



# **How OpenShell Keeps AI Agents Safe:**

An Essential Technical Overview

**SAFE AI Foundation USA**

12 OCTOBER 2026



AI agents are increasingly capable of taking actions on behalf of users, including executing programs, accessing files, communicating with external services, and using credentials. This creates a fundamental security problem: an AI agent may be highly capable while its behavior remains difficult to predict or fully trust.

NVIDIA OpenShell addresses this problem by placing security controls outside the AI agent's own process. Rather than relying on the AI model to obey a safety policy, OpenShell creates a controlled execution environment in which operating-system mechanisms, sandboxing, policy enforcement, network mediation, credential protection, and supervisory controls constrain what the agent can actually do.

OpenShell is therefore best understood as a system-safety runtime for AI agents, rather than an AI model, CUDA technology, or Linux-kernel component. Its architecture can be summarized as AI agent on the upper layer, with OpenShell below it as a user space software running over Linux kernel executing on hardware. The central security principle is that an AI agent may request an action, but it does not have unilateral authority to perform that action.

**Keywords:** AI Safety; Runtime Safety; Sandbox; Watchdog; Linux Kernels;

## 1. Introduction

Traditional software is normally written so that the program itself determines what it will do. Autonomous AI agents introduce a different problem. An agent can dynamically generate code, invoke tools, access data, communicate with remote systems, and make decisions that were not explicitly programmed in advance. Consequently, simply instructing an AI agent “not to access a file” is not equivalent to technically preventing it from accessing the file. A secure agent architecture therefore needs an **enforcement layer** outside the agent itself. NVIDIA’s OpenShell is designed around this principle. Instead of trusting the agent to enforce its own restrictions, OpenShell places the agent inside a sandbox and uses **operating-system security mechanisms** and **a trusted supervisory layer** to constrain its actions.

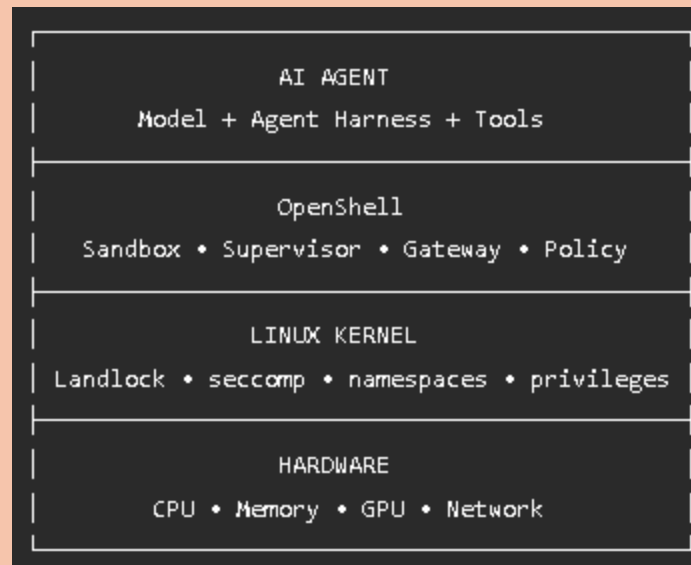


Fig. 1: The resultant architecture.

## 2. What Is OpenShell?

OpenShell is NVIDIA's open runtime for secure execution of AI agents. It provides several software mechanisms for controlling:

- filesystem access
- process execution
- Linux privileges
- network connectivity
- DNS resolution
- external API access
- credentials
- policy
- selected HTTP/WebSocket traffic
- policy changes

The important architectural property is that these controls are not implemented solely inside the AI agent. Rather, it is implemented outside the agent, directly beneath it but above the Linux kernel. This can be explained with Figure 2 below.

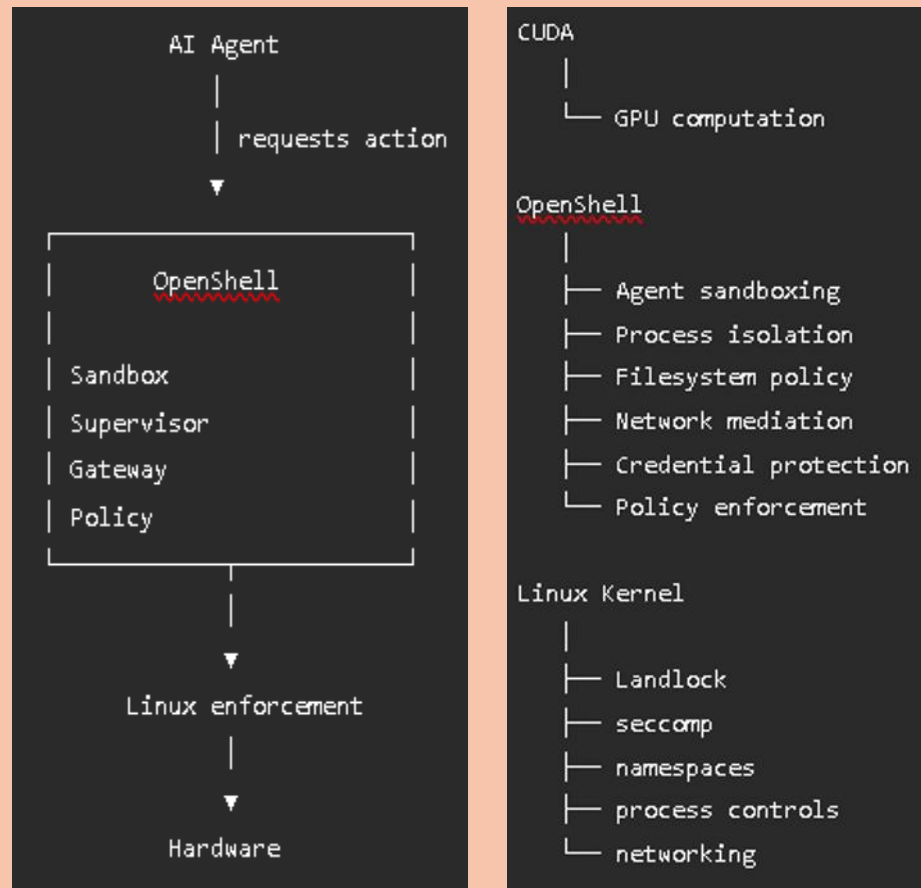


Fig. 2 (left): Openshell sits in between the agent and the linux kernel.

