2nd Workshop on Safe AI (SafeAI) | UAI 2026, Amsterdam | 21 August 2026

# Policies on paths

Securing AI agents at runtime, and a grammar for their actions

**Maurits Kaptein**
Eindhoven University of Technology | Kyvvu

TU/e | kyvvu.

m.c.kaptein@tue.nl

# About me

- **TU/e** — statistics and machine learning. Broadly: reinforcement learning, contextual bandit (cMAB) problems, causal inference, efficient computation, and — more recently — AI / agent security.
- **Kyvvu** — a well-funded, fast-growing start-up (and hiring) building an *Agent Security Kernel*: a new security layer for AI agents.

This talk is a mix of engineering and theory.

# Outline

**Setup.** What an agent is, what "securing" one means, and where in the system security can live.

**Part I — Theory.** Paths as the object of interest; policies as functions on paths; the scores they induce; and the some interesting open questions.

**Part II — Practice.** To run this across many agents you need a shared vocabulary of actions — the Agent Action Grammar (AAG) — and an open standard behind it.

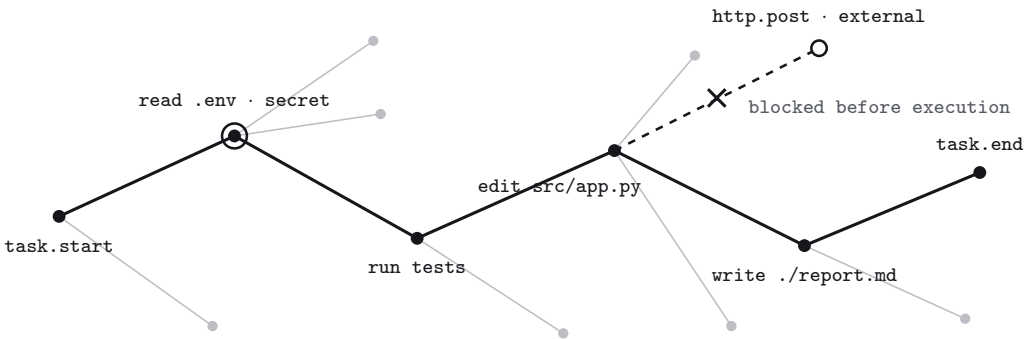
# Aim and definitions

**Agent.** Software that, given a *plan* produced by a language model, uses a *harness* to parse and execute it. The plan is a sequence of steps (tool calls); the harness runs them, feeds back results, and loops.

**Path.** One run of an agent is a *path*: a sequence of steps together with their arguments. The set of all paths is every sequence the agent could execute — unbounded, and often not very useful to enumerate.

**Securing an agent.** Not every path is acceptable. A *secure* agent is one that only follows *desirable* paths — those consistent with what it is permitted to do (confidentiality, integrity, availability; cf. Saltzer & Schroeder, 1975).

# A path, and the paths not taken



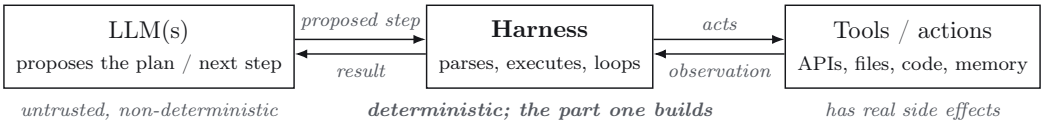
# What is new about agents

Agents take actions, as software always has. The difference is where the control flow comes from.

- Traditional software: the sequence of actions is written by a programmer *at design time* — reviewable, testable, enumerable in advance.
- Agents: the sequence is chosen by the model *at runtime*. Control flow is *data* — produced step by step, differently on every run.
- So the path is not known ahead of time, and the same task yields different paths on different runs.

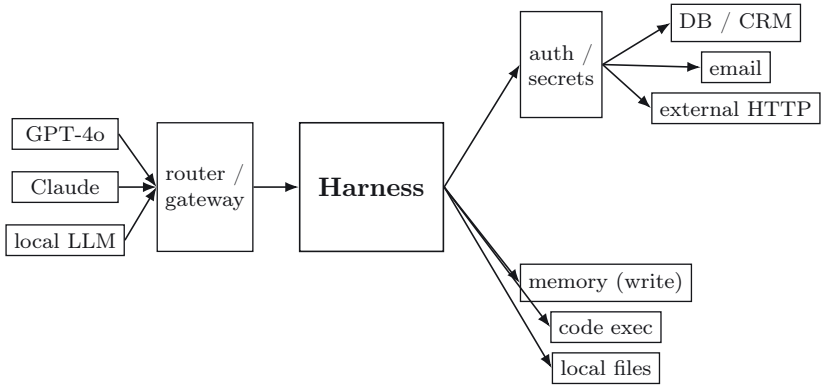
Many things bear on agent security — injection, sandboxing, identity, supply chain, etc. The runtime-decided path is one of the novel challenges.

