Fail-Closed Provider Execution for Governed AI Agent Runtimes

Gustavo Venegas*, Edison Vazquez*, Danilo Naranjo†, Benjamin Gonzalez*
*Chilean Chamber of Artificial Intelligence †Ocular
gustavo@gobiernoia.cl, edison@jhedai.com, dan@ocularsolution.com, ben@nirvana-ai.com

***Abstract*—Governed AI agent runtimes interleave model calls, tool decisions, and provider operations, and a single request that crosses the provider boundary can trigger external calls, data movement, network access, or credential use. A recurring hazard is that declared provider metadata is treated as executable authority: a contract that names an external provider is allowed to select it. We present ArgorixLang's *fail-closed provider boundary*, which separates declarative provider contracts from executable provider adapters and applies capability, policy, and runtime-profile checks before any provider operation is planned. Only providers explicitly enrolled in an executable registry are callable; a named contract is metadata, not authority. We give a lightweight formal model of the planning rule—** $\text{Plan} \Rightarrow \text{MayExecute}$ **, with** $\text{Declared} \not\Rightarrow \text{Executable}$ **—and an authorization algorithm in which any unknown reference, denied capability, failed policy, or non-executable provider terminates in a recorded denial or review state rather than falling through to execution. On the existing ArgorixLang corpus of 33 request directories (27 complete, 6 source-only), the evaluated profile is simulated-only: every complete trace records an external-provider execution blocked event, network denied, and secrets/key-material denied. We hold a strict boundary: VM-level denial does not prove operating-system sandboxing, container isolation, host network containment, secret-store isolation, adapter correctness, absence of side effects, or production isolation.**

***Index Terms*—AI agents, governed runtime, fail-closed execution, provider boundary, runtime security, policy enforcement, capability checks, agent governance, negative controls.**

I. Introduction

The provider boundary is the most consequential decision point in a governed AI agent runtime. Everything upstream of it—prompt parsing, planning, tool selection—is internal deliberation; the moment an agent *executes* a provider operation, the runtime can issue an external call, move data off the host, open a network connection, or read credential material. If that boundary is permissive, the rest of the governance stack is decoration: a system that carefully records policy decisions but then lets any declared provider run has recorded nothing that constrains behavior.

The specific failure mode we target is the conflation of *declaration* with *authority*. Agent programs, prompts, and configuration routinely *name* providers—an OpenAI provider, a search tool, a remote model. A naive runtime treats the appearance of a provider name as permission to call it, so that adding a line of metadata, or coaxing the model into emitting one, selects an external execution path. Governed runtimes must not allow prompts, source-level declarations, or contract metadata to *become* executable authority. A declared provider contract should describe an intended boundary; it should not, by itself, make a provider callable.

ArgorixLang [1], [2] enforces this separation at the virtual-machine level. Its provider boundary maintains two distinct registries: a set of *declarative provider contracts* (capabilities, limits, intended operations) and a set of *executable provider adapters* (callable implementations). A contract's presence does not enroll its provider for execution. Before any operation is planned, the runtime evaluates capability constraints, policy permission, and a runtime profile, and consults the executable registry. The governing rule is *fail-closed*: when a reference is unknown, a capability is incompatible, a policy denies or requires review, or a provider is not executable, the request path terminates in an auditable denial or review state rather than falling through to a provider call.

We are deliberate about what this does and does not establish. Fail-closed here is a control-flow property of the implemented VM/application path: missing authorization does not reach execution. It is *not* a proof of operating-system sandboxing, container isolation, host-level network containment, secret-store isolation, live-provider safety, adapter correctness, or absence of side effects outside instrumentation. We keep that boundary explicit throughout.

Contributions.

1) A fail-closed execution model for governed AI agent runtimes, in which absence of authorization yields denial or review rather than fallthrough execution.
2) A provider-boundary design that separates declarative contracts from executable adapters, with an executable registry as the sole source of callable providers.
3) A lightweight formal model of provider-execution authorization, stated as an interpretation of the implemented planning rule rather than a verification proof.
4) An observational evaluation on the existing ArgorixLang corpus showing blocked external execution, network denial, and secret/key-material denial.
5) A strict claim-boundary analysis distinguishing VM-level denial from production isolation.

A scope note distinguishing this paper from the companion Agent Passport and EvidenceBundle papers is given in Section XIII.

