Orchestrating WASM-based MCP Tool Runtimes for AI Agents across Edge-Cloud Continuum

Moohyun Song
Department of Artificial Intelligence
Hanyang University
Seoul, Korea
moohyunsong@hanyang.ac.kr

Hayoung Kim
Department of Data Science
Hanyang University
Seoul, Korea
kimhayoung@hanyang.ac.kr

Kyoohyun Lee
Department of Artificial Intelligence
Hanyang University
Seoul, Korea
khyhlee@hanyang.ac.kr

Jae Gi Son
Korea Electronics Technology Institute
Seoul, Korea
jgson@keti.re.kr

Kyungyong Lee[†]
Department of Data Science
Hanyang University
Seoul, Korea
kyungyong@hanyang.ac.kr

***Abstract*—While the Model Context Protocol (MCP) standardizes logical agent-tool interactions, current container-based execution models fail within the Edge-Cloud Continuum, where edge devices are particularly resource-constrained. Since persistent deployment is infeasible on the edge, the reliance on heavy containers for on-demand execution causes prohibitive cold-start latencies, disrupting real-time agentic workflows. To bridge this gap, we propose an orchestration framework that decouples tool semantics from execution infrastructure, informed by our thorough analysis of MCP ecosystem heterogeneity. Based on these insights, we introduce the *Agent Tool Dispatcher*, a middleware that exposes detailed execution metrics to facilitate precise workload placement across Device, Edge, and Cloud tiers. Complementing this, we present WasmMCP, a WebAssembly-based runtime achieving architecture independence and rapid cold starts, leveraging Wasm's sub-millisecond module instantiation to deliver end-to-end startup in tens of milliseconds to enable efficient deployment of MCP tools across constrained heterogeneous devices. Extensive evaluation using realistic agent scenarios demonstrates that WasmMCP achieves 10–21$\times$ faster cold starts compared to container-based baselines. Furthermore, our dispatcher achieves zero constraint violations while maintaining competitive end-to-end latency, ensuring seamless and scalable agent operations across edge-cloud continuum.**

***Index Terms*—WebAssembly, Edge-Cloud Continuum, Model Context Protocol, AI Agent**

I. Introduction

Recent advances in Large Language Models (LLMs) have extended the scope of generative AI from static natural language processing to dynamic, goal-oriented execution, enabling the emergence of autonomous agents. Unlike traditional models constrained by pre-trained knowledge bases, these agents leverage iterative reasoning capabilities to decompose complex objectives into executable plans and interact with external environments. This architectural evolution facilitates the deployment of LLMs in sophisticated domains, where the primary utility shifts from semantic knowledge retrieval to the autonomous execution of multi-step workflows.

As agents progress toward handling increasingly complex tasks, their operational capability becomes bounded by the effective utilization of external tools that are the interfaces bridging the model's reasoning with external APIs, enterprise databases, and IoT devices. However, the heterogeneity of proprietary interfaces has introduced a significant interoperability gap. Addressing the $N \times M$ integration complexity between diverse agent frameworks (e.g., LangChain, AutoGen) and an expanding tool ecosystem has traditionally necessitated the maintenance of ad-hoc adaptation layers. To mitigate this challenge, the Model Context Protocol (MCP) [1] has emerged as a unified industry standard. By defining a universal schema for tool discovery and invocation, MCP decouples the agent's planning from low-level tool implementations, facilitating the development of modular and interoperable agentic systems.

While MCP successfully standardizes the logical interaction between agents and tools, it remains ignorant of the physical execution environment. As agents are increasingly integrated into real-world applications requiring low latency and data privacy, their operation inevitably spans a distributed Edge-Cloud Continuum. However, deploying across this heterogeneous landscape introduces severe architectural hurdles that the protocol alone cannot address.

Current MCP implementations predominantly assume homogeneous environments, typically local desktops or centralized cloud servers, where high-bandwidth connectivity and uniform hardware are guaranteed. However, real-world IoT and edge scenarios are characterized by Instruction Set Architecture (ISA) heterogeneity (e.g., mixing x86 servers with ARM/RISC-V sensors or devices) and constrained network conditions. In such resource-constrained environments, maintaining permanently running containers is infeasible, necessitating an on-demand execution model. However, the standard practice of packaging tools as heavyweight containers incurs

[†]Corresponding author

substantial initialization overhead. This results in prohibitive cold-start latencies that disrupt the real-time responsiveness required for interactive agentic workflows [2], [3], hindering the seamless migration of agents to the network edge.

To address these challenges, we began by conducting a comprehensive analysis of the rapidly evolving MCP tool ecosystem. We observed that while tools share a unified logical interface, they exhibit distinct operational characteristics, ranging from lightweight data transformations to heavy hardware-dependent computations. Existing architectures, however, force a static binding between tools and agents, ignoring these heterogeneity and leading to inefficient resource utilization. To resolve this, we introduce the *Agent Tool Dispatcher*, a profile-based orchestration layer that routes tool execution on the appropriate infrastructure tier. By mapping agent tool requirements to the corresponding infrastructure tier in the edge-cloud continuum, this module formalizes high-level tool semantics into quantitative execution parameters, enabling the scheduler to optimize placement decisions.

To execute the placement decisions made by the Dispatcher, we developed *WasmMCP*, a lightweight serverless runtime. On resource-constrained edge devices, maintaining persistent processes for every potential tool execution is infeasible, and an on-demand execution model is required to preserve limited available resources, such as memory. However, traditional container runtimes impose significant overhead that conflicts with this intermittent execution model, incurring prohibitive cold-start latencies. WasmMCP addresses this by leveraging a lightweight WebAssembly (WASM) [4] runtime to abstract underlying hardware heterogeneity, enabling architecture-independent deployment across diverse edge architectures. By implementing a custom stripped-down initialization path and a dual-transport architecture, WasmMCP achieves rapid cold starts in tens of milliseconds, allowing the efficient hosting of many idle tools on resource-constrained devices with marginal operational overheads.

We validate the proposed architecture through a full-scale system implementation integrated with standard agent frameworks to our custom Wasm runtime and the dispatcher module. We evaluate the system using representative agentic workflows derived from real-world usage scenarios. Our empirical results demonstrate that WasmMCP reduces cold-start latency by 10–21$\times$ and memory footprint by 59–72% compared to container-based baselines. The Agent Tool Dispatcher achieves zero constraint violations across all scenarios while reducing end-to-end latency by up to 2.3$\times$ compared to naive cloud-only deployment, effectively balancing the trade-offs between data locality and computational resources.

In summary, our key contributions are as follows:

- **MCP Ecosystem Analysis:** Analyzing the heterogeneity of MCP workloads to identify the limitations of static deployment in edge-cloud environments.
- **Agent Tool Dispatcher:** Proposing a middleware that dynamically optimizes tool placement across device, edge, and cloud tiers based on workload characteristics.

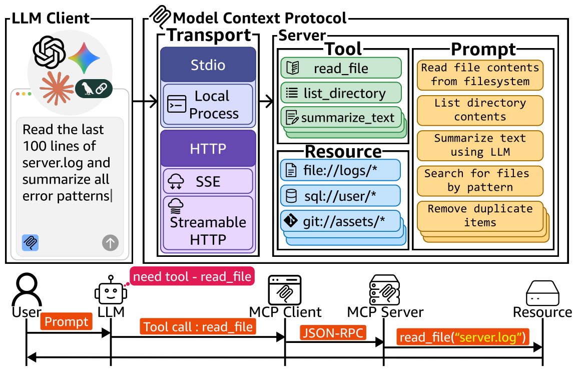


Fig. 1: Overview of MCP primitives and transport mechanisms with tool invocation flow

- **WasmMCP Runtime:** Developing a Wasm-based serverless runtime for MCP tools that leverages sub-millisecond Wasm instantiation to achieve end-to-end cold starts of tens of milliseconds on constrained devices.