Fig. 3 (right): Openshell is not based on CUDA.

The agent is, therefore, treated as an untrusted or potentially compromised workload. Agents' requested are intercepted, checked, and acted upon, depending on its safety implications and risk levels. An important misconception or misunderstanding is that because OpenShell comes from NVIDIA, it must be implemented using CUDA. It is not. CUDA is NVIDIA's platform for programming NVIDIA GPUs. OpenShell is primarily **systems software**, with its core implementation centered on Rust programming language and Linux operating-system facilities. An AI agent running inside OpenShell can use a GPU and CUDA if its workload requires GPU computation. But CUDA is not what makes OpenShell safe.

OpenShell's agent safety boundary is fundamentally an operating-system and runtime safety architecture. OpenShell is primarily implemented using **Rust**, a modern systems-programming language. Rust is comparable in purpose to C and C++ in that it provides low-level control and high performance, but it provides stronger compile-time protections against many classes of memory-safety problems. Examples include protection against:

- buffer overflows
- use-after-free errors
- dangling references
- data races
- memory-corruption bugs

This is particularly relevant to security infrastructure. OpenShell operates close to the operating system, handling processes, networking, sandboxing, safety, and security boundaries. A memory-corruption vulnerability in such software could potentially undermine the very safety boundary it is intended to enforce. Rust, therefore, provides a foundation for building this type of systems software. However, this does not exclude the use of C++ or other programming languages suitable for low level runtime functions.

**OpenShell does not run inside the Linux kernel.** Instead, OpenShell is user-space software that uses Linux kernel security mechanisms. The conceptual stack is shown by Fig 4 below:



Fig. 4: Openshell sits on top of the Linux Kernel

On Linux, OpenShell relies on Linux kernel existing facilities such as “Landlock” and “Seccomp”. When running through environments such as Docker Desktop on macOS or Windows, the relevant Linux security mechanisms operate within the Linux environment/VM (virtual machine) rather than directly inside the host operating system's kernel.

OpenShell does not require a special NVIDIA kernel. It communicates with Linux through standard kernel interfaces, such as Linux system-call interfaces and kernel security APIs. It does not need to modify the Linux kernel. Three particularly important mechanisms are:

1. **Landlock** – file system control
2. **Seccomp** – system-call control
3. **Linux process APIs** and **namespace** mechanisms

### Let's explain these individually:

**Landlock** is a Linux security facility that allows applications to restrict filesystem access. OpenShell sandbox process makes Linux API system calls to construct a filesystem ruleset. OpenShell can establish rules such as:

#### 1. Agent filesystem permissions

/home/agent/data    READ + WRITE

/home/agent/code    READ

/etc                DENY

/root               DENY

The critical point is that OpenShell does not merely tell the AI agent not to access /etc directory. Instead, it establishes restrictions that the Linux kernel enforces. This is illustrated by Fig 5 below:

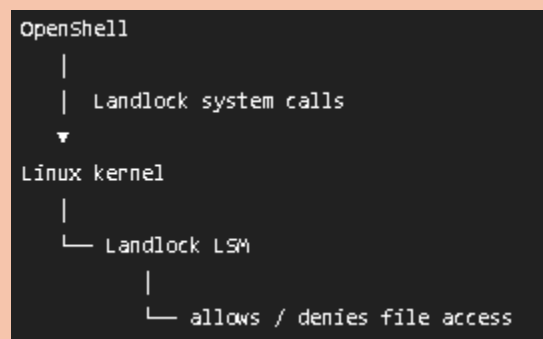


Fig. 5: Landlock filesystem control

If the agent subsequently attempts an unauthorized filesystem operation, the kernel will intercept and reject it, providing the safety feature needed to stop AI from running rogue.

## Seccomp: Controlling System Calls

Every Linux process ultimately interacts with the operating system through system calls. Examples of system calls include: `open()`, `read()`, `write()`, `execve()`, `clone()`, `connect()`, `socket()`, etc. OpenShell can use **seccomp** to restrict system calls made by the agent, as illustrated by Fig. 6 below:

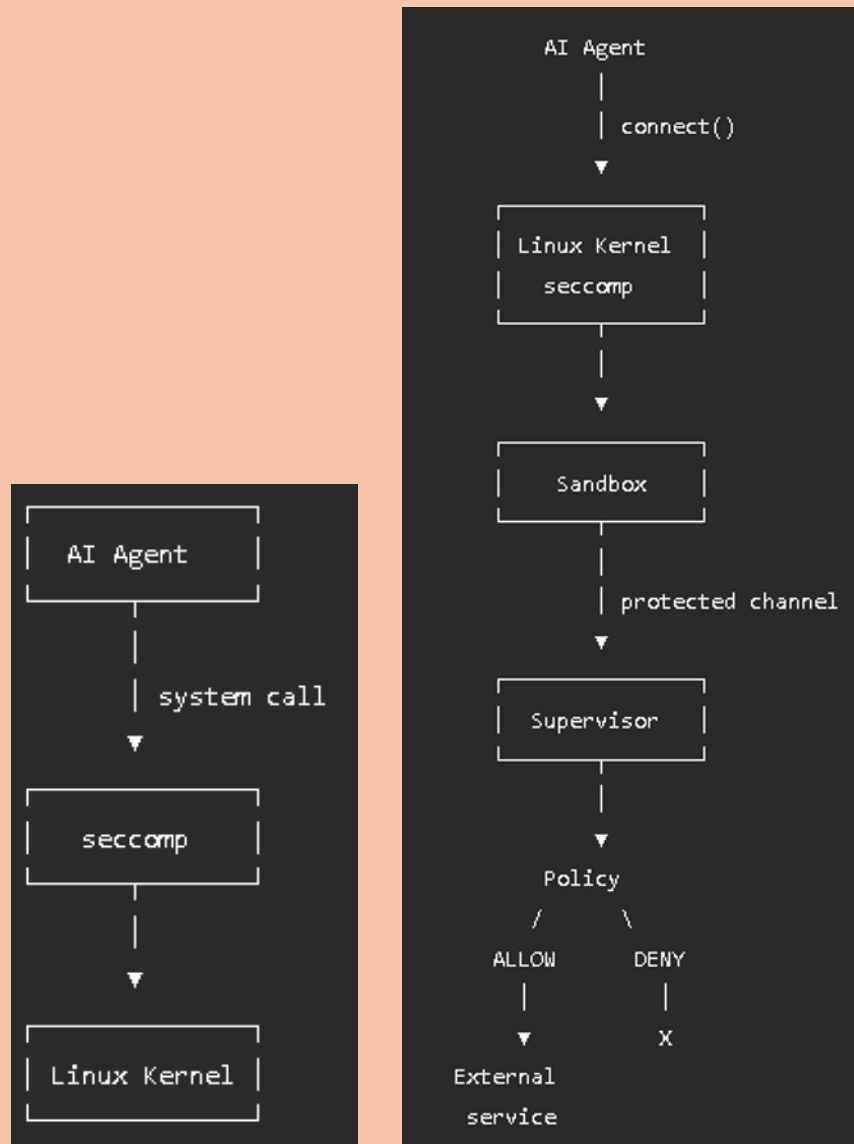


Fig. 6 (left): Seccomp intercepting system calls.

Fig. 7(right): Network control in OpenShell

This, therefore, provides another layer of defense. Even if an agent generates malicious code, the generated program still has to operate within the system-call restrictions imposed on its execution environment. There is no way for it to jump out of this constrained environment to create chaos.

**The Network Security Path:** Network access is especially important for autonomous agents. An unrestricted agent could potentially:

- upload sensitive files
- communicate with an attacker
- download malicious software
- access unauthorized APIs
- exfiltrate credentials
- interact with arbitrary Internet services
- replicate itself elsewhere

OpenShell, therefore, places network access behind policy controls. The important design principle is that an agent does not receive unrestricted network authority merely because it requested a connection.

For example, an agent might attempt:

```
connect("api.example.com", 443)
```

Instead of simply allowing the request to go directly to the Internet, OpenShell can mediate supported networking operations by intercepting this call, identifies the requesting program, request is sent to a Supervisor, policy is then evaluated before a decision to allow or deny the request. If allowed, the supervisor can then establish the approved external connection and relay traffic. This gives OpenShell a significant safety property that network authorization is determined outside the AI agent's own decision-making process and ability.

## 2.1 OpenShell's Components: Sandbox, Supervisor, and Gateway

OpenShell delegates safety responsibilities among several crucial components, such as the sandbox, supervisor and gateway.

### 2.1.1 Sandbox

The sandbox is responsible for establishing and maintaining the agent's restricted execution environment. It is associated with process isolation, filesystem restrictions, system-call intercepts and restrictions, network isolation, process identity, and communication with the supervisor.

### 2.1.2 Supervisor

The supervisor is a trusted component outside the agent's direct control. It can evaluate policy, mediate network access, manage credentials, perform approved upstream connections, support middleware and coordinate with the gateway.

### 2.1.3 Gateway

The gateway acts as a control-plane component. It manages things such as: sandbox lifecycle, configuration, policy, authorization, provider configuration, logging and coordination. The resulting trust separation can be represented by Fig 8 below.

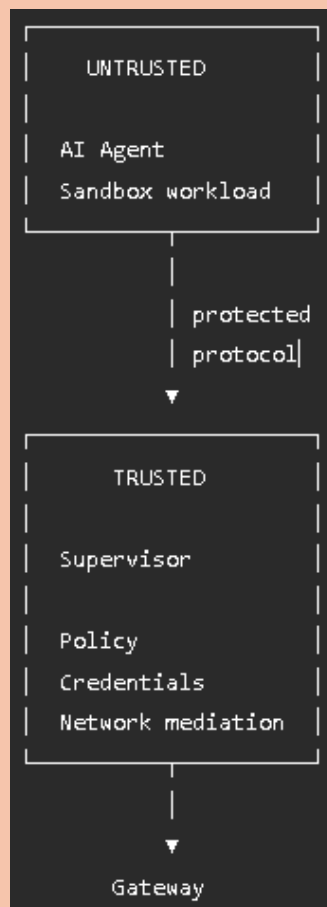


Fig 8: Sandbox, supervisor and gateway

### 2.1.4 The Sandbox Protocol

The sandbox and supervisor communicate using a protected communication channel. The OpenShell Sandbox Protocol provides an abstraction between the sandbox environment and trusted supervisory components. Depending on the deployment environment, underlying communication can use mechanisms such as Unix sockets, TCP, or vsock.