II. Background and Motivation

A governed agent runtime is a composition of model calls, tools, provider adapters, policies, and evidence. These layers

fail differently, and the provider layer fails hardest, because it is the only one whose outputs leave the runtime. A wrong tool plan can be re-planned; an executed external provider call cannot be un-sent.

It helps to separate concepts that informal usage collapses. A *provider declaration* is the appearance of a provider name in a program or request. A *provider contract* is structured metadata describing a provider's capabilities, operation limits, and intended boundary. An *executable adapter* is a runtime-callable implementation. A *capability permission* is an agent's right to perform an operation class. A *runtime profile* is the per-execution policy on network, secrets, key material, tools, and external execution. A *policy approval* is a decision over a specific operation. And *actual external execution* is the act of invoking a live provider. These are not synonyms, and a safe runtime requires *all* of the gating conditions to hold before reaching the last one.

Fail-open behavior—defaulting to execution when a check is missing, unrecognized, or errors—is dangerous precisely because the dangerous path is the default. A misconfigured registry entry, an unhandled lookup miss, or an ambiguous policy result becomes an external call. Fail-closed inverts the default: the absence of a positive authorization is itself a denial. This makes *negative controls*—recorded events showing that an operation was blocked, that network was denied, that secrets were withheld—first-class evidence. A negative control is positive evidence of denial behavior. It is not, and we do not read it as, proof of host-level containment.

III. Provider Boundary Model

We define the boundary in terms of six elements; Table I summarizes their implemented role and what each does *not* claim.

(1) Declarative provider contract. A contract names a provider and states its capabilities, operation limits, and intended boundary. It is metadata only. A contract is inert with respect to execution: it cannot, by itself, cause a provider call. In the evaluated chatbot artifacts, an `OpenAIProvider` contract and an `OpenAISandbox` descriptor are present as declarations.

(2) Executable provider adapter. An adapter is a runtime-callable implementation. It is reachable only if explicitly enrolled in the executable provider registry, and even then only after capability, policy, and runtime-profile checks succeed.

(3) Executable provider registry. The registry is the sole source of callable providers. A provider named in a contract is not callable unless it also appears in the registry. In the observed configuration the only executable provider is `simulated`; external providers named by declarations remain non-executable.

(4) Runtime profile. The profile denies, for the demonstrated path, network access, plaintext secrets, key material, tools, agent execution, and external execution, and requires audit, evidence, security reports, and sandbox metadata. It is consulted before planning.

TABLE I
Provider boundary concepts.

Concept	Meaning	Implemented role	Not claimed
Declarative contract	Provider name, capabilities, limits	Inert metadata; intended boundary	Executable authority
Executable adapter	Callable implementation	Reachable only if registered	Adapter correctness/safety
Executable registry	Set of callable providers	Sole source of execution; `simulated` only	Live external providers
Runtime profile	Per-run denials	Denies net/secrets/keys/ext. exec	Host-level containment
Capability check	Agent's right to an op	Gate before planning	Identity verification
Boundary event	Recorded denial/block	Auditable negative control	OS sandbox proof

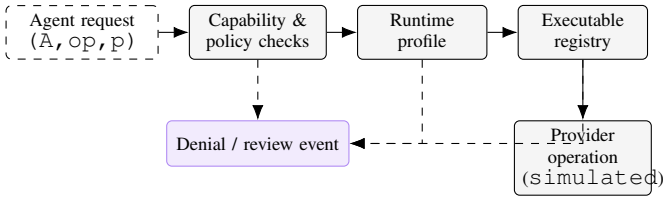

Fig. 1. Provider boundary architecture. An agent request passes capability/policy checks, the runtime profile, and the executable registry before any provider operation is planned; any failing gate diverts to a recorded denial or review event.

(5) Provider boundary events. The runtime emits auditable events at the boundary, including an external-provider execution blocked event, a runtime network denied event, and a runtime secrets denied event, each recorded in the trace.

(6) The key distinction. The model's load-bearing statement is that declaration does not imply executability:

$$\mathrm{Declared}(p) \;\not\Rightarrow\; \mathrm{Executable}(p).$$

A contract may name an external provider; a contract does not make a provider executable.

IV. Formal Model

We give a lightweight formalization. It is a precise *runtime interpretation* of the implemented fail-closed boundary, not a full formal-verification proof.

