Table of Contents

Table of Contents

TL;DR: Uses AI agents to lift trusted formal semantics of real languages into Lean 4, decomposing each language into a surface AST, a minimal core, and a desugaring between them, then validating by differential testing against the reference implementation. Targets Pkl (warm-up) and Zig's comptime evaluator (main). Output: a Lean 4 semantic-lifting toolkit and validated semantics. Venue: PLDI 2027.

Simon Henniger + Nada Amin (Harvard University) | Claimcheck: Towards More Reliable LLM-Based Formalization

TL;DR: Improves LLM-driven formalization (natural language to formal spec) by detecting 'intent drift', where a generated spec subtly fails to capture what the user meant. Builds on the claimcheck round-tripping technique (informalize a spec, then compare it to the original NL) and extends it with new techniques, benchmarks, and domains. Output: an open-source tool plus a submission to OOPSLA (Oct 2026) or another conference.

Quinn Dougherty (Forall R&D) | Apart x Atlas | Verifying non-interference preservation across MLIR lowering

TL;DR: The MLIR compilation pipeline that turns an ML model into an executable is an under-defended attack surface. Chen et al. (S&P 2026) showed lowering passes can break integrity by injecting backdoors; this project addresses the parallel confidentiality problem, where lowering passes introduce information channels (data-dependent control flow, memory aliasing, dynamic shapes) that leak protected model weights even when the source program is non-interferent. Existing MLIR translation-validation tools (mlir-tv, Regehr PLDI'25) check semantic equivalence, not non-interference; the project extends them with a self-composition SMT encoding plus per-value security labels, and proves a syntactically-checkable 'data-independent lowering' subset sound in Lean 4. Evaluated on ~10 HEIR secret-dialect lowerings, aiming to surface at least one leak class in upstream lowerings. Output: a translation-validation tool for non-interference preservation across MLIR lowering. Target: CSF 2026, with TACAS or FMCAD as backup.

Ryan Marinelli (University of Oslo) | Geometric Canaries: Detecting Adversarial Failures in LLM-Assisted Formal Verification

TL;DR: Detects a failure mode where adversarially perturbed theorem statements steer LLMs toward plausible-but-invalid proofs. Applies geometric chain-of-thought analysis (two metrics: Average Angle Error and Number of Mistakes) to flag reasoning instability mid-generation, before the proof checker runs, giving a lightweight 'canary' for any LLM-assisted verification pipeline. Output: a peer-reviewed paper plus an open-source Python toolkit. Target: CSF 2027 or TACAS 2027.

Luiza Cristina Corpaci (TU Wien / AMD) | Semantic Diff-ing in LLM-Assisted Formal Representations

TL;DR: Catches a silent failure mode: LLM translations between formal representations that compile and pass tests but violate the semantic invariants that made the original meaningful (valid but not faithful). Builds a semantic-diff checker that tests whether translations preserve equational invariants, using SMT/Lean over the Equational Theories Project as a decidable ground-truth fragment. Output: semantic-diff and semantic unit-test frameworks. Target: TACAS (paper Oct 15 / artifact Oct 30, 2026).

Shaowei Lin (Beneficial AI Foundation) | Deductive Vericoding

TL;DR: Starts from the Curry-Howard idea that proofs are programs, so code can be synthesized deductively the same way tactics synthesize proofs. Builds on the SuSLik deductive-synthesis approach and a double-categorical framework, arguing this scales better than the usual generate-then-verify loop for correct-by-construction verified code. Output: a public Lean library for deductive vericoding plus a blog post on using it with AI agents.

Jocelyn Qiaochu Chen (NYU / University of Alberta) | Mining Enforceable Specifications for LLM-Agent Tool Use

TL;DR: The rules governing LLM-agent tool use are scattered across prompts, tool descriptions, tool code, orchestrators, and runtime config. This project extracts those

latent rules, classifies each by what is needed to check it (static, action-history, data-flow, or runtime), and compiles the checkable ones into validators, stateful monitors, provenance checks, or runtime guards. Validated on SQL, file-system, and web-retrieval agents. Output: a framework and middleware prototype plus a paper targeting FSE, OOPSLA, or TACAS.

Tomasz Nguyen (Confluent / IBM) | BabelBench

TL;DR: Tests whether LLMs reason about proofs structurally or just pattern-match on system-specific syntax, which matters because spec-validation pipelines depend on faithful translation between formal systems. Builds a benchmark of ~150 paired theorems measuring within-paradigm (Coq to Lean, TLAPS to Dafny) versus cross-paradigm (type-theoretic to model-theoretic) proof translation, with verifier-grounded scoring and a predicted asymmetric failure pattern. Output: the benchmark plus a workshop or conference paper. Target: TACAS 2027 (ETAPS, early Oct 2026).

Max von Hippel (Anduril Industries) | Agent Permission System

TL;DR: Replaces 'LLM-as-judge' agent safety (no formal guarantees, fragile to prompt injection) with a permission and runtime-monitoring system. Policies are written in a high-level DSL, compiled to monitorable LTLf formulas, and machine-checked sound in Lean 4, combining Claude Code-style per-tool prompts with AWS-IAM-style expressiveness plus information-flow labels. Evaluated on adversarial benchmarks (AgentDojo, AI control). Output: an open-source implementation plus a short paper at ETAPS/TACAS 2027.

Kiran Gopinathan (Basis Research Institute) | Fuzzing Proof Assistants

TL;DR: Every theorem Lean 4 checks rests on a trusted computing base (C++ runtime, elaborator, type checker, compiler) that the proofs cannot themselves verify, so a bug there can silently invalidate everything. Builds systematic fuzzing infrastructure targeting each pipeline stage via coverage-guided and grammar-aware test generation

(preliminary work already found a heap overflow affecting every Lean 4 version). Output: an open-source fuzzing toolkit plus an academic paper. Target: ICSE 2027 (~Oct 2026 deadline).

Jeffrey Chang + Kanghee (Theorem) | Bootstrapping Semantics with Mutation Testing

TL;DR: Aims to bootstrap program semantics automatically, with no human oversight, by building a harness that tells models whether the specification they generated is correct. Correctness is tested by sampling mutations of the source program and checking behavioral equivalence against the candidate spec (a good spec accepts the program but rejects behavior-changing mutants). The fellowship work is empirical ML: building a dataset, running evals, and finetuning open models.

Shriram Krishnamurthi | Formalizing properties

Shriram Krishnamurthi (Brown University) | Formalizing properties using PICK

This is a shorter version of the full proposal here: open in Google Docs

Summary

A central part of applying formal methods is obtaining formal properties. Even when people have a sense of what properties they want, they have struggled to express these in the formal notations demanded of tools. Now, help is in sight: language models are proving quite effective at translating prose to formal properties.

But how do we know whether we got the properties right? As Perlis said, we can't proceed from the informal to the formal through purely formal means. We need a methodology by which people can judge the correctness of the resulting property.

We have been working on a methodology called PICK that helps with this. The goal would be to expand PICK to a variety of formal property languages.

Target output

My target output is both systems and (if the work yields interesting research fruit) papers.

I think it's premature and inappropriate to focus on conferences and deadlines at this point. Those follow once the work is done and we see what got done, when, and what is the most appropriate view. I have extensive experience mentoring people at various levels to publication, and will apply that experience in this process.

Theory of impact

Formal methods tooling depends on getting properties right. Indeed, they are indifferent to the correctness of properties: even if a property does not reflect the user's actual intent, a synthesis tool will gladly generate the wrong system, and a verification tool will gladly give false confidence. Thus, an essential precursor to using formal methods is to make sure the properties are indeed the ones that the user desires. Yet, this obvious and central problem has received almost no attention in the long history of formal methods.

Desired expertise and skill sets

The ideal candidate will be a good programmer and will have some familiarity with one ore more formal methods specification languages (including logics, dependent types, etc.). They must be willing to experiment, but must also be able to work in a contemplative way.

What will not work well is the kind of person who spawns a dozen agentic tasks and then just passes along the work without further inspection. We'd much rather work with a person who runs just a single agent and intersperses its use with long periods of reflection.

High-level background

PICK builds on a variety of work in cognitive science, formal language theory, and synthesis. It is inspired by CEGIS, CEGAR, and RLHF, but also by works in perceptual psychology. We can provide a full reading list to participants.

Detailed technical background

We have a prototype of PICK in place (which runs in VSCode). The most visible version of it generates regular expressions, though we now have versions of it that produce access-control policies and LTL expressions. A paper on PICK has been accepted but is not yet formally in print, so the best thing to read is our blog post of the regex tool (and mentally extrapolate from that to other property languages): https://blog.brownplt.org/2025/12/11/pick-regex.html

Proposed methodology

We have already formalized these things in the PICK paper, which will soon be public.

Proposed timeline / milestones

I don’t think this project is going to work along such strict lines. First, it has a real exploratory dimension. Second, it depends a lot on the individual, their background, their availability, etc. Someone who is currently between jobs and able to work 40 hours/week is very different from someone doing it as a side quest while holding down a regular job.

Preliminary experiments

Again, the PICK paper describes several user studies that show its effectiveness.

Risks, uncertainties, monitoring & mitigation

Ultimately, the problem we face is a cognitive one. We need to do things that help people see things for themselves, to uncover latent beliefs. That is a fundamentally hard problem. It requires thought and attention to detail. Many people are poor at this.

I have been describing (and building towards) what I call the Responsible Programmer (RP) model. The RP is still human; they have all the frailties that the rest of us do. But they actually care about getting things right, and are willing to exert more than an epsilon of effort (but not much more) in that direction. How best can we help them?

But it is possible that no such RP exists, or there are far too few of them for it to matter, or that even for an RP, the load is just too high. I think the main output then will be a negative scientific contribution: this is just too hard to do.

In addition, PICK depends on certain key mathematical properties to be enjoyed by the property language, which not all languages enjoy. We will need to use approximations in those cases. That necessarily introduces some errors, which in turn may lead to false confidence. Will this be problematic? Will this be more problematic than the start-of-the-art (hopes and prayers)? We can’t know until we’ve tried.

Shriram Krishnamurthi | Specs from examples

Shriram Krishnamurthi (Brown University) | Specs from examples using Alloy/Forge

This is a shorter version of the full proposal here: open in Google Docs

Summary

I have been interested in the problem of specification elicitation for decades now. I consider it the dark underbelly of formal methods: all our great tooling doesn't do anything if we don't have a specification of what we want the system to do. Yet specification elicitation is mostly ignored in favor of math, algorithms, and code.

My team has worked extensively with lightweight formal methods tools because of their power to help people with property elicitation. But these tools were still inaccessible to most developers. I think we're in a position now to make them much more accessible.

The goal is to use formal methods tooling to generate interesting concrete examples that people can examine and classify. These in turn will lead to inducing specifications, both positive and negative, of intended system behavior.

Target output

My target output is both systems and (if the work yields interesting research fruit) papers.

I think it's premature and inappropriate to focus on conferences and deadlines at this point. Those follow once the work is done and we see what got done, when, and what is the most appropriate view. I have extensive experience mentoring people at various levels to publication, and will apply that experience in this process.

Theory of impact

They will make the use of models much safer, because they will help people figure out what properties they need independent of implementations. Those properties would then feed into a variety of verification and validation technologies (property-based testing, model-based testing, model-checking, proof assistants, run-time verification, etc.) that ensure that any freedom given to the agents for generating code do not lead to untoward behavior. We can't do any of this without specs, and that's the focus here. That said, we'll be sticking to attributes that can be

captured in formal logics. This (a) misses out on some valuable properties, and (b) can lead to properties that are very difficult to check with existing v&v technologies.

Desired expertise and skill sets

You should be comfortable programming and be willing to use languages that may be new to you (e.g., Racket) to drive the project. You should have at least a little awareness of formal methods. You have to be open to experimentation, but also able to sit back and work contemplatively and thoughtfully.

Most of all, YOU should be willing to put in some thinking. If your style these days is to spin up a dozen agent instances and just pass on their output, that would be a bad fit. We'd much rather work with someone working with a single agent instance, periodically, with non-trivial periods of reflection in-between. Token-maxxing is an anti-goal.

High-level background

I think it would be very valuable to read the Lightweight Formal Methods manifesto. Though it has somewhat different aims, it is the closest to articulating a vision that we think makes sense in this space. Do things that are quick and effective, and assume partiality at all points. Our big challenge is to proceed from partiality to totality, and we can't do that if we assume totality at the outset.

