AI Harnesses for Lean Proofs

AI for Mathematics (General Elective) — Week 11

16 June 2026

Formal proof: the hard part is generation

Same loop as Week 10 — generate → verify → repair → accept — but a different step is the hard one.

- Natural-language proof (5B): the hard part is **verification**. An AI verifier reads the proof and decides, and it can be wrong either way. Most of the work went into making that judgment trustworthy.
- Formal proof (Week 11): verification is easy — the Lean kernel checks the proof and cannot be fooled. The hard part is **generation**: producing something the kernel will accept.

Week 11 uses the Lean kernel

Now "accept" means the Lean kernel agreed:

- `lake env lean` returns with no errors
- `#print axioms` shows only the three axioms Lean trusts: `propext`, `Classical.choice`, `Quot.sound`

The kernel is a compiled program. It will not accept a wrong proof. The trust model is just the kernel plus those three axioms.

What mechanical acceptance looks like:

```
$ lake env lean B3.lean
  -- 0 errors, 0 sorry

#print axioms
  putnam_2025_b3
[propext,
 Classical.choice,
 Quot.sound]
```

Roadmap —Week 11

Part A Autoformalization: what AI does when it writes Lean

Part B Ecosystem timeline + layer map

Part C Case study #1: numina-lean-agent

Part D Case study #2: Archon

Part E Across the harnesses

Part F The Lean-LSP-MCP shared layer

Part A · Autoformalization

Autoformalization is several different jobs

Most attention goes to filling the `sorry`. Here are the distinct jobs:

- Prove a stated theorem (fill the `sorry`) — in practice you sketch a skeleton (have-chain), pull out helper lemmas, and fill each; these go together, not in sequence
- Formalize statements and definitions — turn a theorem or a definition into Lean; usually done together, since a statement needs Lean versions of the definitions it mentions
- Refactor — shorten, generalize, or speed up a finished proof
- Document — write the blueprint outline and name declarations
- Translate — port a Rocq / Isabelle / Lean 3 proof to Lean 4

Finding the right Mathlib lemma (*retrieval*) is a method used while proving, not a separate job — it came from LeanDojo in 2023.

Which system does which job

The six systems this lecture surveys — none does all of it:

- numina-lean-agent — fills sorries, proposes helper lemmas, retrieves premises (one theorem at a time)
- Archon — scaffolds statements + definitions, proposes skeletons, fills sorries, refactors, writes the blueprint (whole project)
- Leanstral — fills sorries, translates Rocq, synthesizes code + proof
- LeanAgent — fills sorries across many repositories; trains its own premise retriever
- In-editor copilots (Lean Copilot, LLMStep, Sagredo) — suggest one tactic at a time
- AutoformBot — translate a paper into Lean files (statements + definitions)

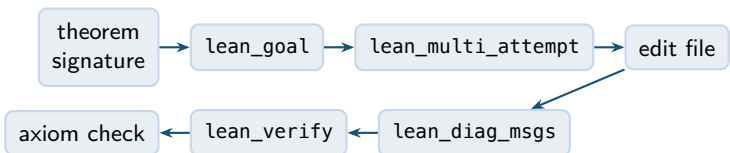
Parts C and D go deep on numina and Archon.

Prove a stated theorem (fill the `sorry`)

Takes `theorem t ... := by sorry`; produces a proof the kernel accepts (`sorry`-count $= 0$).

Not one move but three, interleaved: sketch a have-skeleton, pull the hard parts out as helper lemmas, and fill each — recursively, until no `sorry` remains. (Skeleton and lemmas: next two slides.)

One fill step, as an agent runs it:



Done means `lake env lean` is clean and `#print axioms` lists only `propext`, `Classical.choice`, `Quot.sound`.

Proving, part 1: the skeleton

Lay the proof out as a have-chain whose leaves are `sorry`:

```
theorem T : P := by
  have h1 : P1 := by sorry
  have h2 : P2 := by sorry
  ...
  combine h1 h2 ...
```

DeepSeek-Prover-V2 does this with two models: a big general model (DeepSeek-V3) sketches the have-decomposition, a smaller prover model fills each leaf.

A have lives inside one proof; when a sub-goal is reusable or large, it is promoted to a standalone helper lemma instead — the next slide.

Proving, part 2: helper lemmas

Pull the hard sub-goals out as smaller, stated `lemmas` (proofs left for later). Euler's partition proof — the helpers numina extracted:

Lemma	Role
`two_adic_odd_factorization`	every $x > 0$ is uniquely $2^r \cdot a$, a odd
`ordProj/ordCompl_two_pow_mul`	recover 2^r and a from $2^r \cdot a$
`psi_sum`	Ψ preserves the sum (still a partition of n)
`psi_odd`	every part of Ψ is odd
`phi_sum`	Φ preserves the sum
`phi_nodup`	Φ's parts are distinct
`psi_phi`	$\Psi \circ \Phi = \mathrm{id}$ (odd side)
`phi_psi`	$\Phi \circ \Psi = \mathrm{id}$ (distinct side)

Each helper is its own `sorry` to fill — one hard proof becomes several easier ones.

Premise retrieval: a method, not a task

While proving you constantly need the right Mathlib lemma for the current goal. Finding it is *retrieval* — a method, first used by LeanDojo's ReProver (2023). Today it is offered as six tools in `lean-lsp-mcp`:

```
lean_local_search    ripgrep over .lake/packages — fast, no rate limit
lean_leansearch      leansearch.net — natural language to Mathlib name
lean_loogle          loogle.lean-lang.org — type-pattern search
lean_leanfinder      Hugging Face endpoint — semantic by meaning
lean_state_search    premise-search.com — goal to closing lemmas
lean_hammer_premise  Lean Hammer server — premises for simp/aesop
```

numina wraps these as its Leandex search; Archon uses a `mathlib-analogist` subagent. (How these retrievers are trained is a Week 12 topic.)

Euler's partition theorem (the running example)

In 1740 Philippe Naudé asked Euler in how many ways n can be written as a sum of *distinct* positive integers. Euler's study of that question (published 1748) launched partition theory and gave:

partitions of n into **distinct** parts $=$ partitions into **odd** parts (in number).

Example $n = 6$, four each:

- distinct: $6 \mid 5{+}1 \mid 4{+}2 \mid 3{+}2{+}1$
- odd: $5{+}1 \mid 3{+}3 \mid 3{+}1{+}1{+}1 \mid 1{+}1{+}1{+}1{+}1{+}1$

Euler proved it with generating functions; the formalization here builds the *constructive* bijection instead. We use it as the running example below.

Formalize statements and definitions: one job

Usually these come together: to state a theorem in Lean you first need Lean definitions for everything it mentions. A statement drags in its definitions.

When the definitions are already in Mathlib it is mostly the signature — Euler's partition theorem reuses Mathlib's `Nat.Partition`:

```
theorem euler_partition (n : ℕ) :
    (Nat.Partition.distincts n).card = (Nat.Partition.odds n).card := by sorry
```

The work is in the encoding, not the syntax:

- reuse Mathlib's `distincts` / `odds` — so it is genuinely Euler's theorem
- `.card` counts the partitions (a `Finset`)
- your own non-equivalent "distinct" predicate would compile but prove the wrong thing

Done by: Archon (scaffold phase), numina, AutoformBot.

When the definitions are new

The statement reused Mathlib, but the *proof* needs definitions Mathlib does not have: Euler's bijection maps Φ, Ψ. The agent writes them.

```
-- not in Mathlib: Ψ replaces each part 2^r·a (a odd) by 2^r copies of a
def psi (μ : Multiset ℕ) : Multiset ℕ :=
  μ.bind fun x => Multiset.replicate (ordProj[2] x) (ordCompl[2] x)
```

Design choices when writing the map:

- computable or not: a plain `def` with no `Classical.choice` keeps `#eval` working
- which data structure: `Multiset ℕ` (order ignored, multiplicities kept)
- how to factor a part: `ordProj[2]` / `ordCompl[2]` (the 2-adic split)

Done by: numina / Archon write the maps; Archon scaffold writes new structures.