Definitions. Let A be the requesting agent, op the requested operation, and p the target provider. Let C be the set of declared provider contracts and E the executable provider registry. Define predicates: $\mathrm{Declared}(p) :\Leftrightarrow p \in C$; $\mathrm{Executable}(p) :\Leftrightarrow p \in E$; $\mathrm{Cap}(A, op)$, agent A holds the capability for op; $\mathrm{PolicyPermit}(A, op, p)$, the policy permits the operation; $\mathrm{RuntimePermit}(op, p)$, the runtime profile permits it. Let $\mathrm{Plan}(A, op, p)$ mean the VM may plan execution of op on p, and $\mathrm{Deny}(A, op, p)$ mean the VM terminates the request path with a denial or review event.

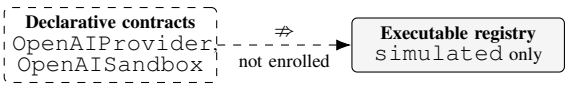

Fig. 2. Declarative vs. executable split. Contract metadata (left) names providers but does not enroll them; only the executable registry (right) yields callable providers. A contract does not imply an executable adapter.

TABLE II
FAIL-CLOSED AUTHORIZATION PREDICATES.

Predicate	Definition	Interpretation
$\mathrm{Executable}(p)$	$p \in E$	Provider enrolled as callable
$\mathrm{Declared}(p)$	$p \in C$	Provider named by a contract
MayExecute	Exec $\wedge$ Cap $\wedge$ Policy $\wedge$ Runtime	All gates hold
Plan $\Rightarrow$ MayExecute	planning needs authorization	Fail-closed rule
$\neg$MayExecute $\Rightarrow$ Deny	contrapositive	No fallthrough
Declared $\not\Rightarrow$ Executable	contract $\neq$ adapter	Boundary statement

Authorization predicate.

$$\begin{aligned}\mathrm{MayExecute}(A, op, p) := \mathrm{Executable}(p) \\ \wedge\ \mathrm{Cap}(A, op) \\ \wedge\ \mathrm{PolicyPermit}(A, op, p) \\ \wedge\ \mathrm{RuntimePermit}(op, p)\end{aligned}$$

Fail-closed planning rule. Planning requires authorization, and its contrapositive is denial:

$$\mathrm{Plan}(A, op, p) \ \Rightarrow\ \mathrm{MayExecute}(A, op, p),$$

$$\neg\mathrm{MayExecute}(A, op, p) \ \Rightarrow\ \mathrm{Deny}(A, op, p).$$

Declaration is not executability. The boundary statement and its consequence:

$$\mathrm{Declared}(p) \ \not\Rightarrow\ \mathrm{Executable}(p),$$

$$\mathrm{Declared}(p) \wedge \neg\mathrm{Executable}(p) \ \Rightarrow\ \neg\mathrm{MayExecute}(A, op, p).$$

In particular, an external provider that is declared but not enrolled is blocked:

$$\begin{aligned}\mathrm{ExternalProviderDeclared}(p) \wedge p \notin E \\ \Rightarrow\ \mathsf{ExternalProviderExecutionBlocked}.\end{aligned}$$

Reading. The conjunction in MayExecute is the formal content of fail-closed: every conjunct must hold to plan, so the failure (or absence) of any one forces Deny. We do not present this as proof of authenticity, isolation, or external truth; it is a statement about the implemented control flow, in which non-authorization cannot reach a provider call. Table II restates the predicates.

```
Algorithm 1: Fail-Closed Provider Planning
Input : agent A, operation op, provider p,
        registry E, policy state Pi,
        runtime profile R
Output: plan or deny

 1. Resolve A, op, p.
 2. if A unknown or op unknown: deny.
 3. if p not in E:
        emit ExternalProviderExecutionBlocked; deny.
 4. if not Cap(A, op): deny.
 5. if not PolicyPermit(A, op, p):
        require review or deny.
 6. if not RuntimePermit(op, p):
        emit relevant denial event; deny.
 7. plan op only if (2)-(6) all pass.
 8. record decision in trace + security report.
```

Fig. 3. Fail-closed provider planning. Each gate can only deny or pass; planning is reached only when every gate passes.