II. Model Context Protocol Ecosystem

The MCP [1], released by Anthropic [5] late 2024, establishes an open standard interface for LLMs to securely access external data and tools. Prior to MCP, integrating N LLM applications with M external services required $N \times M$ custom-built adapters. MCP standardizes these external services as *Servers* and LLM applications as *Clients*, defining a standard interface based on JSON-RPC 2.0 that reduces integration complexity to $O(N + M)$.

A. MCP Primitives and Communication Model

Figure 1 illustrates the MCP architecture and communication flow. MCP server is composed of three primary components of *Tools*, *Resources*, and *Prompts*. *Tools* represent executable functions for the LLM, whereas *Resources* expose data sources such as file systems or database records. This distinction is critical for scheduling in distributed environments, as the location of a *Resource* (e.g., a local log file) imposes data locality constraints on the corresponding *Tool* (e.g., a log analyzer), dictating where the computation must occur. *Prompts* serve as reusable templates for interaction.

The protocol supports two primary *Transport* mechanisms which dictate the deployment model. The *stdio transport* utilizes standard input/output streams for local inter-process communication (IPC). While serving as the default for local host applications, this approach enforces tight coupling, mandating that the tool and the LLM client reside on the same physical machine. Conversely, the *HTTP transport* facilitates remote execution via Server-Sent Events (SSE) or Streamable HTTP. The standard implementations predominantly rely on SSE for bidirectional messaging. Unlike the stateless Streamable HTTP model, SSE necessitates long-lived connections to push updates. This approach implicitly assumes an always-on server architecture, incurring continuous resource overhead

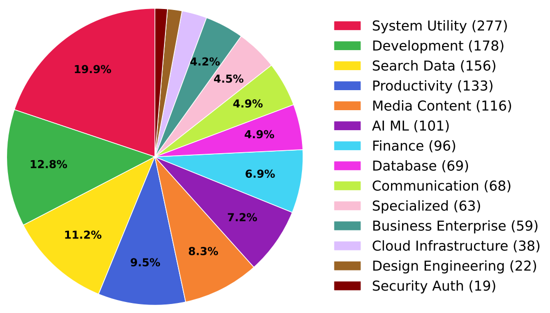


Fig. 2: Category of MCP tools collected from 1,467 servers

even when tools are idle that can become a significant limitation for efficient distributed deployment.

Crucially, existing MCP servers often suffer from implementation-level coupling, where the transport layer is hard-coded at build time. This prevents a tool designed for local usage (stdio) from being seamlessly repurposed for remote access (HTTP) without code modifications, severely limiting deployment flexibility.

B. Empirical Analysis of MCP Ecosystem

To understand the ecosystem's characteristics, we conducted an empirical analysis using data from major registries including Smithery [6], DeepNLP [7], and PulseMCP [8].

1) Data Collection and Dataset Overview: To extract precise execution definitions beyond simple metadata, we attempted to initialize each server in an isolated environment and invoke the `tools/list` method. However, only 1596 servers (approximately 18%) were successfully executed. The majority failed due to missing API keys, complex environmental configurations, and unresolved runtime dependencies where such a ratio is observed in other prior work [9]–[11]. By aggregating and deduplicating these validated entries based on repository URLs, we constructed a final dataset comprising 17,566 distinct tools from 1,467 unique servers.

We classified these servers into 14 major categories using a keyword-based approach [12]. As shown in Figure 2, the ecosystem is dominated by System Utility (19.9%), Development (12.8%), and Search Data (11.2%). This distribution suggests that MCP is currently evolving beyond simple API wrappers into a domain-agnostic interface for handling heterogeneous and system-intensive workloads.

To understand execution characteristics of each tool, we profiled representative tools in container environments on an Intel NUC (i5-1240P, 16GB RAM, 1Gbps Network). Figure 3 presents the resource utilization breakdown for each tool, showing the proportion of execution time spent on Disk I/O, Network I/O and CPU computation. The results reveal significant workload heterogeneity.

2) Key Findings: Based on a thorough analysis of the dataset, we identified three critical design implications for building scalable and efficient tool execution systems.

a) Finding 1: Heterogeneous resource demands necessitate flexible workload placement: As presented in Figure 3, tool execution profiles exhibit significant heterogeneity. Network-bound tools (e.g., *fetch*, *summarize_text*) spend over 90% of execution time on network I/O awaiting external API responses. Compute-intensive tools (e.g., *resize_image*, *image_hash*) are dominated by CPU processing. Furthermore, advanced workloads like AI inference may benefit significantly from hardware acceleration, while I/O-bound tools (e.g., *read_file, write_file*) incur high disk I/O ratio, favoring execution proximate to the data source. This operational diversity indicates that a monolithic deployment strategy is insufficient, and an optimal system must support dynamic workload distribution tailored to these distinct resource characteristics.

b) Finding 2: Static transport binding hinders efficient resource scaling: Our analysis of MCP servers registered on Smithery reveals a clear separation in transport implementation. Approximately 22.3% of tools exclusively support Stdio (local-only), while 77.7% rely solely on HTTP (remote-only). Notably, no servers in our dataset support both transports simultaneously. This static binding forces a rigid build-time determination. Stdio-bound tools cannot be offloaded to the cloud to alleviate local congestion, while HTTP-bound tools necessitate an always-on server architecture. Given the inherently sporadic nature of LLM tool invocation, maintaining persistent servers for these HTTP tools results in significant resource wastage. To resolve this, a unified runtime capable of dynamically switching between local and remote execution without code modification is essential.

c) Finding 3: Functional metadata lacks operational context for scheduling: Figure 4 illustrates a typical MCP tool definition. Current MCP schemas effectively describe *what* a tool does (functional semantics via name and description) but fail to specify *how* and *where* it should optimally run (operational requirements). For instance, the schema does not distinguish between network-agnostic utilities and tools sensitive to data locality or specific hardware affinities. This absence of explicit operational metadata prevents tool orchestrators from making informed, locality-aware placement decisions.

III. Challenges of Running MCP Servers in Edge-Cloud Continuum

While our findings confirm the necessity of an Edge-Cloud Continuum to handle heterogeneous tool workloads [13], [14], transitioning from the current static MCP model to a dynamic distributed environment presents three fundamental challenges of the tight coupling between tool selection and execution placement, latency overhead of conventional serverless runtimes, and the lack of support for lightweight environments.

A. Mixing Tool Selection with Placement

A naive approach to distributing tools to edge-cloud continuum environment is to expose location-specific endpoints to the LLM (e.g., `read_file_edge` vs. `read_file_cloud`). However, this creates a combinatorial explosion in the context window. As motivated by prior

