Building Proof Agents

LLMs, natural proofs, Lean, retrieval, search, and repair

AI for Mathematics — Week 5

Ma Jiajun · May 5, 2026

Why a Proof Agent?

An LLM can write plausible proof text. A proof agent controls a loop that proposes, checks, retrieves, repairs, and records evidence.

LLM

Good at informal strategy, language, analogy, and local code proposals.

Lean

Good at exact proof checking and exposing proof states and errors.

Agent

The harness that makes model, library, verifier, search, and memory cooperate.

Running example: **the sum of the first n odd numbers is n^2.**

How an LLM Writes an Answer

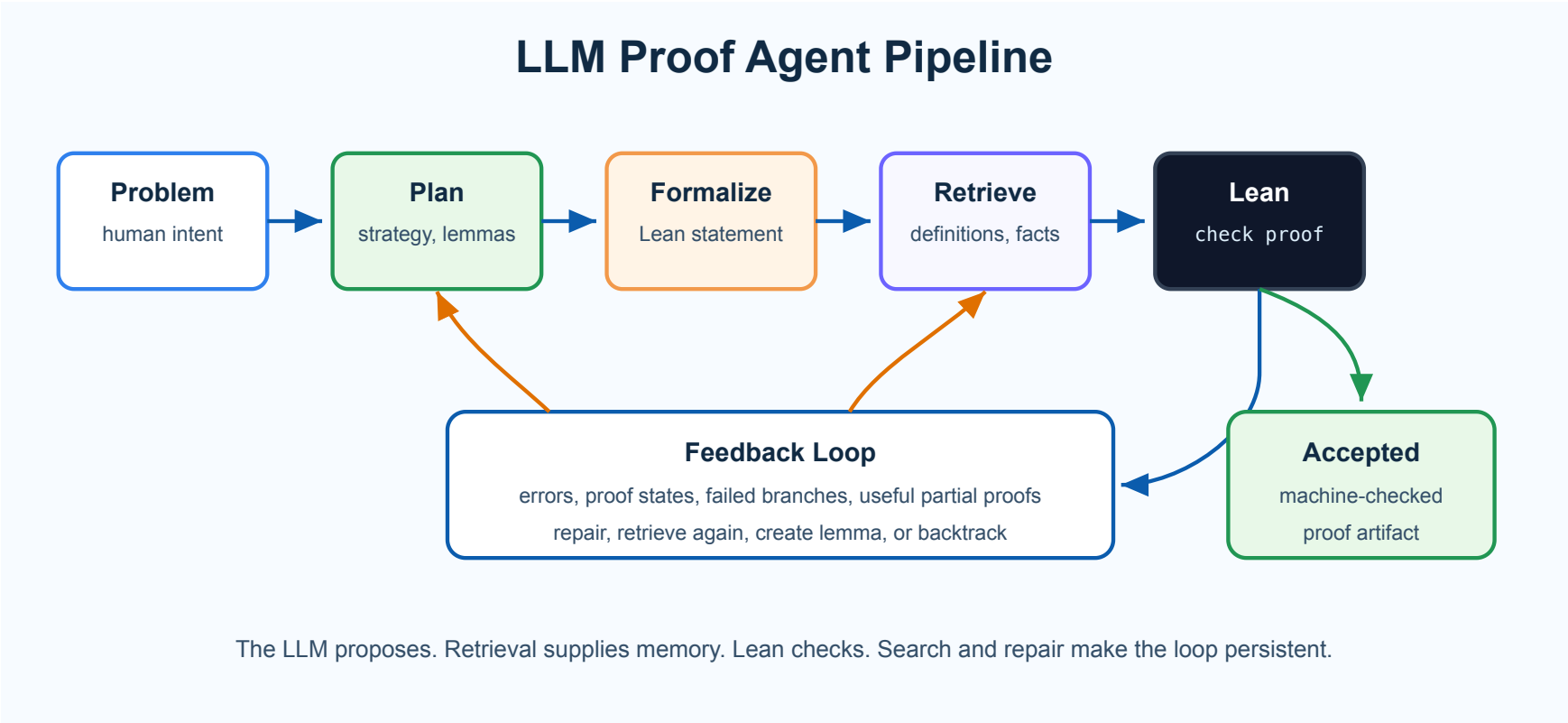
read the problem → write a small next piece → append it to the answer → repeat

```
Prompt:
  Prove that the first n odd numbers sum to n^2.

Model begins:
  We prove by induction on n. For n+1, add the next odd number ...
```

The model is not born knowing the proof. It has learned which next pieces of text or code usually fit this kind of problem.

The small pieces are called tokens, but the important idea is simpler: the model keeps extending the answer one piece at a time.

The Main Pipeline



Locally generated explanatory diagram. Source anchors: LeanDojo/ReProver, COPRA, Prover Agent.

One Problem, Three Artifacts

Natural proof

Use terms 2k+1 for k=0,...,n-1. In the step from n to n+1, add 2n+1, so n^2 becomes $(n+1)^2$.

Lean target

```
example (n : Nat) :
(Finset.range n).sum (fun k =>
2*k + 1) = n^2 := by ...
```

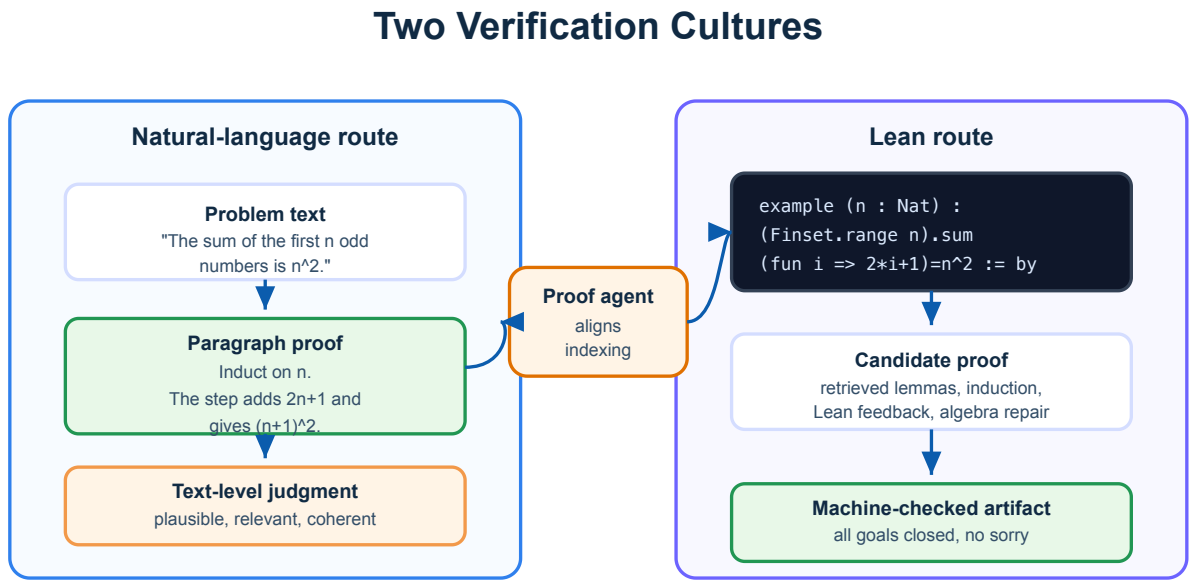
Checked artifact

Lean accepts only when all goals are solved using available definitions, theorems, tactics, and imports.

Fluent proof text is not the same as a checked proof.

Natural-language proof systems such as NaturalProofs and NaturalProver study proof text; Lean-centered systems use proof assistants for final checking.

Natural Proofs and Formal Proofs

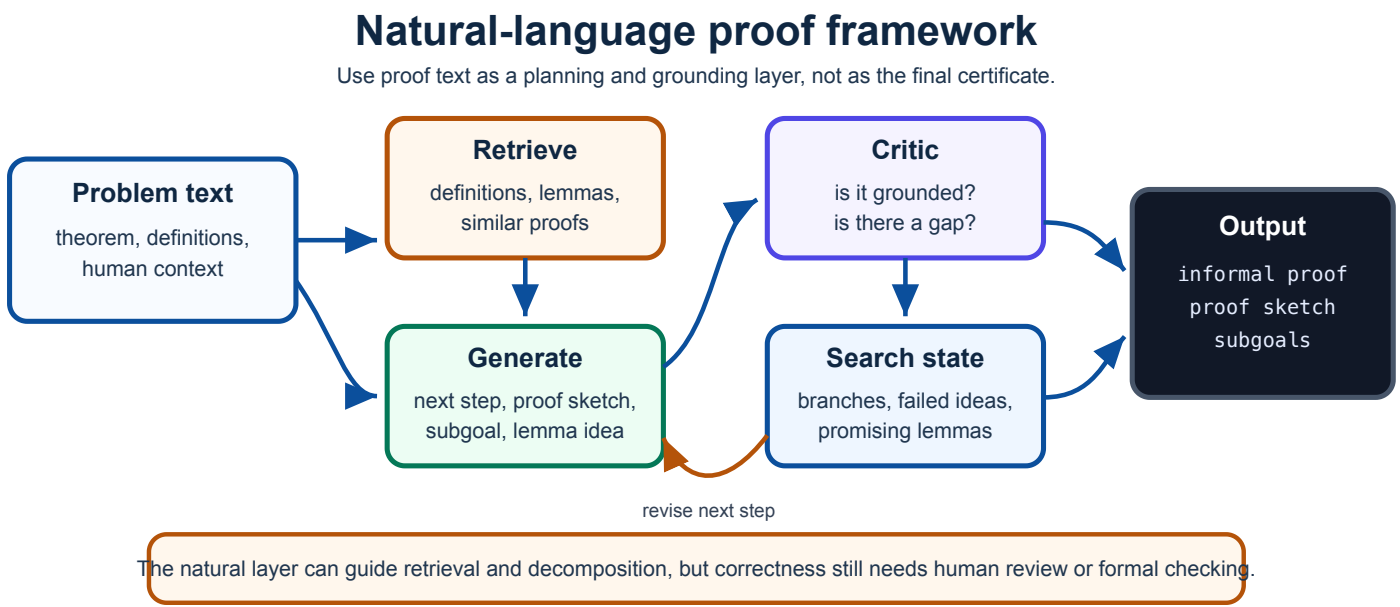


Natural proofs hide indexing; Lean-centered agents must expose Finset.range and let feedback guide repairs.