Detailed technical background

Two valuable things we've done to look at are described in these blog posts on my group's site:

1. Forge: https://blog.brownplt.org/2024/04/21/forge.html
2. Differential analysis: https://blog.brownplt.org/2024/06/27/differential-analysis.html

These embody the background and kinds of systems that I think will be useful, and the intellectual and tooling background we would like to bring to bear.

Proposed methodology

In the interests of time, I'm respectfully going to pass on providing a full research plan here. Suffice it to say I've done this many, many times and can figure it out, but also, sometimes these kinds of problems require a lot more exploratory work before you can pin down the above.

Proposed timeline / milestones

I don't think this project is going to work along such strict lines. First, it has a real exploratory dimension. Second, it depends a lot on the individual, their background, their availability, etc. Someone who is currently between jobs and able to work 40 hours/week is very different from someone doing it as a side quest while holding down a regular job.

Preliminary experiments

See the above systems (Forge and Margrave). Been working on these aspects for decades, long before they mattered. 😁

Risks, uncertainties, monitoring & mitigation

Ultimately, the problem we face is a cognitive one. We need to do things that help people see things for themselves, to uncover latent beliefs. That is a fundamentally hard problem. It requires thought and attention to detail. Many people ae poor at this.

I have been describing (and building towards) what I call the Responsible Programmer (RP) model. The RP is still human; they have all the frailties that the rest of us do. But they actually care about getting things right, and are willing to exert more than an epsilon of effort (but not much more) in that direction. How best can we help them?

But it is possible that no such RP exists, or there are far too few of them for it to matter, or that even for an RP, the load is just too high. I think the main output then will be a negative scientific contribution: this is just too hard to do.

Erik Meijer (Leibniz Labs) | Universalis

Erik Meijer (Leibniz Labs) | Universalis / Automind: Provably Safe Agentic Compute

This is a shorter version of the full proposal here:
Apart Project Proposal — Provably Safe Agentic Compute

Summary

Tool-using LLM agents are unsafe by construction. Recent incidents such as a Chevy dealer chatbot agreeing to sell a Tahoe for $1, OpenAI's Operator running a $31.43 unauthorised purchase, Claude Code recursively deleting 106,120 user files, an internal Meta agent leaking confidential data to a forum, an AI Safety director's own agent ignoring three stop commands while deleting 200+ emails, show that alignment + evals + runtime guardrails are insufficient.

We propose **provably safe agentic compute**, operationalising the principle *in dubio pro securitate*: an agent's plan is unsafe until proven safe. The model emits a program in **Universalis** (a literate Prolog-style DSL); a Z3-MCP verifier checks `plan ⊆ policy` (i.e. `¬∃x. plan(x) ∧ ¬policy(x)`) before any tool fires.

We validate on standard benchmarks like AgentDojo, InjecAgentvs. NeMo-Guardrails / Invariant Labs / native function-calling baselines. Our core claim is **static detection**: every attack expressible in the policy language is caught *before* any tool executes, with quantified false-rejection rate on benign tasks. Runtime defences are bounded by what they can recognise *after* damage starts.

Target output

A static-verification layer for LLM-generated agent workflows that proves safety before execution. Target venue: **ICSE 2027** (full-research track, abstract Aug 2026 / paper Sep 4, 2026). Backup venue: **TACAS 2027** (Oct 2026) or **CSF 2027** (Feb 2027).

Motivation in one paragraph (*in dubio pro securitate*)

Criminal law presumes innocence because wrongful conviction is far worse than wrongful acquittal. The agent-safety problem is the dual: permitting an unsafe action (data breach, irreversible deletion, financial loss) costs far more than withholding a safe one (a slow agent step). The same decision rule therefore yields the opposite presumption: *in dubio pro securitate*, **unsafe until proven safe**.

We propose to operationalise this rule mechanically: every action an agent proposes must be

discharged against an explicit policy, with a machine-checkable proof, before it executes. The 2024–2026 incident record (see Summary) is the empirical evidence that the current presumption is wrong.

Theory of impact

Provably safe agentic compute

(1) eliminates whole classes of prompt-injection and tool-poisoning attacks at the source, providing deterministic guarantees that runtime guardrails, bounded by the fundamental limits of pattern matching against natural language, cannot.

(2) Unblocks safe deployment of agents in regulated domains: finance (transaction approval workflows), healthcare (patient-data handling), infrastructure (deploy/rollback).

(3) Replaces the audit-after-the-fact posture of current safety stacks (logs + evals + reactive guardrails) with auditable, machine- checkable workflows aligned with SOX, HIPAA, and AI Act compliance regimes.

(4) Composes with alignment: verification narrows the action space; alignment chooses well within it.

(5) Opens new use-cases for end-user-programmable AI for spreadsheet-class users by letting non-experts trust agents because the trust is mathematical rather than empirical.

Proposed methodology

Hypothesis

> **H1.** A generate-verify-execute pipeline using Universalis + Z3 reduces prompt-injection attack-success rate by ≥10× relative to the strongest runtime-guardrail baseline, while preserving task-success within 5 percentage points of unrestricted function calling.
>
> **H2.** Proof-carrying workflows, where the model also emits a small SMT-LIB2 proof script,reduce verifier wall-clock time by ≥5× over solver-only inference, making the verifier viable on agent-loop budgets.

Novel techniques

1. **Workflow IR.** A constrained subset of Universalis (logic-programming AST) sufficient to express the typical agent-loop idioms (sequence, conditional, bounded loop, parallel

calls) while remaining decidable under SPACER.

2. **Two-tier policy language.** Information-flow policies (source/sink with arg-conditional sinks) plus capability policies (tool-effect annotations) that compose. Both compile to Horn clauses. Operational form: refinement check `plan ⊆ policy`, discharged as `¬∃x. plan(x) ∧ ¬policy(x)` (UNSAT iff verified).

3. **Proof-carrying workflows.** Following Necula, the model emits not only the workflow but an SMT-LIB2 witness; the verifier becomes a microsecond-scale checker rather than a search engine. Asymmetry analogous to Sudoku: *check* in O(n²), *solve* NP-complete on n×n grids.

4. **Frame-condition synthesis.** A pass that augments user-stated postconditions with frame conditions to exclude "delete the world"-style frame anomalies (cf. the 200-email and 106,120 -file incidents).

Experimental design

- **Benchmarks.** AgentDojo (Debenedetti et al., NeurIPS 2024); InjecAgent (Zhan et al., ACL 2024). We additionally seed an **IncidentReplay** suite drawn from documented 2024–2026 failures (Tahoe-for-$1 prompt manipulation; Operator unauthorised purchase; Meta internal-forum leak; Yue inbox runaway; Claude Code recursive deletion; Alibaba covert-mining), translated into reproducible AgentDojo-style task harnesses.

- **Models.** Claude Sonnet 4.7, GPT-5.5, Gemini 2.5 Pro with fixed temperatures and decoding parameters across conditions
  .
- **Conditions / baselines.**
  - **Vanilla** native tool calling, no defence.
  - **Prompted defence** system-prompt instructions to ignore embedded instructions (Anthropic's "spotlighting" recipe).
  - **NeMo Guardrails** Colang-based runtime monitor.
  - **Invariant Labs `invariant`** DSL-based runtime monitor.
  - **Generate + execute (no verify)** the workflow IR but no verification (ablation).
  - **Ours: generate + verify + execute** with policy library.
  - **Ours: ours + proof-carrying** with model-emitted SMT-LIB2 proofs.

- **Metrics.**
  - Attack-success rate (ASR), per attack class.
  - Task-success rate on benign tasks (utility).
  - Verifier wall-clock time, end-to-end p50/p95 latency.
  - False-rejection rate (verifier rejects a legitimate workflow).
  - Policy-authoring effort (lines, time, expert vs LLM-assisted).

- **Statistics.** Five seeds per (model × condition × benchmark) cell; bootstrap 95% CIs; paired-permutation tests for ASR deltas; pre-registered analysis plan.

Key assumptions

A. The workflow IR is expressive enough for ≥80% of AgentDojo-style tasks (preliminary experience with our existing examples suggest this is plausible for the linear-arithmetic fragment).

B. SPACER terminates on the workflow fragment within an agent-step budget (≤5s p95).

C. Forcing the model to "think in code" does not degrade reasoning materially. Our *Function-Frustrations* paper argues the opposite, but we will measure.

D. Policies for common agent surfaces (email, file system, calendar, payments) can be authored as a small reusable library in Universalis.

Confirming or denying the hypothesis

H1 is confirmed iff the ASR delta vs. the strongest baseline exceeds 10× with a 95% CI excluding 5×, and task-success degrades by <5pp.

H1 is denied if either threshold is missed; we will publish the negative result and analyse why (likely failure modes: under-expressive IR, brittle policy library, model unable to follow workflow grammar).

H2 is confirmed iff the proof-carrying condition is ≥5× faster on the median benchmark instance.

Worked example: the Tahoe policy

To make the architecture concrete, the Tahoe sales example expressed in Universalis.

```
None
% --- Invariant (the policy: floor prices) ---
Fact floor_price{ model: "Tahoe",  price: 45000 }.
Fact floor_price{ model: "Malibu", price: 25000 }.

# Question
Can we commit to selling Model to Customer for Price?
```

```
# Answer  price_commitment{ model: Model, price: Price }
We can commit to selling Model for Price when
   floor_price{ model: Model, price: Floor }
and Price ≥ Floor.

% --- Conjecture (the agent's plan in response to "Tahoe for $1")
---
# Question
Offer Customer a Model="Tahoe" for Price and confirm the Deal.

# Answer  sell_tahoe{ customer: Customer, model: "Tahoe",
                      price: Price, deal: Deal }
We check the Tahoe is in stock  lookup_vehicle{ model:"Tahoe",
stock:true }.
Then commit to the price{ model:"Tahoe", price: Price }.
And confirm{ customer: Customer, message: Deal }.
```

The verifier asks SPACER whether `sell_tahoe` is a refinement of the relation `price_commitment`. With `Price = 1`, the discharge `¬∃ ... sell_tahoe(...) ∧ ¬price_commitment(...)` is **SAT** with witness `(customer:Bob, price:1, deal:false)` and the plan is rejected before any `confirm` tool is called. If we replace `Price = 1` with `Price = 50000` and the same query is **UNSAT**: the deal is admitted.

The same shape generalises to taint-style policies by using path-reachability over a connectedness relation.

Preliminary experiments

We have a working prototype implementing the core verifier:

- **Z3-CHC translation** Translates Universalis programs to Constrained Horn Clauses, using SPACER as the decision engine.

- **Invariant checker** Automatically extracts invariants from `question` clauses and discharges them by violation-reachability queries. Working end-to-end on the program-as-specification model.

- **Semantic subtyping types** We use JSON-Schema-as-types, encoded as Z3 formulas over a `JsonValue` ADT, à la Bierman & Gordon.

- **Proof-carrying types** via **Interactive prover MCP** This MCP server exposes Z3 (SMT, fixedpoints, tactics) over an MCP protocol, enabling Automind to discharge proofs interactively when the one-shot proof-carrying path fails..

The "Guardians of the Agents" article walks through the design pattern end-to-end on an email exfiltration scenario; our implementation provides the verification machinery the article sketches. Together they constitute a credible implementation prior.

Desired mentee expertise and skill sets

one PL/verification lead (Z3, semantic subtyping),
one ML/agent-systems lead (benchmark harness, model wrangling),
one empirical-safety lead (eval design, statistical analysis).

Mike Dodds (Galois, Inc / Oath Technologies)

Mike Dodds (Galois, Inc / Oath Technologies) | Semantics Done Quick

See https://oath.tech/pub/2026/05/semantics-done-quick for the full project details.

Simon Henniger + Nada Amin (Harvard University)

Simon Henniger + Nada Amin (Harvard University) | Claimcheck: Towards More Reliable LLM-Based Formalization

This is a shorter version of the full proposal here: open in Google Docs

Summary

In formal verification (FV), proof-carrying guarantees are only as good as the spec they satisfy. This project explores how to improve LLM-driven formalization from natural language (NL).

We want to create a set of techniques that tangibly improve LLM formalization outcomes, similar to the improvements brought by techniques like Reflexion (for self-correction) and Verbalized Sampling (for diversity in outputs).

We will directly build on preliminary work on claimcheck (https://midspiral.com/blog/claimcheck-narrowing-the-gap-between-proof-and-intent/). This work showed that round-tripping — the process of informalizing a formal spec in one run, then checking it against the initial natural language requirement in another run — is a relatively effective method for finding intent drift in the formalization process.