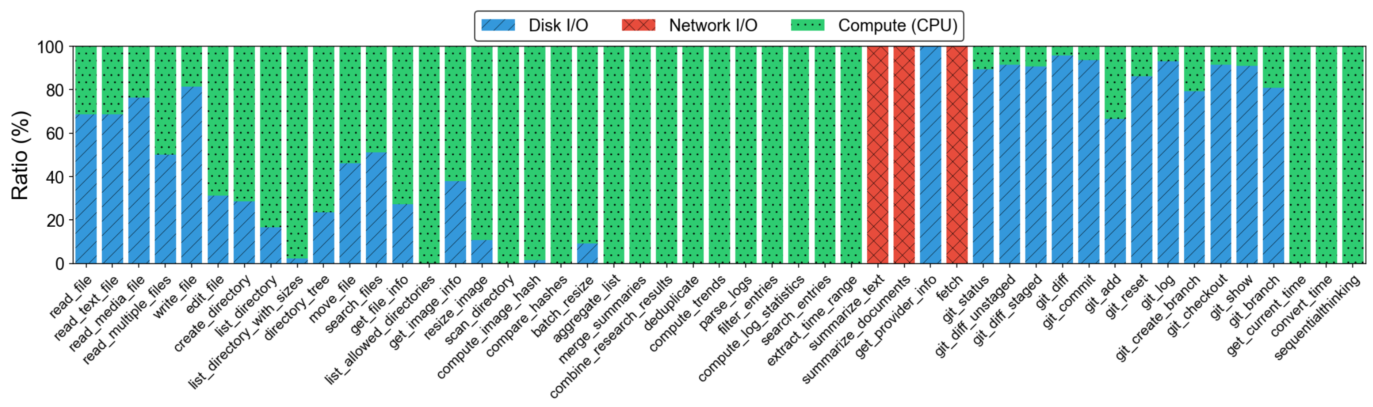


Fig. 3: Resource utilization breakdown of MCP tools showing Disk I/O, Network I/O, and Compute (CPU) ratios

```
{
  "name": "create_or_update_file",
  "description": "Create or update a single file in
  ↪  a repository",
  "input_schema": {
    "type": "object",
    "properties": {
      "owner": {"type": "string"},
      "repo": {"type": "string"}
    },
    "required": ["owner", "repo", "path",
    ↪  "content"]
  }
}
```

Fig. 4: Example of MCP tool schema from Git MCP Server

works [15], for example, replicating a baseline suite of 25 tools across three tiers (Device, Edge, Cloud) triples the search space to 75 endpoints. Such expansion is critical because LLM tool selection accuracy degrades sharply as the candidate set grows, leading to increased hallucination [16].

More fundamentally, this approach conflates two orthogonal concerns of *semantic selection* (WHAT to run) and *operational placement* (WHERE to run). LLMs are designed to interpret user intent based on semantic context, not to optimize infrastructure scheduling based on real-time system states (e.g., network latency, hardware load). As identified in Finding 3, current tool definitions lack the necessary operational metadata. Consequently, delegating placement decisions to the LLM not only increases cognitive overhead but also results in suboptimal execution due to the model's blindness to infrastructure constraints.

Challenge 1: Lack of Tool's Operational Metadata. Current MCP schemas provide only functional metadata (e.g., name, description), but lack essential operational attributes. Without this context, tool scheduling remain blind to infrastructure states, preventing optimized placement in the continuum.

B. Latency and Portability Trade-offs in Serverless Computing

To mitigate the resource overhead of maintaining persistent servers for HTTP-bound tools (Finding 2), a serverless execution model is theoretically ideal for sporadic workloads [17]. However, standard container-based serverless runtimes introduce unacceptable trade-offs for interactive agents workloads.

a) The Cold-Start Issue: Interactive agents require near real-time responsiveness. Conventional container runtimes (e.g., Docker, Kubernetes) typically incur initialization overheads ranging from hundreds of milliseconds to seconds [2], [3]. In multi-step agentic workflows where tools are invoked sequentially, these accumulated cold-start delays may violate strict latency requirements, rendering standard serverless platforms impractical for interactive use cases.

b) Heterogeneous System Architecture: Unlike homogeneous cloud data centers, the edge-cloud continuum is inherently heterogeneous across diverse Instruction Set Architectures (ISAs) such as x86, ARM, and RISC-V [18]. Relying on native binaries or standard containers necessitates maintaining separate build artifacts for every target architecture [19]. This requirement for architecture-specific builds creates a prohibitive maintenance burden hindering the seamless migration of tools between edge devices and cloud nodes.

Challenge 2: Inefficiency of Conventional Runtimes. Existing container-based serverless runtimes incur prohibitive overheads for interactive agents workloads [20]. The heavy initialization causes cold-start delays that violate real-time responsiveness, while the tight coupling to specific CPU architectures (e.g., x86 vs. ARM) creates a operational complexity, hindering transparent tool execution across distinct edge-cloud nodes.

C. Integration Challenges with Emerging Runtimes

To address the aforementioned runtime limitations, WebAssembly (Wasm) [4] emerges as a promising solution. Wasm is a portable binary instruction format designed for secure and sandboxed execution at near-native speeds, making it inherently independent of underlying hardware architectures. Ideally, porting MCP servers to Wasm would resolve both the cold-start latency [21], [22] and the ISA heterogeneity issues. However, adopting Wasm for MCP introduces a significant integration challenge.

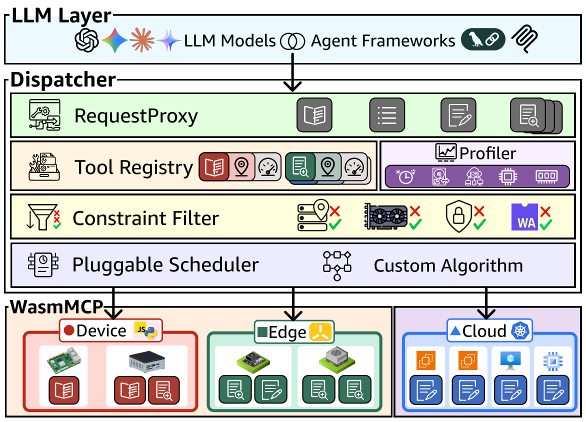


Fig. 5: Overall architecture of the proposed system

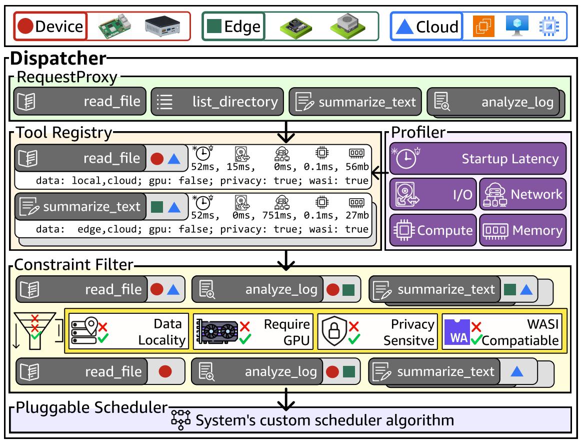


Fig. 6: Agent Tool Dispatcher architecture and request flow

a) Incompatibility with Current SDKs: Current MCP Software Development Kits (SDKs) are designed for long-running, heavyweight OS processes. They rely heavily on POSIX primitives, such as threading, signals, and raw sockets, that are either absent or restricted in the strict WebAssembly System Interface (WASI) [23], [24] that is the standardized API designed to allow Wasm to access system resources. Existing workarounds, such as running a sidecar proxy process to bridge HTTP to Wasm (e.g., `mcp-proxy` [25]), reintroduce the complexity and resource overhead that Wasm aims to eliminate. Thus, the lack of a WASI-native protocol implementation remains a primary barrier to deploying lightweight, standalone MCP modules.

Challenge 3: Absence of Wasm-Native Protocol Support. Standard MCP SDKs rely on heavyweight OS primitives that are incompatible with the strict WASI sandbox. Consequently, adopting Wasm currently necessitates inefficient sidecar proxies to bridge connectivity, which negates the resource and security benefits of Wasm and hinders the deployment of Wasm to standalone, lightweight serverless runtimes.

IV. Overall System Design

To address the aforementioned critical challenges, we propose an workload-aware Wasm-native orchestration framework tailored for distributed MCP tool execution. The proposed system is built upon the tight integration of two core components of **Agent Tool Dispatcher**, a workload-characteristics-aware orchestration layer designed to dynamically route tool execution across the device-edge-cloud continuum, and the **WasmMCP**, a WASI-native framework that enables the deployment of standalone, resource-efficient tool modules across heterogeneous resources.

The proposed architecture is guided by the principle of *Separation of Concerns*. We strictly decouple the *semantic intent* (WHAT to run) from the *physical execution* (WHERE to run). This abstraction allows the system to scale across heterogeneous continuum tiers without requiring modifications to the agent's cognitive logic.