Refactor: clean up a finished proof

Takes a proof the kernel already accepts; produces an equivalent one that meets a quality bar. Four kinds:

- Shorten — a 50-line `calc` chain → one `omega` / `linarith`
- Generalize — a lemma over `Nat` → over `[Semiring α]`
- Speed up — an 8-second proof → under 1 second
- Migrate — propagate a Mathlib rename across every use site

Done by: numina's `golfer` (after a long proof succeeds); Archon's polish phase.

Document: blueprint outline and naming

Two jobs: write the natural-language blueprint, and name declarations so tactics can find them.

A blueprint block (Archon scaffold phase):

```
\begin{theorem}[name_for_humans]
  \lean{namespace.theorem_name}
  \uses{def:related_definition, lem:supporting_lemma}
  % SOURCE: [Hartshorne] III.5.1
  Informal statement in the project's notation.
\end{theorem}
```

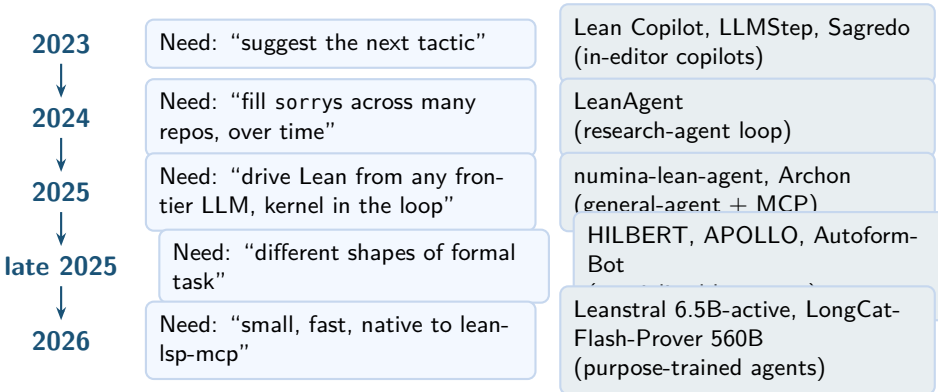
Naming: `<namespace>.<conclusion>_<connective>_<assumption>`, e.g. `add_comm`, `Nat.lt_of_lt_of_le`. Good names are what let `rw`, `simp`, and `exact?` find a lemma.

Three less-common jobs

- Translate — port a Rocq / Isabelle / Lean 3 proof to Lean 4, keeping the meaning across different type systems. Leanstral does this.
- Code + proof together — from a Lean spec, write a program and a proof it meets the spec (verified compilers, crypto, parsers). Leanstral.
- Refute — given a suspected-false statement, produce a counterexample and a Lean proof of its negation. Catches a bad statement formalization; no S06 system does it as a main job.

Part B · The Ecosystem

B1 · Timeline of tools by year



B2 · Tool layers and first appearance

Layer	First appeared	Example systems
shared layer	2025	lean-lsp-mcp
in-editor	2023	Lean Copilot, LLMStep, Sagredo
general-agent	2025	numina-lean-agent, Archon
purpose-trained	2026	Leanstral, LongCat-Flash-Prover, Kimina
autoformalizer	late 2025	AutoformBot
minimal-agent	late 2025	APOLLO (repair-focused)
closed-research	2024–2026	Seed-Prover, Aristotle, Aleph (PutnamBench)

- The **shared layer** row is the same across the whole stack.
- In-editor copilots predate the shared layer — they run inside Lean's metaprogramming layer.
- General-agent and purpose-trained rows both attach to the shared layer.

B3 · Three observations about the stack

1 · Shared MCP layer

By 2026 every agent in the general-agent, purpose-trained, autoformalizer, and minimal-agent rows talks to Lean through the same 22 tools.

Differences between harnesses are above this layer.

2 · Each layer addressed a gap

Each layer was added because the previous layer had a **visible limitation**:

- in-editor: tactic only
- LeanAgent: trained-corpus only
- general-agent: prompt-engineering only
- purpose-trained: protocol internalized

3 · Purpose-trained models extend into the harness

Leanstral and LongCat-Flash-Prover were trained on tool-use trajectories, so they have the **protocol** built in rather than relying on the prompt.

They operate like agents but are distributed as models.

Part C ·numina-lean-agent

C1 · The system and what it proved

What it is

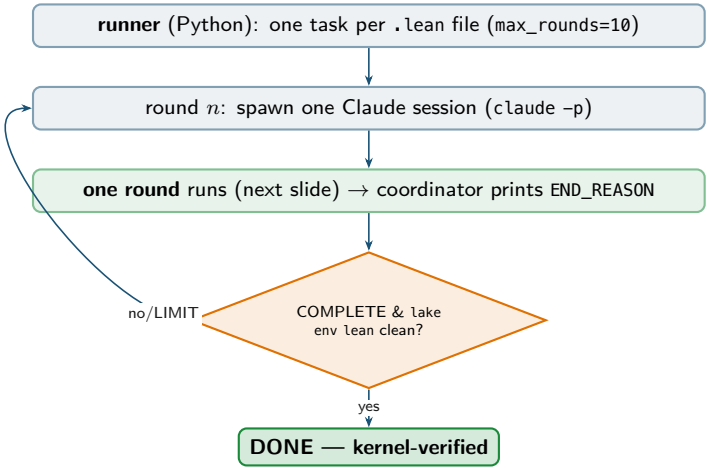
- Claude-Code agent for filling `sorry`s in Lean 4 theorems
- Repo: `project-numina/numina-lean-agent` (MIT, `e9e987b`)
- Paper: arXiv 2601.14027

Results

- **Putnam 2025: 12/12** problems proved end-to-end
- **Brascamp–Lieb** (arXiv 2511.11091): $\sim 8{,}000$ Lean lines
- Leandex: `leandex.projectnumina.ai`

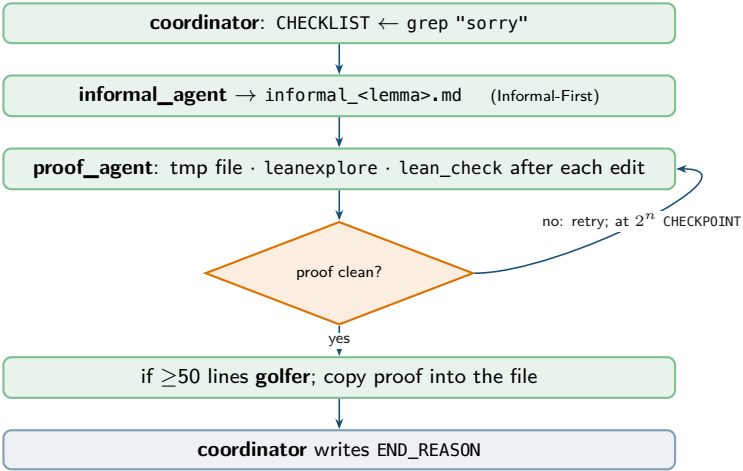
Lean is the only verifier. The architecture follows from that choice.

C2 · The overall run: the round loop



The runner never decides "done": it reads the `END_REASON` sentinel, then `lake env lean` does. On fail/LIMIT it resends the prompt as round $n+1$. `StatementTracker` reverts statement edits; `SafeVerify` (optional) replays.

C2 · Inside one round (one Claude session)



Most retries stay inside `proof_agent`. At attempts $2, 4, 8, 16, \ldots$ it instead returns `CHECKPOINT`; the coordinator respawns `informal_agent` (refine). Next: `should_refine`.

C2 · The CHECKPOINT mechanism

Trigger — `proof_agent.md:108–112`, a pure function of the inner attempt counter:

```
def should_refine(attempts):
    # At 2, 4, 8, 16, 32... trigger refine
    return attempts > 0 and (attempts & (attempts - 1)) == 0
```

Handshake — `proof_agent.md:273–280`: the proof agent does *not* respawn; it exits with the literal result word:

```
Result:  CHECKPOINT
Attempts: 8
Message:  At informal refine checkpoint. Please spawn informal_agent.
Current stuck point: Cannot prove h2, tried simp and omega.
```

The coordinator reads that, spawns `informal_agent` in *refine* mode (passes attempts, stuck point, recent `lean_check` diagnostics), Gemini writes `informal_<lemma>.md` v{N+1}, the coordinator respawns the proof agent. Doubling means v1 gets 1 attempt, v2 gets 2, v3 gets 4, v4 gets 8, ...: **a bad plan is replaced fast; a plausible plan is given room**. "Blocked" is only legal at ≥ 30 attempts.