Locally generated explanatory diagram. Source anchors: NaturalProofs, NaturalProver, LeanDojo/ReProver.

For teaching, keep these two notions separate: a proof paragraph can be mathematically useful before it is formally checkable.

Natural-Language Framework: Three Jobs



Locally generated explanatory diagram. Source anchors: NaturalProofs, NaturalProver, DeepTheorem.

The natural-language layer is where the agent can explore proof ideas cheaply, but it is not where the final correctness certificate lives.

What Natural-Proof Systems Actually Test

Component	Question	Example signal
Reference retrieval	Does the model find useful definitions and lemmas?	cited theorem, similar proof, premise score
Step generation	Can it write the next plausible proof move?	local proof continuation
Grounding	Is the proof tied to supplied references?	required references appear in the argument
Critique	Is there a gap, circular step, or hidden assumption?	human/rubric/LLM judgment
Handoff	Can the sketch guide formal proving?	subgoals, lemma candidates, formalization hints

NaturalProofs, 2021
Dataset and task framing for theorem proving in natural mathematical language, with reference retrieval.

NaturalProver, 2022
Grounded proof generation: the proof should use the supplied theorem references, not just sound plausible.

DeepTheorem, 2025
Uses informal proof quality as a training and evaluation target for mathematical reasoning.

Example: Natural Proof Search

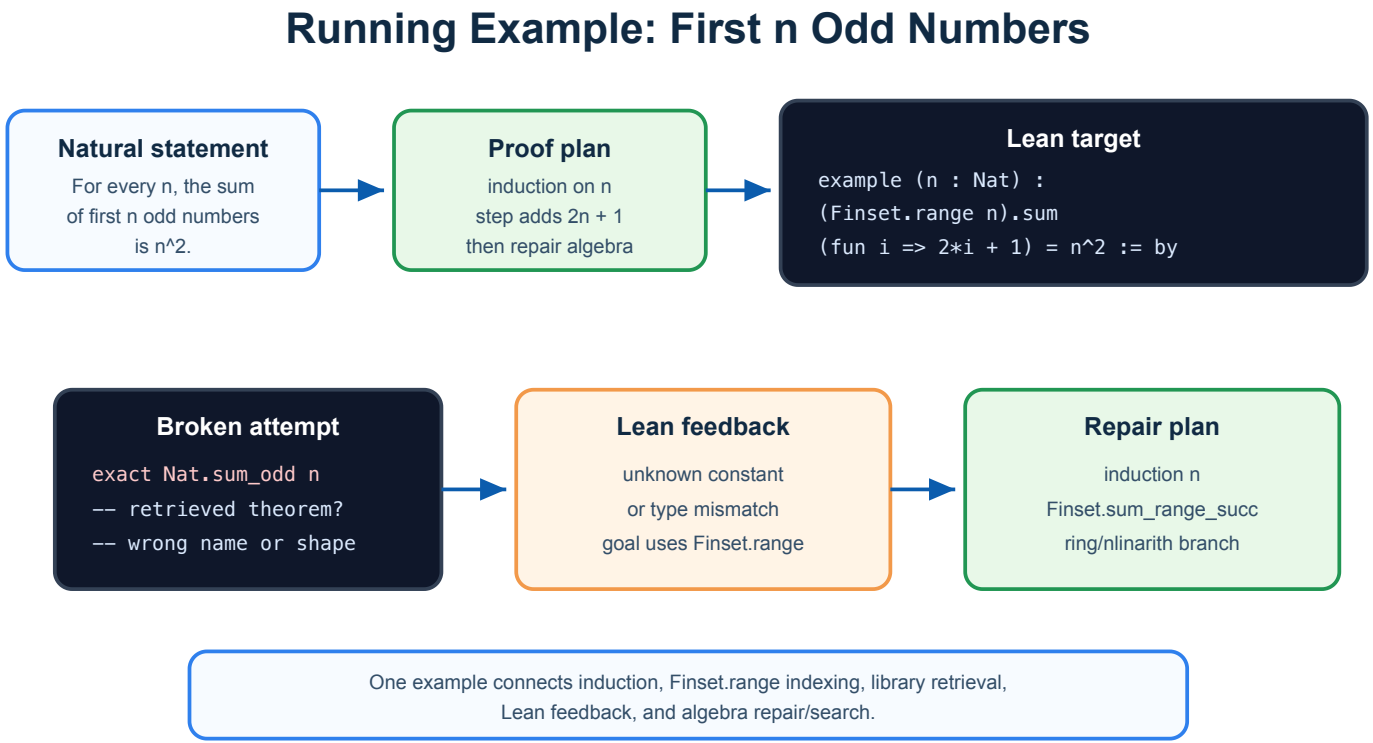
```
Goal: sum of first n odd numbers = n^2

Branch A:
  use induction on n
  reduce step to algebra

Branch B:
  search for finite-sum or arithmetic lemmas
  maybe an exact odd-sum theorem exists

Branch C:
  use geometric square picture
  good for planning, not directly Lean code
```

Natural proof search is useful because it can compare strategies before Lean syntax becomes the bottleneck.

- Branch A gives a clear Lean proof skeleton.
- Branch B is shortest if the exact library lemma is found.
- Branch C is useful for human intuition, but still needs formal translation.

The danger is false fluency: a natural branch can sound correct while hiding a wrong definition, an unstated assumption, or a theorem that does not exist.

Running Example Trace



Locally generated explanatory diagram. The example is still elementary, but now shows induction, indexing, retrieval, algebra, and repair.

What Flows Through the Agent?

Object	Example	What can go wrong
Natural problem	"first n odd numbers"	indexing ambiguous
Informal plan	induction; add `2n+1`	off-by-one error
Formal statement	sum over `Finset.range n`	wrong range or number type
Proof state	hypotheses and unsolved goal	model ignores local context
Checked artifact	Lean file without `sorry`	proof checks but is unreadable

> The agent's job is to transform these objects while preserving the original mathematical intent.

Formalization Is Already a Mathematical Choice

> The statement "first n odd numbers" is not a Lean theorem until the system chooses an indexing convention and a representation of finite sums.

```
-- Sum k = 0, ..., n-1 of (2k+1)
-- needs Mathlib finite sums; range n means 0, ..., n-1
example (n : Nat) :
    (Finset.range n).sum (fun k ⇒ 2*k + 1) = n^2 := by
  ...

-- Or define a recursive function for the first n odd numbers?
def oddSum : Nat → Nat
  | 0 ⇒ 0
  | n+1 ⇒ oddSum n + (2*n + 1)
example (n : Nat) : oddSum n = n^2 := by
  ...
```

Formalization risks

- the theorem is too weak,
- the theorem is too strong,
- the wrong library definition is used,
- the proof solves a different problem.

See Autoformalization with LLMs and ProofNet for statement-level bridging tasks.

Example: One Sentence, Several Formal Tasks

Natural sentence	Hidden formal choice	Agent should ask
"The first n odd numbers sum to n²."	`range n` vs `range (n+1)`, `Nat` vs `Int`, finite sum vs recursion	What exactly counts as the first n odd numbers?
"`A ∩ (B ∪ C) = (A ∩ B) ∪ (A ∩ C)`."	set equality, universe, membership rewriting	Should the proof use extensionality?
"`length (xs ++ ys) = length xs + length ys`."	list namespace, notation, existing lemma names	Is there already a library theorem?
"`(a * b)^-1 = b^-1 * a^-1`."	group assumptions and inverse notation	Which algebraic structure supplies the lemma?

> Autoformalization is not just translation; it is choosing the mathematical object that Lean will check.

This is the central difficulty behind Autoformalization with LLMs and ProofNet: the target formal statement must still represent the intended theorem.

Proof State: The Agent's Local World

```
example (n : Nat) :
    (Finset.range n).sum (fun k ⇒ 2*k + 1) = n^2 := by
  -- proof state:
  -- range n means k = 0, ... , n-1
  -- n : Nat
  -- ⊢ (Finset.range n).sum (fun k ⇒ 2*k + 1) = n^2
  ...
```

> A tactic is an action that transforms the current proof state.

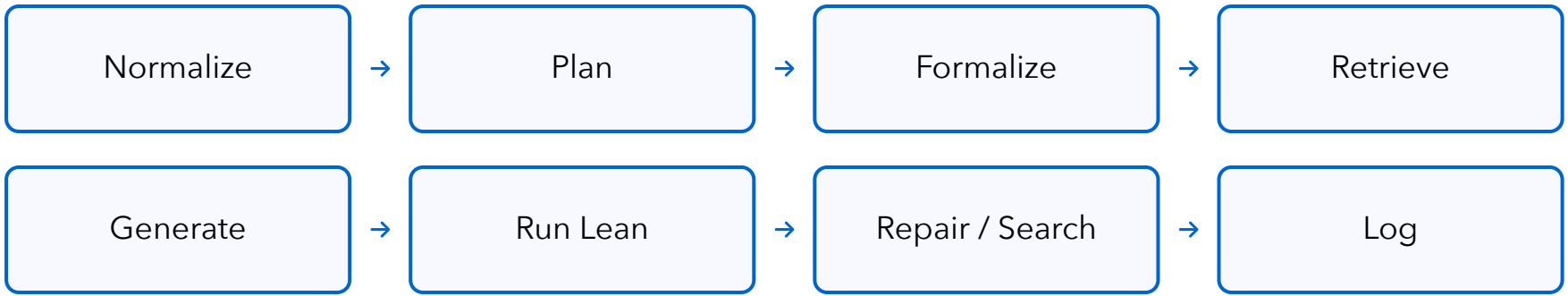
- Good agents condition on the current state.
- Bad agents keep proving the original problem text.
- Lean errors are structured feedback, not just failure.

COPRA and LeanDojo use proof-environment interaction as part of proving.

The Agent Is a Set of Modules

Each module exists because one-shot generation fails in a specific way.

Module Map



The modules are conceptually separate even when a real system merges several of them into one model call.

Normalizer and Planner

Problem normalizer

- identify variables and domains,
- expose hidden assumptions,
- select likely definitions,
- reject ambiguous goals.