The proposed architecture is presented in Figure 5 that is organized into three distinct layers, each serving a specialized role in the execution pipeline.

- LLM Agent Layer : This layer hosts the cognitive agent (e.g., LangChain) responsible for task planning. It interacts only with logical tools, remaining agnostic to the underlying infrastructures. The abstraction removes the need for modifications to the existing agent logic.
- Agent Tool Dispatcher : Acting as the central control plane, this layer intercepts logical tool calls from the *LLM Agent Layer* and resolves them to optimal physical instances. It utilizes rich workload profiles to assess infrastructure constraints and determines the most appropriate execution tier of Device, Edge, or Cloud. Key components include the *RequestProxy* to decouple the LLM agent layer from execution complexity, *Tool Registry* for metadata management, *Constraint Filter* for feasibility checks, and a *Pluggable Scheduler* for final placement decisions (detailed in Section V).
- Unified Execution Layer : This layer provides a consistent execution environment across heterogeneous continuum. It hosts physical tools deployed either as conventional containers or as ephemeral Wasm modules using the proposed WasmMCP runtime (detailed in Section VI).

V. Context-Aware Agent Tool Dispatcher

The *Agent Tool Dispatcher* functions as a specialized control plane that decouples semantic tool selection from physical resource placement. As shown in Figure 6, the dispatcher orchestrates the execution pipeline through distinct phases of relaying, filtering, and placement. This architecture allows the system to enforce hard constraints (e.g., privacy) as well as soft constraints (e.g., latency) via a pluggable interface.

a) Transparent Relaying via RequestProxy: The *RequestProxy* serves as a logical abstraction layer, exposing invocation stubs to the LLM. Generated from the *Tool Registry*, these stubs mask the complexity of distributed infrastructure, allowing the agent to interact with a single logical interface.

By intercepting calls and redirecting them to the internal pipeline, this abstraction reduces the agent's complexity from $\mathcal{O}(N \times M)$ to $\mathcal{O}(N)$, where N is the number of tools and M is the number of execution nodes, effectively preventing hallucination caused by excessive tool candidates.

b) Metadata Management: The *Tool Registry* acts as the central directory mapping tools (e.g., `read_file`) to multiple physical endpoints (Device/Edge/Cloud). It stores static workload description with dynamic workload profiles. To support informed decision-making, the adjacent *Profiler* continuously captures runtime metrics, such as startup latency, I/O throughput, and network overhead, from tool responses. These metrics are fed back into the registry, enabling the system to refine profiles based on historical performance.

c) Constraint-First Filtering: Before selecting the target resource, the *Constraint Filter* prunes the search space by validating requests against hard constraints. It evaluates attributes such as *data_locality* (e.g., ensuring sensor data is processed on-device) or *privacy_level*. As shown in Figure 6, infeasible nodes, such as those lacking specific hardware or violating privacy policies, are discarded immediately.

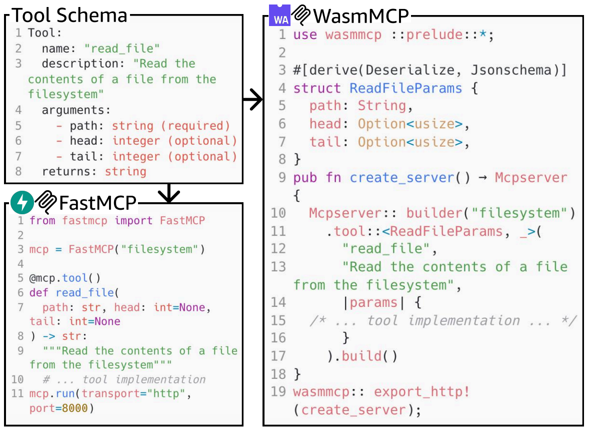
d) Pluggable Scheduler and Cost Model: The *Pluggable Scheduler* determines the final execution tier for the filtered candidates. Designed as an extensible interface, it supports diverse algorithms ranging from simple heuristics to complex optimization models. In this implementation, we adopt a cost-based greedy scheduler. It calculates a total cost $\text{Score}(i, u)$ for executing tool i on node u as defined in Equation (1).

$$\begin{aligned}\text{Score}(i, u) = \alpha_i \big(P_{\text{exec}}(i, u) + \beta_i \cdot P_{\text{net}}(u)\big) \\ +(1 - \alpha_i) \cdot P_{\text{comm}}(u)\end{aligned} \tag{1}$$

P_{exec} represents the normalized execution latency, and P_{comm} denotes the communication overhead between the dispatcher and the node. The term β_i acts as an indicator for network-bound tools, adding external network latency $P_{net}(u)$ to the cost. The weight α_i allows the system to prioritize either computation or data transfer efficiency based on the tool type.

Though we implement a simple scheduling algorithm, it is important to note that the primary objective of this dispatcher is not to enforce a single static scheduling logic, but to serve as an enabling framework. By providing tool profiles and system metrics into the edge-cloud continuum, it empowers developers to implement and test diverse optimization scheduling strategies tailored to specific application needs.

For instance, when a tool's profiled execution latency on Wasm significantly exceeds that of a container (e.g., compute-intensive tasks with large inputs), this disparity is naturally captured by $P_{\text{exec}}(i, u)$, which varies across nodes with different runtime backends. The scheduler can thus route execution to container-based nodes, effectively combining the fast cold starts of Wasm for lightweight tools with the native performance of containers for heavy workloads.



```
Tool:
  name: "read_file"
  description: "Read the contents of a file from the filesystem"
  arguments:
    - path: string (required)
    - head: integer (optional)
    - tail: integer (optional)
  returns: string
```

```
from fastmcp import FastMCP

mcp = FastMCP("filesystem")

@mcp.tool()
def read_file(
  path: str, head: int=None, tail: int=None
) -> str:
  """Read the contents of a file from the filesystem"""
  # ... tool implementation
mcp.run(transport="http", port=8000)
```

```
use wasmmcp ::prelude::*;

#[derive(Deserialize, Jsonschema)]
struct ReadFileParams {
  path: String,
  head: Option<usize>,
  tail: Option<usize>,
}
pub fn create_server() → Mcpserver {
  Mcpserver:: builder("filesystem")
    .tool::<ReadFileParams, _>(
      "read_file",
      "Read the contents of a file from the filesystem",
      |params| {
  /* ... tool implementation ... */
      }
    ).build()
}
wasmmcp:: export_http!(create_server);
```

Fig. 7: Comparison of MCP tool implementation between FastMCP and WasmMCP

VI. WasmMCP: Portable Serverless Runtime for Cloud-Edge Environments

While the *Agent Tool Dispatcher* serves as the intelligent control plane, the overall system latency is ultimately bound by the underlying execution engine. Conventional container runtimes impose significant storage footprints and initialization delays, making them ill-suited for the bursty, high-frequency nature of agentic tool use on the edge. To address this, **WasmMCP** is designed as a lightweight execution engine for high-density tool deployment, exploiting Wasm's sub-millisecond module instantiation to achieve end-to-end startup in tens of milliseconds.

A. Hybrid Execution Strategy

We adopt a hybrid runtime strategy to address the inherent heterogeneity of tool requirements. Heavyweight tools necessitating specific hardware acceleration (e.g., CUDA) or extensive dependencies (e.g., complex Python/Node.js ecosystems) are dispatched to standard container runtimes pinned to cloud or high-performance edge nodes. For the majority of lightweight, stateless logic, we utilize the proposed WasmMCP. Compiling tools into `wasm32-wasip2` binaries achieves near-native execution speeds with an orders-of-magnitude reduction in resource footprint. This allows the Dispatcher to pack hundreds of idle tool functions on a resource-constrained device, waking them only upon invocation without risking memory saturation.