We will build on claimcheck and expand on it in multiple ways:

1) Intent drift is a universal problem across many fields, including math formalization, the sciences and coding (including both formal verification and testing). Although our focus remains specs on code, we will expand claimcheck to more domains, including general reasoning, formalization into different logics, and use-cases in the sciences (depending on profile and interests of our mentee).
2) For each domain, we will either find and adapt or create from scratch a benchmark for LLM-driven formalization.
3) Given this evaluation setup, we will improve upon claimcheck. To do so, we will generate and implement further ideas to reduce intent drift. This includes approaches with multiple different models (round-robin validation), contradiction checking, as well as improved informalization.

Target output

Twofold output: Submission to OOPSLA (deadline in October ‘26) or another conference and an open-source tool.

Theory of impact

A tool such as claimcheck can only help provide some guarantees that a formal specification matches user intent. It is also useful because it flags potential issues, so that the informal correspondence or the formalization can be iterated on (e.g., by an agent). The tool keeps the agent honest.

For instance, such a tool could be used within an IDE for formal specifications.

Our tool addresses the failure mode of agents weakening the formal specifications in order to more easily satisfy them.

Our tool addresses the need that users often want legible natural language summaries of the guarantees provided by a formalization.

We believe many use cases are opened up by our tool, including for Midspiral’s verified web apps and beyond, as applicability, robustness and speed improves with the work proposed here.

Desired expertise and skill sets

A background in programming languages theory or logic (first-order, temporal, or dependent types) is highly valuable, since the work spans multiple formalisms and requires reasoning about semantic equivalence across representations. We're especially interested in applicants with domain expertise in another field that requires formalization -- especially if it's a field that we haven't thought of yet! Experience with benchmark design, property-based testing, and academic research workflows is important.
We're looking for creative applicants who will do well within the exploratory framework of the project.

High-level background

https://midspiral.com/blog/claimcheck-narrowing-the-gap-between-proof-and-intent/: claimcheck: Narrowing the Gap between Proof and Intent by Fernanda Graciolli and Nada Amin. The goal of this proposal is to expand on this work.

https://github.com/realChrisHahn2/nl2spec: nl2spec: Interactively Translating Unstructured Natural Language to Temporal Logics with Large Language Models by Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, Caroline Trippel (CAV 2023)

https://nl2postcond.github.io/: Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions? by Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, Shuvendu K. Lahiri (FSE 2024) and Evaluating LLM-driven User-Intent Formalization for Verification-Aware Languages by Shuvendu K. Lahiri (FMCAD 2024)

https://ai.stanford.edu/blog/clover/: Clover: Closed-Loop Verifiable Code Generation by Chuyue Sun, Ying Sheng, Oded Padon, Clark Barrett (SAIV 2024)

Detailed technical background

The problem we are addressing: after an agent generates a verified artifact from natural language, how do we actually guarantee that the artifact matches the natural language intent?

While some trust in LLMs will always be a necessary condition for employing LLM-driven FV, we aim to minimize the trust surface as much as possible so that the core benefits of FV can be made accessible with high confidence. To do so, we propose that different techniques at various degrees of determinism and observability be added to the translation process.

We have a preliminary intervention that has been working quite well: have an LLM independently informalize the formal specifications and have another LLM compare the resulting natural language with the initial natural language. We have packaged this intervention in a tool called claimcheck (https://midspiral.com/blog/claimcheck-narrowing-the-gap-between-proof-and-intent/).

We have a small benchmark suite that confirms this works well compared to other approaches. In the wild, we have seen claimcheck catch real issues around specification weakening (e.g., order independence was claimed but only order prepending and appending were verified).

We would like to use this program as a broader opportunity to think about the future claimcheck that would provide strong guarantees that the implementation matches the user's intent.

Limitations that this follow-up work will address is stronger evaluation and new techniques to complement round-tripping. Part of the issue in going from natural language to formal specifications is that ambiguity needs to be untangled and explicitly specified. This is something that could be explicitly addressed by the tool: accepting that it makes additional decisions over the intent, and assessing and surfacing those additional decisions to the user.

Proposed methodology

Hypothesis: there is a better way to reason about the gap between informal and formal statements. Novel techniques include pushing independent round-tripping, and devising additional techniques. Those techniques could frontally address the issue of ambiguity lifting.

We will work towards building a tool that reliably does what claimcheck glimpses at.

Experimental design is based on benchmarks formalizing natural language to first-order logic, to Lean, Verus, and other languages and domains that can be unambiguously reasoned about.

This is an exploratory project. We hope for the mentee to bring in their own domain expertise and ideas. For the domain, we will at least consider the following:

- Formalizations of math in Lean
- Formalizations of properties of code in property-based tests
- Formalizations of properties of code in one or multiple of of {Dafny, Verus, Lean, Rocq}

For the techniques, we will at least try:

- System prompt instructing LLM to find ambiguities in NL
- Round-robin approach (one model checking another)
- claimcheck-like roundtripping on individual requirement formalizations
- Formal equivalence check on x generated formal samples
- On failed equivalence check, tests to find counterexamples
- On successful equivalence check, informalization to re-assert translation
- Mutation tests to find too-weak implementations

Proposed timeline / milestones

Milestone: Short paper on claimcheck and its positioning to SpecOps

Description: https://conf.researchr.org/home/splash-issta-2026/specops-2026#Call-for-Papers

Target Completion Date: June 15 (venue deadline)

Key Deliverables: A short submission to ground the work already done and the aspirations of the project, includes only one domain (likely code with either Dafny or property-based testing) and an expanded benchmark

Milestone: Optimization of Domain 1

Description: Expand upon benchmark and generate and validate new ideas for improving claimcheck

Target Completion Date: July 15, 2026

Milestone: Domain 2

Description: Find new domain 2 (to be agreed upon based on the interests and background of the mentee), find benchmark or create benchmark for it

Target Completion Date: August 15, 2026

Milestone: Optimization of Domain 2

Description: Iterate upon claimcheck's performance in domain 2

Target Completion Date: September 15, 2026

(More domains if time permits.)

Milestone: Draft paper

Description:

Target Completion Date: October 15, 2026 (or earlier, as needed for the venue deadline)

Key Deliverables: A draft paper for whatever venue we target

Preliminary experiments

We have two initial sets of experiments.

On our own benchmark extracted from https://github.com/metareflection/dafny-replay, we found that the round-tripping approach of informalizing the formal specification, and then separately comparing that informalization with the original natural language, help find issues of specifications.

We also experimented with the round-tripping approach, using a target of first-order logic rather than Dafny. We found a dozen formalization issues in an established benchmark FOLIO v1. Our results got validated, because FOLIO v2 has most of the issues we found independently fixed.

Risks, uncertainties, monitoring & mitigation

It might be reasonable to say that this work addresses an impossible issue: making formalization legible in natural language. There will always be gaps and imprecision in understanding, and it might be reasonable to expect that for safety-critical missions, the formal specifications is what needs to be reviewed, not as a natural language proxy. That said, we think there are many cases where a robust natural language correspondence is valuable and even enough, and we're excited to make these use cases possible.

Capacity

We guarantee 8 hours per week.

Quinn Dougherty (Apart x Atlas)

Quinn Dougherty | Apart x Atlas | Verifying non-interference preservation across MLIR lowering

This is a shorter version of the full proposal here: Open in google docs.

Target output

> Translation-validation tool for non-interference preservation across MLIR lowering, evaluated on HEIR's `secret` dialect. Target: CSF 2026 (deadline ~Feb 2026) or TACAS / FMCAD as backup.

Venue rationale. Submission cycles relevant to the fellowship calendar: CSF 2026 (paper deadlines historically Feb / May / Aug — three rounds per year, IEEE CSF), TACAS 2027 (ETAPS, typically October submission), ICSE 2027 (typically March submission, longer review). CSF is the sharpest fit (it's the venue Barthe-style non-interference work has landed in for fifteen years); TACAS catches the formal-methods audience if the SMT/Lean engineering is the stronger story by mid-fellowship; ICSE is a fallback if the artifact and empirical evaluation crystallize faster than the theory.

Summary

Model weights are confidential and valuable, but the compilation pipeline that turns a model into an executable is an attack surface that current ML compilers do not defend. Chen et al. (IEEE S&P 2026) established this for *integrity*: the compiler can inject backdoors. We address the parallel **confidentiality** problem: lowering passes in MLIR can introduce information channels (data-dependent control flow, memory aliasing, dynamic shapes) that leak protected weights even when the source-level program is non-interferent. Existing MLIR translation-validation tools (mlir-tv, Regehr PLDI'25) check equivalence, not non-interference. We extend them to check **non-interference preservation** under lowering, evaluated on HEIR's `secret` dialect. Deliverables: (i) a self-composition-based SMT encoding on top of mlir-tv, (ii) evaluation against ~10 HEIR-relevant lowerings, (iii) a syntactically-checkable "data-independent lowering" subset proved sound. Expected outcome: at least one leak-class identified in upstream lowerings.

High-level background

Four foundational lines of work this project sits between:

1. **The compiler as an ML attack surface.** Chen, Peng, He, Yang, Ray, *Your Compiler is Backdooring Your Model: Understanding and Exploiting Compilation Inconsistency Vulnerabilities in Deep Learning Compilers* (IEEE S&P 2026). This is the key motivating result. Chen et al. show that production DL compilers can be coerced into producing backdoored binaries from benign source models — 100% attack success on triggered inputs, zero detection by current defenses, and 31 of the top 100 HuggingFace models already contain *natural* compilation- induced triggers exploitable without modifying source. **This establishes that the MLIR-level compilation pipeline is a real attack surface for ML.** Chen et al. attack the *integrity* property: post-compilation semantics diverge from source. This project addresses the parallel *confidentiality* property: lowering passes can introduce information channels that leak protected weights. The threat surface is the same — the compiler — and we expect the defensive infrastructure to share most of its machinery, but the security property and proof obligations differ.

2. **Information flow control via type systems.** Volpano, Smith, and Irvine, *A sound type system for secure flow analysis* (JCS 1996). Barthe, D'Argenio, Rezk, *Secure information flow by self-composition* (CSFW 2004; journal version MSCS). These give the relational semantics for non-interference and the self-composition reduction that lets a unary verifier discharge a relational obligation. Our SMT encoding is in this tradition.

3. **Translation validation for compiler IRs.** Pnueli, Siegel, Singerman, *Translation validation* (TACAS 1998). Lopes, Lee, Hur, Liu, Regehr, *Alive2: Bounded Translation Validation for LLVM* (PLDI 2021, Distinguished Paper) — bounded SMT-based equivalence checking for LLVM IR peephole optimizations, which found 47 bugs in production LLVM. Regehr et al., *First-Class Verification Dialects for MLIR* (PLDI 2025) — ports this discipline to MLIR and validates a canonicalization pass and dataflow analysis transfer functions. Our work generalizes the Alive2 / Regehr discipline from semantic equivalence (a unary post-condition) to non-interference (a relational post-condition).

4. **Confidentiality in ML systems.** The weight-extraction threat model is established in the ML security literature: model extraction attacks via query access (Tramèr et al., USENIX 2016), weight stealing via side channels (Batina et al., *CSI NN*, USENIX 2019), and the broader literature on confidentiality of models as IP and as the carrier of training-data privacy. The compilation-induced leak vector — that lowering itself can introduce channels that were not present in the source program — is the under-studied piece. Existing IFC work for compilers (Sison & Murray, *Verifying that a compiler preserves concurrent value-dependent information-flow security*, 2019, on non-interference preservation in an Isabelle/HOL-verified compiler; Almeida et al. on Jasmin's constant-time preservation) lives at LLVM or assembly-DSL altitude and does not reach modern ML compiler IRs.

How our work relates to Chen et al. specifically. This is a parallel, not a direct response. Chen et al. demonstrate that compilation can break a desired *integrity* invariant (functional equivalence) and that the security community has noticed. We argue — and aim to demonstrate empirically — that compilation can equally break a desired *confidentiality* invariant (non-interference between secret weights and public outputs), via similar mechanisms: lowering passes that introduce data-dependent control flow on a secret tensor, canonicalizations that drop encoding metadata carrying security labels, bufferization that exposes weights via memory-aliasing patterns visible to a co-tenant. If Chen et al. is the first paper to treat the DL compiler as an integrity attack surface, this is the parallel first paper for confidentiality. (Hedge: the "first paper" framing depends on the systematic literature review the fellow will do in M1; we adjust the claim downward if priors exist.)