The architectural goal is more important than the specific transport. The sandbox should not be able to impersonate or simply replace the trusted supervisory component. This creates a separation between the workload that executes the AI agent and the component responsible for deciding what the workload is allowed to do.

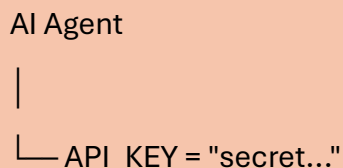
## 2.2 OpenShell Protection Mechanisms

### 2.2.1 Credential Protection

Credentials are secret pieces of information, such as API keys, passwords, access tokens, or certificates that prove an AI agent is authorized to access a particular service or resource. They are needed because external systems such as cloud services, databases, GitHub, or AI-model APIs generally cannot simply trust an incoming request; they need to verify who or what is making the request and what it is permitted to do. Credentials are, therefore, another major safety and security concern. An AI agent may need access to an external service, but giving the agent a raw API key creates a serious problem.

An example of:

BAD DESIGN



If the agent is compromised, manipulated, or tricked into revealing the secret, the credential may be exposed. A more secure architecture is:

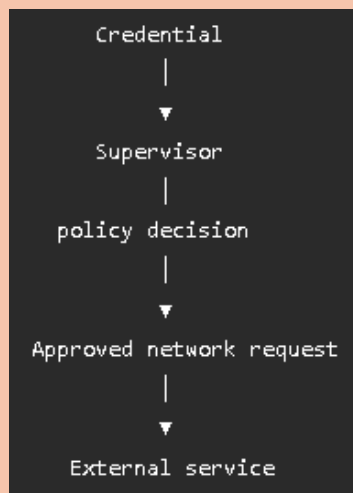


Fig 9 (left): Credentials Protection

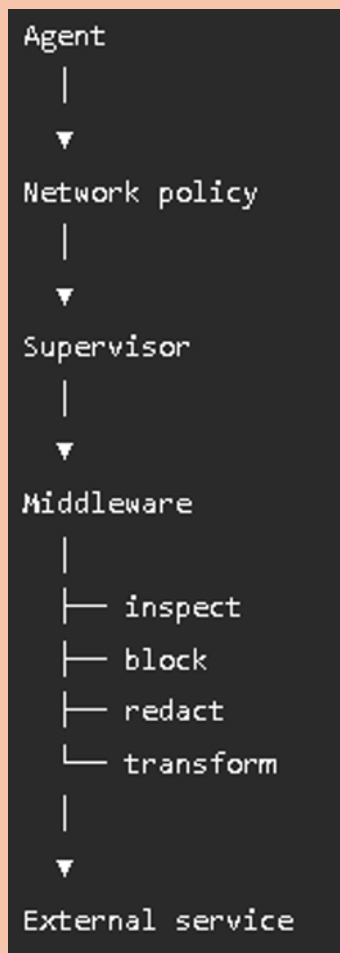


Fig 10 (right) Middleware Protection

The agent does not necessarily need unrestricted visibility of the underlying credential. This is particularly valuable for autonomous agents because the agent's ability to use a service can be separated from its ability to see and exfiltrate the service credential.

### 2.2.2 Middleware Protection

OpenShell can also provide middleware at the supervisor level. This enables inspection or transformation of supported traffic after basic network authorization but before communication with the external service. For example, see Fig. 10. This creates an additional layer of control. Network policy checks if this program is allowed to communicate with this service while middleware layer checks if a particular request or response is or is not acceptable.

### 2.2.3 Process Privileges & Policy Protection

OpenShell also uses conventional Linux privilege-reduction mechanisms. The agent can be run: without root privileges, without Linux capabilities, with no-new-privileges protections and inside appropriate process isolation mechanisms. The principle is based on least privilege and default deny. Even if the agent attempts to execute an operation, the underlying process should possess only the authority necessary for its permitted task.

OpenShell policies describe what an agent is permitted to access. A conceptual policy might look like:

**Filesystem:**

|            |              |
|------------|--------------|
| /workspace | READ + WRITE |
| /etc       | DENY         |

**Processes:**

|                  |       |
|------------------|-------|
| approved tools   | ALLOW |
| privileged tools | DENY  |

**Network:**

|                 |       |
|-----------------|-------|
| github.com      | ALLOW |
| malicious.sites | DENY  |

**Credentials:**

|              |       |
|--------------|-------|
| approved API | ALLOW |
| unrestricted | DENY  |

The key principle is that permissions are explicitly granted rather than assumed. This is particularly important for autonomous agents because their behavior can change dynamically based on prompts, retrieved information, tool results, and generated code.

Runtime enforcement is only one part of the problem. Another problem is what happens if someone changes the safety policy incorrectly? A policy change could accidentally expand the agent's harmful capabilities. NVIDIA's OpenShell architecture uses a Policy Prover concept for formally evaluating proposed policy changes. This creates two distinct safety mechanisms:

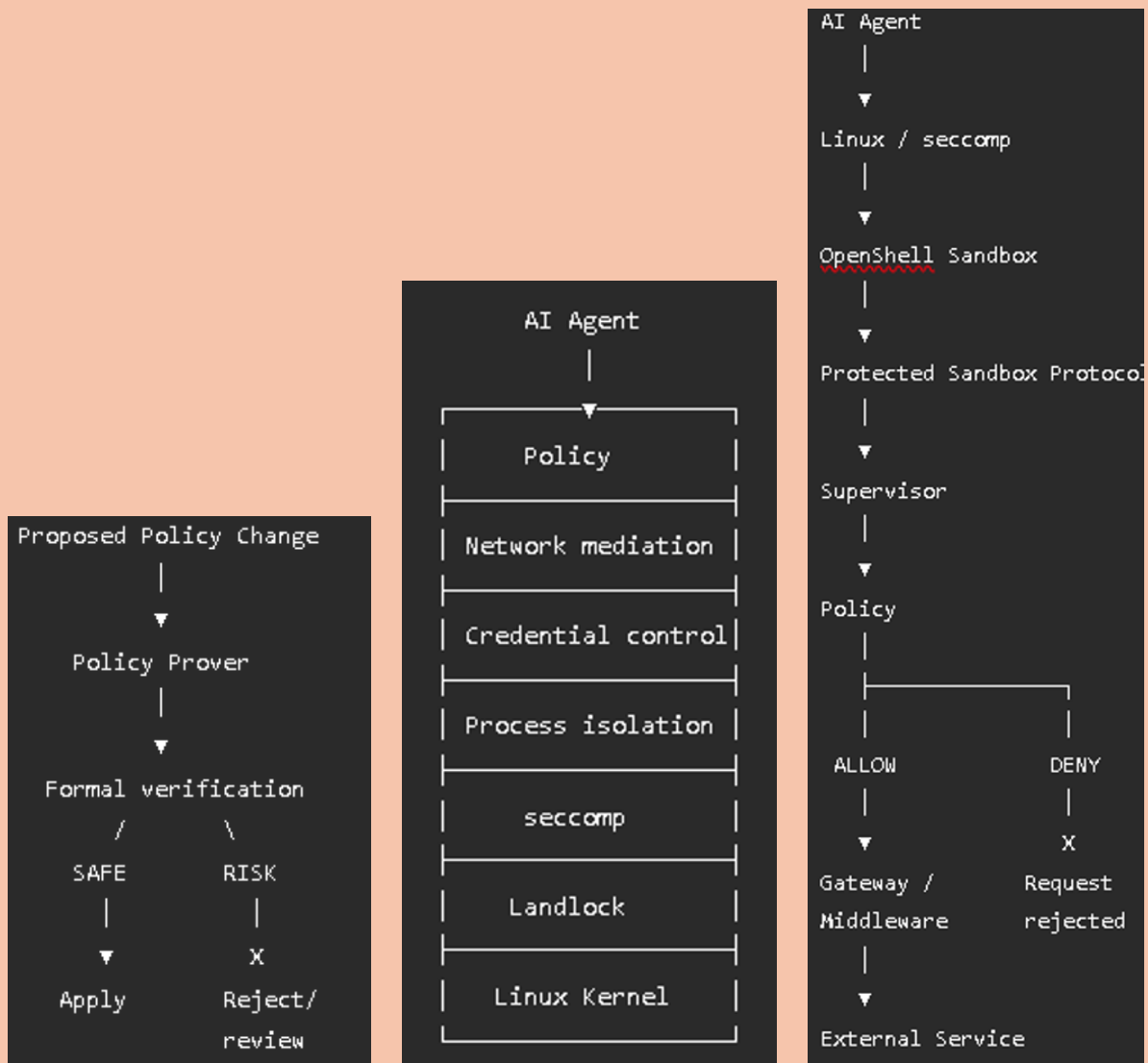


Fig 11 (left): Policy protection via policy prover

Fig 12 (middle): Defense in depth through multiple layers co-operation

Fig 13 (right): Control path of multi-layered safety protection

- A. **Runtime safety:** where it controls what the agent can actually do.
- B. **Policy safety:** where it controls whether a proposed expansion of authority should be accepted.

This distinction is fundamental and useful.

### 2.2.4 Defense-in-Depth Protection

OpenShell does not depend on one safety mechanism. Instead, multiple layers cooperate together to provide defense in depth. If one layer fails to prevent an undesirable action, another layer may still prevent it.

For example, consider an AI coding agent that wants to download information from an external software repository.

- **Step 1: Agent generates a request**  
The AI agent decides to access an external API.
- **Step 2: Request enters the sandboxed environment**  
The request originates from the agent process running inside OpenShell.
- **Step 3: Linux security (safety) controls apply**  
The relevant system-call and networking controls are applied.
- **Step 4: OpenShell identifies the request**  
The sandbox can associate the request with the requesting program.
- **Step 5: Supervisor receives the request**  
The request crosses the protected sandbox/supervisor channel.
- **Step 6: Policy is evaluated**  
The supervisor determines whether the destination and request are permitted.
- **Step 7: Credentials are handled**  
If required, approved credentials can be supplied through the controlled path rather than exposing unrestricted secrets to the agent.
- **Step 8: External connection is established**  
If permitted, the supervisor establishes the upstream connection.
- **Step 9: Traffic is returned**

The response is relayed back through the controlled path, as shown in Fig 13. This provides multi-layered in-depth safety protection.

Table 1 below summarizes what OpenShell can protect against.

| Threat                                  | OpenShell mechanism                                              |
|-----------------------------------------|------------------------------------------------------------------|
| Unauthorized filesystem access          | Landlock                                                         |
| Excessive process privileges            | Non-root execution, capabilities restrictions, no-new-privileges |
| Unauthorized system calls               | seccomp                                                          |
| Unauthorized network connections        | Network isolation and mediation                                  |
| Unauthorized DNS/network requests       | Network policy and mediation                                     |
| Uncontrolled external API access        | Supervisor policy                                                |
| Credential exposure                     | Controlled credential handling                                   |
| Dangerous HTTP/API requests             | Middleware and request policy                                    |
| Unsafe policy expansion                 | Policy verification                                              |
| Agent attempting to bypass its controls | Enforcement outside the agent process                            |

Table 1: Openshell safety features

### 3. Discussions

#### 3.1 OpenShell and the Linux Kernel

It is important not to describe OpenShell as being "inside Linux." A technically accurate description is: OpenShell is a user-space security software that programs and coordinates Linux kernel security mechanisms to enforce restrictions around an AI-agent workload.