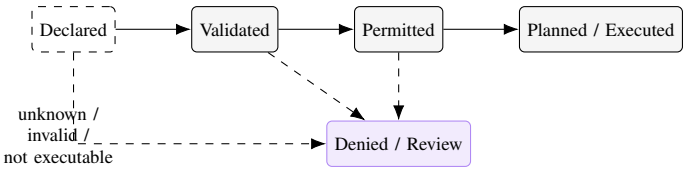

Fig. 4. Fail-closed state machine. A request advances Declared $\rightarrow$ Validated $\rightarrow$ Permitted $\rightarrow$ Planned/Executed only while every gate passes; any unknown, invalid, non-executable, or denied condition diverts to a recorded Denied/Review state.

V. Runtime Execution Sequence

The runtime evaluates a validated program rather than accepting unrestricted provider calls. The path proceeds: (1) load validated bytecode; (2) resolve the requesting agent and handler; (3) resolve target provider metadata; (4) evaluate capability constraints; (5) evaluate policy constraints; (6) consult the runtime profile; (7) consult the executable provider registry; (8) plan the operation only if all checks pass; (9) otherwise emit a denial or review event and do not execute; (10) record trace, security report, and evidence artifacts. Runtime state is retained in an outcome object even when execution fails, so a report can describe the denial without converting it into success. Algorithm 3 states the planning decision; Figure 4 shows the state machine.

VI. Negative Controls

The evaluated profile exhibits a set of *negative controls*: recorded outcomes in which an operation was withheld. These include external-provider execution blocked, network denied, secrets denied, key material denied, simulated-only provider execution, and tools/agent-execution disabled for the demonstrated path; policy may additionally place a request in a review-required state.

We read negative controls as *positive evidence of denial behavior*: the runtime did not merely fail to call an external provider, it recorded that it blocked one, denied network, and withheld secrets. That recorded refusal is auditable and reproducible from the artifacts. What a negative control is *not* is proof of operating-system-level containment: the runtime

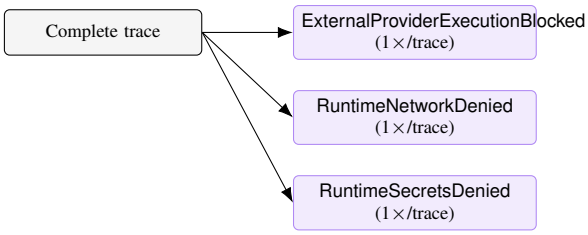

Fig. 5. Negative control evidence. Each complete trace records the boundary denial events as auditable negative controls; they are evidence of denial behavior, not OS-level containment.

did not contact the provider, but this study does not instrument the host to prove that no other process path could. The negative control evidence is summarized in Figure 5 and quantified in Section VII.

VII. Observational Evaluation

We reuse the existing ArgorixLang runtime corpus [1] and add no new experiment, benchmark, or deployment.

A. Corpus

Of **33** request directories, **27** are *complete* and **6** are *source-only* and not assessable for trace, security, or evidence fields. All 27 complete sessions report execution status *completed*, and all 27 complete evidence bundles pass offline verification (27/27). The corpus contains **7,155** counted ledger events and **1,188** detailed policy violations across the complete sessions.

B. Provider boundary and negative controls

The evaluated runtime profile is *simulated-only*: the executable provider registry contains only the `simulated` provider, while declarative contracts name external providers that remain non-executable. Each complete trace records one ExternalProviderExecutionBlocked event, one RuntimeNetworkDenied, and one RuntimeSecretsDenied; key material is likewise denied. The four observed runtime controls are application-level behaviors (Table III).

C. Policy results

The detailed policy objects report `policy_passed=false` and `review_required=true` for the complete sessions. A favorable coarse status field does not override these detailed results: an explicit failure or required review prevents an aggregate pass.

D. Interpretation

The observed profile demonstrates VM-level denial and bounded provider planning: external execution is blocked, network and secrets are denied, and only a simulated provider is callable. It does *not* demonstrate live-provider safety, OS-level sandboxing, or production isolation. Source-only directories remain *unavailable*, never successful executions; a missing artifact is an explicit incomplete observation, not a pass. The repeated structure supports pipeline reproducibility but limits any claim of behavioral diversity.

TABLE III
Observed runtime controls.