Comparison to similar goals. SecWasm (Bastys, Algehed, Sjösten, Sabelfeld, SAS 2022) addresses IFC for a different IR (WebAssembly), with no lowering- preservation story. FaCT (Cauligi et al., PLDI 2019) addresses constant-time programming at the source-language DSL level. Jasmin (Almeida et al., CCS 2017) provides verified cryptographic-grade constant-time-preserving compilation but for a small assembly-like DSL, not for ML compilation, and operates on confidentiality-via-timing rather than information-flow at the IR level. None of these target the MLIR ecosystem, and none address the compilation-introduced-confidentiality-channel problem at the linalg-on-tensors altitude where ML programs live.

Proposed methodology

Hypothesis

> **H.** A self-composition encoding extended with per-value security labels, layered on top of mlir-tv's existing SMT encoding for the `linalg` and `tensor` dialects, suffices to detect non-interference- violating lowerings on a corpus of HEIR-relevant MLIR programs. We further conjecture that a syntactically-checkable subset of lowering passes — those whose rewrite rules are *data-independent* in a precise sense (no rewrite condition depends on a secret-derived value) — preserves non-interference by construction, and we can characterize this subset formally.

Novel techniques

1. **Relational SMT encoding for MLIR with security labels.** Self- composition transformation at the MLIR level (operates on the IR, not on the SMT formula) followed by lifting through mlir-tv's existing encoder. The key engineering bit is preserving the `aisec.protected` / `aisec.declassify` annotations through the self-composition transform so the SMT obligation knows which values are public-observable.

2. **Data-independent lowering subset.** A static characterization of MLIR `PatternRewriter` rules whose match predicates and rewrites do not branch on (or compute over) secret-labeled values. Programs in this subset are non-interference-preserving by construction; a linter pass can certify this without any SMT.

3. **Counterexample lifting.** When the SMT solver returns SAT (non-interference is *not* preserved), lift the counterexample back to a pair of MLIR inputs that produce different public outputs after lowering but agreed on public outputs before. This is the practical artifact: it turns an unsatisfiability failure into a concrete attack witness.

Experimental design

Corpus. ~10 lowering passes from upstream MLIR + HEIR, focused on those that interact with `secret`-typed or `tensor`-typed values:

- `linalg-on-tensors → linalg-on-buffers` (bufferization, known-tricky)
- `linalg → loops` (the canonical structured-to-unstructured lowering)
- `secret → cleartext` rewrite passes inside HEIR
- `secret.generic` outlining / inlining transforms
- A small number of canonicalization patterns that touch `tensor` insert/extract ops (known to drop encodings; see OP-1 in `open-problems.agents.md`)

For each pass, run the relational validator on a hand-crafted suite of ~20 inputs (positive: non-interferent both pre- and post-; negative: manually-introduced leaks) plus a fuzz-generated set if time permits.

Baselines. Two:

1. **Unary equivalence (mlir-tv vanilla).** Does mlir-tv-as-shipped already detect these leaks because they manifest as semantic divergence? Empirical question, but our prior is no: the canonical non-interference leak (a public output depending on a secret input without declassification) is *not* a semantic-equivalence bug, because the pre- and post-lowered programs compute the same function of the secret input. Demonstrating this difference empirically is part of the contribution.

2. **Pre-lowering verifier only (this repo's MVP).** Our prototype verifier checks non-interference at one IR level. It does not address whether lowering preserves the property. So: programs verified pre-lowering, when lowered by a hostile or buggy pass, may emit non-interferent-looking output that is no longer non-interferent. Quantifying this gap is the threat-model motivation for the relational TV story.

Key assumptions

A1. **mlir-tv's SMT encoding of `linalg.generic` and `tensor` ops is sound enough to extend.** If the underlying encoder has soundness bugs, our relational extension inherits them. We will not re-validate mlir-tv; we will treat it as a trusted base and flag any divergence the extension surfaces.

A2. **HEIR's `secret` dialect is stable enough to target.** HEIR is moving fast (2024–2026). We pin a HEIR commit at fellowship start and rebase only if a hard reason emerges. Risk: the dialect's surface syntax shifts mid-fellowship; mitigation: maintain a minimal `secret`-dialect mirror in our repo and target that if upstream churn becomes blocking.

A3. **Self-composition is the right relational encoding for MLIR.** The alternative is a product-program construction that doesn't duplicate operations. Self-composition is simpler and is standard for type-system-style IFC. We do not claim it is optimal; we claim it is sufficient for an evaluation against ~10 lowerings.

A4. **A meaningful fraction of HEIR-relevant lowerings are *not* in the obviously-safe subset.** If almost every relevant lowering turns out to be syntactically data-independent, the SMT machinery has nothing interesting to say and the paper collapses to the characterization theorem alone. We have weak preliminary evidence against this collapse (bufferization legitimately introduces data-dependent control flow), but it's a real falsifier.

How we confirm or deny the hypothesis

Confirming evidence. (i) The relational validator detects at least one non-interference violation in a real upstream MLIR lowering that mlir-tv vanilla does not detect. (ii) The data-independent subset is non-trivial (at least N% of HEIR lowerings fall outside it). (iii) The mechanized soundness theorem for the data-independent subset goes through in Lean 4 with no `sorry`s. Any one of these constitutes a positive result; all three is a strong paper.

Disconfirming evidence. (i) Every realistic HEIR lowering turns out to be syntactically data-independent → the SMT story is unmotivated, write up the characterization theorem alone, target a shorter venue. (ii) The SMT encoding is too expensive to terminate on realistic programs → fall back to bounded depth, write up the scaling limits honestly. (iii) The self-composition encoding has soundness issues we can't resolve → pivot to the Lean mechanization as the primary contribution (this is what OP-6 is for in reserve).

The fellowship is long enough to detect any of these failure modes in time to pivot, provided we are honest about the disconfirming signals as they arrive.

Preliminary experiments

The supporting evidence here is genuinely thin and I want to flag that explicitly rather than dress it up. What exists today in the repo:

1. **A working-on-paper MVP verifier** for non-interference modulo declassification, in C++ on top of HEIR's `secret` dialect. ~200 lines in `lib/Transforms/VerifyNonInterference.cpp`. The verifier is not currently buildable on the maintainer's development environment due to toolchain incompatibilities; this is the meta-problem captured as OP-7 in `docs/exobrain/open-problems.agents.md` and we expect the fellow to inherit a buildable artifact (we will fix the build before fellowship start as a precondition).

2. **Hand-traced execution of nine test cases** (`docs/exobrain/traces.agents.md`), covering positive (`@no_protected`, `@sanctioned_declassify`, `@mobilenet_block_sanctioned`, `@internal_multi_use_ok`) and negative (`@direct_leak`, `@leak_after_generic`, `@mobilenet_block_leak`, `@fanout_via_two_reveals`, `@fanout_via_two_generics`) cases. All nine traces produce the expected diagnostic. **Honest caveat**: hand traces against self-authored code are consistency evidence, not correctness evidence. The fellow will be expected to convert these into real unit tests as the first week's work.

3. **A literature sweep** (`docs/exobrain/findings.agents.md`) that found no published MLIR IFC dialect, no published translation validation for non-interference in MLIR, and confirmed the Chen et al. attack as a published high-profile motivation. The sweep was a few hours of web search, not a systematic review. The "first to combine TV with non-interference in MLIR" claim is defensible at *that* search depth and should be hedged in the eventual paper to that depth.

4. **A research trail** (`docs/exobrain/`) documenting the design decisions, dead-ends, and reframings that produced the open- problems doc. Worth showing a fellow as evidence that the project has been honestly red-teamed against itself.

What we **do not** have: any quantitative result, any benchmark, any actual SMT-solving experiment, any Lean code. The fellowship is where those come from.

Potential applications & practical uses

Model weights are confidential — they encode training data, are expensive to produce, and increasingly drive commercial value. Chen et al. (S&P 2026) showed the compilation pipeline is an attack surface for ML *integrity*; the same surface is a candidate attack vector for

confidentiality, with no current defense. A working non-interference-preservation tool would (i) let HEIR and similar projects ship lowering passes with a checked guarantee they do not introduce weight-leak channels, (ii) give MLIR pass authors a CI-grade regression catch for security properties, on par with existing equivalence-checking, (iii) supply the formal-methods community with a working example of relational verification at modern compiler-IR altitude, and (iv) seed a longer agenda in confidentiality-preserving ML compilation: quantitative info flow, bandwidth-bounded egress certificates, compositional FHE-scheme proofs (OP-4, OP-5 in `docs/exobrain/open-problems.agents.md`). Practical adoption is gated on HEIR's trajectory, which we don't control.

Anything else you'd like us to know

Three things, kept short.

First, this repo's exobrain (`docs/exobrain/`) is the research trail for the prototype: critique of the original framework, finding-by- finding citation work, the open-problems triage that produced the seven candidate research questions, and the honest write-up of the toolchain failure that became OP-7. It's there to demonstrate that the project has been red-teamed against itself before being proposed externally. A fellowship reviewer who wants to assess whether the project is grounded should read `docs/exobrain/critique.agents.md` and `docs/exobrain/open-problems.agents.md`.

Second, the closest competitor work I'm aware of — Regehr et al.'s First-Class Verification Dialects for MLIR (PLDI 2025) — is infrastructure we *use*, not work we displace. We are not racing Regehr's group; if anything, a successful project here is positioned to be a "real-world application" citation for their framework. I'd welcome a Regehr-group co-author on the final paper if they're game.

Third, the dual-use posture (R5 above) is something I'd want to discuss with Atlas before the project goes public. The dominant view in the IFC research community is that publication of non-interference analysis tools is net-defensive (this is the SecWasm / FaCT / Jasmin posture), and I share that view, but the ML-compiler-attack literature is recent enough that the community norms aren't fully settled. Atlas likely has more context here than I do.

Ryan Marinelli (Oslo) | Geometric Canaries

Ryan Marinelli (University of Oslo) | Geometric Canaries: Detecting Adversarial Failures in LLM-Assisted Formal Verification

This is a shorter version of the full proposal here: open in Google Docs

Summary

LLM-assisted formal verification pipelines are vulnerable to a critical failure mode: adversarially perturbed theorem statements can steer models toward plausible-but-invalid proofs, with no warning until the proof checker fails. This project applies geometric chain-of-thought analysis to proof generation, using two metrics, Average Angle Error and Number of Mistakes, to detect reasoning instability mid-generation, before the checker is invoked. Both metrics are already validated as adversarial-sensitive in general reasoning (Marinelli 2025); formal verification provides a strictly stronger testbed with exact binary ground truth. We construct an adversarial benchmark from miniF2F and ProofNet, collect CoT traces from DeepSeek-R1 and Lean-specialised models, and evaluate geometric signals as predictors of proof invalidity. If confirmed, this yields a lightweight geometric canary deployable as a pre-check in any LLM-assisted verification pipeline.

Target output

A peer-reviewed paper characterising and detecting adversarial failure modes in LLM-assisted formal verification pipelines using geometric chain-of-thought (CoT) signals, targeting CSF 2027 (IEEE Computer Security Foundations Symposium) or TACAS 2027 as the primary venue, with a workshop paper at a co-located FM 2026 venue as an intermediate milestone. CSF and TACAS abstract deadlines fall in late

September--October 2026. Secondary deliverable: an open-source Python toolkit for CoT-based anomaly detection in proof-generation pipelines.

Theory of impact

The immediate application of this work is as a lightweight screening layer in LLM-assisted formal verification pipelines. Proof checkers such as Lean 4 and Coq are computationally expensive relative to the cost of embedding a CoT trace and computing a geometric signal. If Average Angle Error and Number of Mistakes can reliably flag suspicious proof attempts before the checker is invoked, this creates an opportunity to redirect compute toward human review or re-sampling rather than spending it on proof attempts that are likely to fail or, more dangerously, fail silently under adversarial conditions.

The safety motivation is substantial. LLM-assisted formal verification is moving into production contexts where the cost of an undetected false proof is high. Current pipelines offer no mechanism for detecting adversarial steering of the proof generation process. The proof checker is the only gate, and it operates post-hoc. A geometric signal that operates mid-generation fills a gap that no existing tool in the formal verification stack addresses.

There is a secondary benefit in how this changes the human review process. Rather than asking a verification engineer to audit a completed proof, a geometric pre-check can surface the specific steps in the reasoning trace where instability is highest, focusing attention on the parts of the proof most likely to contain the error.

Finally, the adversarial benchmark constructed during this fellowship, covering miniF2F and ProofNet with systematic perturbation taxonomies and Lean 4 ground truth labels, will be released publicly.