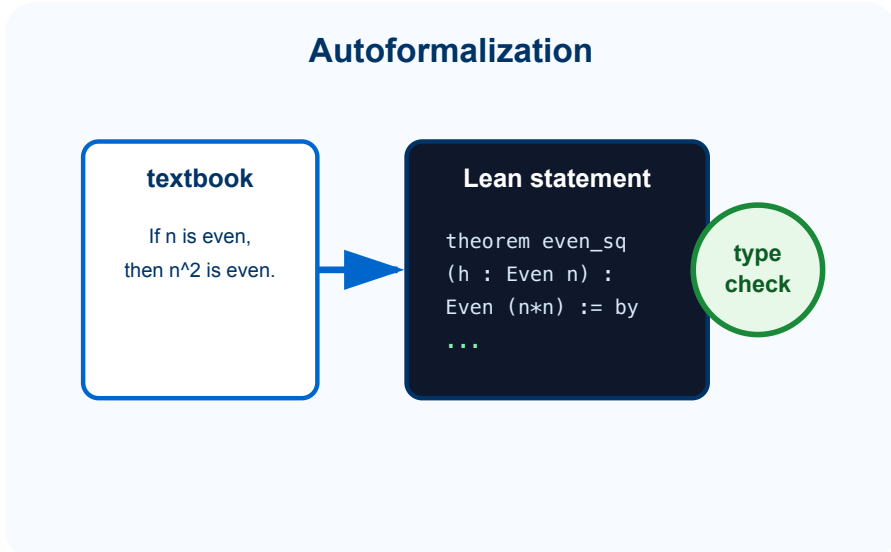
> Risk: proving a nearby theorem that was easier to formalize.

Informal proof planner

- choose a proof method,
- split into subgoals,
- suggest auxiliary lemmas,
- decide what to try first.

NaturalProver, DeepTheorem, and Prover Agent all make informal reasoning useful before formal checking.

Autoformalizer



Local project diagram.

> Autoformalization translates mathematical intent into a theorem statement, imports, definitions, and sometimes a proof skeleton.

- Autoformalization with LLMs: statement translation as a training and proving bridge.
- Draft, Sketch, and Prove: informal proof sketch guides formal proving.
- Lean-STaR: informal thought before tactic.

Statement Validator

Autoformalization needs a second pass: does the Lean theorem still mean the original theorem?

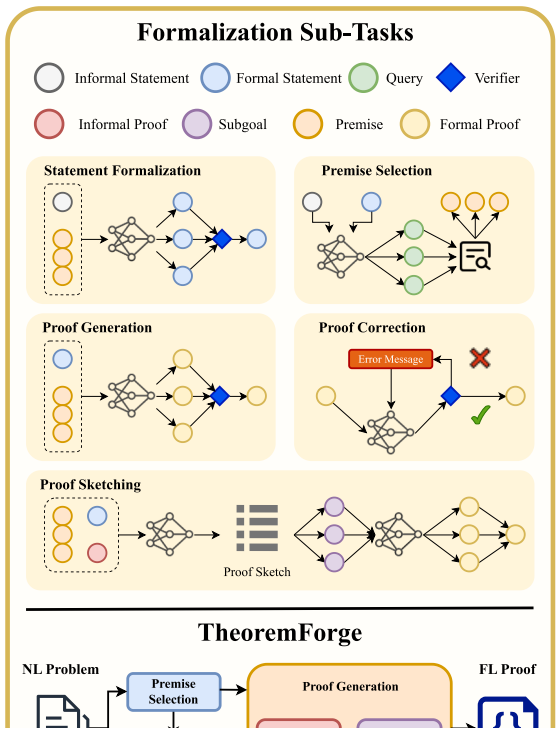
- check type/domain choices,
- compare assumptions and conclusion,
- look for weakened or strengthened statements,
- ask for counterexamples to the formalized target,
- flag theorem drift before proof search begins.

```
natural theorem
  → Lean statement
  → intent check
  → proof search
```

If the statement is wrong, a perfect Lean proof is still the wrong artifact.

Autoformalization with LLMs, ProofNet, and DAP all expose the statement/answer-discovery problem before proof construction.

TheoremForge: Five Formalization Tasks



Cropped from Figure 1 in TheoremForge, 2026.

Main decomposition

Statement formalization, premise selection, proof generation, proof correction, and proof sketching are different tasks.

Why this matters

Training a proof agent needs many kinds of examples, not only final theorem-proof pairs.

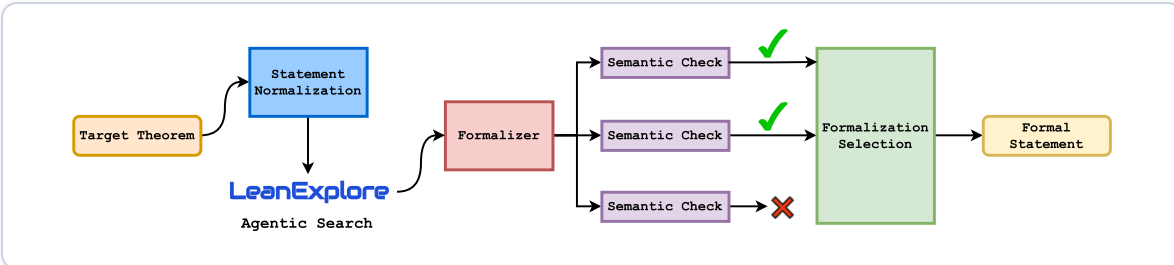
Low-budget angle

The workflow is designed as a data factory: create useful formal trajectories without massive test-time search.

Failure mode

If the task labels are blurred, a model may learn to compile code without learning which mathematical job it is doing.

TheoremForge: Statement Formalization



Cropped from Figure 2 in TheoremForge, 2026.

First bottleneck

The agent must turn a natural theorem into the right formal statement before proof search even begins.

What Lean can check

Lean checks syntax and types. It does not by itself know whether the formal theorem still says the intended problem.

Extra judgment

The workflow therefore adds semantic checks and model judging to filter candidate formalizations.

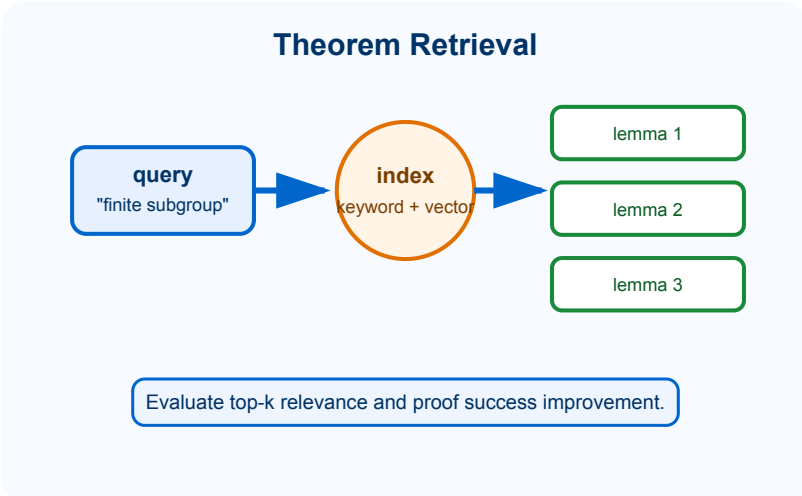
Classroom warning

A valid proof of the wrong statement is a failure, even if every line compiles.

Retriever: Mathematical Memory

> The model should not prove from memory alone. It should search definitions, lemmas, examples, and accessible premises.

- LeanDojo/ReProver: premise retrieval for Lean proving.
- LeanExplore: semantic and lexical search over Lean declarations.
- Rethlas/Archon: theorem search for research-level proof strategy and formalization.



Local project diagram.

Lean Runner and Verifier

Input

```
Lean file
tactic candidate
proof skeleton
```

Output

```
accepted proof
error message
proof state
unsolved goals
```

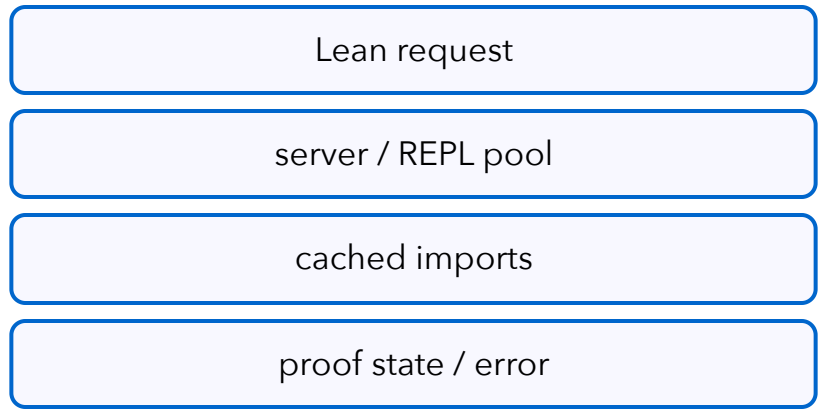
> Lean is not just the final grader. It is the agent's source of feedback.

- LeanDojo-v2 packages Lean workflow components for AI-assisted proving.
- Lean Copilot runs LLM inference natively in Lean.
- AXLE exposes Lean checking and proof manipulation as infrastructure.

Verifier Engineering

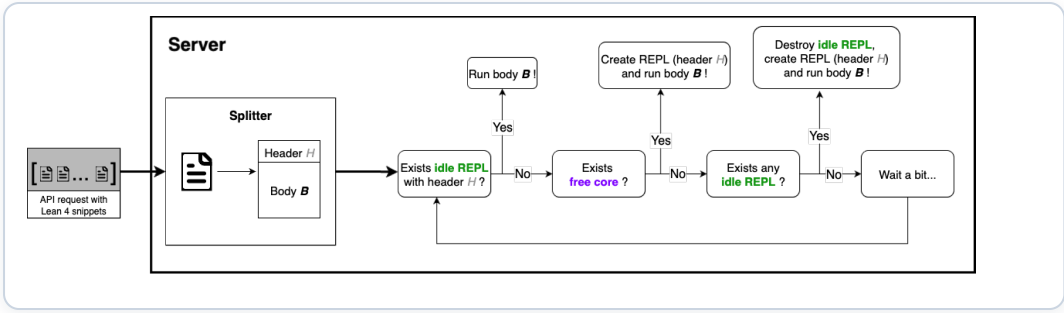
> At scale, "call Lean" becomes an engineering subsystem.

