EDERA

# Performant Isolation for Agentic AI

A Comparative Analysis Proving Edera's Hardened Runtime is Both More Secure and Faster than Docker for AI Workloads

# Executive Summary

AI agents can be a powerful tool for handling complex, multi-stage processes. But what happens when the agent misbehaves? In this paper we propose combining design patterns for multi-stage AI agents with Edera to achieve strong security guarantees with minimal limits to the agent's capabilities. The design patterns provide best practices for how agents interact, and Edera provides strong isolation. So each single-purpose agent can be in its own isolated environment, then securely combined into a powerful application.

We create an example architecture and implementation to demonstrate that running agents in Edera can actually be faster than running them in Docker.

# Introduction

AI agents have proven to be useful at a wide range of tasks from triaging email to searching the internet to handling customer inquiries. But running agents in traditional, shared-kernel container environments comes with risks. If the agent is able to access your filesystem, what if exfiltrates API keys from the host environment? If it's able to read logs to respond to customer questions, can it also read sensitive information and return that to the customer?

We want to allow agents to have enough autonomy to complete complex tasks, but prevent them from wreaking havoc on our systems. A lot of work has been done to make agents more accurate through carefully crafted prompts and splitting large tasks into smaller ones. And researchers have come up with secure design patterns to combine these agents. These design patterns use set interfaces or careful control flow to ensure that a prompt injection can't spread from one agent to the entire system. What we focus on in this work is where these agents are actually running.

When agents are run directly on your machine they can get access to everything else running on your machine, so we propose isolating agents with Edera. These agents, which can be split into separate agents for each task following existing best practices, can run in isolated Edera zones. Input and output can be passed between the agents, but each agent only has access to data that's directly passed in. If the agent has to operate on files, it will operate only on files passed into the zone, and any changes will be made only after the files are passed back as output. This use of Edera can be combined with agent design patterns designed for security, like using the operator pattern and pre-defining steps for each agent.

In this paper, we look at the feasibility of running complex agentic systems in Edera. We create a sample architecture that uses tools like model context protocol (MCP) and retrieval augmented generation (RAG) with a supervisor model to determine if agents can be run in isolated zones, how hard this is to set up, and what the performance impact is. We run the agents in both Docker and Edera to compare the performance and usability of each setup.

# Background

## Edera

Edera is a hardened runtime for containers that provides isolation to workloads through the use of zones. Zones are similar to the guest operating system in a MicroVM, providing a minimal workspace for applications. Zones are managed by Edera's Xen-based type 1 hypervisor. By running workloads in zones, they have a reduced attack surface and no shared operating system with other workloads on the system. Edera acts as a Kubernetes runtime, making it compatible with existing Kubernetes applications. This compatibility allows existing applications to gain isolation without sacrificing their existing Kubernetes-based orchestration or CI/CD pipelines. For more information about Edera, see our **paper**.

## AI Agent Design Patterns

A lot of work has been done on the security of AI agents. We especially want to call out work on secure design patterns for agents that this work built on. **IsolateGPT** proposes that AI agents communicate through set interfaces. This allows for user screening of inter-app communication and setting the context for each agent. **Beurer-Kellner et. al.** propose a series of design patterns for AI agents that reduce the risk of prompt injection. These design patterns limit the abilities of agents, which limits these patterns to single-purpose agents where the designer has some knowledge of what the agent will need access to. In this work we utilize the supervisor pattern, with a deterministic supervisor inspired by these design patterns from the literature.

The agent design patterns discussed provide application-level security. Our work adds a crucial, missing layer: infrastructure-level isolation. Edera ensures that even if an application-level control fails (e.g., via a novel zero-day prompt injection), the blast radius is contained to a single, powerless zone.