The relationship is that OpenShell invokes Linux APIs, establishes landlock rules, establishes seccomp controls, configures process isolation, manages the sandbox which in turn went into linux kernel to enforce restrictions. The kernel, therefore, ultimately has authority over operations performed by the process.

### 3.2 OpenShell and OASP

OpenShell should also be distinguished from NVIDIA's broader Open Agent Safety Platform (OASP) concept. OpenShell represents the runtime/software enforcement layer. Other OASP components address additional aspects of monitoring and hardware/platform-level enforcement. Therefore, when analyzing NVIDIA's AI-agent safety architecture, it is useful to study OpenShell separately from the broader OASP system.

### 3.3 OpenShell, SAIOP and AISM

OpenShell and SAIOP operate at related but different architectural levels. The Safe AI Operating Point (SAIOP) defines the safe operating point/boundary for AI before humans lose control. It is a boundary that all AI systems must not cross to avoid losing control and hitting catastrophe. Hence, SAIOP operates at the upper layer and has broader implications on AI Governance and control.

OpenShell, on the other hand, provides concrete runtime mechanisms for enforcing portions of such a boundary. OpenShell, therefore, should not be viewed simply as an alternative name for SAIOP. OpenShell and Sentry can operate as the underlying layers to support the over-arching governance enforcement provided by SAIOP.

Our earlier paper on “Total AI Safety” outlines a new and integrated AI Safety architecture where SAIOP and AISM (AI Safety Model) can work in tandem with NVIDIA's OASP to provide comprehensive AI Safety. The paper illustrated that an AI-agent safety is not simply an AI-model problem or a runtime problem. **AI safety cannot be solved by benchmarking or evaluation work alone.** The AI safety community should take note of this.

A model may be aligned and still require an execution boundary. A runtime may be secure and still require an appropriate policy. A policy may be well designed and still require reliable enforcement. Effective AI safety therefore involves multiple layers. The SAFE AI Foundation's AISM covers input, model, output and compliance safety. This broadens the safety coverage to a greater extent not covered by runtime agent platform safety.

Comprehensive AI safety requires multiple safety layers to work in tandem in a practical and realizable way. We will be releasing additional documentation on AISM to further explain how AISM incorporates input, model, output, and compliance safety layers to work in tandem with these other layers (SAIOP sitting on top, and OpenShell and Sentry below).

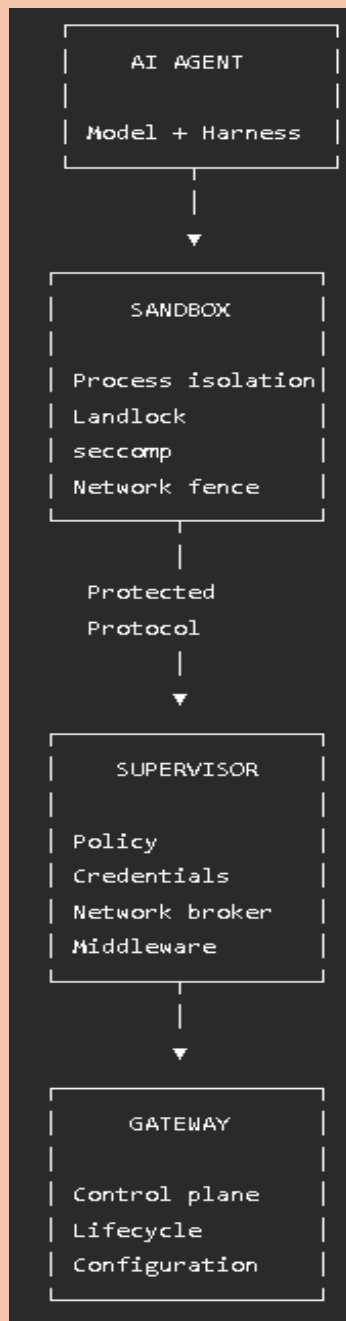
## 4. Conclusions

NVIDIA OpenShell represents a significant shift in how autonomous AI agents can be secured in a practical way. It is through real implementation, not just a conceptual design work or another research paper. Rather than attempting to make the AI agent itself responsible for enforcing its own restrictions, OpenShell creates an external enforcement underlying architecture. Its principal elements include: rust-based systems software, sandboxed agent execution, linux kernel enforcement, landlock filesystem controls, seccomp system-call controls, process and privilege restrictions, network isolation and mediation, trusted supervisory components, credential protection, middleware for higher-level inspection, declarative security policies and policy verification. OpenShell, on its own, is insufficient to solve the existing AI safety problems as it lacks model safety and AI governance. However, integrating SAIOP and AISM with OpenShell and Sentry results in an integrated AI safety architecture that provides comprehensive AI safety at multiple layers.