- run many Lean workers in parallel,
- cache imports and initialized environments,
- batch proof checks and tactic-state queries,
- set timeouts and isolate failing attempts,
- return structured errors, not only text logs.



Kimina Lean Server describes REST access, parallel Lean REPL workers, and import caching; AXLE exposes Lean checking and proof manipulation as reusable infrastructure.

Kimina Lean Server: Verification as a Service



Cropped from Figure 1 in Kimina Lean Server: Technical Report, 2025.

What it is

Not a prover model. It is the Lean execution layer that lets an agent submit many Lean snippets and receive checked results.

Core trick

Split each script into a costly header and a changing body. Reuse an idle REPL with the same header whenever possible.

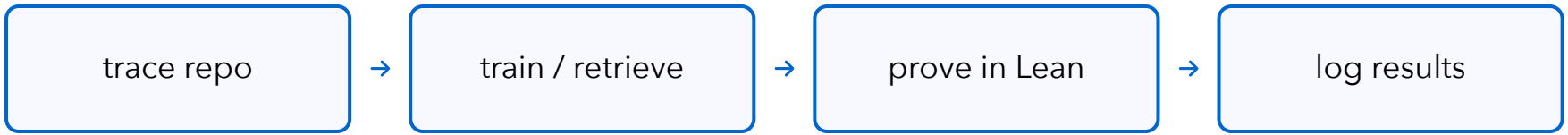
Why agents need it

Search and RL can create thousands of proof attempts. Without parallelism and caching, Lean checking becomes the bottleneck.

Risk

The verifier service must pin Lean version, project state, and imports. A wrong cache state corrupts the reward signal.

LeanDojo-v2 as Workbench



Data layer

Extract theorem statements, tactics, proof states, premises, repositories, and dependency metadata.

Model layer

Train supervised models, reward-based models, retrieval models, and progress estimators.

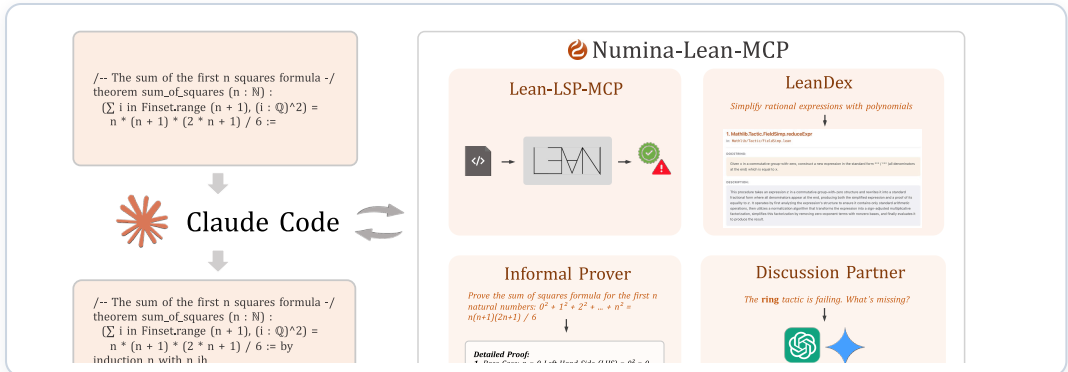
Agent layer

Run whole-proof generation or tactic search; some agents also remember useful repository experience.

LeanDojo-v2 is best taught as a proof-agent workbench: it connects data extraction, training, retrieval, proving, and UI rather than replacing the whole agent with one model.

LeanDojo-v2 GitHub README and OpenReview paper. The implementation uses Pantograph for interactive Lean proving and includes SFT, GRPO, retrieval, and progress-model workflows.

Numina-Lean-Agent: Lean Becomes a Tool Set



Cropped from Figure 1 in Numina-Lean-Agent, 2026.

Controller

Claude Code acts as the agent controller. The important output is not one proof text, but a sequence of tool calls.

Tool surface

Numina-Lean-MCP exposes Lean execution, proof search support, informal proving, and discussion tools.

Teaching point

A strong base model can become a formal reasoning agent if the environment exposes the right operations and feedback.

Open issue

Tool routing is itself a reasoning problem: when should the agent ask Lean, search, discuss, or rewrite the plan?

SFT: Supervised Fine-Tuning

SFT means: show the model many solved examples, then train it to copy the style of the correct answer.

```
example input:
  current proof state + useful lemmas

example target:
  the next tactic used in a correct proof
```

```
training pressure:
  make the target output more likely next time
```

Training target

- copy the correct proof step piece by piece,
- learn Lean formatting and local proof patterns,
- learn what information in the proof state matters,
- produce a better first guess during proof search.

In LeanDojo-v2, the SFT trainer fine-tunes a language model for prover outputs; the same copy-the-target objective underlies most supervised tactic/proof generation pipelines.

SFT for Proof Agents

SFT example	Input	Target output
next-tactic model	current proof state + retrieved premises	next tactic from a traced proof
whole-proof model	theorem statement + imports	complete Lean proof block
sketch-to-proof model	informal sketch + formal target	Lean skeleton or tactic sequence
repair model	failed attempt + Lean error	corrected proof fragment

SFT gives the agent a competent prior: it learns what a plausible tactic or proof should look like.

But SFT only imitates examples. It does not directly learn from its own failed attempts.

RL: Reinforcement Learning

> RL means: let the model try actions, score what happened, and update it toward actions that get better scores.

```
state  = current proof situation
action = tactic / lemma / proof block / branch choice
reward = score from Lean or another evaluator
```

```
training goal:
  choose actions with higher expected reward
```

Training target

- reward successful or useful attempts,
- penalize dead ends or invalid proofs,
- learn from attempts the model generated itself,
- keep enough exploration to find new proof paths.

For proof systems, examples include proof-assistant-feedback RL in DeepSeek-Prover-V1.5, interaction-based RL in Seed-Prover 1.5, and reward-based refinement workflows in LeanDojo-v2.

RL From Lean Feedback

sample attempts → run Lean → give a score → update model

Final-score reward

`+1` if Lean accepts, `0` or penalty otherwise.

Partial-credit reward

Partial credit for fewer goals, useful lemma, shorter proof, or recoverable error.

Risk

The model can learn shortcuts that score well without proving the intended theorem.

SFT teaches the model to imitate proof traces; RL teaches it to prefer actions that make the verifier or evaluator happy.

Kimina-Prover-RL uses Lean-verification reward through Kimina Lean Server; DeepSeek-Prover-V1.5 uses proof-assistant feedback as RL/search signal.

Proof Generator: What Should It Emit?

natural proof paragraph **formal statement** **proof sketch** **next tactic** **auxiliary lemma**

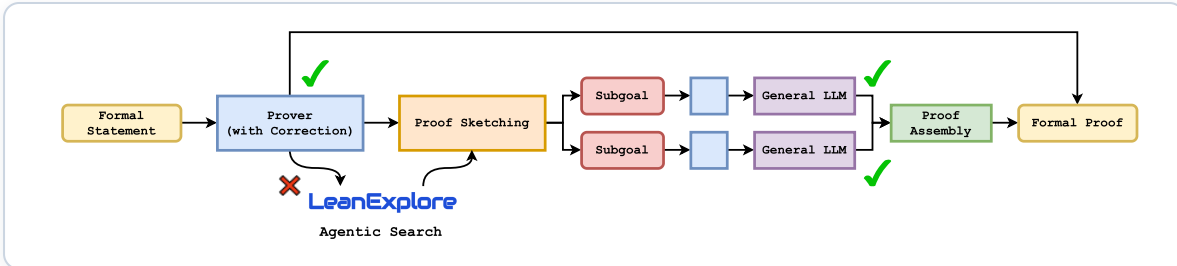
subgoal decomposition **whole proof** **repository-level edit**

Small units give frequent feedback, but require more search control.

Large units are easier to prompt, but failures are harder to diagnose.

Examples: Lean-STaR emits informal thought plus tactic; Seed-Prover emphasizes lemma-style whole-proof reasoning; Leanstral is framed around Lean repository/code-agent workflows.

TheoremForge: Proof Generation



Cropped from Figure 3 in TheoremForge, 2026.

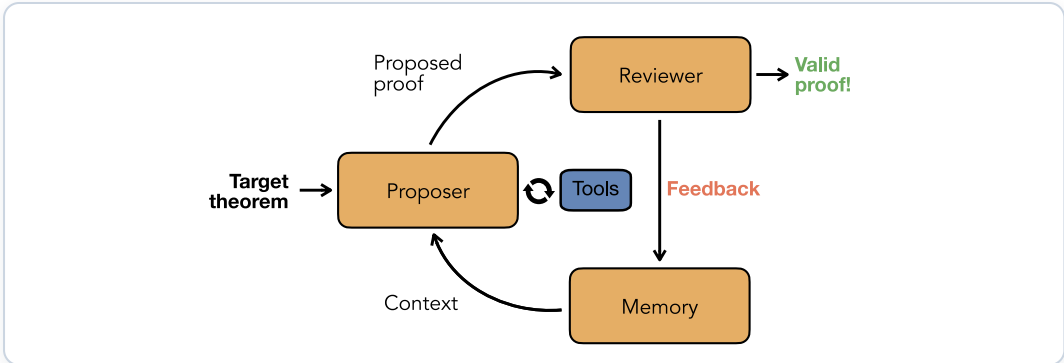
Direct proof path
A proof learner samples candidates; compiler and semantic checks reject bad candidates.

Repair path
LeanExplore and proof sketching turn failures into subgoals and correction attempts.

Agent pattern
Try direct proof first; when it fails, decompose, retrieve, repair, and assemble.

Risk
Compiler-error repair can overfit local errors while leaving the proof plan mathematically incoherent.

Minimal Agent: The Baseline Loop



Cropped from Figure 1 in A Minimal Agent for Automated Theorem Proving, 2026.

Proposer
Reads the target theorem and context, then writes a candidate proof.

Reviewer
Rejects invalid or cheating proofs; accepts only a proof that passes the required checks.