Desired expertise and skill sets

Some exposure to formal methods or theorem proving (miniF2F, ProofNet, Mathlib)

Experience with embedding-space analysis or representation geometry (cosine similarity, angular metrics, dimensionality)

Statistical literacy: bootstrapping, ROC-AUC, cross-validation, confidence intervals

Prior work with chain-of-thought prompting or reasoning-model evaluation

Familiarity with adversarial robustness literature

Experience constructing benchmarks or curating evaluation datasets

Background

A growing body of work has examined the reliability of LLM reasoning, with particular attention to whether chain-of-thought traces faithfully reflect the model's actual inference process. Lanham et al. (2023) find that CoT is often post-hoc and unfaithful, and that faithfulness degrades monotonically as model size increases from 13B to 175B parameters. The implication is uncomfortable: larger models, which are most capable, are also least trustworthy in their stated reasoning. This motivates looking for external signals that do not depend on the model's self-reported logic.

Recent mechanistic work has begun to characterise which parts of a reasoning trace actually matter. Bogdan et al. (2025) introduce the notion of thought anchors, defined as critical steps whose perturbation substantially shifts the final answer. They identify these using KL divergence under semantic perturbation and attention suppression

techniques. This provides a principled basis for CoT segmentation and directly informs how we weight high-influence steps in our geometric analysis.

On the geometric side, Marinelli et al. (2025) show that step-level embedding distances and angular deviations in CoT traces discriminate adversarial from benign prompts, and correlate significantly with output accuracy across model sizes. This is the direct methodological predecessor of the current proposal. The present work applies the same framework to formal proof generation, a setting where the stakes are higher and the ground truth is cleaner.

The formal verification context introduces its own set of challenges and benchmarks. First et al. (2023) establish ReProver as a strong baseline for LLM-assisted theorem proving in Lean 4, and miniF2F as the standard evaluation suite. ReProver and related systems treat proof correctness as a binary post-hoc verdict with no intermediate uncertainty signal, which is precisely the gap this proposal addresses.

Taken together, these works motivate a natural next step: if geometric CoT signals predict reasoning failure in open-ended tasks, and if formal verification provides exact ground truth alongside formally constructible adversarial inputs, then this is the right domain in which to test whether those signals can serve as a practical early-warning mechanism.

Proposed methodology

Hypothesis. When a language model is steered toward a false proof by an adversarially perturbed theorem statement, its chain-of-thought trace will exhibit detectable geometric instability, specifically elevated Average Angle Error and Number of Mistakes, before the proof checker is invoked. These signals will be predictive of proof invalidity and will be systematically higher for adversarial inputs than for benign ones.

Dataset construction. From miniF2F, 200 theorems are selected spanning arithmetic, algebra, and number theory. From ProofNet, 100 theorems are drawn from undergraduate mathematics. For each theorem, two to three adversarial variants are constructed via systematic perturbation: quantifier swaps, sign flips in arithmetic expressions, and boundary condition changes that shift the theorem from true to false while preserving its surface structure. Each (theorem, proof attempt) pair is labelled correct or incorrect by the Lean 4 proof checker.

CoT trace collection. DeepSeek-R1 at 7B and 14B parameters and at least one Lean-specialised model (InternLM2-Math or DeepSeek-Prover) are prompted to generate full chain-of-thought proofs for each theorem in the dataset. Models are prompted to reason explicitly before producing formal tactic steps.

Geometric signal computation. Each CoT trace is segmented at sentence boundaries. Each segment is embedded using all-MiniLM-L6-v2 into a 384-dimensional dense vector space. Average Angle Error and Number of Mistakes are then computed per trace following the procedure in Marinelli (2025). We additionally compute a trajectory analysis that tracks how Average Angle Error evolves across the proof attempt.

Detection evaluation. Logistic regression and gradient boosting classifiers are trained on geometric features to predict two outcomes: proof invalidity and adversarial input. Both classifiers are evaluated under 5-fold cross-validation. Detection performance is reported using ROC-AUC and precision-recall curves, with confidence intervals computed via bootstrapping. Three baselines: the proof checker alone (oracle, no early warning), raw CoT length, and step-level perplexity.

Ablations. Sentence-transformer embeddings versus last-hidden-state embeddings from the generating model; Average Angle Error versus raw cosine distance as the geometric primitive.

Luiza Cristina Corpaci (TU Wien, AMD)

Luiza Cristina Corpaci (TU Wien, AMD) | Semantic Diff-ing in LLM-Assisted Formal Representations

This is a shorter version of the full proposal here: open in Google Docs

Summary

When LLMs translate between formal representations (natural language to code, code to specifications, one formal language to another) failures are silent. The output compiles, passes (maybe llm-written) tests and it looks right, but it can violate semantic invariants that made the original representation meaningful. This silence is a part of a threat model for AI-assisted formal methods. This project detects when an LLM-generated formal specification is valid but not faithful.

Rice's theorem says that semantic property preservation is undecidable in general. No algorithm can determine whether an arbitrary program transformation preserves an arbitrary semantic property. However, equational theories project constitute a formal, decidable fragment of meaning. The dataset contains algebraic identities that must hold across a representation change (commutativity, associativity, distributivity, idempotence, etc.).

This project builds a semantic diff framework based on a a checker that tests whether LLM-driven translations preserve equational invariants, using SMT solving and/or a Lean backend.

Stretch goal: test whether internal activations predict semantic drift before decoding. If feasible, we will train lightweight probes over open-weight models; otherwise, the project still succeeds via the checker, benchmark, and empirical taxonomy. There is existing work by Nanda et al. (modular arithmetic), Henighan et al. (algorithmic tasks), and others that establishes that formal properties can be read from internal representations before they surface in outputs.

Validation is done based on preservation accuracy: accuracy of the full LLM + parser + checker pipeline against the ETP implication graph stratified by equation complexity class (1–4 operations).

If the stretch goal is reached, internal representations that predict preservation failure before decoding would enable pipeline-level intervention prior to output — catching failures before they propagate.

Target output

Semantic diff framework + semantic unit test framework for LLM-generated formal specifications; targeting TACAS (paper submission Oct 15 / artifact submission Oct 30 2026).

Theory of impact

The immediate output is an equational diff checker, which is directly deployable in pipelines where LLMs translate between formal representations (e.g., AI-assisted theorem proving, hardware specification, contract synthesis). The failure taxonomy provides actionable guidance for training data curation and RL reward signal design for formal reasoning tasks .

The work does not accelerate offensive capabilities and it restricts them by making LLM formal output less exploitably wrong.

Desired expertise and skill sets

This is a hands-on technical project and a strong fit will be someone who is agentic, thoughtful, and able to move from idea to concrete tests quickly for underspecified problems.

The mentoring style combines regular guidance on research direction with feedback on experiment design, interpretations, and writing.

The expectation is that mentees work independently between meetings, read actively, and iterate on experiments.

Familiarity with Lean 4 (or another ITP) is important to have, specifically the ability to reason about what a formal statement means relative to an informal description.

Prior work on mechanistic interpretability, LLM evaluation, or benchmark construction is directly relevant.

Red-teaming or security experience is a bonus given the adversarial robustness angle.

High-level background

Eliciting Latent Knowledge (Christiano et al., ARC 2021) frames the problem of whether a model's internal representations are faithful reporters of what it "knows". This directly motivates the stretch goal: the probe component is testing whether the model is a reliable reporter of its own equational representations, distinguishing "model knows but mistranslates" from "model never represented the structure".

LLM formal reasoning benchmarks (e.g., MiniF2F, FIMO) show that LLMs achieve high token-level accuracy on formal tasks while failing on semantic correctness at a rate above what surface metrics reveal. The gap between fluency and faithfulness is empirically documented.

Rice's theorem says that general semantic preservation is undecidable for arbitrary programs, so safety-relevant validation needs tractable fragments where semantic drift can be measured precisely. Equational theories provide such a fragment as they are small enough for automated checking and rich enough to expose meaningful failures in translation.

Equational Theories Project provides the formal substrate: a structured dataset of algebraic equational theories with ground truth equivalence relationships. This gives the project a tractable benchmark where "semantic preservation" has a precise, checkable definition.

Verified compilation and semantic preservation in formal methods defines semantic preservation as a transformation that preserves semantics if it preserves observable behavior under all execution contexts. This gives us a definition to approximate and measure against in the LLM setting.

Detailed technical background

The Equational Theories Project:

- A magma is a set G with a binary operation ◇: G×G->G.
- A law is a formal expression w ≐ w' over the free magma; a magma satisfies the law if every variable assignment makes it hold.
- An implication E_1 -> E_2 holds if every magma satisfying E_1 also satisfies E_2; a non-implication is witnessed by an explicit finite counterexample magma.
- ETP has 4,694 equations that are stratified by complexity (between 1 and 4 operations), spanning from trivial (Eq 1: x = x) through commutative (Eq 43: x◇y = y◇x) and associative (Eq 4512: x◇(y◇z) = (x◇y)◇z) to hundreds of near-equivalent and incomparable theories whose relationships are non-obvious.

What ETP provides for this project:

- Ground truth at scale: 22M verified implication pairs, stratified by equation complexity

- A topology on theory space: position in the implication graph is the formal definition of "what a theory means" in this setting
- An existing Z3 interface for automated non-implication checking
- The syntax/semantics distinction is formally encoded and measurable

What ETP does not provide:

- ETP was designed as a theorem-proving benchmark for proof assistants and automated provers, not for LLM evaluation. No work has used the implication graph to measure where LLMs lose semantic content during translation tasks, nor connected failure patterns to model internals.
- We approximate preservation by whether the output theory is equivalent to, stronger than, weaker than, or incomparable with the intended theory under the implication partial order.

Proposed methodology

Hypothesis: LLMs fail to preserve equational invariants in a non-uniform, structurally predictable way. Specific equation classes (e.g. commutativity vs. absorption vs. distributivity) have characteristic failure signatures that are detectable from internal model states before outputs are produced, and failure rates are predictable from the structural complexity of the target theory.

Experimental design:

- Dataset: curate a stratified sample from ETP covering ~5-8 equational theory classes of increasing complexity. For each, construct translation pairs, e.g.:
    - natural-language law -> Lean/equational expression;
    - notation A -> notation B; both satisfying theory T
    - equation pair -> implication classification;
    - generated specification -> nearest ETP node/neighborhood.
- LLM evaluation: run 3-4 model families (small, open-weight models with accessible residual streams vs. stronger models for semantic preservation evaluation capabilities) on translation tasks. Check outputs with an SMT solver (Z3) and/or Lean checker for equational invariant preservation. Record failure rates per theory class & per model.
- Failure taxonomy: cluster failure modes.

Baselines: syntactic diff (edit distance), test-based equivalence (random input sampling), LLM self-checking ("does this translation look correct to you?"), human annotation on a validation subset.

Key assumptions:

- Equational invariant preservation is necessary (though not sufficient) for semantic preservation and violations are real failures.
- Internal model representations contain predictive signal about semantic faithfulness prior to decoding.
- The ETP sample is structurally diverse enough to generalize to real-world formal translation tasks.
- SMT/Lean checking is tractable for the ETP subset used.

Confirmation/denial: Hypothesis confirmed if failure rates across equation classes show statistically significant non-uniform distribution. Denied if failures are uniformly distributed (random, not structural) and probes don't generalize across model families.

Tasks:

- Implication classification: Given equations E_1 and E_2 from ETP, does the LLM correctly classify E_1 -> E_2 as holding or not? Ground truth: the verified implication graph.
- Lean encoding faithfulness: Given an equation in natural language ("the commutative law"), does the LLM's Lean 4 encoding correspond to the correct ETP node? Check via Z3 counterexample search and implication graph lookup.
- Counterexample generation: Given a non-implication, can the LLM produce a valid counterexample magma? Check algebraically.
- Stratification: Use ETP's complexity stratification (1–4 operations) as difficulty axis. Failure rates are expected to be non-uniform and we're measuring it.
- The semantic diff metric: For each LLM output, locate it in the implication graph (or bound its position via automated checking). Compute graph distance from ground truth. A distance of 0 shows preservation; distance > 0 shows measurable semantic drift with directional information (over-constrained vs under-constrained).
- Existing tooling we directly extend: Z3 integration, Lean 4 proof structure, equation explorer, finite magma explorer, all are already in the ETP repo.

