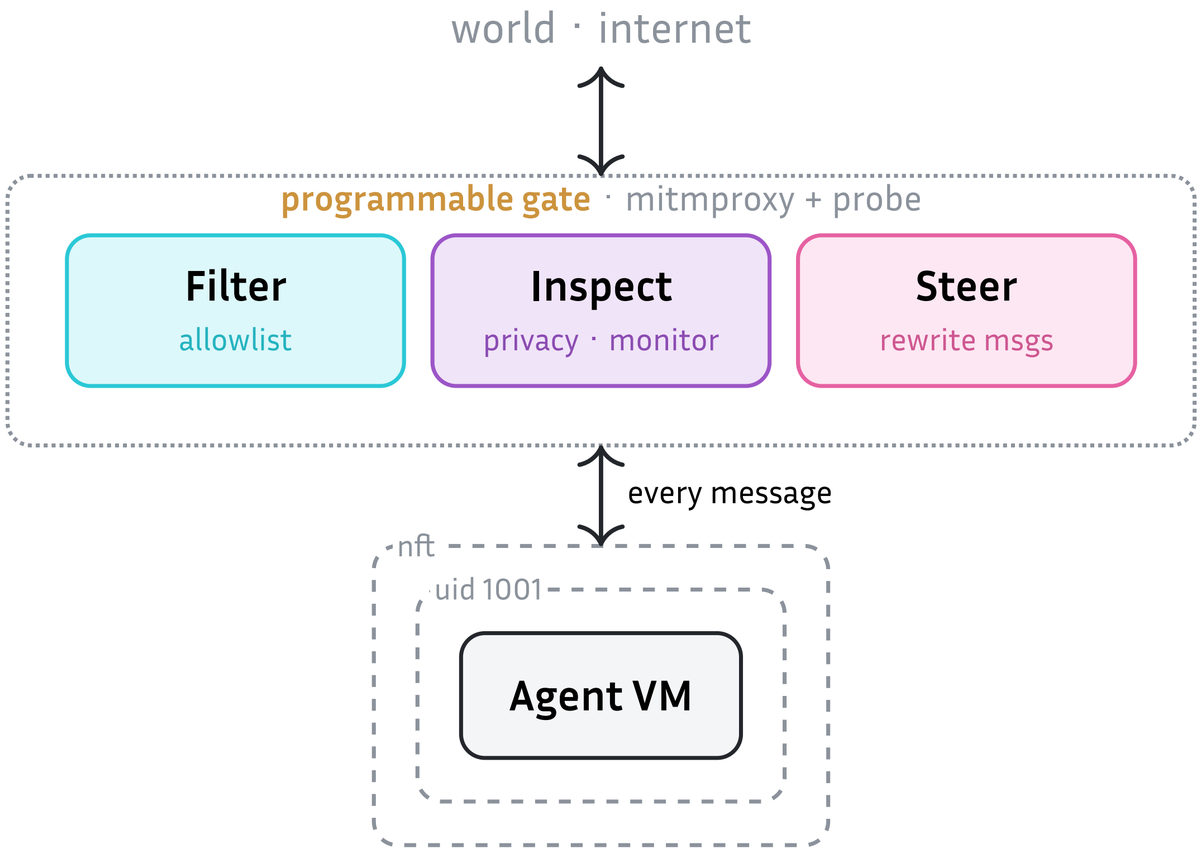
Steerability via constraints: a substrate for scalable oversight of coding agents

Thomas Winninger
Télécom SudParis & ENS Paris-Saclay

Context

Sandboxes secure the agent *while* it writes code:



But the sandbox ends when you need to execute the code yourself. The bottleneck is no longer capability, it is **verification**: can a human, or a trusted small model, audit the code before it runs?

Formal methods answer this (Lean, Rust + Verus, TLA$^+$), but they assume a language built for proofs. What about **experimental Python**, PyTorch, CUDA, code that changes daily, where none of that stack applies?

Decades of human engineering already solved oversight at scale: access control, network policy, strict conventions enforced by tooling. We apply the same to the agent's **substrate** rather than its prompt:

- a start-to-end system built from tools each under 100 LoC;
- a small set of Python substrate conventions that make oversight tractable;
- a controlled experiment lifting a small reviewer's recall from **54.5%** to **90.9%** (Gemma 4 e4b, 8B-class).

Method — steerability via constraints

Prompts are advisory — the model may honor them or not, and a decade of adversarial research says that will not change soon.

Substrate constraints are not: linters, type checkers, runtime contract validators and frozen dataclasses (`ruff`, `pyrefly`, `beartype`, `pydantic`) refuse to run code that violates them. The same logic holds at the machine layer — the VM, egress filter and inference proxy above. Two moves:

1. Enforce conventions in the substrate, not in an `AGENTS.md` or `ARCHITECTURE.md` that drifts as the code grows. A rule the linter already excludes is one neither the agent nor the reviewer must remember, and a function's contract becomes textually visible without prose:

Kind	Unconstrained	Constrained
Linters	`ruff` defaults	`ruff` (security, bugbear, bandit), `pyrefly`
Runtime checks	none	`beartype`, `pydantic` strict
Typing	Plain `Tensor`	`Float[Tensor, "b d"]` (jaxtyping)
Dataclass	Plain	Frozen, slotted and kw-only
Collections	`list`, `dict`	`tuple`, `Mapping`
Discriminators	`kind: str`	`Literal[...]`
Invariants	In `ARCHITECTURE.md`	In docstrings
Layout	Flat	Pure `core/` with impure `runners/`

2. Source as the detailed specification. A 200-LoC `docs` CLI projects the repository as a topological map — headers, docstrings, and module `Invariants:` — so reviewing a function needs only that function plus its module header, never the whole codebase. Invariants sit next to the code they constrain, not in a central file a reviewer never re-reads.

Experiment

A 2×2 study — **substrate × tools** — on 11 backdoors stratified by detection span (L1 local syntactic → L5 global invariant), each silent on the clean test suite. A fixed 8B reviewer (Gemma 4 e4b) inspects the codebase **one function at a time**.

→ Writing constrained code makes it harder to hide backdoors, and easier to detect them.

Setup	Caught	Recall	L1-2	L3	L4-5
Unconstrained, no tools	6/11	54.5%	3/4	2/3	1/4
Unconstrained, docs CLI	9/11	81.8%	3/4	3/3	3/4
Constrained, no tools	9/11	81.8%	3/4	2/3	4/4
Constrained, docs CLI	10/11	90.9%	3/4	3/3	4/4

L1 is local syntactic, **L2** is local semantic, **L3** is multi-file flow, **L4** is in-module invariant violation, and **L5** is global invariant violation.

Recall climbs **54.5% → 90.9%**; substrate and tools contribute **independently**. Constraints alone cost precision — the reviewer over-flags without an efficient way to check invariants — and the `docs` CLI restores it. The review **protocol** matters as much as the substrate: a naive single-pass reviewer reverses the effect.

Limitations

- Experiment is preliminary, synthetic and small
- Constraints might lower accuracy → need for a full capability experiment.
- While the reviewer was chosen small to fit the scalable oversight setup, bigger models might behave differently → need for bigger models.

thomas.winninger@telecom-sudparis.eu