# Architecture

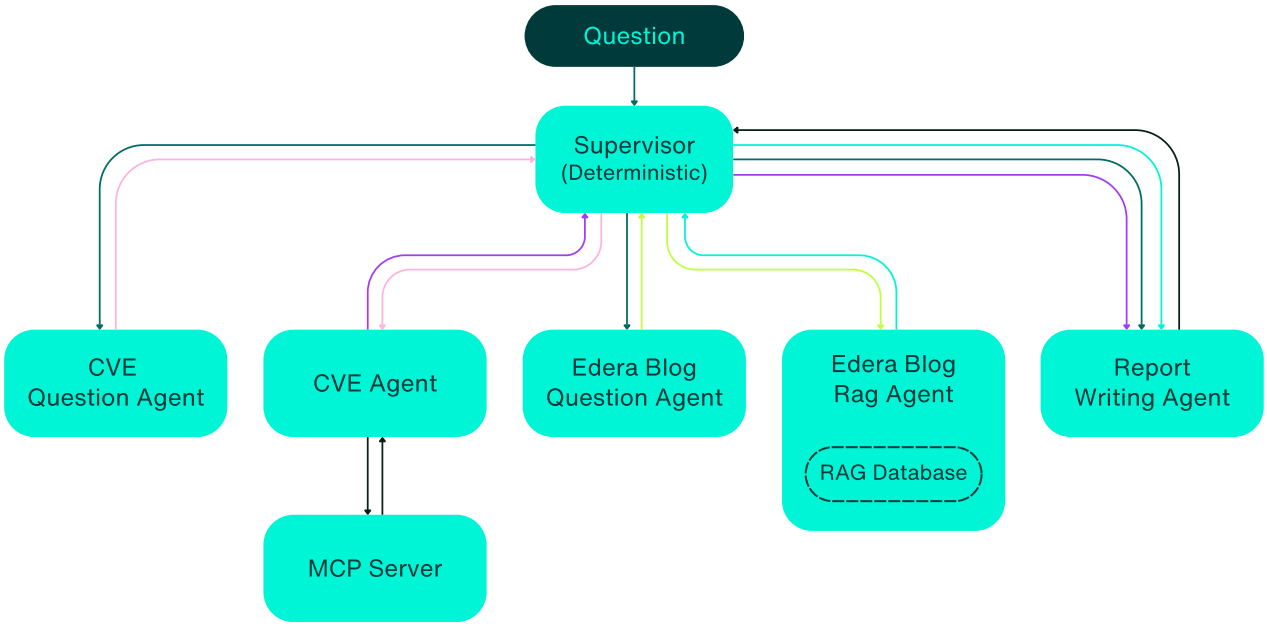
We designed a sample agent architecture to put the idea of agent isolation into practice. The goal is to see how isolation works in practice, and to observe the performance for different types of AI operations including the use of knowledge bases and MCP servers. Each agent in this architecture can be put in its own Docker container or Edera zone, with networking enabling communication between the agents. The architecture is summarized in the diagram below.

The agents in this architecture write a report in answer to a question about container isolation. Each agent is given a small task, with the supervisor orchestrating calls to the other agents. For this experiment, we used a deterministic, non-AI supervisor. The supervisor passes inputs between the agents, but does not read this data itself, meaning it is safe from prompt injection by the other agents. While this supervisor model demonstrates how isolated agents can be combined with other agentic best practices, the isolation mechanisms would work similarly with an agentic supervisor or other agent architecture.

The supervisor first gathers data about CVEs related to the input question from an MCP server. It calls a CVE question agent which generates a relevant question about CVEs, then passes this question to a CVE agent. The CVE agent responds to the question with access to a search engine MCP server. This MCP server is able to read the top Google results for a query. The CVE agent creates an answer to the question and sends this back to the supervisor.

The supervisor then looks for relevant information from a knowledge base. This knowledge base is a RAG database that uses ChromaDB and is seeded with information about container isolation from Edera's blog. Similar to the above, the supervisor first asks the blog question agent for a relevant query for the knowledge base, then asks the RAG agent for a response to that query. This response is sent back to the supervisor.

Finally, the answers from both the CVE agent and the RAG agent are sent to a report writing agent to generate the final report.




**Key:**

Edera Zone

Arrow Color = Identical Content

**Pseudocode:**

```
Q1 = CVE_question_agent(question)
R1 = CVE_agent(Q1)
Q2 = Edera_blog_question_agent(question)
R2 = Edera_blog_RAG_agent(Q2)
R3 = report_writing_agent(question, R1, R2)
```

# Implementation

We implemented the above architecture in both Docker and Edera to compare the performance and security of each. The Docker implementation uses a Docker Compose file to build and deploy all of the agents. The Edera implementation translates this Compose file into Kubernetes yaml with the Edera runtime class. Docker and Kubernetes (with the Edera cri) provide the networking for the agents to communicate. The source code for both implementations is available **here**.

The agents are written in python using LangGraph and use the OpenAI API. They are packaged as Docker containers in order to run in both implementations. In Docker each agent is a Docker container, and in Edera each agent is a Kubernetes container inside a zone.