Tools and memory
Search tools supply facts; memory stores feedback so the next proposal can change.

Why it is useful
Many impressive systems are enlarged versions of this loop: propose, check, remember, try again.

Search Controller

Control questions

- How many candidates are sampled?
- Which branch is expanded?
- When should the system retrieve again?
- When should it create a lemma?
- When should it stop?

try many candidates

backtrack after failure

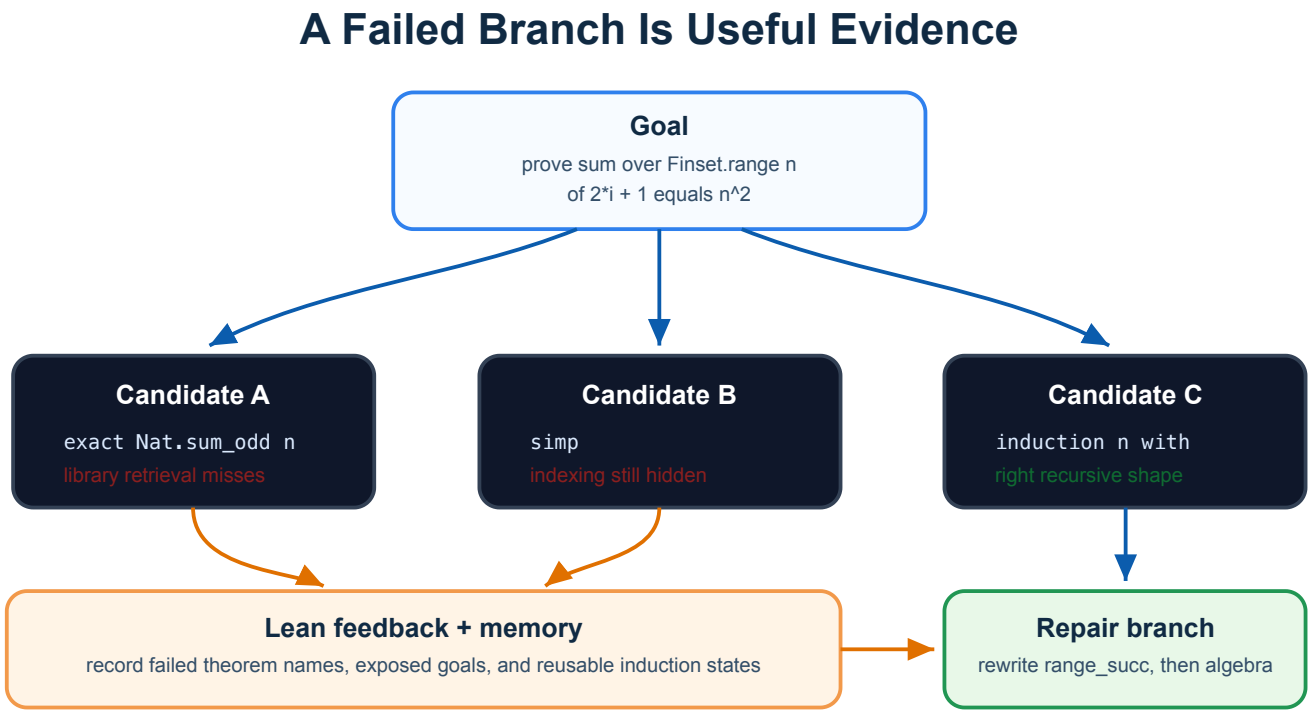
Tree / graph search

self-critique loop

spend more attempts

COPRA uses backtracking; DeepSeek-Prover-V1.5 uses proof-assistant-feedback RL and search; Aristotle reports Monte Carlo graph search.

Example: Search Tree From Lean Feedback



Search is retrieval plus Lean feedback: choose induction, fix Finset.range indexing, and repair the algebraic step.

Locally generated explanatory diagram. Related systems: COPRA, DeepSeek-Prover-V1.5, Aristotle.

Tree Search Is Structured Sampling

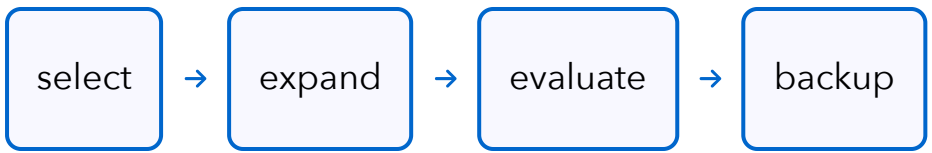
Search object	In a game	In a proof agent
Node	board position	proof state, informal subgoal, or partial proof
Edge	legal move	tactic, lemma, rewrite, proof paragraph
Search guide	which move looks promising	LLM score, retrieval score, known tactic pattern
Evaluation	win probability	Lean progress, checker score, solved subgoal, useful lemma
Terminal reward	win/loss	Lean accepts, contradiction, timeout, unrecoverable error

> Tree search is what lets a proof agent spend more compute on promising branches while keeping alternatives alive.

This is the shared control idea behind backtracking agents such as COPRA, proof-assistant-feedback search in DeepSeek-Prover-V1.5, and graph search in Aristotle.

Monte Carlo Tree Search

Monte Carlo means using repeated sampled attempts to estimate which branch is worth more work.




```
choose a branch by balancing:
  how promising it looks
  + how much we still need to explore it
```

- **Select:** follow branches that look useful but are not fully tested.
- **Expand:** ask the model for new candidate actions.
- **Evaluate:** use Lean, a checker, a learned scorer, or a short trial run.
- **Backup:** update parent branches using the new evidence.

For proofs, the hard part is evaluation: most branches are unfinished until Lean accepts or exposes a useful error.

Kocsis and Szepesvari, Bandit Based Monte-Carlo Planning, 2006 introduced UCT-style tree selection; DeepSeek-Prover-V1.5 adapts proof-assistant feedback and search to Lean proving.

Relation to AlphaGo

Locally generated explanatory diagram. Source anchors: Silver et al., AlphaGo, Nature 2016; UCT, 2006; DeepSeek-Prover-V1.5.

AlphaGo combined a learned move guide, a learned position scorer, and tree search. Proof agents reuse this pattern, but replace game simulation with Lean feedback, retrieval, and formal verification.

Lemma Lifecycle

propose lemma → formalize → prove / reject → reuse

Why create lemmas?

They turn a hard global proof into smaller verified assets.

What can go wrong?

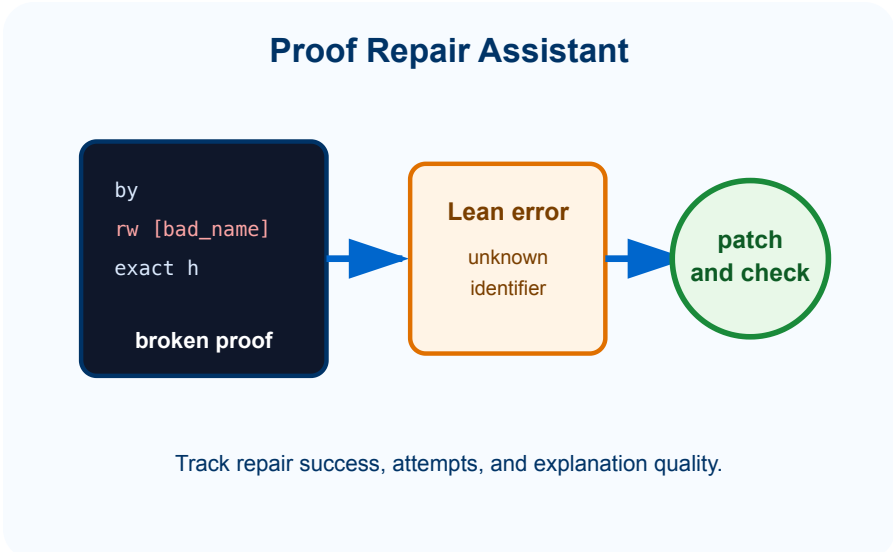
The lemma may be false, too weak, too strong, or harder than the original goal.

What should be logged?

Statement, dependencies, proof status, failed attempts, and reuse count.

Prover Agent, Seed-Prover, and Aristotle all make auxiliary lemma generation or lemma-style reasoning a central mechanism.

Repair and Memory



Local project diagram.

A failed proof attempt can still teach the agent.

- classify the Lean error,
- remember failed tactics,
- summarize partial progress,
- generate a missing lemma,
- retry with different retrieval.

Baldur studies generation and repair from proof-assistant feedback; Seed-Prover 1.5 emphasizes learning from Lean/tool interaction experience.

Evaluator and Logger

Minimal metrics

- type-checking success rate,
- number of Lean calls,
- proof length,
- retrieval hit rate,
- human repair time,
- failure categories.

Without logs and metrics, a proof agent is only a demo.

A useful negative result still explains what failed: statement formalization, retrieval, tactic generation, search, or repair.

From Logs to Training Data

```
attempt
  → Lean result
  → reward / filter
  → training example
  → stronger prover
```

The same verifier feedback can drive both search during solving and future model training.