B. WasmMCP Architecture and Implementation

To facilitate this hybrid execution model, WasmMCP incorporates three key architectural mechanisms of dual-transport capability, compile-time schema derivation, and a stripped-down initialization path.

a) WASI-Native Dual-Transport: A critical limitation of existing Wasm implementations is the lack of native networking support, often forcing reliance on inefficient sidecar proxies. WasmMCP overcomes this by embedding dual-transport capabilities directly within the binary. Leveraging Rust's feature flag system, developers can compile the same source

TABLE I: MCP Server Configuration and Static Mapping

MCP Server	Provided Tools	Static Mapping	Rationale
filesystem	read_file, write_file, read_multiple_files, list_directory, list_directory_with_sizes, directory_tree, search_files, get_file_info, create_directory, move_file, edit_file, read_text_file, read_media_file, list_allowed_directories	DEVICE	Local data affinity
git	git_status, git_log, git_diff, git_diff_staged, git_diff_unstaged, git_show, git_commit, git_add, git_reset, git_create_branch, git_checkout, git_branch	DEVICE	Local repository
image_resize	scan_directory, get_image_info, resize_image, batch_resize, compute_image_hash, compare_hashes	EDGE	Compute at data
data_aggregate	aggregate_list, merge_summaries, combine_research_results, deduplicate, compute_trends	EDGE	Intermediate processing
log_parser	parse_logs, filter_entries, compute_log_statistics, search_entries	EDGE	Preprocessing
summarize	summarize_text, summarize_documents, get_provider_info	CLOUD	External LLM API
fetch	fetch	CLOUD	External data access

TABLE II: Scenario Overview

ID	Scenario	Dataset	Used MCP Servers
S1	Code Review	Defects4J	git, filesystem, summarize
S2	Log Analysis	Loghub	filesystem, log_parser, summarize
S3	Research Assistant	S2ORC	fetch, summarize, filesystem
S4	Image Processing	COCO	filesystem, image_resize, data_aggregate

code into two distinct artifacts of *CLI Mode* (`wasi:cli`) for local stdio execution and *HTTP Mode* (`wasi:http`) for standalone serverless deployment. This unified codebase approach eliminates the need for external proxies and IPC overhead, significantly reducing deployment complexity.

b) Compile-Time Schema Derivation: To minimize implementation complexity, WasmMCP offers the syntactic expressiveness of high-level dynamic frameworks (e.g., FastMCP) within a statically typed environment. Figure 7 compares tool implementation between FastMCP (Python) and WasmMCP (Rust). Adopting the declarative programming paradigm of FastMCP, WasmMCP brings the ease of development offered by FastMCP to the WASM ecosystem, leveraging procedural macros to automatically derive JSON schemas from Rust struct definitions. This enables compile-time validation and type-safe tool development unlike decorator-based runtime schema generation. Furthermore, existing Python or Node.js MCP servers can be easily migrated to WasmMCP with assistance from AI coding tools, which automate the bulk of type conversion and Rust pattern generation, with subsequent manual debugging as needed.

c) Optimized Cold-Start Latency: To meet the strict latency requirements of interactive agents, WasmMCP implements a custom, lightweight JSON-RPC handler. Unlike standard SDKs that initialize heavy asynchronous runtimes and thread pools, WasmMCP strips away persistent connection logic redundant for stateless functions. This optimization ensures negligible startup latency, effectively solving the cold-start issue inherent in serverless architectures [26], [27].

VII. Implementation Details

The prototype of the proposed system comprises a Python-based Dispatcher and a Rust-based Wasm runtime, approximately 7,900 and 11,600 lines of code, respectively. The source code and artifacts are available as open-source[1].

A. Agent Tool Dispatcher

A primary implementation goal was to ensure seamless interoperability with existing agent ecosystems. We implemented the *Tool Dispatcher* by extending the `BaseTool` interface of **LangChain** [28], a standard framework for LLM agents.

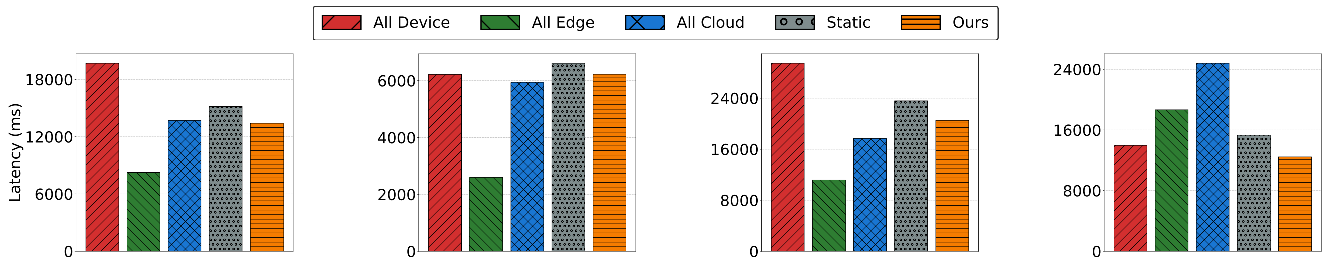
a) Location-Aware Request Proxy: We introduced a specific class, `LocationAwareRequestProxy`, which wraps standard MCP tool definitions. This proxy retains the original metadata (JSON schema) but overrides the execution method (`_run`). Upon invocation, instead of executing logic locally, the proxy triggers the Dispatcher's resolution pipeline via `get_location_for_call(tool_name, args)`. This design allows developers to inject distributed capabilities into standard agents via the `EdgeAgentMCPClient` without modifying the agent's core cognitive logic or prompt.

b) Unified Transport Abstraction: To handle the heterogeneity of execution endpoints, the Dispatcher implements a unified `ClientSession` abstraction. It dynamically switches between `mcp.client.stdio` for local process-based tools and a custom **Streamable HTTP Client** for remote Wasm instances. This ensures backward compatibility, allowing legacy Node.js/Python MCP servers to coexist with our serverless Wasm architecture within the same session.

[1] https://github.com/ddps-lab/EdgeAgent, https://github.com/ddps-lab/WasmMCP

TABLE III: WasmMCP Runtime Comparison

MCP Server	**Native Language**	**Cold Start (ms)**		**Exec Time (ms)**		**Memory (MB)**		**Binary Size**	
		WasmMCP	Container	WasmMCP	Container	WasmMCP	Container	WasmMCP	Container
filesystem	Node.js	53.2	527.3	2.1	13.0	28.2	68.9	471 KB	80 MB
git	Node.js	53.8	723.1	2.2	5.1	25.8	74.0	475 KB	167 MB
fetch	Python	53.4	1114.1	1997.9	2540.8	30.8	84.2	1.42 MB	80.5 MB
summarize	Python	53.1	1102.4	478.4	1228.2	26.2	92.6	317 KB	82 MB
log_parser	Python	53.2	1038.3	196.3	0.2	37.8	79.3	1.24 MB	80.1 MB
image_resize	Python	53.1	1119.1	261.4	90.1	35.9	94.2	1.17 MB	159 MB



(a) S1. Code Review (b) S2. Log Analysis (c) S3. Research Assistant (d) S4. Image Processing

Fig. 8: End-to-end latency across scenarios and scheduling policies

B. *WasmMCP as a Serverless Runtime*

For the execution plane, we constructed a specialized serverless environment based on Knative Serving [29]. This setup leverages the millisecond-level cold start of Wasm [21] to implement an aggressive scale-to-zero policy, optimizing resource usage in constrained edge environments.

a) Stateless HTTP Transport: Standard MCP implementations rely on SSE for progress logging, which necessitates persistent connections. This conflicts with the ephemeral nature of serverless functions. To resolve this, WasmMCP handles each tool invocation as an independent HTTP POST request with Streamable HTTP, allowing the runtime to terminate instances immediately after processing to maximize resource utilization.

b) Kubernetes Integration and Storage Extension: We utilize the `containerd-shim-wasmtime` from the `runwasi` project [30] to orchestrate Wasm modules as native Kubernetes pods.[2] By registering `wasmtime` as a `RuntimeClass`, Wasm workloads can be scheduled alongside standard containers simply by specifying `runtimeClassName: wasmtime`.