# Security Validation

Next, let's compare the security between the Docker and Edera setups, as well as comparing to a baseline of running all agents in the same application. But before we do that, let's look at our threat model and what we're hoping to gain from this security,

We assume that an attacker can perform prompt injection in the initial question sent to our system. Specifically, they can input arbitrary text into this input. With this access, the attacker will try to perform a malicious action through the agent such as alter the file system, alter permissions, or return malicious output. Their end goal is to compromise the host system or alter agent resources like the RAG database.

We consider non-malicious agent actions to be out of scope. For example if the attacker gets the agent to write a report about dogs, this is not considered a successful attack.

So considering this goal, how do our different agent environments compare?

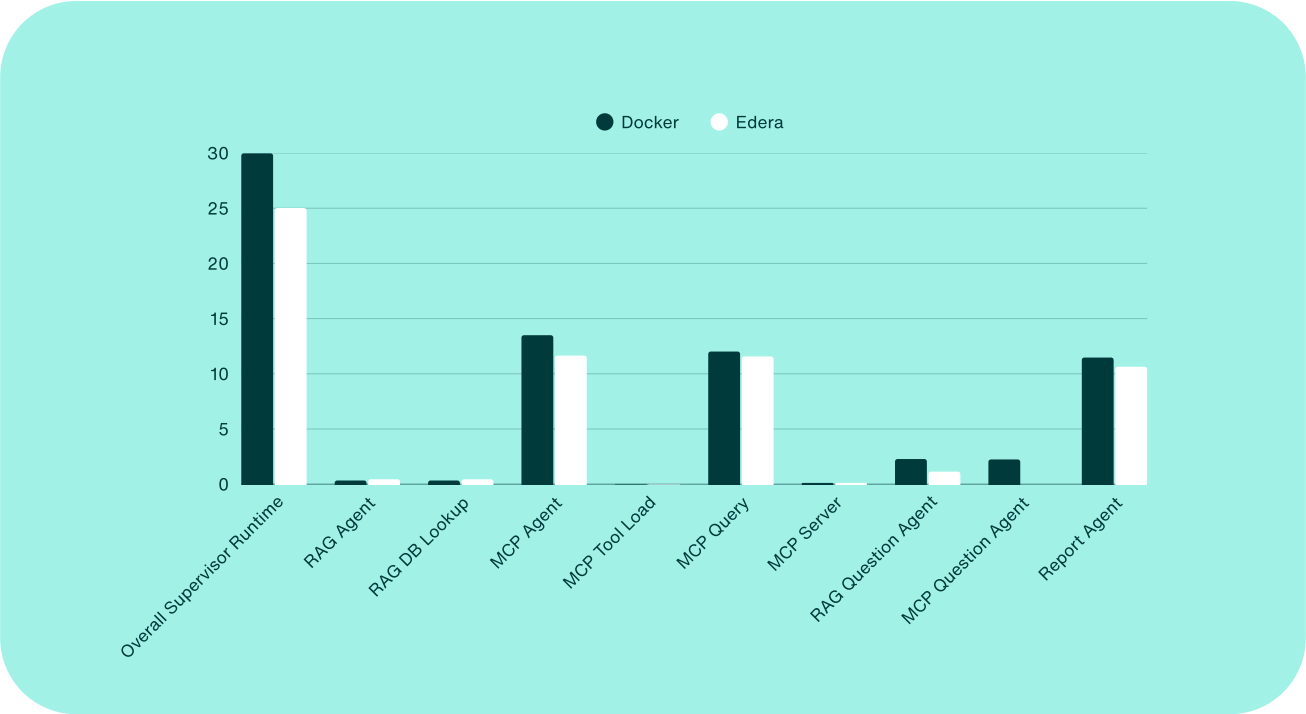
Without any containerization, all agents are running on the host system directly. So a malicious agent, or one under the influence of a prompt injection, could directly impact the system.

For example, the agent could add malicious data to the RAG database or exfiltrate sensitive data like OpenAI API keys.

By putting agents in separate Docker containers, we run them inside containers rather than directly on the host system. These containers prevent a malicious agent from viewing processes running in other agents or the host system, and provide some resource separation. But containers still share a host file system, and we have seen that all it takes is a single vulnerability in the Linux kernel to allow containers to access data in other containers or on the host machine.

Adding Edera zones around containers adds a security boundary to the containers. Now each agent has its own operating system, and cannot access any data from other zones or the host system. This means that even if a prompt injection convinces an agent to behave badly, it can only compromise its own zone rather than the entire system.

So in summary, containerizing agents provides some logical separation between them and the host system. Adding Edera turns this into a security boundary.




# Benchmarks

Next, we compare the performance and usability of agents in Docker and Edera.