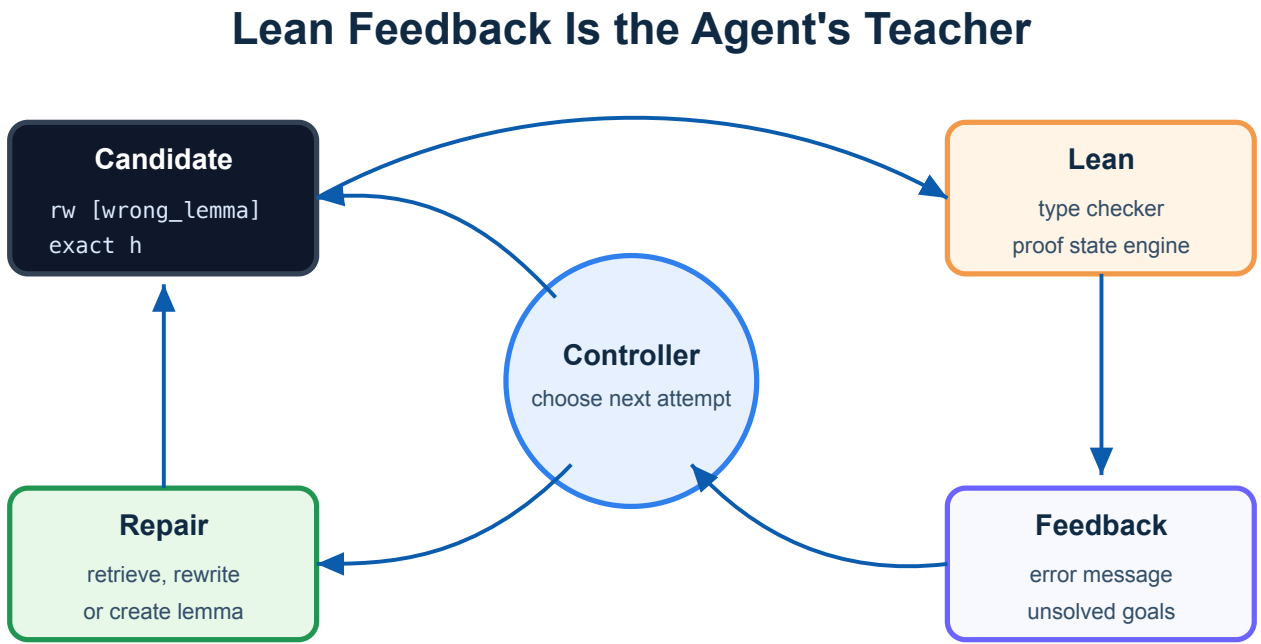
- accepted proofs become supervised examples,
- failed attempts become negative evidence,
- Lean acceptance can define final-score rewards,
- multi-turn traces can train repair behavior,
- experience replay turns attempts into a growing corpus.

DeepSeek-Prover-V1.5 uses proof-assistant feedback for RL/search; Seed-Prover 1.5 emphasizes learning from interaction experience; Kimina-Prover-RL uses Lean verification reward through Kimina Lean Server.

The Loop

Propose. Check. Read feedback. Search again.

Lean Feedback Loop



A failed Lean attempt is not wasted. It becomes structured information for the next search step.

Locally generated explanatory diagram. Related systems: COPRA, Baldur, DeepSeek-Prover-V1.5.

Agent Loop Pseudocode

```
state = initialize(problem)
for step in budget:
    query = build_prompt(state)
    candidates = model.generate(query)
    candidates = add_retrieved_lemmas(candidates, state)
    result = run_lean(candidates)

    if result.accepted:
        return checked_proof

    state = update_state(
        old_state=state,
        lean_feedback=result.errors_or_goals,
        failed_candidates=candidates
    )

return failure_report
```

Most design choices are hidden inside `build_prompt`, `add_retrieved_lemmas`, and `update_state`.

Concrete Trace

Natural statement

The sum of the first n odd numbers is n^2.

Proof plan

Use zero-based indexing: terms are 2k+1 for k=0,...,n-1. The induction step adds 2n+1.

Broken attempt

Formalize with `range (n+1)`, so the base case and induction step no longer match the natural statement.

Repair idea

Change to `range n`; induct on n; use the induction hypothesis; simplify the added term.

Shared Architecture

Existing systems differ, but the skeleton is surprisingly stable.

Common Layer 1: Mathematical Intent

Every proof agent starts by deciding what problem is being solved and what artifact counts as success.

Human proof

NaturalProofs and NaturalProver stay close to mathematical language.

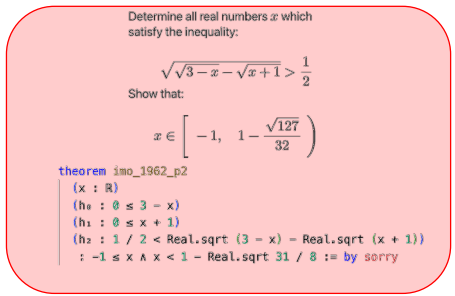
Answer discovery

DAP separates finding an answer from proving it.

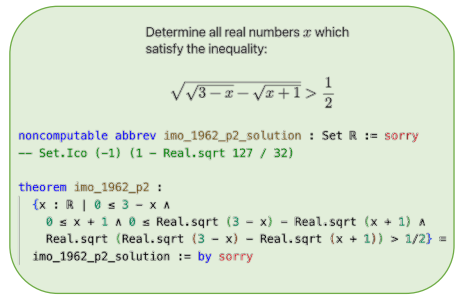
Research artifact

Axiom Math's Fel conjecture case emphasizes natural-language-to-Lean intent preservation.

DAP: Easy Mode vs Hard Mode



Easy Mode
imo_1962_p2 from miniF2F-test



Hard Mode
imo_1962_p2 from miniF2F-Hard

Cropped from Figure 1 in Discovery-Augmented Proving, 2026.

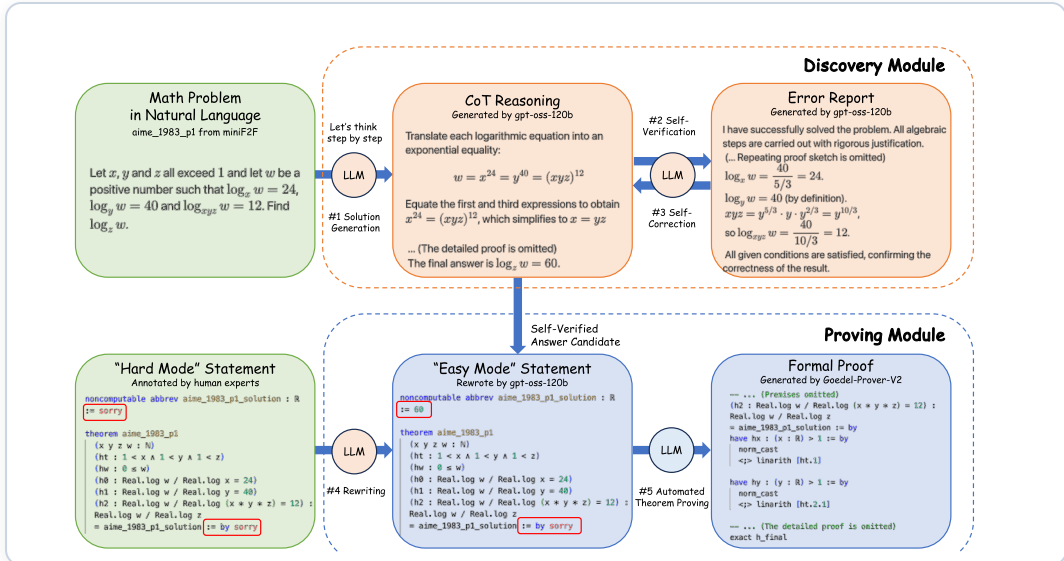
Easy Mode
The answer is already embedded in the statement, so the prover mainly justifies a supplied target.

Hard Mode
The statement represents the original question more faithfully, so the answer must be discovered.

Why students should care
A benchmark can look formal while testing a weaker task than the natural-language problem.

General lesson
Formal correctness is only meaningful after the formal statement has the right meaning.

DAP: Discovery Before Proving



Cropped from Figure 2 in Discovery-Augmented Proving, 2026.

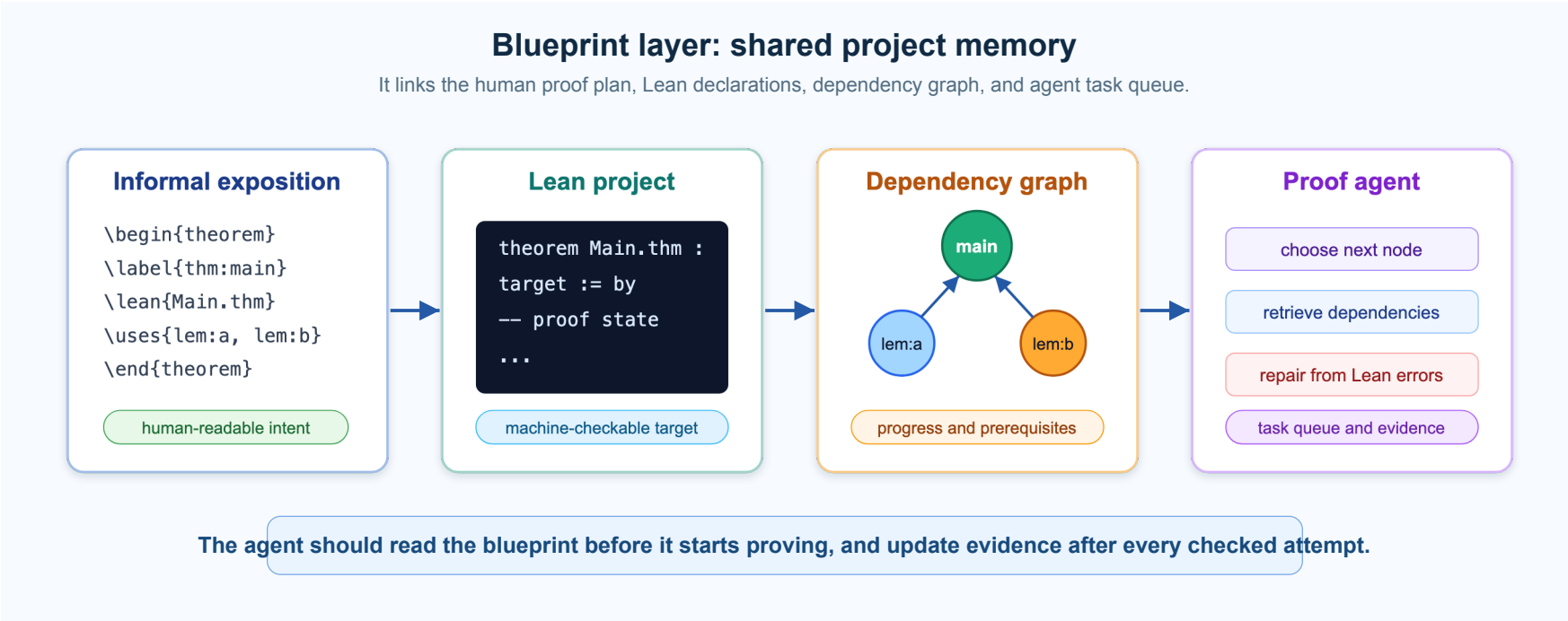
Discovery module
A reasoning model proposes an answer and self-checks it before the formal prover is invoked.