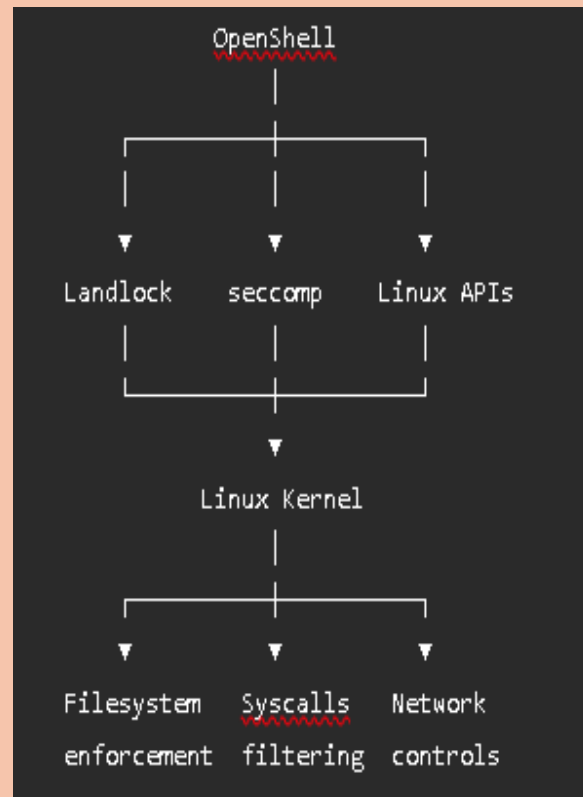
## References

1. NVIDIA OpenShell documentation — Architecture.
2. NVIDIA OpenShell documentation — How OpenShell Works.
3. NVIDIA OpenShell documentation — Policies.
4. NVIDIA OpenShell documentation — Support Matrix and Linux requirements.
5. NVIDIA OpenShell GitHub repository.
6. NVIDIA OpenShell Sandbox Architecture documentation.
7. NVIDIA OpenShell Supervisor Middleware documentation.
8. SAFE AI Foundation “Total AI Safety Architecture”, October 2026. Available at: [https://files.safeaifoundation.com/Total\\_AI\\_Safety.pdf](https://files.safeaifoundation.com/Total_AI_Safety.pdf)
9. SAFE AI Operating Point (SAIOP): The Upper Safe Limit of AI before humans lose control. Available at: <https://files.safeaifoundation.com/AI-SAIOP-paper2026.pdf>
10. SAFE AI Foundation’s Input-Model-Output-Compliance AI Safety Model (AISM). See brief explanation: <https://safeaifoundation.com/ai-benchmarking>
11. SAFE AI Foundation – “AI Safety is not AI Security”, Technical Report, September 2026. PDF available at: <https://files.safeaifoundation.com/AI-Security-Safety-Paper.pdf>

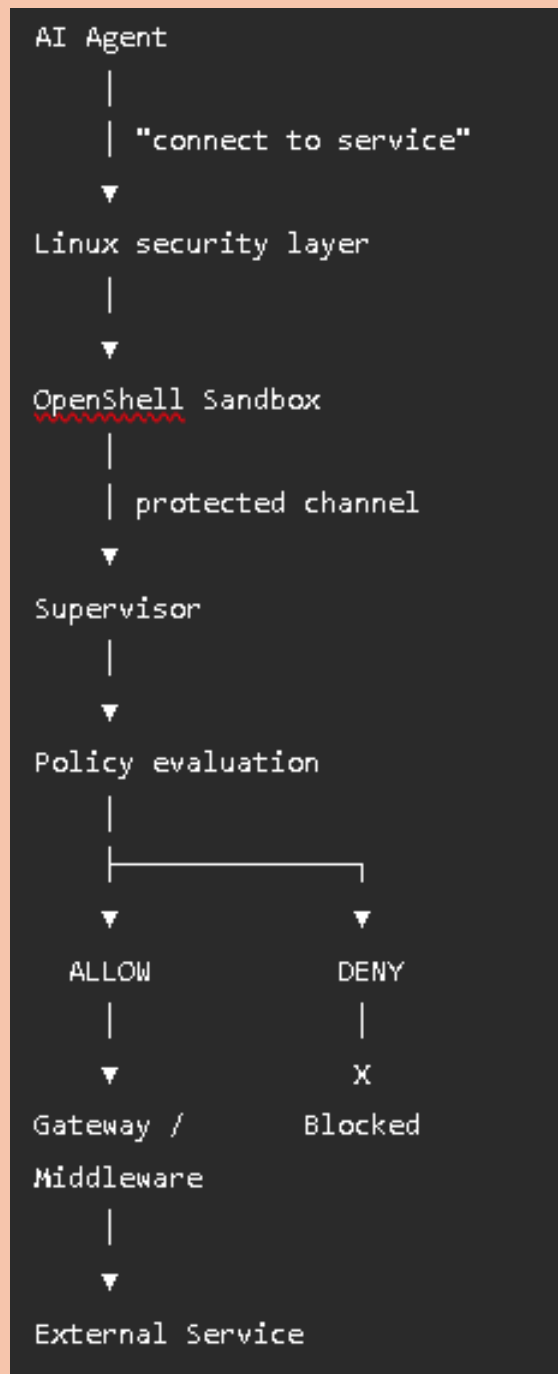
## Appendix A — OpenShell Architecture at a Glance