## Performance

We send the supervisor the question “Tell me about container escape vulnerabilities from the past year.” several times to access the performance  of each system. We run the system 5 times in both Docker and Edera and use the average performance to compare the systems. Note that due to the non deterministic nature of agentic systems, there was some variance in the runtimes for each system. As such these numbers should be taken as a sample to understand the performance characteristics of each system, rather than as the actual expected runtime.

The table below shows performance numbers for each system overall, in addition to specific measurements of individual agents and operations. These measurements were taken by orchestrating the agents so they provide timing information. In the table, we show the total runtime of each agent (in bold), in addition to the runtime of operations like the RAG database lookup or the MCP query. The overall supervisor runtime shows the runtime for the system end-to-end. All data is in seconds.

Overall, Edera performed slightly faster than Docker. All timings were similar between the two implementations, with the MCP agent adding the most to the performance difference. We speculate that Edera’s networking may contribute to this performance difference.

## Usability

We next compare the usability of each implementation. We start with a quantitative assessment of the configuration complexity, then discuss some qualitative usability measures.

For the configuration complexity we count the lines of code for the code and configurations involved in our implementations, ignoring dependencies for simplicity.

Both implementations use the same Docker containers for the agents. We count the lines of the python code in these containers, shown below. The RAG agent has the most, partly due to the code needed to store objects in the RAG database. The MCP server has the least as it is a very minimal implementation and heavily relies on the FastMCP library.

| | Python loc |
|---|---|
| Supervisor: | 58 |
| CVE question agent: | 43 |
| RAG question agent: | 41 |
| MCP agent: | 49 |
| MCP server: | 26 |
| RAG agent: | 93 |
| Report agent: | 43 |
| **Total** | **353** |

The Docker compose file has 76 lines of code, plus 21 lines in the additional openai compose file used to separate the API token. This totals to 97 lines of configuration for the Docker implementation.

The Edera implementation configuration consists of a service and deployment for each agent. The services are all almost identical at 13 lines each, and the deployments have an average of 34 lines. With 7 agents, this comes to a total of 332 lines of configuration.

So from these numbers, Edera requires more configuration than Docker, though a lot of that configuration is duplicated between the agents.

We then move on to more qualitative analysis about the different usability between these implementations. As the Edera implementation is basically a Kubernetes implementation with an added line that says “runtimeClassName: edera”, a lot of this comes down to the difference between Docker and Kubernetes. That said, the main qualitative takeaways from building these implementations were:

01 The Docker setup had a bit less boilerplate because it just ran the containers rather than setting up a service and deployment for each one. The Kubernetes setup is probably more production-ready, but for this small experiment Docker’s flexibility was nice.

02 The Edera debugging tooling was handy. Edera adds some additional debugging capabilities on top of Kubernetes, so when something wasn’t working I could look at “protect zone logs” and inspect workloads directly from the zones they are running in. This helped me find some bugs that weren’t obvious from a “kubectl describe pod”. This “protect zone log”' feature was a significant, Edera-specific differentiator, providing deep, hypervisor-level introspection that is impossible with standard “kubectl” commands.

03 Updating images took less steps in Docker. Because I was using Docker images, the Compose setup built them for me whenever they changed, while I had to upload them to a registry to get the changes reflected in the Kubernetes setup. This difference would disappear in a CI/CD-driven production environment which always pushes to a registry and is standard practice for both Docker and Kubernetes deployments.

04 The Docker setup required healthchecks to make sure everything ran in the correct order (because the supervisor would just crash if the agent servers weren’t up yet). In Kubernetes I just deployed the supervisor last. This could maybe be fixed with separate compose files in a more complex deployment.

So in summary, the Edera implementation required more lines of configuration, though most of that was Kubernetes boilerplate. And the usability mostly came down to that of Docker vs Kubernetes. Though there was some added debugging directly through Edera.

# Conclusion

Agentic AI creates new, severe security risks from remote code execution to data exfiltration. Shared-kernel containers are not designed to solve these problems. Edera provides true, hypervisor-enforced isolation for each agent, turning “logical separation” into a hard security boundary through the use of zones. This contains the blast radius of any single compromised agent.

Our research demonstrates that these security gains come without a performance penalty for complex systems of AI Agents. In our realistic, multi-agent architecture, we show that the Edera-based deployment was ~16% faster than the Docker Compose equivalent. For any organization deploying agentic AI in production, Edera's hardened runtime is the clear choice for achieving performant, scalable, and truly secure isolation without platform re-architecture.

Get in Touch