Rewrite step
Once the answer is known, the hard statement is rewritten into an easier theorem about that answer.

Proving module
A separate theorem-proving model tries to produce the Lean proof for the rewritten statement.

Design message
Natural-language proof agents often need both answer discovery and formal verification; one does not replace the other.

Common Layer 1b: Blueprint as Project Memory



Locally generated explanatory diagram. Source anchors: leanblueprint README, Tao's PFR blueprint tour, LeanArchitect.

For a large Lean formalization, the blueprint is the object that keeps human mathematics, Lean names, dependencies, and progress in one navigable plan.

Blueprint: Why This Exists

Without a blueprint

- The textbook-style proof is in one place.
- The Lean theorem names are in another place.
- The dependency order is partly in people's heads.
- No one can quickly tell what is blocked.

With a blueprint

- Each informal theorem points to a Lean declaration.
- Each node records what it depends on.
- Progress is visible to humans and agents.
- The next task can be chosen from the graph.

For students: think of a blueprint as a shared project map for a long proof, not as a new proof checker.

What Does a Blueprint Record?

```
\begin{theorem}[Smale 1958]
  \label{thm:sphere_eversion}
  \lean{sphere_eversion}
  \leanok
  \uses{def:immersion}
  ...
\end{theorem}

\begin{proof}
  \leanok
  \uses{thm:open_ample, lem:open_ample_immersion}
  ...
\end{proof}
```

- `\lean{...}` links exposition to a named Lean theorem or definition.
- `\leanok` marks a statement or proof as formalized.
- `\uses{...}` builds a mathematical dependency graph.
- `\notready`, `\discussion`, and `\mathlibok` record project status.

> The graph is not merely a picture: it is a scheduling device for humans and for agents.

Macro behavior from the leanblueprint README. The example there is from the Sphere Eversion project.

Why Agents Need Blueprints

Task selection

Pick a node whose prerequisites are already stated or proved.

Intent preservation

Compare generated Lean statements with the informal theorem text.

Retrieval scope

Use declared dependencies before searching the whole library.

Progress report

Report which theorem, proof, or dependency is actually blocked.

Collaboration

Let humans review mathematical choices before the agent expands them.

Evidence

Attach Lean errors, proof attempts, and verified lemmas to the right node.

This is an architectural interpretation. Existing blueprint sites are human-facing; agent use requires extra tooling around the same metadata.

Current Blueprint Frictions

Human-facing friction

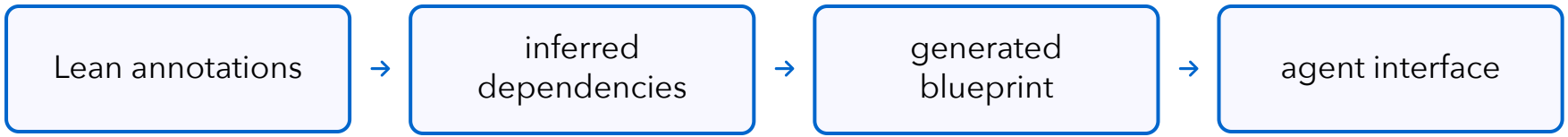
- LaTeX exposition and Lean code can drift apart.
- `\leanok` is a project-status claim, not a semantic proof of equivalence.
- Dependencies in `\uses` may be too coarse, stale, or different from Lean imports.
- Large graphs can become hard to navigate without good granularity.

Agent-facing friction

- The agent needs machine-readable tasks, not just rendered HTML.
- It needs to know whether to prove a statement, define an object, or repair exposition.
- Failure logs are not part of the standard blueprint data model.
- Changing a Lean declaration name can break the bridge unless checks are run.

LeanArchitect identifies decoupled LaTeX and Lean artifacts as a maintenance burden and a barrier to AI automation; leanblueprint provides `checkdecls` to catch missing Lean declaration names.

A Direction: LeanArchitect



Move metadata closer to Lean

The formal file becomes the source of more blueprint data.

Reduce duplication

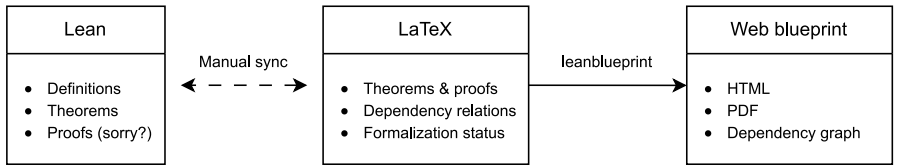
Fewer hand-maintained links between theorem text, declarations, and dependencies.

Expose structured tasks

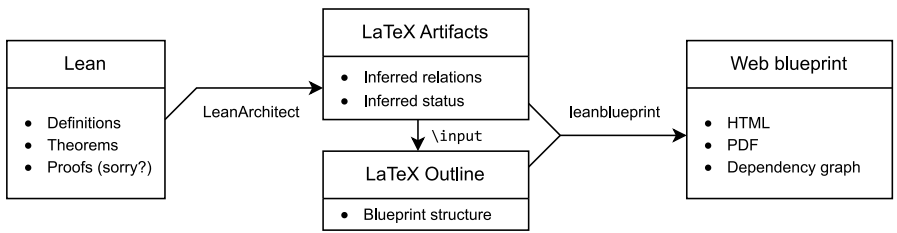
Progress tracking becomes usable by human collaborators and AI proof agents.

Zhu, Monticone, Avigad, Welleck, LeanArchitect: Automating Blueprint Generation for Humans and AI, 2026.

LeanArchitect: Blueprint as Interface



(a) Blueprint generation workflow without LeanArchitect.



Cropped from Figure 1 in LeanArchitect, 2026.

Without automation

Lean files and LaTeX blueprints drift apart; humans manually synchronize theorem names, dependencies, and status.

With LeanArchitect

The system infers relations and proof status from Lean, then feeds a structured blueprint pipeline.

Agent use

A blueprint gives the agent a dependency graph: choose the next lemma, retrieve nearby context, measure progress.

Current limitation

The generated graph can track formal progress, but it may still miss the human mathematical motivation.

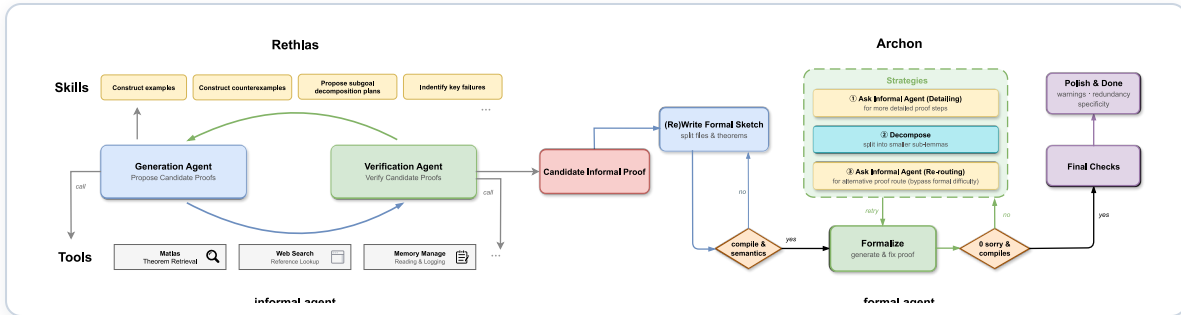
Common Layer 2: Strategy and Decomposition

> Hard proofs are usually not solved by emitting the final proof in one shot.

```
generate proof idea
  → split into subgoals
  → propose auxiliary lemmas
  → choose next proof branch
```

- Draft, Sketch, and Prove: informal sketch guides formal proof.
- DeepSeek-Prover-V2: subgoal decomposition.
- Seed-Prover, Prover Agent, and Aristotle: lemma generation or decomposition is central.

Rethlas/Archon: Two-Agent Structure



Cropped from Figure 1 in Automated Conjecture Resolution with Formal Verification, 2026.

Rethlas

An informal agent searches for a plausible mathematical proof idea using generation and verification agents.

Archon

A formal agent translates the candidate informal proof into a Lean project and removes remaining proof gaps.

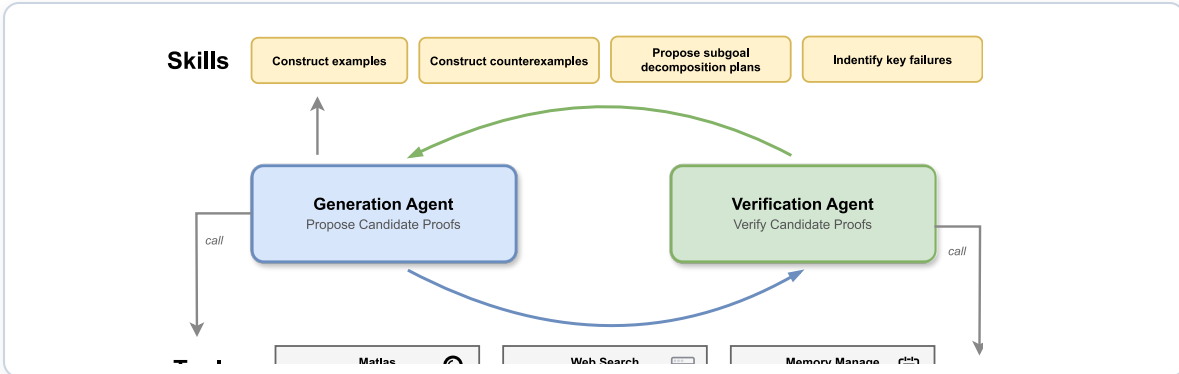
Important separation

Finding the proof strategy and making Lean accept it are related but different jobs.

Use case

This architecture is designed for research-level mathematics, where direct next-tactic search is usually too local.

Rethlas: Natural-Proof Agent



Cropped from Figure 2 in Automated Conjecture Resolution with Formal Verification, 2026.