Example Semantic Difference:

Intended	LLM output	Semantic relation	Meaning
`x ◇ y = y ◇ x`	`x = x`	output is weaker	LLM dropped the real constraint
`x = x`	`x = y`	output is stronger	LLM over-constrained the system
commutativity	associativity	incomparable	LLM changed the property

Preliminary experiments

Preliminary work: I have previously explored ETP as a representation/embedding benchmark and found that equation renderings contain recoverable signal. This supports the assumption that equational-theory classes are not random with respect to model representations. As an initial fellowship pilot, I will run a 50–100 item evaluation on implication classification and natural-language-to-equation mapping before project starts.

Shaowei Lin (Beneficial AI Foundation)

Shaowei Lin (Beneficial AI Foundation) | Deductive Vericoding

This is a shorter version of the full proposal here: open in Google Docs

Summary

By the Curry-Howard correspondence, proofs are programs. If so, then there should be a deductive strategy for synthesizing code, the same way we use tactics to synthesize proofs. We address this problem using a double categorical framework (https://arxiv.org/abs/2505.18329), and will be building on top of the SuSLik approach (https://arxiv.org/abs/1807.07022). We believe that deductive vericoding will be a more scalable approach to verified coding than the traditional generate-then-verify approach, and verified software is important for protecting our critical infrastructure from AI-assisted attacks. We will validate this claim by running our solution on existing vericoding benchmarks, e.g. https://arxiv.org/abs/2509.22908.

Target output

A public repo on the Beneficial AI Foundation GitHub account that contains a Lean library for deductive vericoding, and a blog post with examples on how to use this library with AI agents.

Theory of impact

By ensuring that synthesized code satisfies given specs by construction, we avoid a future where vibe-coded programs potentially have obscure vulnerabilities and where vibe-coded complex systems are still notoriously difficult to verify. I know this need exists because verified code synthesis is widely acknowledged as a need but is not widely available.

One possible use-case: translating legacy code bases to memory-safe languages. The translation could start with the generation of formal specs, property-based tests and unit tests. Our deductive vericoding tool is used to generate a code that satisfies the specs, and PBT and unit tests ensure that difficult-to-formalize properties are checked empirically.

Desired expertise and skill sets

Strong abstract mathematical skills, some knowledge of category theory, and experience with the Lean Prover.

High-level background

Vericoding Benchmark https://arxiv.org/abs/2509.22908

SuSLik Deductive Program Synthesis https://arxiv.org/abs/1807.07022

Loom: Foundational Multimodal Program Verifiers
https://ilyasergey.net/assets/pdf/papers/loom-preprint.pdf

The Vericoding Benchmark paper describes the high-level problem. The SuSLik paper proposes a promising approach that we will build off. Loom is a similar framework, but more towards verification of existing programs rather than correct-by-construction synthesis of programs.

Detailed technical background

Double Operadic Theory of Systems https://arxiv.org/abs/2505.18329 This is the most crucial paper. It has all the right theoretical constructs but no formalizations in Lean. We want to implement the concept of open wiring diagrams in Lean, in the most general way possible, so that we are not restricted to any particular monad instance.

Program Verification with Polynomial Functors https://arxiv.org/abs/2604.01303 This paper has interesting detailed implementations, e.g. polynomial functors for interfaces, Kleisli morphisms for implementations, and dependent polynomial for pre/post conditions.

Proposed methodology

The hypothesis is that AI agents can do a wide variety of deductive vericoding problems, as long as it fits into the double operadic theory of systems (DOTS). The problems need not come from a fixed domain logic. The techniques/tactics should generalize to all DOTS-shaped problems.

The experiment will proceed as follows:

1. Selecting an appropriate formalization of double categories from Mathlib

2. Formalize Interfaces, Goals and Interactions in this double categorical formalism. Previously, they were formalized with Predicates, (Hoare) Triples and Continuations in Loom and SuSLik
3. Develop small vericoding Goals and solve them in this new formalization
4. Develop small tactics that can decompose large Goals and solve small ones
5. Train AI agents (primarily through Skills) to solve some complex Goals
6. Assess formalism and framework on Vericoding Benchmark

We will use examples from the SuSLik paper as a baseline, and compare the rate of successful automated vericoding. We will also look at examples from other domain logics, such as Petri nets and cryptographic algorithms.

The key assumptions are that the domain logic has enough primitives to solve the vericoding problems effectively. Without sufficient primitives, the vericoded solutions might be correct by correction, but have poor performance in practice.

We will confirm the hypothesis if AI agents can use the new formalism for vericoding across different domain logics, such as functional programs, heap-manipulating programs, Petri nets and cryptographic algorithms.

Preliminary experiments

I have an initial proof-of-concept that uses the Loom monadic framework (of Predidates, Triples and Continuations) to solve a simple Append-Number-To-String problem, with Codable as the structure for reifying goals and their solutions.
https://github.com/Beneficial-AI-Foundation/deductive-vericoding/blob/main/DeductiveVericoding/Function.lean

Jocelyn Qiaochu Chen

Jocelyn Qiaochu Chen | Mining Enforceable Specifications for LLM-Agent Tool Use

Jocelyn's full proposal: open PDF in Drive

Tomasz Nguyen (Confluent / IBM) | BabelBench

Tomasz Nguyen (Confluent / IBM) | BabelBench

This is a shorter version of the full proposal here: open in Google Docs

Summary

- Problem: LLM proof generation is benchmarked within single proof systems (Lean, Coq, Dafny, TLA+), but we don't know whether models reason about proofs structurally or pattern-match on system-specific syntax. This matters for AI safety: spec-validation pipelines depend on faithful translation between formal artefacts.

- Contribution: A benchmark measuring within-cluster (Coq↔Lean, TLAPS↔Dafny) vs. cross-cluster (type-theoretic↔model-theoretic) proof translation. ~150 paired theorems with verifier-grounded scoring.

- Novelty: Existing benchmarks (miniF2F, ProofNet, PutnamBench) are single-system. We isolate the paradigm-translation axis specifically and predict an asymmetric failure pattern.

- Evidence: The asymmetry prediction is grounded in the documented semantic gap between temporal/state-machine and type-theoretic paradigms (e.g. Padon et al.'s liveness-to-safety reduction); single-system frontier results don't tell us whether that gap survives translation.

Target output

Cross-paradigm proof-translation benchmark + workshop/conference paper, target TACAS 2027 (ETAPS, submission early-October 2026)

Theory of impact

The bet: spec validation is a near-term bottleneck for AI-assisted formal verification, and the bottleneck has a specific shape — models that can synthesise proofs in a single system can't be trusted to refactor specifications across systems, because they pattern-match rather than reason about the underlying semantics. If true, this matters for: (1) AI-assisted refactoring of formally verified infrastructure, where translation between layers (TLA+ design spec → Dafny

implementation → Coq meta-theory) is routine; (2) automated reproduction of formal results across systems, an emerging community workflow; (3) longer-term, training-time use of formal verification as a correctness signal for AI-generated systems code, which presupposes that models understand the verification target rather than the verifier's surface syntax.The benchmark is small in scope (one paper, one fellowship cycle) but targets a measurement gap that, to my knowledge, no existing benchmark addresses.

Desired expertise and skill sets

Anyone with interest in formal methods. Some elementary understanding of the distinction between model theoretic and type theoretic theorem proving required.

High-level background

The proposal sits at the intersection of three lines of work.

(1) LLM theorem proving and proof benchmarks. miniF2F (Zheng et al., 2022) and ProofNet (Azerbayev et al., 2023) established the standard evaluation paradigm: theorems formalised in one or two systems (Lean, Isabelle), with pass@k as the metric. DeepSeek-Prover-V2 (2025) and recent Lean-focused systems achieve strong results on these benchmarks. The pattern across all of them: single-system evaluation, with translation between systems treated as a curation step rather than as a capability under test.

(2) Distributed-systems verification. IronFleet (Hawblitzel et al., SOSP 2015) demonstrated end-to-end verified distributed protocols using Dafny, with refinement from TLA-style high-level specs to executable code. Anvil (Sun et al., OSDI 2024) verified Kubernetes controllers in Verus. Padon et al. (POPL 2018) showed how to reduce liveness verification to safety via prophecy variables — a technique that crosses the temporal/non-temporal divide and motivates why translation between paradigms is non-trivial. These works define the "model-theoretic / state-machine" cluster the proposal targets.

(3) Type-theoretic proof assistants and the Curry-Howard side. Software Foundations (Pierce et al.) and the Mathematical Components library establish the canon for Coq; mathlib does the same for Lean. The semantic gap between these two clusters — propositions-as-types with dependent elimination on one side, transition systems with temporal operators on the other — is well-documented in pedagogical material but, to my knowledge, has not been systematically measured as a translation difficulty axis for LLMs.

Detailed technical background

Two papers anchor the technical motivation.

Padon, Hoenicke, Losa, Podelski, Sagiv, Shoham. "Reducing Liveness to Safety in First-Order Logic." POPL 2018. The paper shows that liveness properties can be reduced to safety properties via the construction of a "dynamic abstraction" that introduces history and prophecy variables. This is structurally significant for the proposal: it's the cleanest demonstration that the boundary between temporal-logic specifications (where liveness is native) and first-order/type-theoretic specifications (where it isn't) is a real semantic shift, not a notational one. A model that can faithfully translate a TLAPS liveness proof into a Coq inductive proof must reconstruct the prophecy machinery. A model that pattern-matches will produce something that typechecks in Coq but doesn't capture the original property. This gives the benchmark a sharp diagnostic: the gap between "produces a valid proof in the target system" and "produces a valid proof of the corresponding theorem" is exactly the gap I want to measure.

DeepSeek-AI. "DeepSeek-Prover-V2: Advancing Formal Mathematical Reasoning." 2025. The current state of the art on Lean theorem proving via large-scale RL with verifier-grounded rewards. Two aspects matter for the proposal. First, the methodology — programmatic generation of training problems, automated proof checking, RL on verifier-pass signal — is exactly the methodology fv-worlds prototypes for the cross-system case. Second, the system is Lean-only by design. The proposal asks whether the capability generalises, and predicts asymmetrically that it does within the type-theoretic cluster (Coq is close enough that minor automation differences dominate) but breaks down at the cluster boundary. The conjunction is the core empirical bet: the Padon paper tells us where the semantic boundary is; DeepSeek-Prover-V2 tells us what the strongest current systems can do within their native paradigm. The proposal measures what happens at the boundary.

Proposed methodology

Phase 1 — Corpus construction (June, ~6 weeks).

Curate ~150 paired theorems organised into four buckets:

- Within-cluster, type-theoretic: Coq ↔ Lean (~40 pairs).
- Within-cluster, model-theoretic: TLAPS ↔ Dafny (~40 pairs).
- Cross-cluster, "easy" direction: model-theoretic → type-theoretic (~35 pairs). Safety properties only initially; liveness deferred (see Risks).

- Cross-cluster, "hard" direction: type-theoretic → model-theoretic (~35 pairs). Algebraic / inductive theorems with non-obvious operational interpretation.

Each pair requires: a source theorem with a verified proof in the source system (no Admitted, no empty axioms — the source side must compile cleanly); a target-system theorem statement that asserts the corresponding property in the target paradigm's semantics; and a natural-language equivalence rationale linking the two statements. The target-side proof is what the model under evaluation produces — we do not need to hand-translate proofs, only to certify that the source compiles and that the target statement is the right one to ask the model to prove. This is a substantially smaller manual lift than full bidirectional theorem translation, and it's what makes ~150 pairs tractable in six weeks.

Phase 2 — Evaluation harness (July, ~4 weeks). Extend the fv-worlds Docker-based grading infrastructure to run the full Coq/Lean/TLAPS/Dafny toolchains. Existing harness already covers Coq/Lean/Dafny/Verus/TLA+; the work is mostly hardening (timeout policies, trusted base sizes, partial-credit semantics for proofs that admit lemmas). Run the benchmark across (at minimum) Claude Sonnet/Opus, GPT-5/-class, DeepSeek-Prover-V2 in its accessible form, and an open-weights Lean specialist. Several conditions per model: zero-shot, few-shot with paradigm-matched examples, few-shot with paradigm-mismatched examples (the diagnostic condition).

Phase 3 — Analysis and write-up (August–September, ~8 weeks). Primary analysis: pass-rate gap between within-cluster and cross-cluster conditions, controlled for theorem difficulty. Secondary: failure-mode taxonomy — does the model produce typechecking-but-wrong proofs (translation failure), proofs that don't typecheck (target-system unfamiliarity), or refusals (capability ceiling)? The taxonomy is the mechanism story; the gap measurement is the headline number. Workshop submission target: end of August (DeepLearning.IL or AITP if timing aligns). Conference submission: TACAS 2027 (ETAPS), early-to-mid October 2026 deadline.