Control / event	Observed value	Interpretation
Provider execution	`simulated` only	Bounded executable registry
External execution	Blocked (1×/trace)	Denial recorded, not isolation
Network access	Denied (1×/trace)	VM-level egress refusal
Secrets / key material	Denied (1×/trace)	No plaintext secret/key use
Complete directories	27 / 33	Assessable sessions
Source-only directories	6 / 33	Unavailable, not passes
Bundles verified offline	27 / 27	Internal consistency only
Ledger events	7,155	Structured event emission
Policy violations	1,188	Negatives preserved
Detailed policy result	passed=false, review=true	Review state preserved

VIII. Security Analysis

We analyze threats as attack paths across trust boundaries, not as proof that a named control defeats every adversary. Table IV states each bounded mitigation alongside its remaining gap.

Threats. A prompt- or source-level attempt to *select* an external provider; confusing a declarative contract with an executable adapter; capability escalation; policy bypass; network egress through a live adapter; secret or key leakage; a malicious executable adapter once registered; registry misconfiguration; runtime-status confusion (reading a coarse pass over a detailed failure); host-level side effects outside instrumentation; and consumer overinterpretation of VM denial as production security.

Bounded mitigations. The split between declarative and executable registries prevents a contract from becoming callable merely by being named. The executable registry is the sole execution source; capability and policy checks gate planning; the runtime profile denies network, secrets, key material, and external execution; the fail-closed rule forces denial on any missing authorization; and boundary denials are recorded as auditable events preserved in evidence artifacts.

Remaining risks. A malicious or defective adapter, *once enrolled*, is within the trust boundary and is not evaluated here. OS-level leakage, host network misconfiguration, and secret-store exposure are outside the instrumented path. Evidence bundles are unsigned, so an adversary controlling artifacts and bundle together can produce a self-consistent replacement. There is no controlled adversarial test and no proof that all side effects are blocked. The strongest supported statement is conditional: within the observed runtime path, provider selection is mediated by capability, policy, and an executable-provider allowlist; denials are recorded; and complete artifacts are offline-consistent. Security outside that path remains unmeasured.

IX. Related Work

We compare carefully and claim no equivalence. OPA [3] is an external policy *decision* engine evaluating Rego;

TABLE IV
THREATS AND BOUNDED MITIGATIONS.

Threat	Bounded mitigation	Remaining gap
Select external provider via metadata	Declarative/executable split; registry is sole source	Adapter behavior once enrolled
Capability escalation	Capability check before planning	No identity verification
Policy bypass	Policy permit / review before planning	Decision not canonicalized
Network egress	Runtime profile denies network	No host egress proof
Secret / key leakage	Secret and key-material denial	No secret-store isolation
Malicious adapter	Registry enrollment required	Registered adapter untested
Status confusion	Detail dominates coarse status	Schema-level precedence pending
Overinterpretation	Explicit claim boundary	Consumer discipline required

TABLE V
DECLARED VS. EXECUTABLE PROVIDER DISTINCTION.

Statement	Supported?	Explanation
A contract may name an external provider	Yes	Contracts are metadata
A named contract makes a provider callable	No	Declared $\not\Rightarrow$ Executable
Only registered providers are executable	Yes	Registry is sole source
The evaluated profile is simulated-only	Yes	`simulated` is the only entry
Declared-but-unenrolled provider is blocked	Yes	ExternalProvider-ExecutionBlocked
Blocking proves OS-level isolation	No	VM-level control only

ArgorixLang instead compiles and records runtime policy and provider decisions inside its own artifacts rather than externalizing the decision. WASI [4] provides a capability-oriented host interface with a sandbox model; the ArgorixLang fail-closed boundary is a VM/application-level control and is not a proof of WASI-style host sandboxing. MCP and A2A [5], [6] standardize agent interaction and interoperability surfaces; this paper concerns local provider execution control, not live interoperability. in-toto [7] provides signed supply-chain provenance binding steps to keys; ArgorixLang records runtime evidence but offers no supply-chain signer guarantee. The OWASP Top 10 for LLM Applications [8] catalogs prompt-injection and tool/plugin risks; ArgorixLang constrains operation planning through capability and policy checks but does not claim universal injection prevention. The NIST AI RMF [9] frames risk management and is not a certification we claim to satisfy. Agent languages such as Jason/AgentSpeak [10] focus on agent behavior and reasoning execution; this paper focuses narrowly on the provider boundary and fail-closed runtime enforcement. The ArgorixLang implementation is in Rust [11].