C2 · The four agents: consumes and produces

Agent	Consumes	Produces
`coordinator` (router)	target task; subagent results; run state	Task dispatches; CHECKLIST + status; `END_REASON` line
`informal_agent`	one `sorry`; at checkpoints, failed attempts + diagnostics	`informal_<lemma>.md`: strategy, lemma names, pitfalls
`proof_agent`	that `.md` + the goal + search tools	tactic proof in `.lean`; `lean_check` result; CHECKPOINT
`golfer`	a working proof $\geq$50 lines	a shorter proof, still kernel-clean

The Python `runner` is not an agent: one `.lean` file per task, spawn the session, count rounds, run the kernel check. The four agents share one Claude session and pass work through files.

C3 · The coordinator prompt — main_entry.md

From `prompts/autosearch/main_entry.md` (verbatim; ellipses skip narration only):

```
Carefully read the prompts under prompts/autosearch/
subagent_prompts/ (common.md, coordinator.md,
proof_agent.md, informal_agent.md, golfer.md).
Also read skills/SKILL.md and each sub-skill's SKILL.md
(search, verification, llm, code-transform, sorrifier)...

You should act as coordinator agent to complete the target.

When you launch a subagent, you MUST:
1. Describe the assigned task clearly
2. Include the absolute path to its corresponding prompt file
3. Include the absolute path to the reference resources...
4. Instruct the subagent to update its entry in CHECKLIST.md
5. All verification must go through lean_check.py
   (never `lake build` for per-file checks).

...at most 2 subagent in parallel. ...each subagent
is working on EXACTLY ONE task.
```

C3 · coordinator.md — routing constraint

The coordinator behavior is defined in `subagent_prompts/coordinator.md` (485 lines). The key block (`coordinator.md:42–55`):

```
YOU ARE FORBIDDEN FROM DOING PROOF WORK DIRECTLY.

ALL work MUST go through Task tool subagents.
This is NON-NEGOTIABLE. No exceptions. Ever.

WHY: Context explosion. Your context will blow up
     if you try to do work yourself.
     Subagents have isolated context that gets
     discarded after they finish.
```

The coordinator is a router only. Its context holds the checklist, run state, and dispatch decisions — not proof bodies. Without this constraint the context grows with every proof attempt and the agent runs out of space within one round.

C3 · Informal-First protocol + 2^n refine

Informal-First
(coordinator.md:184–211)

- Every **new** sorry: spawn `informal_agent` first
- Wait for `informal_<lemma>.md` v1 to exist
- Then spawn `proof_agent` with the file reference
- Do not skip the informal step on a first attempt

2^n refine checkpoints
(proof_agent.md:108–118)

At attempts 2, 4, 8, 16, 32…

- Proof agent returns CHECKPOINT
- Coordinator re-spawns informal agent with failed attempts + diagnostics
- Informal agent asks Gemini to revise strategy; writes v{N+1}
- Proof agent re-reads updated file

Only mark **blocked** if `attempts` $\geq$ `30` (coordinator.md:333). Below 30: keep as *in_progress* and continue.

C3 · Prompt design worth copying

1. **Router-only coordinator.** The prompt forbids doing proof work directly — all work goes through Task subagents with isolated context, so the long-lived agent's context stays small.
2. **Informal-first.** Every new `sorry` gets a natural-language strategy file before any Lean is attempted.
3. 2^n **refine checkpoints.** At attempts 2, 4, 8, 16…the proof agent stops and the strategy is revised — escalate the thinking, not just the retry count.
4. **Fixed tool and tactic order.** Search: `leanexplore` → `loogle` → `leanfinder`. Tactics: `hint` → `grind` → manual. No per-call guessing.
5. **Sentinel protocol.** The agent ends a turn with `END_REASON:COMPLETE/LIMIT`; the runner — not the model — calls it done, and only after a kernel check.

C4 · Round loop with bounded budget

```
PAT_REASON = re.compile(
  r"(?m)^\s*END_REASON:(LIMIT|COMPLETE|"
  r"SELECTED_TARGET_COMPLETE)\s*$", re.I)
```

The runner reads a sentinel line written by Claude to decide when a round ends.

Sentinel	**Runner action**
`COMPLETE`	Run `on_complete_callback` (kernel check all files); break on pass, resend on fail
`LIMIT`	Continue same session; after 2 consecutive LIMITs reset to fresh session
`SELECTED_TARGET_COMPLETE`	Continue same session, move to next target
(none)	Defensive: resend full prompt as new session

Default `allow_sorry=False`: COMPLETE is honored only when both errors and sorries are gone.

Budget: `max_rounds=10` (outer), up to 30+ inner attempts per round.

C4 · on_complete_callback — kernel check on every COMPLETE

```
def on_complete_callback() -> bool:
    lean_files = find_lean_files(check_path)
    results = check_lean_files_parallel(lean_files)
    if task.allow_sorry:
        errors = [f for f,e,_,_,_ in results if e]
    else:
        errors = [f for f,e,s,_,_ in results if e or s]
    return len(errors) == 0
```

Three kernel checkpoints per task:

1. **Proof agent**: `lean_check.py` after every meaningful edit
2. **Runner**: `check_lean_files_parallel` on every COMPLETE signal
3. **SafeVerify** (optional): independent kernel replay of original vs. submission

There is no separate verifier service. `lake env lean` is the verifier.

C5 · The three tools the proof_agent reaches for

No MCP. Every tool is a Python CLI under `skills/cli/`; the agent invokes it from `Bash`, reads stdout, logs to per-task `CLI_LOG_PATH`.

The agent is told to reach for these three in this order (`proof_agent.md:289–308`, "High-Frequency Tools (use liberally!)"):

```
python skills/cli/lean_check.py <file>      # compile + diagnostics
python skills/cli/leanexplore.py "QUERY"    # semantic Mathlib search
python skills/cli/discussion_partner.py     # ask Gemini/GPT for strategy
```

- `lean_check.py` — `lake env lean <file>` → structured JSON; the only thing the agent trusts.
- `leanexplore.py` — `leandex.projectnumina.ai`; the primary search, "invoked as `python skills/cli/leanexplore.py QUERY`" (`main_entry.md:12`).
- `discussion_partner.py` — free-form chat (Gemini or GPT) when stuck.

C5 · Reference: the full CLI surface (11 scripts)

Skill group	Scripts (under `skills/cli/`)
Verification	`lean_check.py`; `axle.py verify-proof / disprove`
Search	`leanexplore.py` (PRIMARY); `loogle.py`, `leanfinder.py`, `leansearch.py`, `state_search.py`, `hammer_premise.py`
LLM	`informal_prover.py` (3-model panel), `discussion_partner.py`, `code_golf.py`
Code-transform	`axle.py repair-proofs / simplify-theorems / sorry2lemma / extract-theorems`

Every call writes one line to `cli.log` (per-task `CLI_LOG_PATH`); `cli_stats.json` rolls up called/ok/fail per tool.

C5 ·Skill grouping in SKILL.md files

Skill group	Scripts covered
`skills/search/SKILL.md`	leanexplore, loogle, leanfinder, leansearch, state-search, hammer-premise
`skills/verification/SKILL.md`	lean-check, verify-proof, disprove
`skills/code-transform/SKILL.md`	repair-proofs, simplify-theorems, sorry2lemma, extract-theorems
`skills/llm/SKILL.md`	informal-prover, discussion-partner, code-golf
`skills/sorrifier/SKILL.md`	workflow: lean_check + axle sorry2lemma to isolate failing proof steps

Search priority (enforced in all three major prompts): `leanexplore` → `loogle` → `leanfinder` / `leansearch`. Tactic priority: `hint` → `grind` → manual.

C5 · Code-transform: what the axle commands do

axle command	What it does
`repair-proofs`	Auto-fix a broken proof using configurable repair strategies — tried before manual editing
`simplify-theorems`	Remove unused tactics and bindings from a verified proof — a final cleanup step
`sorry2lemma`	Lift each `sorry` sub-goal into a standalone named lemma, so the holes can be attacked independently
`extract-theorems`	Split a Lean file into structured per-theorem records — for analysis or understanding file structure

All four run through the external project-numina `axle` tool (needs `AXLE_API_KEY`), wrapped by `skills/cli/axle.py`, which just spawns the subprocess and logs one line. The `sorrifier` workflow chains `lean_check` + `sorry2lemma` to isolate a failing proof step.

