P. Basavaraju, and J. Burke, "ScaleMCP: Dynamic and auto-synchronizing model context protocol tools for LLM agents", *arXiv preprint arXiv:2505.06416*, 2025.

[51] J. Qiu et al., "Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution", *arXiv preprint arXiv:2505.20286*, 2025.

[52] V. Narajala, K. Huang, and I. Habler, "Securing GenAI multi-agent systems against tool squatting: A zero trust registry-based approach", *arXiv preprint arXiv:2504.19951*, 2025.

[53] Y. Guo, G. Zhu, K. Liu, and G. Xing, "SensorMCP: A model context protocol server for custom sensor tool creation", in *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services*, 2025, pp. 747–752.

[54] A. Ullah et al., “Towards a decentralised application-centric orchestration framework in the cloud-edge continuum”, in *2025 IEEE 9th International Conference on Fog and Edge Computing (ICFEC)*, IEEE, 2025, pp. 37–41.

[55] A. Pashaeehir, S. Shariati, S. Shafaghi, M. Moghimi, and M. Momtazpour, “KubeDSM: A Kubernetes-based dynamic scheduling and migration framework for cloud-assisted edge clusters”, *arXiv preprint arXiv:2501.07130*, 2025.

[56] S. Song, M. Xu, K. Hu, W. Guo, and K. Ye, “TD3-Sched: Learning to orchestrate container-based cloud-edge resources via distributed reinforcement learning”, *arXiv preprint arXiv:2509.18957*, 2025.

[57] W. Liu et al., “Resource scheduling algorithm for edge computing networks based on multi-objective optimization”, *Applied Sciences*, vol. 15, no. 19, p. 10 837, 2025.

[58] P. Giannakopoulos, B. Knippenberg, C. Joshi, N. Calabretta, and G. Exarchakos, “Morpheus: Lightweight RTT prediction for performance-aware load balancing”, *arXiv preprint arXiv:2510.20506*, 2025.

[59] O. Michaelis, S. Schmid, and H. Mostafaei, “L3: Latency-aware load balancing in multi-cluster service mesh”, in *Proceedings of the 25th International Middleware Conference*, 2024, pp. 49–61.

[60] K. Peng et al., “Delay-aware optimization of fine-grained microservice deployment and routing in edge via reinforcement learning”, *IEEE Transactions on Network Science and Engineering*, vol. 11, no. 6, pp. 6024–6037, 2024.

[61] L. Wang, X. Liu, H. Ding, Y. Hu, K. Peng, and M. Hu, “Energy-delay-aware joint microservice deployment and request routing with DVFS in edge: A reinforcement learning approach”, *IEEE Transactions on Computers*, vol. 74, no. 5, pp. 1589–1604, 2025.

[62] L. Wang et al., “A survey on large language model based autonomous agents”, *Frontiers of Computer Science*, vol. 18, no. 6, p. 186 345, 2024.

[63] modelcontextprotocol. “MCP Python SDK”. https://github.com/modelcontextprotocol/python-sdk.

[64] tbxark. “mcp-proxy”. https://github.com/TBXark/mcp-proxy.

[65] eclipse-paho. “Paho.mqtt.python”. https://github.com/eclipse-paho/paho.mqtt.python.

[66] gRPC. "gRPC". https://grpc.io/.

[67] R. V. Rasmussen and M. A. Trick, "Round Robin scheduling–a survey", *European Journal of Operational Research*, vol. 188, no. 3, pp. 617–636, 2008.

[68] M. Mitzenmacher, "The power of two choices in randomized load balancing", *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094–1104, 2002.

[69] M. Harchol-Balter, *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press, 2013.

[70] Z. Lin, B. Ruan, J. Liu, and W. Zhao, "A large-scale evolvable dataset for model context protocol ecosystem and security analysis", *arXiv preprint arXiv:2506.23474*, 2025.

[71] M. Wu et al., "MCPZoo: A large-scale dataset of runnable model context protocol servers for AI agent", *arXiv preprint arXiv:2512.15144*, 2025.

[72] X. Fei, X. Zheng, and H. Feng, "MCP-Zero: Active tool discovery for autonomous llm agents", *arXiv preprint arXiv:2506.01056*, 2025.

[73] Z. Wang et al., "MCP-Bench: Benchmarking tool-using LLM agents with complex real-world tasks via MCP servers", *arXiv preprint arXiv:2508.20453*, 2025.

[74] Z. Luo et al., "MCP-Universe: Benchmarking large language models with real-world model context protocol servers", *arXiv preprint arXiv:2508.14704*, 2025.

[75] Z. Xu et al., "TOUCAN: Synthesizing 1.5M tool-agentic data from real-world MCP environments", *arXiv preprint arXiv:2510.01179*, 2025.

[76] C. Bandi et al., "MCP-Atlas: A large-scale benchmark for tool-use competency with real MCP servers", *arXiv preprint arXiv:2602.00933*, 2026.

[77] G. Mo et al., "LiveMCPBench: Can agents navigate an ocean of MCP tools?", *arXiv preprint arXiv:2508.01780*, 2025.