# The anatomy of an agent



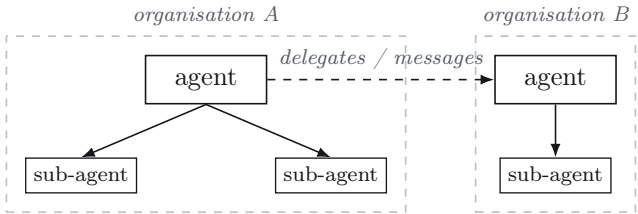

- The model produces a *plan*; it is not a record of what actually happened.
- The harness is the component that both *executes* and therefore *observes* every action.

# A realistic single agent



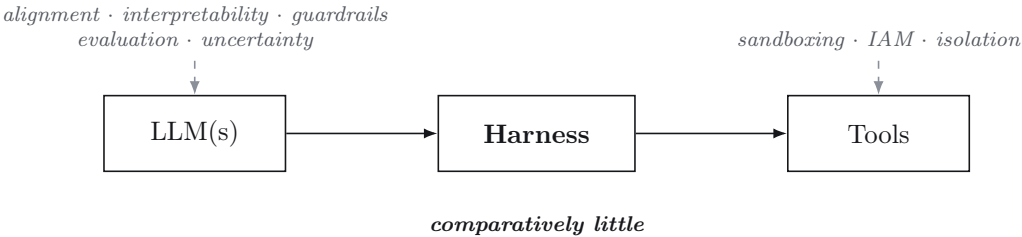

Several models behind a router / gateway; one harness; an auth / secrets layer in front of *some* tools (external ones) but not others (memory, code, local files).

# Fleets of agents




- Agents spawn sub-agents, delegate, and message other organisations' agents; a path can run across agents one does not own.
- Note we lack even a settled definition: what exactly is one *agent* versus a *sub-agent*, or a fleet?

# Where safety research concentrates




- Most safety research targets the **model**: alignment / RLHF, interpretability, guardrails and safety classifiers, evaluation, and uncertainty quantification.
- Classical security targets the **tools**: sandboxing, scoped credentials, IAM.

## The harness as a further layer

The model-side and tool-side work above is essential. The harness is a *further* layer at which to add security — complementary to it, not a correction of it.

- **It is the layer builders actually control.** The model is often third-party and non-deterministic — out of your hands, and not something you can fully bound. The harness is code you write and can inspect.
- **It is where the path is visible.** The harness executes every action, so it is the natural place to observe — and reason about — the sequence of actions, which is the object we care about.

Part I

# Theory: policies on paths

# Formalising the path

From the informal picture (a path is a sequence of steps with arguments) to something we can compute on.

A **step** is a triple

$$s = (\tau,\, d_{\text{in}},\, d_{\text{out}}) \in \mathcal{T} \times \mathcal{D} \times \mathcal{D},$$

with $\tau$ a step *type* (model call, tool call, memory op, …) from a finite set $\mathcal{T}$, and $d_{\text{in}}, d_{\text{out}}$ the input and output data.

A **path** is a sequence $P = (s_1, \ldots, s_n)$. The **partial path** $P_i = (s_1, \ldots, s_i)$ is what is observable when the next step must be decided.

The **proposed next step** is $s^* = (\tau^*, d^*_{\text{in}})$ — its type $\tau^*$ (e.g. "send external email") and input, with the output not yet known because it has not executed. This is the object a runtime check sees.

## Policies, and the scores they induce

A **policy** is a deterministic function

$$\pi_j(A,\ P_i,\ s^*,\ \Sigma)\ \longrightarrow\ [0,1],$$

read as the probability that executing $s^*$, here on this path, violates policy $j$. Arguments: agent $A$, partial path $P_i$, proposed step $s^*$, and shared organisational state $\Sigma$ (other agents, audit log).

Existing mechanisms are special cases: access control depends only on the agent $A$ and the proposed step's *type* $\tau^*$, ignoring $P_i$ and $\Sigma$.

Aggregate across the active policies, then along the path, into a **score**:

$$v_i\ =\ 1-\prod_j\big(1-\pi_j(A,P_i,s^*,\Sigma)\big),\qquad V(P_i)\ =\ 1-\prod_{k\le i}(1-v_k)\ \in[0,1].$$

Kaptein, Khan & Podstavnychy (2026), *Policies on Paths*, arXiv:2603.16586.

## Open questions (largely statistical)

The framework gives a clean target object; it says little about how to estimate or decide well.

1. **Calibration.** $\pi_j$ is treated as a probability, but is produced as a score. Calibrating it to be meaningful is challenging.
2. **Policy generation.** How to exclude as many undesirable paths as possible with as few, and as simple, rules as possible — and evaluate them quickly?
3. **Behavioural metric.** Track $V(P_i)$ over time as a summary of behaviour: drift, anomalies, a fleet moving out of distribution.
4. **Decision-making under risk.** Given scores, costs, and a risk appetite, which action — allow, redirect, block, escalate — is optimal, within and across agents?