C5 · Code-transform: repair strategies and sorry isolation

`repair-proofs` chooses from named repair strategies (`--repairs`):

strategy	effect
`remove_extraneous_tactics`	strip tactics left unreachable after the proof already closed
`apply_terminal_tactics`	replace `sorry` with a terminal tactic — default `grind`, or `--terminal-tactics grind,aesop,simp,rfl`
`replace_unsafe_tactics`	convert unsafe tactics (e.g. `native_decide`) to safer alternatives

`sorry2lemma --reconstruct-callsite` replaces each `sorry` with a *call* to the lemma extracted from it: the main proof keeps its structure while every sub-goal becomes an independent, separately-provable lemma. This is the move the `sorrifier` workflow is built on.

All axle commands take `--names` / `--indices` to target specific theorems, and are rule-based transforms (only `AXLE_API_KEY`, no model) — unlike the Gemini-backed `code_golf`.

C6 · Leandex semantic Mathlib search

`skills/cli/leanexplore.py` (93 lines) is the primary premise-retrieval tool:

```
def search(query: str, num_results: int = 5) -> None:
    url = "https://leandex.projectnumina.ai/api/v1/search"
    params = {"q": query, "limit": num_results,
              "generate_query": False,
              "analyze_result": False}
    ...  # streams SSE, parses data: JSON, prints declarations
```

Comparison: trained vs. tool-based retrieval

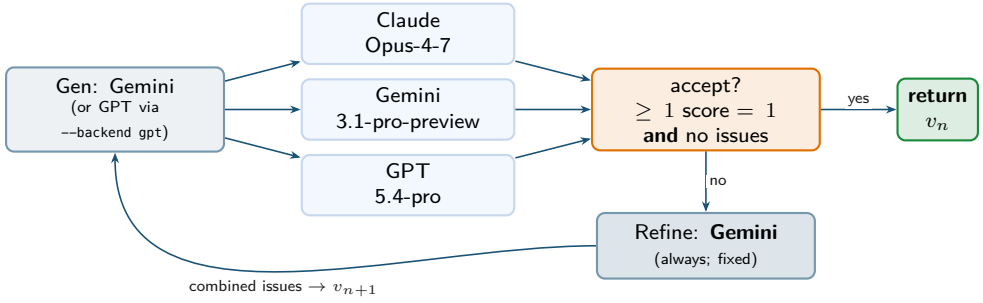
	ReProver (trained)	**Leandex (tool)**
Retrieval	inside one forward pass	separate CLI call by agent
Library	fixed snapshot	whatever Leandex has indexed
Decision	model-internal	agent decides when to retrieve

Both answer "which Mathlib lemma is relevant to this goal?"

In numina-lean-agent, the agent decides when to retrieve; retrieval is a tool call.

C7 · The verifier panel in informal_prover.py

`informal_prover.py` is a **1 generate / 3 verify / 1 refine** loop — asymmetric by design, not a committee:



- accept (`informal_prover.py:272`): `passing_scores >= 1 and not issues` — lenient on PASS, strict on reject.
- refine is always Gemini (`informal_prover.py:317`), even when generator is GPT.
- three families → partly-independent blind spots.

C7 · Informal reasoning stays outside Lean

The three informal channels

- `informal_<lemma>.md` — versioned per-lemma file, written by informal agent, read by proof agent before every attempt
- `informal_prover.py` — one-shot generator + 3-model panel
- `discussion_partner.py` — free-form Gemini/GPT chat when stuck

What the informal text does not do

- It does not enter the `.lean` file
- Agent comments use `/- (by claude) ... -/`
- `informal_<lemma>.md` is not on the Lean compile path
- Panel verdict applies to the draft, not to the Lean kernel

Compare Rethlas (Week 10), where the informal blueprint is itself verified. Here the informal text is scaffolding for the proof agent; the kernel verdict is the record.

C8 · Per-task isolation — StatementTracker + audit trail

Directory layout (`results/<run_id>/<task_id>/`):

`cli.log`	every CLI skill call (start / ok / fail)
`cli_stats.json`	rolled-up per-tool counts
`round_N.json`	end_reason, duration, token cost, statement changes
`claude_raw/round_N.jsonl`	full Claude stream-json transcript
`result.json`	final summary including USD cost and SafeVerify verdict

StatementTracker (`scripts/statement_tracker.py`):

- Snapshots all `theorem`/`lemma` signatures at task start
- After each round: diff added / modified / removed
- Modified or removed: `restore_initial_statements` before next round
- The proof body may change; the theorem statement may not

Every state transition runs through `lake env lean`. The `.lean` file is kernel-clean or it is not; `round_N.json` records which.

C10 · Limitations: scope of the guarantee

Intended statement, not just a valid proof The kernel guarantees the proof body is correct and depends only on the three axioms (`propext`, `Classical.choice`, `Quot.sound`). It does *not* guarantee the agent proved the *intended* statement: "do not edit the statement" is enforced by the runner's diff-and-restore, not by the kernel — a different theorem with the same name still compiles. SafeVerify (kernel-level original-vs-submission replay) closes part of this gap.

StatementTracker race window A statement modification *within* a round is not blocked when it happens; the runner catches it only at round end and restores for the next round, so that round's work is wasted.

Verifier-panel disagreement not surfaced `informal_prover.py` accepts iff at least one model scores 1 and no model reports an issue, so a 1-vs-0.5 split with no issue is accepted. This affects only the natural-language draft, never the kernel verdict; disagreement is logged, not shown to the proof agent.

C10 · Limitations: cost, environment, orchestration

Wall-clock and token cost Every round re-sends a long prompt (`main_entry` + five subagent prompts + skill files) plus up to three external models in `informal_prover.py`; a hard Putnam problem burns tens of thousands of tokens $\times$ up to 10 rounds. Cost is in `result.json`; no published per-problem number.

`max_rounds` ceiling vs 30-attempt floor One round must fit ≥ 30 inner attempts plus search and informal-refine; a hard problem can spend the whole budget on inner search and return only a LIMIT-then-COMPLETE envelope.

No cross-task memory Each `.lean` file starts fresh; a lemma won in A1 does not help A2. No shared cache, no learned-hint database.

Environment and discovery `lean_check.py` walks up for `lean-toolchain` — outside a buildable project every check errors. And the agent only knows a tool exists if it reads the SKILL.md files: enforced by prompt repetition, not at code level.

C11 · Contrast with Archon

	numina-lean-agent	Archon
Unit of work	Single theorem / `.lean` file	Entire Lean project, multi-file
Orchestration	All subagents inside ONE Claude Code session (Task tool); Python only counts rounds	Runner enforces plan / prover / review separation; persistent state files
Memory	No cross-task memory; each file is fresh	State maintained across sessions
Informal channel	External Gemini/GPT/Claude panel; versioned `.md` files	Plan documents + blueprint chapter
Loop control	Soft outer (max_rounds=10); strict inner ($\geq$30 before blocked)	Patient (90+ min before escalating); stops on closed sorry or blocker

Shared: Claude Code runner, lean-lsp tooling, kernel as trust anchor, per-task isolation

numina-lean-agent targets one sorry in one file per task. Archon targets a multi-file Lean project across many sessions.

Part D · Archon

Archon: system and provenance

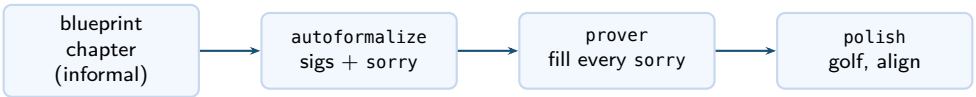
From the `README.md:12–14`:

- Project-scale Lean 4 formalization — multi-file repos, not single benchmarks.
- Plan agent = strategy; prover agents write Lean (analysis separate from execution).
- Three phases: scaffold → prove → polish.