Max von Hippel (Anduril Industries)

Max von Hippel (Anduril Industries) | Agent Permission System

This is a shorter version of the full proposal here: open in Google Docs

Summary

Modern AI agents take open-ended actions — reading files, calling tools, making network requests — yet the dominant deployed safety mechanism is "LLM-as-judge," which offers no formal guarantees and is itself susceptible to prompt injection (Greshake et al., 2023). We will build a permissioning and runtime-monitoring system that combines the ergonomics of Claude Code's per-tool permission prompts with the expressive policy language of AWS IAM, grounded in classical results from runtime verification (linear temporal logic on finite traces, De Giacomo & Vardi, 2013) and language-based information-flow control (Sabelfeld & Myers, 2003). Policies are written in a high-level DSL, compiled to monitorable LTLf formulas, and machine-checked in Lean 4. We integrate the monitor into a popular agent framework and evaluate against an adversarial "live-off-the-land" benchmark drawn from the AI control literature (Greenblatt et al., 2024) and AgentDojo (Debenedetti et al., NeurIPS 2024), measuring both attack-prevention and benign-task preservation. The result is the first formally-mechanized monitoring system for agents and the first quantitative methodology for comparing agent safety mechanisms head-to-head.

Target output

A short paper at ETAPS/TACAS 2027 (submission window late-Sept/early-Oct 2026) introducing a formally-grounded permission and runtime-monitoring system for AI agents, with an open-source implementation and evaluation on an adversarial benchmark.

Theory of impact

The only widely-deployed agent-safety mechanism today is "LLM-as-judge" — unprincipled, unauditable, and adversarially fragile. As agents move into higher-stakes settings (developer tools, enterprise workflows, autonomous research), the absence of formal-method-grade guardrails is a clear and growing gap. Our work delivers (1) the first end-to-end formally-grounded monitoring system for agents with machine-checked soundness; (2) an open-source artifact integratable with mainstream frameworks; (3) a quantitative benchmark methodology future agent-safety papers can compare against. The design pattern — declarative

high-level DSL, temporal-logic IR, mechanized monitor — generalizes beyond agents to any setting where a non-deterministic actor (an LLM, a user, an automated pipeline) operates over sensitive resources.

Desired expertise and skill sets

Knows basic set theory and first order logic. Either can code or is familiar with using Claude …

High-level background

Five strands of prior work motivate this project.

1. AWS Identity and Access Management (IAM). A widely-deployed, declarative, resource-level authorization system whose semantics have been formalized for SMT-based analysis in the Zelkova project (Backes et al., FMCAD 2018). IAM demonstrates that fine-grained, expressive policies can scale to industry use; it also demonstrates that policy authoring is genuinely hard, motivating Zelkova-style "is policy P more permissive than policy Q?" tooling.
2. Linear Temporal Logic on Finite Traces (LTLf). De Giacomo & Vardi (IJCAI 2013) show LTLf is decidable in PSPACE and admits efficient automaton-based monitor synthesis. This is the natural logical substrate for monitoring agent trajectories, which are finite by construction.
3. Runtime verification (RV). Bauer, Leucker & Schallhart (TOSEM 2011) and the broader RV community give us 20 years of theory on online monitors that emit verdicts (sat / unsat / inconclusive) against temporal specifications.
4. Language-based information-flow control (IFC). Denning's lattice model (CACM 1976) and Sabelfeld & Myers (JSAC 2003) provide the formal vocabulary — labels, noninterference — for reasoning about what data the agent can transmit where, which simple resource ACLs cannot capture (e.g., "the agent may read ~/secrets, and the agent may make network requests, but it may not do both in the same trace").
5. Claude Code's permission system. A live, widely-used UX baseline for per-tool, per-invocation human-in-the-loop authorization. We treat this as the ergonomic floor: any policy system that is harder to use than Claude Code's permission prompts will not be adopted.

How we relate to other agent-safety work: research on prompt-injection defenses (AgentDojo; Greshake et al., 2023) and AI control (Greenblatt et al., ICML 2024) is largely about detecting misbehavior post-hoc. We are instead constraining what actions the agent can take in the first place, with provable bounds — a complementary defense layer.

Detailed technical background

The two works our methodology most directly extends.

(1) De Giacomo & Vardi, "Linear Temporal Logic and Linear Dynamic Logic on Finite Traces" (IJCAI 2013). They establish that LTLf is closed under standard temporal operators, has finite-state monitor automata constructable by tableau methods, and admits PSPACE satisfiability checking. We will use LTLf as the target intermediate representation of policy compilation: every policy in our DSL desugars to an LTLf formula over a fixed signature of action predicates (read(path, label), net(host, label), exec(cmd), tool(name, args, label)). The original paper treats abstract atomic propositions and is silent about how the propositional alphabet is grounded; our extension is to define a concrete action signature for agentic systems, embed it together with the LTLf semantics in Lean 4 / Mathlib, and prove monitor soundness (a trace is accepted by the monitor iff it satisfies the policy formula).

(2) Cedar — Cutler et al., "Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization" (OOPSLA 2024). Cedar is the most recent and most rigorously designed policy language in the AWS family; importantly for us, the Cedar team modeled the language in Lean and proved soundness of its policy validator. We adopt Cedar's design philosophy (analyzability through restricted expressiveness, mechanized meta-theory, push-button policy comparison) and extend it along the temporal axis: where Cedar policies are stateless authorization predicates, our policies are temporal formulas over agent traces. Limitations of Cedar we address: (i) no temporal operators ("never X after Y"), (ii) no information-flow labels (a policy can authorize access to a file but cannot constrain where that data subsequently flows), (iii) no notion of "the principal is itself partially under attacker control," which is the defining feature of an agent under prompt injection.

The gap our work fills: neither runtime-verification nor IAM-style authorization was designed for AI agents. The interesting safety properties for agents are temporal and information-flow-shaped; the action vocabulary is open-ended; and the "principal" is itself a partly-adversarial system. Joining these threads is, to our knowledge, novel.

Proposed methodology

Hypothesis. A policy-driven runtime monitor over LTLf with information-flow labels can prevent a substantial fraction of adversarial agent behaviors with low false-positive rate on benign tasks, outperforming both LLM-as-judge defenses and ad-hoc per-tool allowlists.

Approach.

1. Policy DSL. Surface syntax inspired by Cedar and Open Policy Agent's Rego, extended with (a) temporal operators (never, until, after) and (b) information-flow labels drawn from a user-defined lattice.
2. Compilation. Desugar to LTLf formulas over a fixed action signature.
3. Monitor synthesis. Construct deterministic finite automata via standard LTLf-to-DFA constructions, mechanized in Lean 4. Soundness theorem: for all traces τ and policies φ, the monitor accepts τ iff $\tau \models \varphi$.
4. Integration. Wrap an existing agent framework (we propose Claude Code, with LangChain as a fallback) so every tool invocation is mediated by the monitor. Rejected actions surface to the user via a Claude-Code-style permission prompt, preserving the existing UX.
5. Validation. Run on adversarial benchmarks (next section) and measure (a) attack-prevention rate, (b) benign-task success rate, (c) policy authoring time and correctness in a small user study (stretch).

Baselines. (i) No protection; (ii) Claude Code's default permission system; (iii) LLM-as-judge in the style of Constitutional Classifiers; (iv) hand-written per-tool allowlists.

Key assumptions. Tool calls are observable and synchronous (true for current frameworks). IFC labels can be assigned to data sources by operators with reasonable effort (validated qualitatively in the user study). The interesting attack surface is dominated by chained tool misuse (read X → exfiltrate via Y) rather than novel zero-days inside individual tools.

Falsification criteria. If on the chosen benchmark our monitor either (a) blocks fewer than 30% more attacks than the LLM-judge baseline at equal false-positive rate, or (b) blocks more than 20% of benign trajectories the unprotected baseline completes, we will report a negative result.

Preliminary experiments

We have surveyed agent-safety benchmarks and identified AgentDojo (97 tasks, 629 security test cases) and the AI control suite as the most credible adversarial substrates: AgentDojo for prompt-injection-driven misbehavior, AI control for "live-off-the-land" subversion by a knowing adversary. A first-pass categorization of AgentDojo failure modes shows the majority involve cross-context data flow ("read X from one source, exfiltrate to Y") — exactly the pattern IFC labels are designed to capture. We have also prototyped a Lean 4 encoding of LTLf semantics on a small action signature (~30 LOC), which we will extend.

Kiran Gopinathan (Basis Research Institute)

Kiran Gopinathan (Basis Research Institute) | Fuzzing Proof Assistants

This is a shorter version of the full proposal here: open in Google Docs

Summary

Formal verification is gaining traction as a defence against insecure AI-generated code, making the correctness of proof assistants themselves a safety-critical concern. Every proof checked by Lean 4 rests on a trusted computing base (C++ runtime, elaborator, type checker, and compiler) that sits outside the reach of the proofs it certifies. A bug in any of these components can silently invalidate every theorem built on top of them.

We propose building systematic fuzzing infrastructure for Lean 4, targeting each stage of its compilation pipeline. Our approach combines coverage-guided fuzzing (AFL++) with grammar-aware test generation to exercise the elaborator, type checker, and code generator. In preliminary work, we fuzzed a verified Lean library and discovered a heap buffer overflow in the Lean runtime affecting every version of Lean 4 to date. The verified application code had zero memory safety bugs across 105 million executions. We will evaluate by measuring code coverage per pipeline stage, classifying discovered bugs by severity (soundness-critical, memory safety, crash), and assessing whether the methodology generalises to other proof assistants.

Target output

Academic paper on systematic fuzzing of Lean 4's trusted computing base, targeting ICSE 2027 (submission deadline ~October 2026).

Theory of impact

Formal verification is emerging as a key defense against insecure AI-generated code, making proof assistant correctness a safety-critical concern. A bug in Lean's runtime or type checker does not just affect one program—it can silently invalidate every theorem Lean has ever checked. Trail of Bits recently showed that implementation bugs in ZK proof systems can forge proofs; analogous bugs in proof assistants carry similar risks. Our fuzzing infrastructure will be open-source and designed for continuous integration, providing ongoing assurance as Lean

evolves. The methodology generalizes to other proof assistants (Rocq, Isabelle) and any language with a complex type system. By systematically auditing the tools that underpin formal verification, we strengthen the entire foundation on which verified software—and the AI safety case for verified code—rests.

Desired expertise and skill sets

The ideal mentee has hands-on experience with at least one of the following: (1) fuzzing and software testing infrastructure (AFL++, libFuzzer, or similar coverage-guided fuzzers), (2) compiler or programming language implementation (particularly familiarity with type checkers, elaborators, or code generators), or (3) working with proof assistants or dependently typed languages (Lean, Rocq, Agda). Experience with C/C++ and comfort reading runtime code is valuable, since a significant portion of the work involves writing fuzzing harnesses against Lean's C++ runtime. Familiarity with sanitisers (AddressSanitizer, UBSan) and debugging tools (Valgrind) is a plus. A mentee with a systems engineering or security background who is curious about formal methods would be a strong fit, as would a PL researcher interested in testing and verification tool reliability. Graduate students (MSc or PhD) in programming languages, software engineering, or security are well-suited, though strong undergraduates with relevant systems experience are also welcome.

High-level background

1. CSmith (Yang et al., PLDI 2011): The gold standard for compiler fuzzing. CSmith generates random C programs guaranteed to avoid undefined behaviour and uses differential testing across compilers to detect disagreements. Over several years it found over 325 previously unknown bugs in GCC and LLVM, many in mature, heavily tested optimisation passes. CSmith demonstrated that systematic fuzzing can find deep defects in production language implementations. Our work adapts this methodology, random program generation plus differential testing, to the qualitatively different setting of a dependently typed proof assistant.

1. Chaliasos et al., “Well-typed programs can go wrong” (OOPSLA 2021): A systematic study of typing-related bugs in JVM compilers, finding that many bugs cluster in the type checker rather than code generation. This directly motivates our focus on the elaborator and type checker of proof assistants, where analogous complexity exists but with higher stakes—a type checker bug in Lean can render proofs unsound.

1. lean-zip (Kim, 2026, GitHub): A verified implementation of zlib in Lean 4, proved correct end-to-end by 10 autonomous agents. This was the target of our preliminary fuzzing experiments. The results were promising: we found the verified application code was

robust across 105 million fuzzing executions, while both bugs we found sat outside the proof boundary—validating both the power of verification and the vulnerability of the unverified trusted computing base.