X. LIMITATIONS

We state limitations strictly. There is *no* operating-system sandbox proof; *no* process or container isolation proof; *no* host-level network containment proof; *no* secret-store isolation proof; *no* malicious-adapter evaluation; *no* live-provider execution evaluation; *no* controlled adversarial prompt experiment; *no* guarantee of the absence of side effects outside the instrumented runtime; *no* compliance certification; *no* identity or credential verification; *no* blockchain immutability; *no* post-quantum security; and *no* production-readiness claim. The complete sessions are structurally uniform, supporting reproducibility but limiting generalization across prompts, policies, providers, and deployments. Fail-closed means only that, within the implemented VM/application path, missing authorization, unsupported providers, denied capabilities, or non-executable adapters terminate in denial or review rather than falling through to execution.

XI. FUTURE WORK

Natural extensions, none claimed as present: host-level sandbox measurement; a WASI and container comparison; a live provider adapter exercised inside a controlled sandbox; adapter capability manifests; a signed provider adapter registry; runtime egress monitoring; secret-store integration tests; differential testing of fail-closed behavior across runtime versions; an adversarial prompt and source corpus; policy-decision canonicalization into a single tri-state result; an independent audit harness; and formal verification of the planning rule.

XII. CONCLUSION

Fail-closed provider execution gives governed AI agent runtimes a safer local planning boundary: declarations do not become executable authority, executable providers must be explicitly registered, and capability, policy, and runtime-profile checks precede operation planning. On the existing corpus, the evaluated profile is simulated-only, external execution is blocked, and network and secret/key material are denied—auditable negative controls that demonstrate VM-level denial behavior. Production isolation and host-level security are out of scope and remain future work: a verified denial in the VM is exactly that, and not a claim about the operating system, the host network, or live providers.

XIII. SCOPE

This paper differs from the companion Agent Passport paper by focusing on provider execution control rather than sovereign metadata, and from the companion EvidenceBundle paper by focusing on the runtime decision boundary—how capability, policy, runtime profile, and the executable registry gate provider planning—rather than offline digest verification. All empirical values are reused from the original ArgorixLang corpus [1], and no new experiment, deployment, or security guarantee is introduced.

REFERENCES

[1] G. Venegas, E. Vazquez, D. Naranjo, and B. Gonzalez, "ArgorixLang: Compiled governance, sovereign agent metadata, and offline-verifiable runtime evidence," Zenodo preprint, 2026, original ArgorixLang preprint; source material for this derived paper. [Online]. Available: https://doi.org/10.5281/zenodo.21014780

[2] Argorix Labs, "ArgorixLang," 2026, secure, verifiable programming language and runtime for AI-agent systems. [Online]. Available: https://github.com/argorixlabs/argorixlang

[3] Open Policy Agent Project, "Open policy agent (OPA)," Official project documentation, 2026, general-purpose policy engine and Rego policy-as-code language. [Online]. Available: https://www.openpolicyagent.org/docs

[4] WASI Subgroup, "Introduction," Official standards portal, 2026, standards-track APIs and capability-based sandbox model. [Online]. Available: https://wasi.dev/

[5] Model Context Protocol Contributors, "Model context protocol specification, version 2025-06-18," Protocol specification, 2025. [Online]. Available: https://modelcontextprotocol.io/specification/2025-06-18

[6] A2A Protocol Working Group, "Agent2agent (A2A) protocol specification," Protocol specification, 2026. [Online]. Available: https://a2a-protocol.org/v1.0.0/specification/

[7] S. Torres-Arias, H. Afzali, T. K. Kuppusamy, R. Curtmola, and J. Cappos, "in-toto: Providing farm-to-table guarantees for bits and bytes," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1393–1410. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/torres-arias

[8] OWASP Foundation, "OWASP top 10 for large language model applications," Security guidance, 2025. [Online]. Available: https://owasp.org/www-project-top-10-for-large-language-model-applications/

[9] E. Tabassi, "Artificial intelligence risk management framework (AI RMF 1.0)," National Institute of Standards and Technology, NIST AI 100-1, Jan. 2023. [Online]. Available: https://doi.org/10.6028/NIST.AI.100-1

[10] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons, 2007. [Online]. Available: https://doi.org/10.1002/9780470061848

[11] Rust Project, "Rust programming language," Official project website, 2026. [Online]. Available: https://rust-lang.org/