Provenance: `github.com/frenzymath/Archon`
Apache-2.0 · commit 68d1780 (v0.2.0)
Blog: `frenzymath.com/blog/archon-firstproof/`
Paper: arXiv 2604.03789

From blueprint to Lean

Archon turns one informal blueprint chapter into one finished Lean file by advancing through three *stages*:



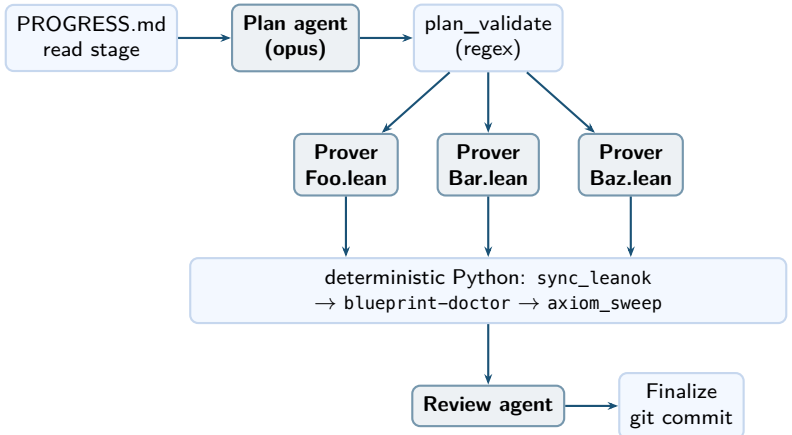
`PROGRESS.md` — the loop's state file:

- Current stage: `autoformalize` → `prover` → `polish` → `COMPLETE`.
- This iteration's file objectives.
- Plan agent owns it; advances the stage only once the current one is verified done.
- `\leanok` markers sync back from compilation, so the blueprint tracks formal progress.

Each stage runs many plan → prover → review iterations (next slide).

One iteration: plan, prover, review

One iteration of `archon loop`, within the current stage:



Bold = LLM agent (the three phases). Plain = deterministic Python / state. Full command: `archon loop /path/to/your-lean-project`

Separation of concerns by phase

scaffold (autoformalize) Read blueprint chapter. Build file structure: modules, theorem signatures, `sorry` placeholders. Success: file compiles with sorries. Agent must not write proofs at this stage.

prove Fill every `sorry` for one `.lean` file. Success: no sorry, zero axioms introduced, file compiles cleanly.

polish Golf proofs, refactor to Mathlib, extract reusable helpers. Success: still compiles, no sorry, no axioms, proofs shorter.

Three patterns banned in *prove* (prover–prover.md:39–45):

- Reflexive-iso: `X ≅ Y` replaced by `Nonempty (X ≅ X) := ⟨Iso.refl _⟩`.
- `Classical.choice (α := X) ⟨witness⟩` — unreachable through `unfold`.
- Empty `proof_wanted` — decl never enters env; `\lean{...}` breaks.

Litmus (prover–prover.md:45): `unfold` the decl — if it stops at `Classical.choice` / `Iso.refl _` / nothing, ship typed `sorry`.

Stage data contracts

Each stage is the same prover agent under a different contract: what it reads, what it must produce, and what it may not touch.

	`autoformalize`	`prover`	`polish`
In	blueprint chapter (informal proof)	file: signatures + `sorry`	file: 0 `sorry`, compiling
Out	file: signatures + `sorry`, compiles	every `sorry` filled, 0 new axioms	golfed / Mathlib-aligned, helpers extracted
Must not	write any proof; change a protected signature	weaken the type (the 3 banned patterns)	change any statement or signature; add `sorry`/axiom

Acceptance is the kernel: `autoformalize` = compiles with `sorry`; `prover` = 0 `sorry` and 0 new axioms; `polish` = still 0, statements unchanged, proofs shorter. Signatures are fixed in `autoformalize` and frozen by `archon-protected.yaml` — only bodies change, so each stage's output is the next stage's input.

golf and align: the polish stage's two tools

Both require the proof to *already compile*, and neither may change a statement, add an axiom, or create a commit.

`/lean4:golf` — tactic-level: shorten / clean a compiling proof. The prompt's own "instant wins":

before	after
`ext x; rfl`	`rfl`
`apply f; exact h`	`exact f h`
`constructor; exact h1; exact h2`	`exact ⟨h1, h2⟩`
`by exact t`	`t`

`/lean4:refactor` (Mathlib-align) — strategy-level: replace a hand-rolled argument with the matching Mathlib lemma, or extract a reusable helper. E.g. a hand-written injectivity proof → `PrimeSpectrum.comap_injective`.

Order: align first (pick the better approach), then golf (tighten the tactics).

How a skill prompt is written: `golf.md`

A skill command (e.g. `golf.md`) is a Markdown file: YAML frontmatter + a body of Usage / Inputs / Actions / Policy / Safety.

```
---
name: golf
description: Improve Lean proofs for directness, clarity,
             performance, and brevity
user_invocable: true
---
# Lean4 Golf  ...  Usage | Inputs | Actions (1–7) | Policy | Safety
```

The work is in the *policy*, not the prose — it stops the agent from "winning" by trading clarity for length:

- **Score:** correctness → directness → clarity → performance → length. Length is a *tiebreaker*, not the goal.
- **Hard reject:** moving up the tactic-complexity ladder (`rfl`/`exact` < `rw`/`apply` < `simp only` < `simpa` < `simp`/`omega`/`grind`) for a 1-line win; removing meaningful names.
- **Safety:** needs a passing build; reverts on failure; never commits.

Plan / Prover / Review —phase agents

Role contract, enforced by the file permissions below:

- Plan decides what / how — never writes a tactic.
- Prover writes tactics for one file — never edits the blueprint or PROGRESS.
- Review writes the journal + semantic markers — never edits `.lean`.

File permissions (`CLAUDE.md` shipped with each project):

File	Plan	Prover	Review
`PROGRESS.md`	RW	R	R
`STRATEGY.md`	RW	–	–
`task_pending.md`	RW	R	R
`task_results/<f>.md`	R	W (own)	R
`proof-journal/`	R	–	W
`.lean` files	–	W (own)	–
`blueprint/src/*.tex`	W	–	W
`archon-protected.yaml`	R	R	R

Phase prompts (`.archon-src/prompts/`, lines): `plan` 390; `prover-autoformalize` 80; `prover-prover` 130; `prover-polish` 85; `review` 290.

Phase agents: consumes and produces

Agent	Consumes	Produces
Plan (opus)	PROGRESS.md stage; blueprint; references; critic + blueprint-doctor JSON; USER_HINTS	STRATEGY.md; blueprint chapters; prover dispatch — never a tactic
Prover (per file, per lane)	one file's blueprint chapter + its `.lean` with sorries + lean-lsp	filled proofs for that ONE file (sorry=0, no new axioms) — never edits blueprint
Review	the whole-project Lean + blueprint	proof-journal entries; semantic markers (`\mathlibok`, `\lean{}`) — never edits `.lean`

Each role is confined by file permissions (previous slide). Deterministic Python — `plan_validate`, `sync_leanok`, `blueprint-doctor`, `axiom_sweep` — runs between the agents.

The 9 subagents (1): dispatched in the plan phase

Each is a Markdown descriptor whose `write_domain` is enforced at dispatch — a read-only subagent literally cannot edit the project; it writes only its `task_results/` report.

Subagent	Access	Role
`blueprint-reviewer`	R/O, mandatory	whole-blueprint audit; the HARD GATE — a file may feed a prover only if its chapter is complete *and* correct
`strategy-critic`	R/O, mandatory	fresh-context critic of `STRATEGY.md`; flags sunk cost; SOUND / CHALLENGE / REJECT
`progress-critic`	R/O, mandatory	per-route convergence audit: CONVERGING / CHURNING / STUCK / UNCLEAR
`mathlib-analogist`	R/O	align a design to Mathlib's idiom, or search a distant domain for the same shape
`blueprint-writer`	writes 1 chapter	drafts / updates a single blueprint chapter from references
`reference-retriever`	writes `references/`	downloads original source files (PDF / TeX) into `references/`
`refactor`	writes **	the only project-wide writer; structural Lean edits; inserts `sorry` at broken sites, never fills

The 9 subagents (2): review phase, and the descriptor schema

Subagent	Access	Role
`lean-auditor`	R/O, mandatory	whole-project Lean audit, no strategy bias; treats "temporary wrong def" comments as red flags
`lean-vs-blueprint-checker`	R/O, mandatory	per-file, both ways: does the Lean follow the chapter, and is the chapter detailed enough

Every descriptor is YAML frontmatter + a prompt body:

- `write_domain` — the only paths it (or a child) may write, enforced at dispatch time, not by the prompt.
- `read_only`, `can_spawn`, `default_enabled`.
- `mandatory: [plan] / [review]` — an audit warning fires if that phase skips it with no recorded rationale.