Part II

# Practice: a grammar of actions

# From formalism to a running system

The formalism says nothing about implementation. Running it against a live agent needs two concrete things.

**You need paths.**

An actual, ordered stream of actions — and, if a policy is to be portable across harnesses, actions that *mean the same thing* everywhere.

**You need policies.**

Functions $\pi_j$ over *classes* of paths, not one hand-written rule per concrete call.

## Writing policies, in practice

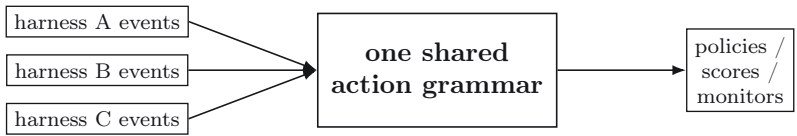
A useful policy is a predicate over trajectories: it must match *every* path of a forbidden shape, whatever the concrete steps in between.

```
# "no external write once the path has touched classified data"
trigger: step.message (verb: POST) # a send
rule: target is external
        AND path_contains( step.resource reading
                           data.classification = confidential )
decision: block
```

- One rule covers unboundedly many concrete paths (any read-then-send, any tools, any order between). This is how we write them today — see `docs.kyvvu.com/core-concepts/policies`.
- It works, but leaves open questions: which rules, how few, how fast, how learned (see Part I).

# Paths need a shared vocabulary of actions

A path is a sequence of *actions*. For a policy to be portable, and for us to reason or aggregate across a fleet of diverse agents, an action must carry the same meaning everywhere.




- Today each harness emits its own events (OpenAI `tool_calls`, Anthropic `tool_use`, Gemini `functionCall`, …); every consumer re-integrates per vendor.
- The goal: write a policy once and have it hold over any conforming agent.

# The Agent Action Grammar (AAG)

A small, *closed* vocabulary for what an agent *does* while carrying out a task — so diverse agents can be reasoned over uniformly. It *describes*; it does not enforce.

- Emitted by the **harness**, not the model — the model states a plan, not what happened. A **task** is a stream of actions sharing one `task_id` (and one memory scope).
- **Twelve action types, closed**: `task.*` lifecycle (`start`, `end`, `error`, `idle`) and `step.*` behaviours (`resource`, `message`, `model`, `self`, `credential`, `exec`, `gate`, `unknown`).
- Each step is a triple {`properties`, `input`, `output`}. An absent `output` marks an **intended** action (pre-execution) — so a check can act *before* the effect; with an output it is **completed**.
- A **verb** (`GET / POST / PATCH / DELETE`) records data-flow direction — does task data leave, or does external data enter — not the literal HTTP method.

# How the AAG differs from OpenTelemetry

A natural question: is this not just tracing? The AAG composes with OpenTelemetry — a task maps to a trace, an action to a span — but adds two things a span cannot carry.

1. **The intended action.** An OTLP span is, by construction, a record of something that *already happened*. The AAG's output-less *intended* action is the moment a security layer allows or denies, *before* any effect.
2. **A closed vocabulary.** AAG's twelve types and four verbs are a fixed set a policy can rely on; OTLP span names are an open surface, advisory once projected.

## RFC-0001 — open for comment

The action model is deliberately small and stable; the genuinely unsettled points live in a public RFC. This is where I would value the community's input.

- **Fixed properties** — which leaves (e.g. `target.trust`, `data.classification`) to promote from suggested to required, and when.
- **Recording blocked actions** — how a denied intended action leaves an unambiguous trace, so "was blocked" and "never happened" are not confusable.
- **Input / output granularity** — raw content vs. a summary / hash / reference, given confidentiality and storage.
- **Asserted vs. verified properties** — marking which values the harness *observed* rather than was *told*; a verb for `step.exec`; version negotiation; and multi-agent representation.

Spec and RFC: `github.com/Kyvvu/AAG`.

Comment via GitHub Discussions or an issue referencing `RFC-0001`.

## Take-aways

1. An agent chooses its **path** at runtime; the object worth securing is the path, not the isolated action.
2. The harness is **another layer** of defence — complementary to alignment, guardrails, and evaluation, and often the layer a builder actually controls.
3. **Theory.** A policy is $\pi_j(A, P_i, s^*, \Sigma) \to [0, 1]$. This can be aggregated into a score $V$. Most existing mechanisms are special cases. There are nice open questions — calibration, policy generation, decision-making under risk, etc.
4. **Practice.** Running this across a fleet needs a shared, closed vocabulary of actions: the AAG (harness-emitted, intended / completed, distinct from observability). Its RFC is open.

Paper: arXiv:2603.16586 | AAG: `github.com/Kyvvu/AAG` | `docs.kyvvu.com`

# Thank you

Questions, and comments on the RFC, are welcome.

TU/e | kyvvu.

m.c.kaptein@tue.nl