A critical limitation in the upstream `runwasi` implementation is the lack of automatic mapping between Kubernetes `volumeMounts` and WASI preopened directories. This prevents Wasm modules from accessing persistent storage or configuration maps. To address this, we extended the shim implementation to introspect the Pod's `VolumeMounts` specification during initialization. Our modified shim dynamically maps these volumes to WASI preopens, enabling WasmMCP tools to seamlessly perform I/O-intensive operations within the strict sandbox environment.

[2]Our fork with PVC support: https://github.com/ddps-lab/runwasi

VIII. Evaluation

We evaluate our system along with two dimensions. First, we measure the performance characteristics of WasmMCP as a lightweight runtime for MCP servers. Second, we assess the effectiveness of the Agent Tool Dispatcher in optimizing tool placement across the edge-cloud continuum.

A. *Environment Setup and Workloads*

Our testbed consists of three tiers of a Raspberry Pi 4 (8GB RAM, ARM Cortex-A72) as the Device tier, an Intel NUC (16GB RAM, i5-1240P) as the Edge tier, and an AWS EC2 x8aedz.xlarge instance (4 vCPUs, 128GB RAM, up to 15 Gbps network) as the Cloud tier. Device and Edge tiers are located in Northeast Asia, while the Cloud tier is deployed in AWS us-west-2 (Oregon) to represent a geographically distributed edge-cloud deployment. The measured round-trip latency between Device and Cloud is approximately 130 ms. To simulate a realistic multi-hop local area network between Device and Edge, we add 30 ms RTT using Linux `tc` to the Edge tier, resulting in approximately 30 ms between Device adn Edge, 130 ms between Edge and Cloud, and 160 ms between Device and Cloud. The Edge tier runs WasmMCP exclusively, while the Cloud tier uses containers only. The Device tier supports both runtimes, enabling comparative analysis.

We used four representative AI agent scenarios involving MCP tools using real-world datasets defined in Table II. S1 (Code Review) uses the Defects4J [31] Lang repository to analyze git repository changes and generate review reports. S2 (Log Analysis) uses the Loghub [32] dataset to parse Apache logs and summarize errors. S3 (Research Assistant) uses the S2ORC [33] dataset to retrieve and summarize academic papers. S4 (Image Processing) uses the COCO [34] dataset to detect duplicate images and generate thumbnails.

When the experiments need to invoke LLM service, to ensure reproducibility, we first execute each scenario using GPT-4o-mini with temperature=0 and seed=42 to obtain deterministic tool call sequences. We then extract the resulting tool invocation order and replay it where necessary, removing the variability of LLM inference result.

B. *WasmMCP Performance*

Table III compares WasmMCP against containerized runtimes on Intel NUC. Each row reports the average values reported across all tools provided by the corresponding MCP server in the row. Memory consumption is measured using Linux *cgroups* for containers and *psutil* for Wasm processes. Both report peak usage recorded after runtime stabilization.

a) Cold Start Latency: WasmMCP achieves consistent cold start latency of approximately 53 ms across all tools, representing a 10–21× improvement over containers. Node.js-based tools (e.g., filesystem, git) show 10–13× reduction, while Python-based tools achieve up to 21× faster initialization latency by eliminating interpreter startup overhead. This result validates WasmMCP's suitability for serverless systems.

b) Execution Time Trade-offs: Execution time varies by workload characteristics. For I/O-bound tools (e.g., filesystem, git) and network-bound tools (e.g., fetch, summarize), WasmMCP performs comparably to containers since execution time is dominated by external latency rather than computation. However, compute-intensive tasks exhibit different behavior depending on input size. For small inputs, WasmMCP's fast cold start compensates slower execution time, resulting in comparable end-to-end latency. As input size grows, container runtimes outperform WasmMCP due to native SIMD optimizations unavailable in the WasmMCP environment. For example, parse_logs with 200 lines completes in 1030 ms (WasmMCP) versus 1066 ms (Container), but at 2000 lines, container execution (1079 ms) significantly outperforms WasmMCP (4895 ms). This suggests that for warm, compute-heavy workloads with large inputs, containers remain preferable. This result emphasizes the heterogeneity of MCP workload characteristics, and there is not a globally optimal runtime regardless of the tools. The placement algorithm should consider the workload characteristics, and the proposed *Agent Tool Dispatcher* provides rich set of information to make informed decision.

c) Resource Efficiency: Memory footprint is reduced by 59–72% across all tools, where the average of WasmMCP being 31 MB and 82 MB for containers. Binary sizes show more improvement, averaging under 1 MB for WasmMCP and 80–167 MB for container images. This reduction enables rapid distribution to bandwidth-constrained edge nodes and allows more tool deployment on resource-limited devices.

C. *Dispatcher Effectiveness with Real-World Agent Workloads*

To present the effectiveness of the proposed *Agent Tool Dispatcher*, Figure 8 presents end-to-end latency for real-world AI agent workloads across five scheduling policies. Three baseline policies (all-device, all-edge, all-cloud) force execution on a single tier with data pre-positioned accordingly.

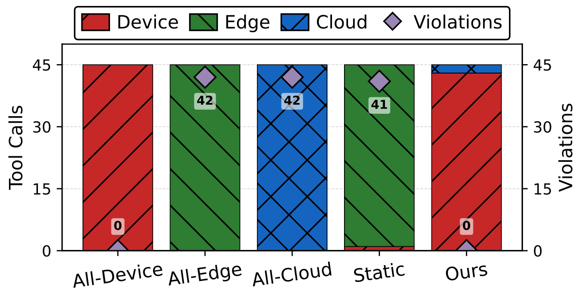


Fig. 9: Tool placement distribution and constraint violations across scheduling policies for S4 (Image Processing)

The static policy uses manually optimized mappings presented in Table I. Our proposed policy implements the constraint filtering and data locality heuristics.

Single-tier policies achieve optimal latency only when workload characteristics align with the chosen tier. In S1 (Code Review) and S2 (Log Analysis), All-Edge achieves the lowest latency because these workloads are I/O-bound, benefiting from the faster storage and CPU of the Intel NUC compared to the Raspberry Pi. However, it may violate locality-constraint that our proposed system follows. All-Device in S3 (Research Assistant) incurs 2× higher latency than our policy due to limited network throughput for external API calls. All-Edge and All-Cloud in S4 (Image Processing) show higher latency than our proposed system. Thorough analysis reveals that tools are compute-bound, but they also incur significant network overhead to transfer intermediate outcomes between tool invocations where such overheads are reflected in the profiling steps in the proposed system.

By combining profiled tool characteristics with constraint-first filtering, our policy achieves competitive performance across all scenarios without manual per-scenario tuning. In S4, our policy outperforms all baselines by keeping I/O-bound on Device while offloading only external API calls to Cloud. In S1 and S2 where All-Edge excels due to hardware advantages, our policy remains within 2.5× while satisfying all placement constraints that All-Edge violates.

1) Placement Analysis: Figure 9 illustrates placement decisions and constraint violations for S4 (Image Processing), which involves 45 tool invocations. We omit observation from other scenarios as they show similar patterns. In this experiment, we define constraints as enforcing data locality (e.g., filesystem operations restricted to Device) and privacy requirements (sensitive data should not be executed on Cloud). While single-tier baselines execute successfully with pre-positioned data, they would violate these operational constraints in production. All-Edge and All-Cloud each incur 42 violations, while the Static policy incurs 41 violations due to its Edge-centric assignment without runtime validation.