Stopping the loop from thrashing (1/2): fresh-context critics

Failure mode of a long autonomous loop: it spirals — "+3 helpers each iter, residual unchanged." The plan agent, hundreds of iters deep, is the worst-positioned judge of whether the route still works.

Two **mandatory, read-only** critics, each handed a *deliberately narrow* context so it judges cold:

- `strategy-critic`: sees only `STRATEGY.md` + references index + blueprint summary + the goal — not the iter history or journals. Verdict: SOUND / CHALLENGE / REJECT.
- `progress-critic`: sees per-route signals (sorry count, helpers added, recurring blockers). Verdict per route: CONVERGING / CHURNING / STUCK / UNCLEAR.

The doer's accumulated context *is* the bias — a reader who never saw the 40 failed iters can say "this route isn't working."

Stopping the loop from thrashing (2/2): the stuck-protocol gate

The verdict feeds a gate the plan agent must obey (`plan.md:222–228`):

- **CHURNING** — STOP. Do not add more helpers; run the named corrective.
- **STUCK** — STOP. Pivot the route.
- Disagree? Rebut *in writing* in `iter-NNN/plan.md`, citing the signals — silent overrides are forbidden.

Subtle: churning $\neq$ unsoundness.

- A `sorry` that won't close may be a *false* statement (missing hypothesis, wrong quantifier), not a hard gap.
- Before burning >1 iter, try to *disprove* it (small / degenerate models, ask for a counterexample) — if one turns up, fix the statement, don't formalize it.

Multi-lane providers

Supported lanes (`MULTILANE.md`):

Provider	value	API key env var
Anthropic (Claude Code)	`anthropic`	(Claude Code login)
Moonshot (Kimi)	`moonshot`	`MOONSHOT_API_KEY`
DeepSeek	`deepseek`	`DEEPSEEK_API_KEY`

- Each lane = one `(provider, model)` in its own worktree under `.archon/lanes/<id>/`
- Lanes run on every file in `## Current Objectives` in parallel
- **First clean wins**: fully-clearing lane starts a 600s grace timer; partial wins don't preempt
- **Per-file merge agent**: when multiple lanes finish cleanly, pick best proof per declaration
- Provider redirected via env vars —same `claude` binary, no settings file on disk

Different providers fail differently; parallel execution gets the union of strengths.

Harness router: an engine per responsibility

Open proposal — frenzymath/Archon PR #25, off by default: route each responsibility to a named *harness*.

- **harness = engine**: `CLI + model + effort + prompt-variant + MCP + auth`.
- Routable units: plan, prover, review, each subagent, each multilane lane.
- `AgentRunner` protocol + `build_runner` factory; additive config, defaults unchanged (no `harnesses.*` override → the normal `ClaudeAgent`).
- First non-Anthropic engine: **Codex** (`agents/codex.py` + `codex.md` variant). Subagent gets a `harness:` key; a lane gets a `harness` field.

vs multi-lane *providers*: a provider is the same `claude` binary at another endpoint; a harness is a different engine / CLI.

Why add a Codex engine: cost and faithfulness

Two reasons the PR gives:

- **Cost.** The loop is headless `claude -p`. Anthropic's 2026-05-14 change (in effect 2026-06-15) meters programmatic use on a separate API-rate pool, no rollover — long unattended runs cost more.
- **Faithfulness.** Codex proves Lean well, but is more willing than Opus 4.8 to pass the checker *without* honestly proving the goal; its cheats evade text matching.

Consequence:

- Routing to Codex wants a *kernel-level* valve — `#print axioms` + statement-equivalence, not a textual check.
- Same failure class as the banned patterns and hidden `sorry` — but now the engine itself is the adversary.

Inner-git at `.archon/git-dir/`

Archon keeps a **private git repo** nested inside `.archon/git-dir`. Same working tree as the outer (mathematician's) repo; separate `GIT_DIR`.

Per-phase commit messages:

```
archon[042/plan]:         stage=prover (137s)
archon[042/prover]:       all-provers sorry=58 (1284s)
archon[042/marker-sync]: 6 added, 0 removed
archon[042/review]:       journal session (412s)
archon[042/finalize]:     iteration 42 complete
```

A second git repo is used because:

- Committing every phase to the outer repo floods the mathematician's history
- Not committing at all means no time machine, no recovery from bad iters

Rewinding with `archon branch`:

```
archon branch try-route-B --from archon[033/plan]
archon loop
```

Working tree rewinds to before a bad refactor; outer git untouched. The original timeline is preserved on the previous inner branch.

Frozen signatures via `archon-protected.yaml`

archon-protected.yaml prevents agents from weakening theorem statements to make them trivially provable.

Every agent reads this file before proposing edits; none may modify a listed signature.

```
# archon-protected.yaml
# Declarations whose signatures must not be modified.
# The mathematician owns these; agents are read-only.

# Example:
#   DecouplingMomentCurve/Main.lean:
#     - main_decoupling_theorem
#     - decoupling_torsion_curve
```

Failure mode this prevents:

- Prover weakens hypotheses after 30 minutes of failure
- Weakened theorem compiles; dashboard shows `sorry=0`
- But the mathematician's claim has been silently replaced

Exception: the `refactor` subagent may move a protected declaration to another file (name + signature verbatim) and must update the path key in the YAML.

Why frozen signatures matter

A clean compile verifies the *proof*, not the *statement* — an AI proof can pass the kernel yet not state what you intended ("semantic hallucination").

A public "machine-verified `P = NP`" repo (`N_PEqualsNP.lean`, 337 lines, cites Cook / Karp / Tardos):

- Headline: `def P_equals_NP : Prop := True` — the "theorem" *is* `True`, proved by `trivial`.
- The hard steps (Cook, Karp) are `axioms` — assumed, not proved.
- Compiles, 0 `sorry` — convincing to anyone who never reads line 99.

Guards (human-owned):

- Freeze the headline signature (`archon-protected.yaml`) — the claim can't be silently redefined as `True`.
- Read the body for `axioms`; run `#print axioms` (catches assumed axioms / `sorryAx`).

Persistent user guidance

Three channels (README.md:183–191):

Mechanism	When to use	Lifetime
USER_HINTS.md	Mid-run corrections	One-shot (cleared after plan)
/- USER: ... -/	File-specific hints	Persistent (in .lean)
Prompts and skills	Behavior change	Permanent (every iter)

How USER_HINTS.md is handled:

1. Plan agent does NOT read or clear the file —the loop does it deterministically
2. File is read before the plan starts; cleared only after plan succeeds (crash preserves hints)
3. Cleared state = template; no "yesterday's hint" drift

From the plan prompt —hints as async override:

```
You decide; you never wait. The loop is autonomous — it often runs
unattended overnight. Every strategy-level choice is YOURS to make.
The user changes the plan by adding hints to USER_HINTS.md *if and
when* they disagree — treat that as an asynchronous override you'll
honour the next iter it appears, never a gate you wait on.
```

Blueprint-driven formalization

LeanBlueprint (PatrickMassot) installed by `archon setup`; configured by `archon init`.

Blueprint block with citation discipline:

```
\begin{theorem}[name_for_humans]
  \label{thm:some_label}
  \lean{namespace.theorem_name}
  \uses{def:related_definition, lem:supporting_lemma}
  % SOURCE: [Hartshorne], III.5.1, p. 174
  %         (read from references/hartshorne.pdf)
  % SOURCE QUOTE: "A morphism f: X -> Y of schemes ..."
  Informal statement, in the project's notation.
\end{theorem}
```

Marker ownership (split agent vs Python):

- `\leanok` → `sync_leanok` phase. *Mechanical*: `sorry_analyzer` + lake set it iff the decl compiles with 0 `sorry` — a fact, not the LLM's call.
- `\mathlibok` → review agent. *Semantic*: a judgment that the decl just re-exports a Mathlib lemma (nothing left to prove).
- `\lean{...}` → review agent (after a rename).