1. *Lean 4* (de Moura and Ullrich, 2021): The proof assistant we target. Its architecture, extensible syntax via user-defined macros, a complex multi-stage elaboration pipeline, and compilation to C, defines our attack surface and introduces challenges absent in traditional compiler fuzzing: generated test inputs must not only be syntactically valid but also conform to Lean's macro expansion and dependent type checking to reach interesting code paths.

Detailed technical background

1. Our preliminary work: "Lean proved this program was correct; then I found a bug" (blog post, 2026)

We took a verified Lean library—lean-zip, a zlib implementation with machine-checked correctness proofs—stripped it of all theorems and documentation to avoid biasing the fuzzer, and systematically fuzzed the resulting code. Using AFL++ with AddressSanitizer and UndefinedBehaviorSanitizer, Claude ran 16 parallel fuzzers across 6 attack surfaces (ZIP extract, gzip decompress, raw DEFLATE inflate, tar extract, tar.gz, and compression), accumulating 105 million fuzzing executions over 19 hours.

The results: (a) zero memory safety bugs in the verified application code, (b) a heap buffer overflow in \lean_alloc_sarray\allocsarray\`—the C++ function that allocates every scalar array (\ByteArray\, \FloatArray\, etc.) in Lean 4—caused by an integer overflow when the capacity approaches \SIZE_MAX\, and (c) a denial-of-service in lean-zip's unverified archive parser. The heap overflow affects every version of Lean 4 and was later fixed. A 156-byte crafted ZIP file with a ZIP64 \compressedSize\ of \0xFFFFFFFFFFFFFFFF\ is sufficient to trigger it.

The key insight driving the proposed work is that verification dramatically improves code quality where it is applied, but the trusted computing base—the runtime, elaborator, and compiler—remains unverified and vulnerable. Our preliminary work only exercised the runtime through a single library's allocation paths. The proposed work scales this methodology from one library to Lean's entire compilation pipeline: parser, macro expansion, elaboration, type checking, and code generation.

2. CSmith (Yang et al., PLDI 2011)

CSmith is a random program generator for C that avoids undefined behaviour by construction. It generates C programs, compiles them with multiple compilers, and flags any disagreement in output as a compiler bug. This methodology—generate valid programs, compile, check for

anomalies—is what we adapt to the proof assistant setting. One of CSmith's core insights was that the "valid program" constraint is essential: randomly generated bytes almost never reach interesting compiler code paths, while well-formed programs exercise the type checker, optimiser, and code generator.

The key challenge in adapting CSmith's methodology to Lean 4 is that "valid" is substantially harder to define and generate. For C, CSmith must avoid undefined behaviour (uninitialised reads, signed overflow, etc.), which is tractable because C's type system is simple. For Lean 4, generating programs that reach interesting elaboration paths requires navigating dependent types, universe polymorphism, tactic blocks, and user-defined macros. We address this in stages: starting with syntactic-level generation that exercises the parser and early elaboration, then incrementally adding type awareness to reach deeper pipeline stages.

A secondary gap is that CSmith's differential testing requires multiple independent compilers for the same language. Lean 4 has only one implementation. We compensate with three differential testing strategies: comparing interpreted vs. compiled execution, comparing across Lean versions, and comparing Lean's kernel against independent checkers via lean4export.

Proposed methodology

Hypothesis: Lean 4's trusted computing base contains undiscovered bugs—including potential soundness bugs—that can be systematically uncovered through targeted, multi-stage fuzzing.

Approach: We target each stage of Lean's compilation pipeline with an appropriate fuzzing strategy:

1. Runtime fuzzing: Build AFL++ harnesses with AddressSanitizer and UBSan for the C++ runtime (memory allocation, reference counting, IO primitives). This extends our preliminary work—which only exercised runtime paths reachable through lean-zip—to the full runtime API surface.

1. Grammar-aware elaborator fuzzing: Develop a program generator that produces syntactically valid Lean syntax, including custom DSLs, tactic invocations, and macro applications. Mutations operate at the AST level rather than the byte level, ensuring inputs reach the elaborator and type checker instead of being rejected by the parser.

1. Differential testing: Compare execution results between (a) Lean's interpreter and its compiled C output, (b) different Lean versions, and (c) Lean's kernel and independent checkers via lean4export. Disagreements indicate bugs in either the compiler or the checker.

1. Targeted component testing: Build focused harnesses for known complex subsystems—the \grind\ tactic, universe checking, instance resolution—informed by analysis of historical bug reports. Our preliminary analysis of Lean's issue tracker suggests historically patched components exhibit recurring vulnerabilities.

Baselines: We compare against (a) byte-level AFL++ without grammar awareness, (b) random syntactically valid Lean programs without type awareness, and (c) the existing Lean test suite's coverage.

Validation: We confirm or deny the hypothesis by measuring: code coverage achieved per pipeline stage, number and severity classification of bugs found (soundness-critical, memory safety, crash, performance), and time-to-first-bug for each strategy. A positive result is finding at least one previously unknown soundness or memory safety bug. A null result (no new bugs found) would itself be informative, providing evidence for the current robustness of Lean's implementation, and would be reported as such.

Key assumptions: (1) Lean's pipeline has bugs reachable by fuzzing (supported by our preliminary heap overflow finding). (2) Grammar-aware generation can achieve meaningful coverage of the elaborator (we validate via coverage measurement). (3) Found bugs will be actionable by Lean maintainers (supported by the accepted bug report and fix from our preliminary work).

Proposed timeline / milestones

Milestone: Runtime and Parser Fuzzing Infrastructure

Description: Build AFL++ fuzzing harnesses for Lean 4's C++ runtime (memory allocation, reference counting, IO primitives) and parser. Set up continuous fuzzing infrastructure. Reproduce existing known bugs (stack overflows, segfaults) as validation of the harness. Establish coverage baselines across runtime functions.

Target Completion Date: Month 2 (~Week 8)

Key Deliverables: Fuzzing harnesses for Lean runtime, continuous fuzzing CI, initial bug reports, coverage baseline report.

Milestone: Grammar-Aware Elaborator Fuzzing and Differential Testing

Description: Build a grammar-aware Lean program generator that produces syntactically valid terms including dependent types, tactics, and macro invocations. Implement differential testing framework comparing Lean's interpreter vs. compiled output, across Lean versions, and against

lean4export's independent kernel reconstruction. Target elaborator, type checker, and tactic framework (especially \grind\, instance resolution, universe checking).

Target Completion Date: Month 4 (~Week 16)

Key Deliverables: Grammar-aware fuzzer, differential testing framework, bug reports, coverage comparison against baseline strategies.

Milestone: Systematic Evaluation and Paper Submission

Description: Run full evaluation campaign across all pipeline stages. Classify all discovered bugs by severity (soundness-critical, memory safety, crash, performance). Measure and report code coverage per pipeline stage and per fuzzing strategy. Assess generalisability by running the grammar-aware fuzzer against at least one additional proof assistant (Rocq). Write and submit paper to ICSE 2027.

Target Completion Date: Month 6 (~Week 24)

Key Deliverables: Paper submission to ICSE 2027, open-source fuzzing infrastructure release, comprehensive bug report with responsible disclosure.

Preliminary experiments

Our primary preliminary result is the blog post "Lean proved this program was correct; then I found a bug". Over 105 million fuzzing executions against a verified Lean library, we discovered:

- A heap buffer overflow in \lean_alloc_sarray\allocsarray\` in the Lean 4 runtime, caused by an integer overflow in allocation size computation. A 156-byte crafted ZIP file triggers it. The bug affects every version of Lean 4, has an accepted bug report, and a subsequent fix.

- A denial-of-service in lean-zip's unverified archive parser, where a crafted ZIP header claiming an exabyte-sized payload causes an out-of-memory crash.

- Zero memory safety bugs in the verified application code across all 105 million executions.

These results validate our key assumptions: fuzzing can find real, impactful bugs in Lean's trusted computing base with relatively modest effort, and verified code is dramatically more robust than unverified code, motivating focused attention on the unverified infrastructure.

Additionally, we have begun cataloguing existing Lean 4 bugs (stack overflows in \makeLambdaFVars\, segfaults on specific architectures, \grind\ tactic stack overflows) and analysing whether historically patched locations exhibit recurring vulnerability patterns. We have built preliminary fuzzing infrastructure (lean-fuzz).

Risks, uncertainties, monitoring & mitigation

The strongest argument against this work is that Lean 4 is already well-engineered and actively maintained, and fuzzing may yield diminishing returns after the low-hanging fruit (like the runtime overflow we already found) is picked. We mitigate this by targeting multiple pipeline stages with qualitatively different strategies—byte-level runtime fuzzing and grammar-aware elaborator fuzzing exercise fundamentally different code paths, and historically, even mature compilers like GCC continued to yield bugs to CSmith for years.

A second risk is that grammar-aware fuzzing for a dependently typed language is technically hard. Generating well-typed Lean terms that exercise interesting elaboration paths may require significant engineering effort. We mitigate this by starting with syntactic-level generation (which already suffices for parser and early elaboration bugs) and incrementally adding type awareness, ensuring each stage independently produces useful results even if deeper stages prove more difficult than expected.

A third concern is dual-use: our fuzzing infrastructure could be used to find exploitable bugs before fixes are available. We mitigate this through responsible disclosure practices—as we did with our initial Lean runtime bug report—and by working directly with the Lean FRO maintainers. This risk is modest in practice: proof assistant bugs primarily threaten the integrity of mathematical proofs rather than enabling exploitation of deployed software. The work is overwhelmingly defensive—strengthening verification infrastructure benefits everyone who relies on formal proofs.

Capacity

10-15 hours per week (PI), plus a PhD student contributing complementary effort on bug cataloguing, harness development, and experimental evaluation.

Jeffrey Chang + Kanghee | Bootstrapping Semantics

Jeffrey Chang + Kanghee (Theorem) | Bootstrapping Semantics with Mutation Testing

Summary

The goal of the project is to develop a pipeline for bootstrapping semantics of programs in an automated fashion, without any oversight from humans. In order to do this, we must build a harness that provides feedback on whether models have generated the correct specification for a given piece of software.

- We have conducted preliminary work to derisk the research direction. During the fellowship, you will conduct empirical ML research, which will include building a dataset, conducting evaluations, finetuning open-source models, and communicating research findings to a broader audience.
- The core aesthetic for working on our projects is that we want to minimize human effort required in specification, and want to develop methodologies that scale to all existing and future software.

Theory Intuition

A specification is a property of programs, `spec : program → Proposition`.
(`Proposition`, written `Prop` in Rocq and Lean, can be thought of as like `bool`, but with no requirement of being computable)
A good spec has three properties:

1. It is true (of the program `p` you care about – `spec p`)
2. It sufficiently constraints the program (`forall p', p' and p have different behavior → ¬ spec p'`) (or if you want to be more constructive, `forall p', spec p' → p and p' behave identically`)
3. It is not overly constraining (e.g., it does not say exactly what `p`'s source code should be) (`forall p', if p and p' behave identically, then spec p'`)
4. It is simple (/ readable / short / concise)

Taken together, 2 & 3 say that a good spec is equivalent to "semantic equivalence to the program of choice" (modulo, of course, the notion of behavioral equivalence you care about).
(1) then just says that your notion of behavioral "equivalence" is reflexive.
((4) is about the syntax of your property, and not about its semantics)

In this sense, the best spec is just the shortest way of writing down a (possibly declarative) reference implementation of the program, together with a description of what behavioral equivalence means.

Specs can be tested against (2) and (3) by randomly sampling mutations of the source program, checking for behavioral equivalence, and checking the spec.

Aditional Questions from Mentors

Mentor-Specific Screening Questions

Shriram Krishnamurthi | Formalizing properties using PICK (Brown)

Please tell us about your experience using rich type systems, model checkers, or other formal methods apparatus.

Shriram Krishnamurthi | Specs from examples using Alloy/Forge (Brown)

Please tell us about your experience with formal modeling and with programming in non-mainstream languages.

Erik Meijer | Universalis/Automind (Leibniz Labs)

Universalis lives at the seam of hacking and research: implementation and proof, in the same day. We're looking for people who enjoy that oscillation. Tell us about a time you did both on one project, and what you got out of switching modes.

Shaowei Lin + Jin-Xing Lim | Deductive Vericoding (Beneficial AI Foundation)

Do you have any experience with interactive theorem provers (e.g. Lean, Rocq, Agda, Isabelle)? Which one? A few words about what you did?

Jocelyn Qiaochu Chen | Mining Enforceable Specifications for LLM-Agent Tool Use

What is an example of a rule that is easy for a human to understand but surprisingly hard for software to check?