Our constraint-first filtering achieves zero violations across all cases. The placement distribution shows 43 invocations on Device (filesystem and image operations) and 2 on Cloud (external API calls). This demonstrates that respecting data locality constraints aligns with performance optimization for I/O-bound workloads, as keeping data-intensive operations local avoids unnecessary transfer overhead.

In summary, while baseline approaches achieved lower latencies by offloading tasks, they failed to respect data boundaries. In contrast, our proposed framework exhibited zero constraint violations, demonstrating its capability to enforce complex data locality requirements while maintaining competitive performance. This highlights our design philosophy of treating correctness as a performance metric of equal importance to raw execution speed in the edge-cloud continuum.

IX. Related Work

MCP Ecosystem and Tool Benchmarking : Recent benchmarks such as LiveMCPBench [9] and MCPToolBench++ [10] evaluate LLM tool-use accuracy using registry metadata, focusing on functional correctness rather than operational placement. Our work complements these by profiling execution behavior for profiling-driven orchestration.

LLM Agent Orchestration : LangChain [28] and AutoGen [35] focus on multi-agent coordination and semantic tool routing. TinyAgent [36] optimizes tool selection for edge deployment. However, these assume tools execute where deployed without dynamic placement. Our dispatcher decouples semantic selection from infrastructure placement, enabling constraint-aware scheduling.

Edge-Cloud Task Offloading : Computation offloading has been extensively studied [13], [14]. Neurosurgeon [15] pioneered DNN partitioning between mobile and cloud, while MAUI [37] and CloneCloud [38] explored energy-aware offloading. However, these works target monolithic workloads with predictable computation graphs. MCP tools are discrete and heterogeneous, requiring rich context for task placement.

Wasm for Serverless Edge : WASM has emerged as a lightweight serverless runtime with significant cold-start reductions [21], [39] and smaller binaries [40] compared to containers. While these establish WASM's edge viability, none address MCP integration. WasmMCP provides WASI-native dual-transport support for seamless deployment.

X. Conclusion and Future Work

In this paper, we addressed the gap between the standardized interaction model of the MCP and the physical constraints of the Edge-Cloud Continuum. We identified that current container-centric deployment models of MCP tools are ill-suited for the bursty, on-demand nature of agentic workloads on constrained devices. To bridge this gap, we proposed a holistic orchestration framework comprising the *Agent Tool Dispatcher* and *WasmMCP*. By decoupling tool semantics from infrastructure dependencies and leveraging WASM's lightweight characteristics, our system achieves 10–21$\times$ faster cold starts and enables efficient tool co-location. This enables the edge devices to function as a viable execution tier for latency-sensitive agent workloads.

For future work, we plan to leverage the proposed framework to develop more sophisticated scheduling algorithms. We aim to explore dynamic multi-objective optimization techniques that balance latency, energy consumption, and privacy constraints to further refine workload placement decisions.

Acknowledgment

This work was supported by Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (RS-2022-00144309 & RS-2025-25441560 & RS-2026-25507506). The authors would like to thank Woohyeok Park for his assistance with figure preparation. Claude Code (Anthropic) was used to support the implementation of the proposed system. All code was reviewed and validated by the authors.

References

[1] "Model context protocol," https://modelcontextprotocol.io/, accessed: 2025-12-15.

[2] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 133–146. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/wang-liang

[3] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *Proceedings of the 11th IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 2018, pp. 181–188.

[4] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, D. Holman *et al.*, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017, p. 185–200.

[5] "Anthropic," https://www.anthropic.com/, accessed: 2025-12-15.

[6] "Smithery," https://smithery.ai/, accessed: 2025-12-15.

[7] "Deepnlp," https://www.deepnlp.org/, accessed: 2025-12-15.

[8] "Pulsemcp," https://www.pulsemcp.com/, accessed: 2025-12-15.

[9] G. Mo, W. Zhong, J. Chen, X. Chen, Y. Lu *et al.*, "Livemcpbench: Can agents navigate an ocean of mcp tools?" 2025. [Online]. Available: https://arxiv.org/abs/2508.01780

[10] S. Fan, X. Ding, L. Zhang, and L. Mo, "Mcptoolbench++: A large scale ai agent model context protocol mcp tool use benchmark," 2025. [Online]. Available: https://arxiv.org/abs/2508.07575

[11] W. Wang, P. Niu, Z. Xu, Z. Chen, J. Du *et al.*, "Mcp-flow: Facilitating llm agents to master real-world, diverse and scaling mcp tools," 2025. [Online]. Available: https://arxiv.org/abs/2510.24284

[12] L. Chiticariu, Y. Li, and F. R. Reiss, "Rule-based information extraction is dead! long live rule-based information extraction systems!" in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, D. Yarowsky, T. Baldwin, A. Korhonen, K. Livescu, and S. Bethard, Eds. Seattle, Washington, USA: Association for Computational Linguistics, Oct. 2013, pp. 827–832. [Online]. Available: https://aclanthology.org/D13-1079/

[13] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali *et al.*, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," *Journal of Systems Architecture*, vol. 98, pp. 289–330, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762118306349

[14] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[15] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2017, pp. 615–629.

[16] S. Hao, T. Liu, Z. Wang, and Z. Hu, "Toolkengpt: Augmenting frozen language models with massive tools via tool embeddings," in *Advances in Neural Information Processing Systems*, vol. 36. Curran Associates, Inc., 2023, pp. 45 870–45 894. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/8fd1a81c882cd45f64958da6284f4a3f-Paper-Conference.pdf

[17] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti *et al.*, "Serverless computing: One step forward, two steps back," in *9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org, 2019. [Online]. Available: http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf

[18] A. Barbalace, M. L. Karaoui, W. Wang, T. Xing, P. Olivier *et al.*, "Edge computing: The case for heterogeneous-isa container migration," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 73–87. [Online]. Available: https://doi.org/10.1145/3381052.3381321

[19] M. Copik, E. Alnuaimi, A. Kamatar, V. Hayot-Sasson, A. Madonna *et al.*, "Xaas containers: Performance-portable representation with source and ir containers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 533–555. [Online]. Available: https://doi.org/10.1145/3712285.3759868

[20] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter *et al.*, "SOCK: Rapid task provisioning with serverless-optimized containers," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 57–70. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/oakes

[21] S. Kang, M. Song, T. Kim, S. Lee, J. Han *et al.*, "Hybridserve: Adaptive webassembly-container runtime selection for edge serverless computing," in *Proceedings of the 11th International Workshop on Serverless Computing*, ser. WoSC11 '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1–6. [Online]. Available: https://doi.org/10.1145/3774899.3775011

[22] A. Hall and U. Ramachandran, "A hybrid runtime for function-as-a-service at the edge," in *Proceedings of the 26th International Middleware Conference*, ser. Middleware '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 469–481. [Online]. Available: https://doi.org/10.1145/3721462.3770785

[23] W. C. Group, "Webassembly system interface (wasi) specification," https://github.com/WebAssembly/WASI, 2020, accessed: 2025-12-15.

[24] modelcontextprotocol, "rust-sdk," https://github.com/modelcontextprotocol/rust-sdk, 2025.

[25] "mcp-proxy," https://github.com/sparfenyuk/mcp-proxy, 2025.

[26] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: a Serverless-first, Light-weight Wasm Runtime for the Edge," in *Proceedings of the 21st International Middleware Conference*, ser. Middleware '20, 2020, pp. 265–279.

[27] V. Kjorveziroski and S. Filiposka, "Webassembly as an enabler for next generation serverless computing," *Journal of Grid Computing*, vol. 21, no. 3, p. 34, 2023.