Citation discipline (1/2): quote the source verbatim

The blueprint is the *informal* side — the kernel can't catch a fabricated "Hartshorne III.5.1 says…" written from memory.

So every externally-sourced block carries three things (`plan.md:113–131`):

- `% SOURCE:` — the pointer (book, theorem no., page) *and* the local file actually opened, `(read from references/hartshorne.pdf)` — the real PDF/TeX, not an index card.
- `% SOURCE QUOTE:` — the statement *verbatim*, in the source's original language (French for Bourbaki/EGA, German for Grothendieck, English for Hartshorne), notation char-by-char, no paraphrase, no translation.
- A visible `Source: …` line so the human sees the citation; proof blocks add `% SOURCE QUOTE PROOF:` before the restated proof.

Citation discipline (2/2): why it's a forcing function

The hard rule (`plan.md:129`): *never cite a source you have not just read locally — from memory is fabrication.*

Why verbatim, original-language quotes actually stop fabrication:

- A fake verbatim French / German quote is far harder to forge convincingly than an English paraphrase.
- `(read from references/<file>)` names a file that must exist — grep-checkable against the shipped PDF.
- No local file? You must fetch it first (`reference-retriever` downloads the PDF/TeX) — you can't quote what isn't there.

Detectable, not merely discouraged — the faithfulness guard on the *informal* side. The kernel guards the proof; the verbatim quote guards the claim it rests on.

Blueprint-doctor

A deterministic structural lint that runs every iteration between prover and review.

Four classes of structural bugs checked:

1. Orphan chapters —`.tex` files not `\input`'d by `content.tex`
2. Broken cross-references —`\ref`/`\uses`/`\cref` pointing at undefined labels
3. Malformed annotations —`\uses{}` with empty argument (crashes plastex with RecursionError)
4. New axiom introductions —`axiom` declarations in project `.lean`

Findings written to `iter-NNN/blueprint-doctor.{md,json}` and inlined into the next plan agent's prompt.

Pattern: deterministic Python checker $\rightarrow$ structured JSON $\rightarrow$ next agent's prompt block.
The agent never has to "remember to check the blueprint."

Prompt and contract design patterns

1. **Role contract by file permission** — enforced, not just requested:
   - Plan never writes a tactic.
   - Prover never edits the blueprint.
   - Review never edits `.lean`.
2. **Banned patterns.** The prove prompt names the cheats that compile but erode the statement (reflexive-iso, `Classical.choice` wrap, empty `proof_wanted`), so the agent does not reach for them.
3. **Frozen signatures.** `archon-protected.yaml` is a veto every agent reads; a protected statement cannot be weakened just to compile.
4. **Hints as async override.** The plan prompt says "you decide; you never wait"; the user steers via `USER_HINTS.md` between iterations, never as a gate.
5. **Checker → JSON → next prompt.** `blueprint-doctor` and the critics write structured findings that are inlined into the next agent's prompt — the agent never has to remember to check.
6. **Cite the source.** Blueprint chapters carry `SOURCE` and `SOURCE QUOTE`, anchoring the formalization to the paper.

The 15 CLI commands

Command	Purpose
`archon setup`	Install deps; one-time
`archon init <proj>`	Init project; write PROGRESS.md
`archon loop <proj>`	Main automation loop
`archon dashboard`	Web dashboard (8080–8099)
`archon doctor`	Verify full setup
`archon prove "<s>"`	Single-statement bootstrap
`archon refactor draft`	Interactive directive writer
`archon refactor run`	Execute refactor directive

Command	Purpose
`archon discuss`	Open Claude Code with Archon context
`archon branch`	Inner-git branch create / switch
`archon log`	Pretty-print inner-git history
`archon version`	CLI + project version
`archon update`	Pull newer release
`archon subagent`	Dispatcher worker (not for users)
`archon migrate`	Migration helpers

Limitations

Cost Multi-lane multiplies cost linearly: 3 lanes $\times$ 10 files $\times$ 100 iters $\approx$ 3000 sessions. Recommended: Opus 4.6.

Complexity $\approx$30 Python modules, 8 prompt files, 9 subagent descriptors, TypeScript dashboard. Must learn the layout to debug a stuck run.

Single-theorem mismatch `archon prove` exists; README is explicit: competition benchmarks are not a target. For one IMO problem, plan/prover/review separation is overhead.

Lean-only Every prover prompt assumes Lean 4 + Mathlib + `archon-lean-lsp`. No Rocq or Isabelle.

Strong-model dependence Subagents are context-discipline-heavy. A model that ignores instructions produces verdicts that are not actually fresh.

Hidden-sorry patterns Three banned patterns detected; `axiom_sweep` catches `sorryAx`. Not exhaustive —every Lean idiom that trivializes a goal opens a new hiding place.

Contrast: Archon vs numina-lean-agent

Dimension	Archon	numina-lean-agent
Target	Multi-file project	Single theorem
Plan/prove sep.	Two agent roles	One agent, one loop
Multi-lane	First-clean-wins; provider mix	Single provider
Inner git	Per-phase commits, `archon` branch	Outer git only
Frozen signatures	`archon-protected.yaml`	`StatementTracker` reverts
User guidance	USER_HINTS + `/- USER: -/` + prompts	Prompt + repo flags
Blueprint	LeanBlueprint mandatory	Optional / absent
Subagent system	9 descriptor-driven helpers	Coordinator + 3 prompt-defined roles
Iteration model	plan→prove→review / iter	One session per attempt
Optimization target	Long-arc convergence	Competition success-rate

Shared: Claude Code + `lean-lsp-mcp` + `lean4-skills` + Bash (`lake build`).

numina = chat about one theorem. Archon = codebase running indefinitely.

Follow-up paper: arXiv 2604.03789

Automated Conjecture Resolution with Formal Verification (frenzymath)

The end-to-end pipeline:

1. Open mathematical conjecture
2. Agent searches for a proof candidate
3. **Archon** formalizes that candidate in Lean 4
4. Lean kernel verifies
5. Result: a formally-checked answer to the conjecture

Archon is step 3. The "first proof" in the blog title refers to a conjecture solved end-to-end with this pipeline.

Open question for this course — can the same machinery work *upstream* of formalization (the candidate-search step), not just inside it?

- plan / prover / review separation
- `USER_HINTS` + `protected.yaml` veto surface
- blueprint-first formalization

That is what arXiv 2604.03789 explores.

Part E · Across the Harnesses

Cross-system concerns —numina-lean-agent vs Archon (1/2)

Concern	numina-lean-agent	Archon
proof structure	`informal_<lemma>.md` + `tmp_<lemma>.lean` per sorry	Three-phase scaffold / prove / polish + blueprint chapters
parallelism	$\leq$2 subagents inside one Claude session	Multi-lane (Anthropic / Moonshot / DeepSeek) per-file, first-clean-wins merge
verification	`lean_check.py` after every edit; optional SafeVerify replay	`lean_check.py` + `lean-lsp-mcp` + `lake build`; `sync_leanok`; `axiom_sweep`; review-agent journal
failure memory	`informal_<lemma>.md` v1$\rightarrow$v2$\rightarrow$… + `cli.log` + StatementTracker history	`task_pending.md`, `proof-journal/`, blueprint-doctor JSON, inner-git per-phase log

Cross-system concerns —numina-lean-agent vs Archon (2/2)

Concern	numina-lean-agent	Archon
persistence	Per-task `results/<task_id>/` dirs; no cross-task memory	`.archon/` (inner git + state + journal + dashboard); persists across sessions
guardrails	Coordinator forbidden from proving; `StatementTracker` restore; sorry-count; `informal_prover` 3-model panel	`archon-protected.yaml`; three-banned-patterns rule; `axiom_sweep`; write-domain glob enforcement; `progress-critic` CHURNING verdict
runtime	One Python runner + Claude Code subprocess + 11 CLI skills	Typer CLI + dispatch slot pool + dashboard + multilane executor + inner-git wrapper

Three design principles that apply across harnesses

> *"The kernel is the trust anchor. Whatever the agent does, the kernel either accepts or it does not."*
>
> Leo de Moura, *Proof Assistants in the Age of AI*, Feb 2026

Three principles that hold regardless of which harness is used:

1. Kernel-inspection depth —axiom auditing, statement-preservation tracking, marker discipline
2. Surface to the human —tactic syntax, namespace discipline, Mathlib idiom-density
3. Surface to the agent —LSP introspection, code actions, premise search (`lean-lsp-mcp`)

The C6 decision table applies these as seven questions for seven shapes of task. Harnesses implement answers; the principles remain.