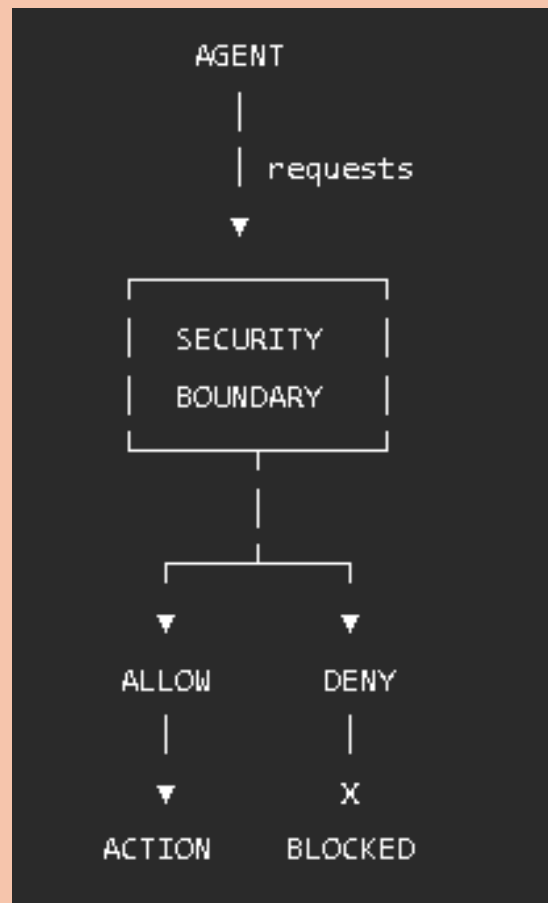
## Appendix B — Kernel Interaction



## Appendix C — Network Request



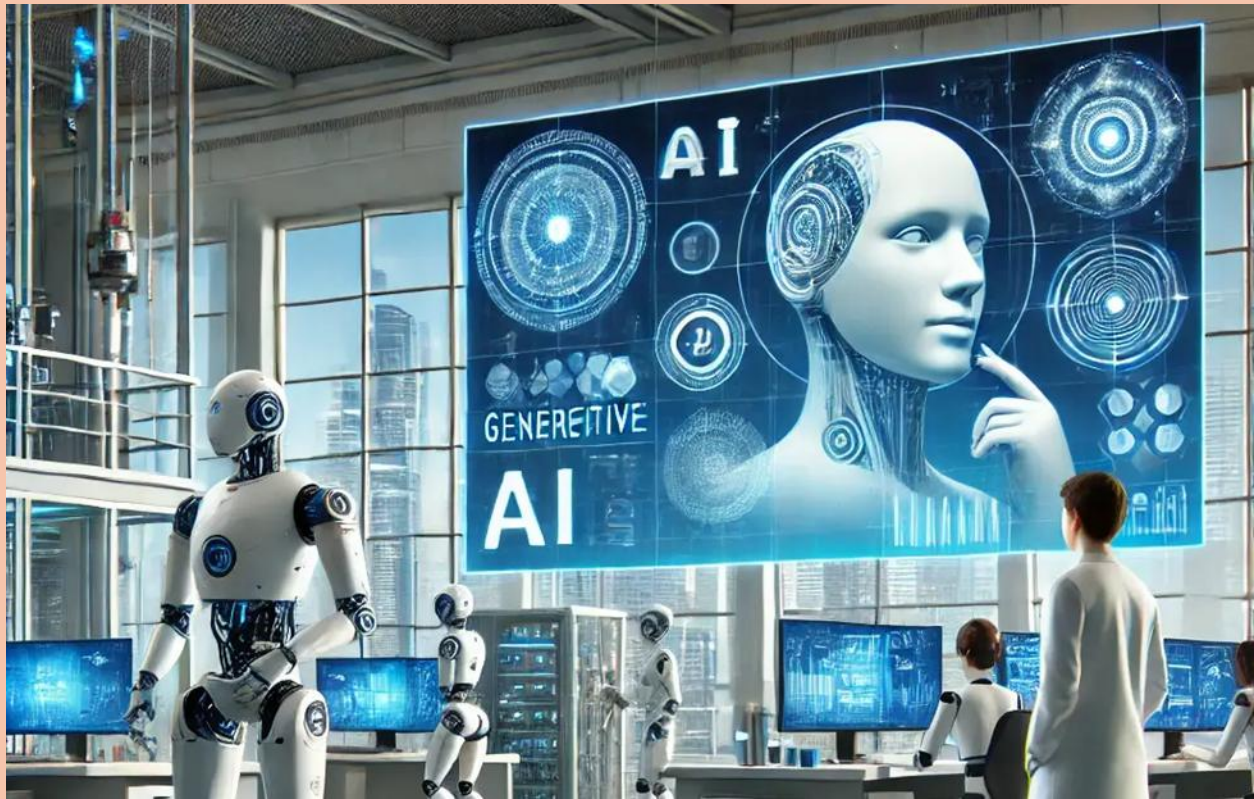
## Appendix D — Core Architectural Principle



The AI agent is not the final authority over its own operating environment.

**Disclaimer:** This technical report is provided free of charge by the **SAFE AI Foundation USA**. It requires strong computer science background for better understanding. ChatGPT has been used to create some of our diagrams to save time. The information in this report is provided “as it is”, by **the SAFE AI FOUNDATION**, USA. The use of the information provided here is subject to the user’s own risk, accountability, and responsibility. The **SAFE AI FOUNDATION USA** and the author are not responsible for the use of the information by the user or reader. Please reference this article if you wish to cite it elsewhere.

**Note:** We do not work with private consultancy firms, universities, or any third parties to produce this technical document. All copyrights are reserved by the **SAFE AI Foundation USA**. Further enquires about this work can be made via email to the author and foundation’s executive director Dr. Chai K Toh at: [contact@safeaifoundation.com](mailto:contact@safeaifoundation.com)



# SAFE AI FOUNDATION

[www.safeaifoundation.com](http://www.safeaifoundation.com)

The SAFE AI Foundation was formed in 2025, in the heart of the Silicon Valley, California, USA. Its mission is to embrace the AI revolution and promote the use of good AI to improve on the quality of life and the economy. It will address the issues of bad AI and its impacts on society. The foundation will inform, educate, and mitigate the dangers of bad AI and help the society to embrace good AI. The foundation will interact with industries, universities and public sector to jointly advance the progress of Good AI. The foundation is a non-profit 501c(3) organization, registered in the State of California, USA.