[28] H. Chase, "Langchain," 2022, released: 2022-10-17. [Online]. Available: https://github.com/langchain-ai/langchain

[29] Knative Project, "Knative: Kubernetes-based platform to deploy and manage modern serverless workloads," https://knative.dev, 2018, accessed: 2025-12-17.

[30] containerd, "runwasi," https://github.com/containerd/runwasi, 2022.

[31] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[32] J. Zhu, S. He, P. He, J. Liu, and M. R. Lyu, "Loghub: A large collection of system log datasets for ai-driven log analytics," in *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2023.

[33] K. Lo, L. L. Wang, M. Neumann, R. Kinney, and D. S. Weld, "S2orc: The semantic scholar open research corpus," in *Proceedings of the 58th annual meeting of the association for computational linguistics*, 2020, pp. 4969–4983.

[34] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona *et al.*, "Microsoft coco: Common objects in context," in *European conference on computer vision*. Springer, 2014.

[35] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li *et al.*, "Autogen: Enabling next-gen LLM applications via multi-agent conversations," in *First Conference on Language Modeling*, 2024. [Online]. Available: https://openreview.net/forum?id=BAakY1hNKS

[36] L. E. Erdogan, N. Lee, S. Jha, S. Kim, R. Tabrizi *et al.*, "TinyAgent: Function calling at the edge," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, D. I. Hernandez Farias, T. Hope, and M. Li, Eds. Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024, pp. 80–88. [Online]. Available: https://aclanthology.org/2024.emnlp-demo.9/

[37] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu *et al.*, "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 49–62. [Online]. Available: https://doi.org/10.1145/1814433.1814441

[38] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 301–314. [Online]. Available: https://doi.org/10.1145/1966445.1966473

[39] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, "Pushing Serverless to the Edge with WebAssembly Runtimes," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2022, pp. 140–149.

[40] C. Marcelino, N. Krennmair, T. W. Pusztai, and S. Nastic, "Lumos: Performance characterization of webassembly as a serverless runtime in the edge-cloud continuum," in *Proceedings of the 15th International Conference on the Internet of Things*, ser. IOT '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 113–121.

Appendix A
Artifact Description

A. Abstract

This artifact provides the complete set of research materials associated with the paper, "Orchestrating WASM-based MCP Tool Runtimes for AI Agents across Edge-Cloud Continuum", including source code, datasets, and evaluation scripts. The artifact comprises four components:

1) **WasmMCP**: A Rust-based WebAssembly runtime for MCP tools, supporting dual-transport (stdio/HTTP) with tens-of-milliseconds module instantiation. (Sections VI–VII, Table III)
2) **EdgeAgent Framework**: A Python-based Agent Tool Dispatcher built on LangChain, implementing constraint-aware workload placement across device, edge, and cloud tiers. (Sections V–VII, Figures 8–9)
3) **MCP Ecosystem Dataset**: Metadata of 17,566 tools from 1,467 MCP servers collected from major registries (Smithery, DeepNLP, PulseMCP), including category labels and resource utilization profiles. (Section II-B, Figures 2–3)
4) **Evaluation Dataset**: Experiment data and scripts to reproduce figures and tables presented in the paper. (Section VIII, Table III, Figures 8–9)

All artifacts are archived on Zenodo with a persistent DOI: https://doi.org/10.5281/zenodo.18640664

B. Artifact Check-list

- **Program:** Rust 1.91.1 (rustc/cargo), Python 3.13.9, wasmtime 39.0.1
- **Compilation:** `cargo build --target wasm32-wasip2` (WasmMCP), `pip install` (EdgeAgent)
- **Run-time environment:** Linux (Ubuntu 24.04+)

- **Hardware:** Any x86_64 or ARM64 machine for source inspection; full evaluation requires a 3-tier testbed (see Section VIII-A)
- **Output:** Figures (PDF), raw experiment data (JSON/CSV)
- **Disk space required:** Under 500 MB
- **Publicly available:** Yes
- **Code license:** Apache License 2.0
- **Data license:** CC BY 4.0
- **Archived:** Yes (Zenodo, DOI: `10.5281/zenodo.18640664`)

C. Description

1) How to Access: The artifact is permanently archived on Zenodo: https://doi.org/10.5281/zenodo.18640664
Development repositories are maintained on GitHub:

- EdgeAgent: https://github.com/ddps-lab/EdgeAgent
- WasmMCP: https://github.com/ddps-lab/WasmMCP

2) Artifact Structure:

```
EdgeAgent-CCGrid2026-Artifact/
|-- README.md
|-- LICENSE
|-- edgeagent/
|   |-- edgeagent/
|   |-- config/
|   |-- infra/
|   +-- scripts/
|-- wasmmcp/
|   |-- wasmmcp/
|   |-- wasmmcp-macros/
|   |-- servers/
|   |-- shared/
|   +-- Cargo.toml
+-- artifacts/
    |-- CODE/
    |-- DATA/
    |   |-- figure2-data/
    |   |-- figure3-data/
    |   |-- figure8-data/
    |   |-- figure9-data/
    |   +-- table3-data/
    |-- FIGURE/
    +-- MCP-SERVER-DATA/
```

3) Software Artifacts: **EdgeAgent Framework** (`edgeagent/`) is implemented in Python (approximately 7,900 lines of code) and extends LangChain's `BaseTool` interface. It includes the `LocationAwareRequestProxy` for transparent tool relaying, the Tool Registry with workload profiling, the Constraint Filter for hard constraint enforcement, and the Pluggable Scheduler. The `config/` directory contains YAML configurations for each MCP server, and `infra/` provides Kubernetes and infrastructure setup for edge-cloud deployment.

WasmMCP (`wasmmcp/`) is implemented in Rust (approximately 11,600 lines of code) and compiles to `wasm32-wasip2` targets. It provides a WASI-native MCP server framework with procedural macros (`wasmmcp-macros/`) for compile-time schema derivation, and dual-transport support (stdio and Streamable HTTP). The `servers/` directory contains MCP server implementations (e.g., filesystem, log parser) used in the evaluation.

4) Data Artifacts: **Research Artifacts** (`artifacts/`) contains all data and scripts for reproducing the paper's figures and tables.

- `CODE/`: Python scripts for generating each figure (`figure2/`, `figure3.py`, `figure8.py`, `figure9.py`).
- `DATA/`: Raw experiment data organized per figure and table (`figure2-data/`, `figure3-data/`, `figure8-data/`, `figure9-data/`, `table3-data/`).
- `FIGURE/`: Pre-generated figure outputs in PDF format.
- `MCP-SERVER-DATA/`: MCP ecosystem dataset containing metadata of 17,566 tool definitions from 1,467 unique servers collected from Smithery, DeepNLP, and PulseMCP registries, along with collection scripts.

5) Mapping to Paper:

- Figure 2 (MCP tool categories) → `artifacts/DATA/figure2-data/` + `artifacts/CODE/figure2/`
- Figure 3 (Resource utilization) → `artifacts/DATA/figure3-data/` + `artifacts/CODE/figure3.py`
- Table III (Runtime comparison) → `artifacts/DATA/table3-data/` (`container-data.csv`, `wasmmcp-data.csv`)
- Figure 8 (E2E latency) → `artifacts/DATA/figure8-data/` + `artifacts/CODE/figure8.py`
- Figure 9 (Placement & violations) → `artifacts/DATA/figure9-data/` + `artifacts/CODE/figure9.py`
- MCP Ecosystem Analysis (Section II-B) → `artifacts/MCP-SERVER-DATA/`
- WasmMCP Runtime (Section VI, Table III) → `wasmmcp/`
- Agent Tool Dispatcher (Section V, Figures 8–9) → `edgeagent/`

D. Notes

The development repositories on GitHub may contain updates beyond the archived version. The Zenodo archive represents the exact snapshot corresponding to the paper's evaluation. For questions or issues, please refer to the GitHub repositories' issue trackers.