What each harness adds over the numina baseline (1/2)

System	Adds over numina baseline	Bought to solve
numina-lean-agent	*(baseline)* coordinator + 3 subagents; `informal_<lemma>.md` channel; StatementTracker revert; 11 CLI skills	Single Putnam-class theorem, one file, ≈30 min, ≈$50
Archon	Plan/prover/review separation; 9 declarative subagents with `write_domain` gates; `sync_leanok` / `blueprint-doctor` / `axiom_sweep`; inner-git; multi-lane providers; `archon-protected.yaml`; Lean-Blueprint mandatory	Multi-file research project, mathematician-owned signatures, hundreds of iterations

What each harness adds over the numina baseline (2/2)

System	Adds over numina baseline	Bought to solve
LeanAgent	Persistent trained ByT5 retriever updated one epoch per repo; best-first search; dynamic-database JSON; curriculum scheduling	Cross-repo lifelong learning; 155 theorems across 23 Lean 4 repos
Leanstral	Training distribution includes MCP-tool-use trajectories; no harness-side prompt engineering for tool format; Pass@k at inference	1/15 the per-task cost of Sonnet at +2.6 score points on FLTEval
in-editor copilots	*(subtracts)* no agent loop, no scratchpad, no failed-path memory; human is the loop	One tactic suggestion; no autonomous closure

LeanAgent and Leanstral both reduce harness load by moving investment into the model. Neither eliminates the harness; the harness shrinks because the model carries more.

Open questions not resolved by the S06 material (1/2)

Wrong proof, clean compile Archon's three-banned-patterns rule + `axiom_sweep` + `archon-protected.yaml` cover *known* failure modes. Any Lean idiom that trivializes a goal opens a new gap; the kernel will not report it.

Harness helps vs harness hinders Archon's README: weaker models may do worse with Archon's structure than without it. No published threshold says when the harness inverts from help to hindrance.

Cost convergence vs divergence Leanstral: $\approx 1/15$ Sonnet per trained task. numina: $\approx$\$50 per Putnam theorem. Archon: hundreds of USD per research iteration. Cost is divergent because the systems target different task shapes.

Open questions not resolved by the S06 material (2/2)

Cross-task memory No S06 system maintains long-term cross-project memory of lemmas proved before. LeanAgent has cross-repo memory by trained retriever —not the same as a personal Mathlib.

Auditability vs autonomy Autonomy and audit-surface are monotone across these five systems: more autonomous $\Rightarrow$ more state must be recorded for review. Whether a more-autonomous, less-auditable design is sustainable in industrial use is open.

Part F · The Lean-LSP-MCP Layer

F1 · What `lean-lsp-mcp` is

All Lean agents in this lecture reach Lean through this one shared component.

- Repository: `github.com/oOo0oOo/lean-lsp-mcp` (v0.26.2, MIT)
- Runtime: Python MCP server wrapping Lean 4's LSP via `leanclient`
- It runs `lake serve` in a Lean project and re-exports LSP request/response shapes as flat MCP tool calls
- **22 MCP tools** —the same set regardless of which agent connects
- Archon, numina-lean-agent, Leanstral, Claude Code, Cursor, VS Code MCP all attach to `uvx lean-lsp-mcp` and see the same tool surface
- Stated in `instructions.py`:

```
This MCP does NOT edit files.  Use other tools for editing.
```

Reading proof state is the server's responsibility; file edits belong to the agent.

F2 · Tool surface —22 MCP tools (1/2)

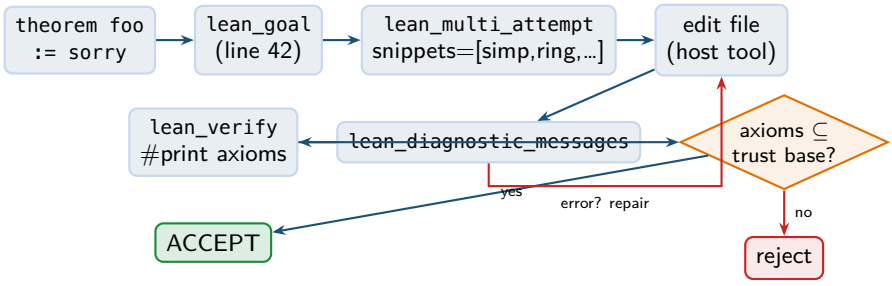
Tool	Purpose
Proof-state inspection	
`lean_goal`	proof state at a position (goals_before / goals_after)
`lean_term_goal`	expected type at a term position
`lean_hover_info`	type signature + docstring at an identifier
`lean_diagnostic_messages`	compiler errors / warnings / infos for a file
`lean_completions`	IDE autocomplete on incomplete code
`lean_file_outline`	token-efficient skeleton: imports + decl signatures
`lean_declaration_file`	source file where a symbol is declared
`lean_references`	all references to a symbol
`lean_code_actions`	resolved LSP actions (captures `Try this:` from simp?)
`lean_get_widgets`	raw user-widget panel data at a position
`lean_get_widget_source`	JavaScript source of a widget by hash
Test without editing	
`lean_multi_attempt`	try candidate tactics; returns goal + diagnostics per snippet
`lean_run_code`	compile a self-contained snippet

F2 · Tool surface —22 MCP tools (2/2)

Tool	Purpose
Theorem-level verification	
`lean_verify`	`#print axioms`; parse axiom list; rg-scan for sorry
`lean_profile_proof`	per-line tactic timing
`lean_build`	`lake` cache + build, then restart LSP
Search —six backends, rate-limited unless self-hosted	
`lean_local_search`	ripgrep over `.lake/packages` (no rate limit)
`lean_leansearch`	leansearch.net (3/30s) —natural-language Mathlib search
`lean_loogle`	loogle.lean-lang.org (3/30s) —type-pattern search
`lean_leanfinder`	Hugging Face endpoint (10/30s) —semantic by meaning
`lean_state_search`	premise-search.com (6/30s) —goal → closing lemmas
`lean_hammer_premise`	Lean Hammer server (6/30s) —premises for simp/aesop

These 22 tools are the complete interface between the agent and Lean.

F3 · Proof loop from the agent's perspective



Trust base: `propext`, `Classical.choice`, `Quot.sound` only.

F4 · Transports: stdio / streamable-http / sse

Standard entry for any client (Claude Code, Cursor, VS Code, Mistral Vibe):

```
{ "command": "uvx", "args": ["lean-lsp-mcp"] }
```

stdio (default) JSON-RPC frames on stdin/stdout. Project root inferred per-call from `file_path`. An agent can move between Lean projects in one session.

streamable-http Single-project HTTP on `127.0.0.1:8000/mcp`. `LEAN_PROJECT_PATH` must be set at startup; project switching rejected. Bearer-token auth via `LEAN_LSP_MCP_TOKEN`.

sse Server-Sent Events —legacy MCP transport. Same single-project constraint as streamable-http.

Optional accelerators: `--repl` (fast `lean_multi_attempt`); `--loogle-local` (bypass rate limit); `ripgrep` (enables `lean_local_search`).

F5 · What `lean-lsp-mcp` does not do

The server answers Lean queries; everything else is the agent's job.

No proof generation No LLM, no tactic synthesis, no goal-conditioned policy. Suggestions come from the elaborator, a hammer service, or a search index —never from this process.

No file edits Stated in `instructions.py:3`. `lean_multi_attempt` and `lean_run_code` use throwaway in-memory edits; they never persist a change.

No memory between calls Caches LSP clients per project and rate-limit counters per search backend. No conversation history, plan state, or prior attempts. Memory lives in the agent harness.

No Mathlib search of its own Delegates to leansearch.net, loogle.lean-lang.org, premise-search.com, Hugging Face, Lean Hammer. Thin caller with response normalization.

No orchestration Planning, retry budgets, multi-attempt voting, and acceptance criteria all sit in the agent loop on top.