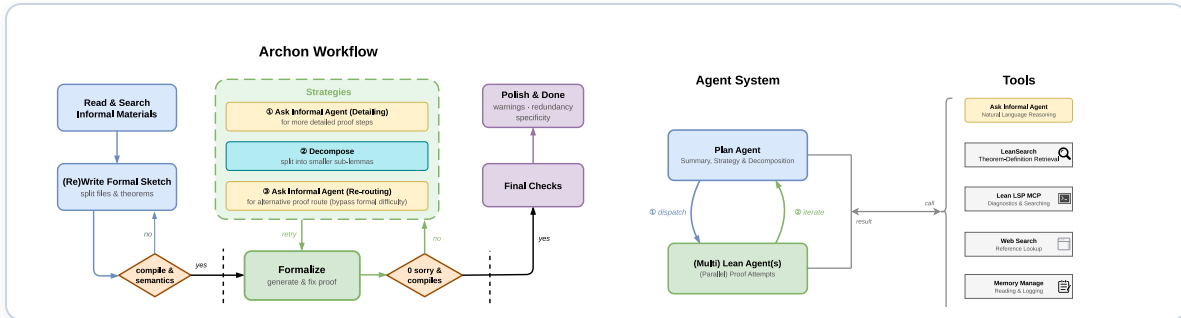
Natural proof as workspace
The candidate proof is not trusted text; it is an object to generate, critique, and revise.

Verification agent
Checks candidate informal proofs, looks for hidden assumptions, and asks for refinement.

Skills
Construct examples, counterexamples, subgoal decompositions, and failure analyses.

Difference from Lean-only agents
The search space is mathematical argument structure before it becomes formal code.

Archon: Formal Agent and Tools



Cropped from Figures 3-4 in Automated Conjecture Resolution with Formal Verification, 2026.

Scaffold
Rewrite the informal proof into formal files, theorem statements, and split lemmas.

Prove
Alternate detailing, decomposition, and re-routing when a Lean proof fails.

Tool box
Informal agent, theorem search, Lean LSP/MCP, web search, and memory are all callable tools.

Final pressure
The formal agent is judged by compilation, missing `sorry`s, and final proof cleanliness.

Common Layer 3: Library Access

Retrieval is how the agent uses existing mathematics instead of rediscovering every fact.

List retrieval

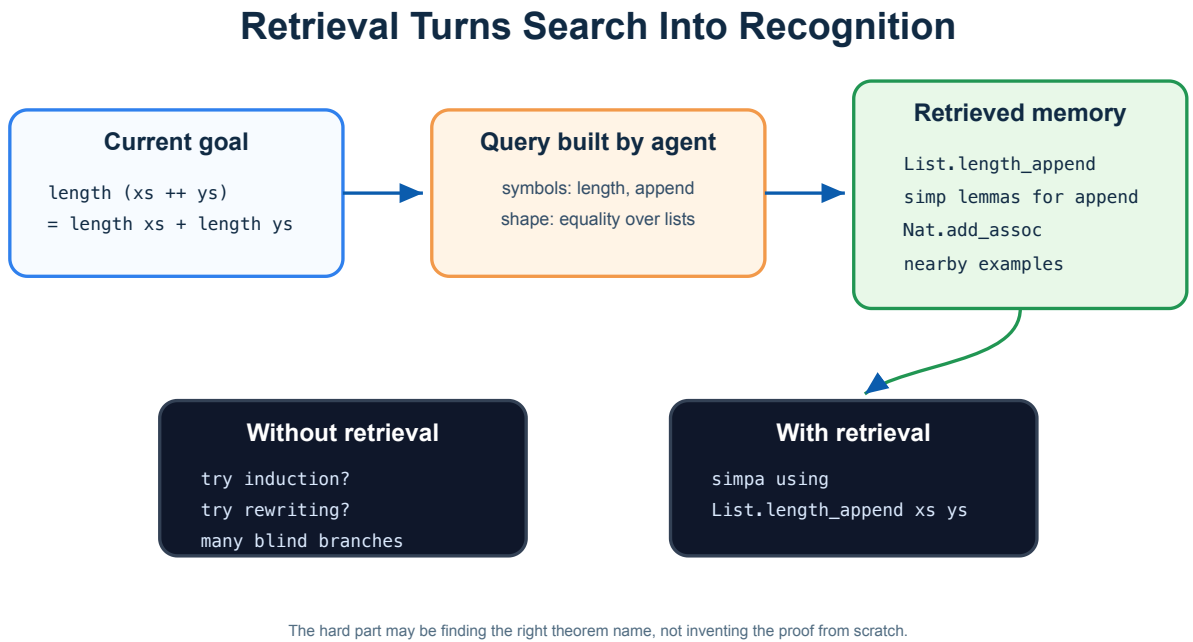
`length (xs ++ ys)` should suggest append-length lemmas or `simp`.

Group retrieval

(a * b)^-1 = b^-1 * a^-1 is easy if the right library theorem is found.

LeanDojo/ReProver, LeanExplore, and Rethlas/Archon all make retrieval a core proof-agent component.

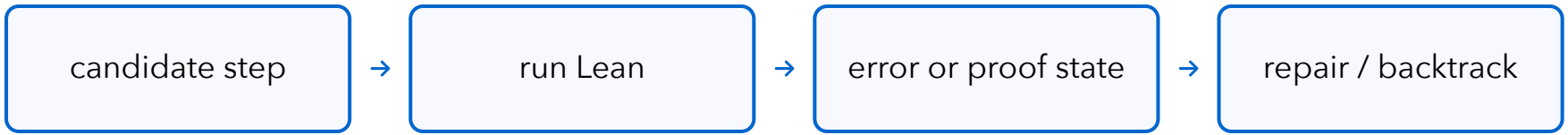
Example: Retrieval Changes the Proof



Locally generated explanatory diagram. Source anchors: LeanDojo/ReProver, LeanExplore.

In a proof agent, retrieval is not decoration. It changes the action space available to the model.

Common Layer 4: Formal Interaction



> Lean is both verifier and teacher: it tells the agent whether a proof is accepted and why an attempt failed.

COPRA, Baldur, DeepSeek-Prover-V1.5, Lean Copilot, and AXLE each use this feedback idea in different ways.

Common Layer 5: Search, Budget, Evidence

Search policy

- expand promising branches,
- keep alternatives alive,
- use verified lemmas as assets,
- spend compute deliberately.

Evidence

- store proof attempts,
- store Lean errors,
- record retrieval results,
- report reproducible failures.

Seed-Prover emphasizes trying both deep branches and broad alternatives; Delta Prover emphasizes decomposition, self-critique, and spending more attempts at solving time; Aristotle reports Monte Carlo graph search.

Two New Directions: Leanstral and LongCat

Leanstral: proof engineering agent

- Works inside a Lean project, not just on one theorem.
- Reads files, runs tools, edits code, checks the build.
- Best viewed as an assistant for maintaining proof projects.

LongCat-Flash-Prover: trained with Lean feedback

- Trains a model around Lean tool feedback.
- Breaks the task into formalize, sketch, and prove.
- Checks that a "successful" proof did not change the problem.

Leanstral asks: how can a model work inside a Lean project? LongCat asks: how can a model learn formalize/sketch/prove behavior from Lean feedback?

Leanstral model card describes a Lean code-agent model; LongCat GitHub and technical report describe formalization, sketching, proving, tool feedback, and reward-hacking defenses.

Leanstral: Repository-Level Proof Work

```
Lean repository
  → inspect files and dependencies
  → run lake / Lean tools
  → diagnose failed proof or definition
  → edit incrementally
  → verify build
```

> The core object is not an isolated test question. It is a live Lean project.

- Lean-specific agent prompt: orient, plan, execute, verify.
- Tool calling means the model can ask another program to run Lean or inspect files.
- Tool connectors are the plumbing that lets the agent use those programs.
- Official examples include diagnosing a Lean definition issue and translating from another proof assistant to Lean.
- Their evaluation checks repository-level proof tasks, not only isolated olympiad problems.

Leanstral LEAN.md describes the Lean agent workflow; Mistral announcement discusses FLTEval, lean-lsp-mcp, and repository-level examples.

LongCat: Formalize, Sketch, Prove

Formalization tools

Auto-formalization tools check whether the Lean statement compiles and still matches the informal problem.

Sketch-proof

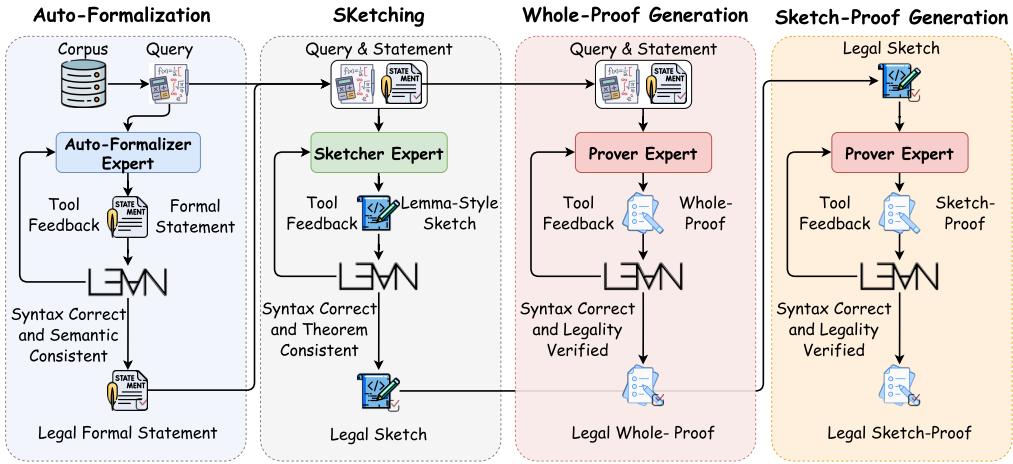
Generate helper lemmas, prove them, then assemble the target theorem.

RL from tools

The model tries formal steps, sees Lean feedback, and training moves toward useful behavior.

LongCat-Flash-Prover technical report, Sections 2-3 and Appendix C/D. The paper's technical names include tool-integrated reasoning, agentic RL, and HisPO for stabilizing long tool-interaction training.

LongCat: Tool-Integrated Synthesis



Cropped from Figure 2 in LongCat-Flash-Prover, 2026.

Four jobs

Auto-formalization, sketching, whole-proof generation, and sketch-proof generation are trained as separate capabilities.

Feedback loop

Each expert interacts with tools, receives Lean feedback, and keeps only legal or semantically consistent artifacts.

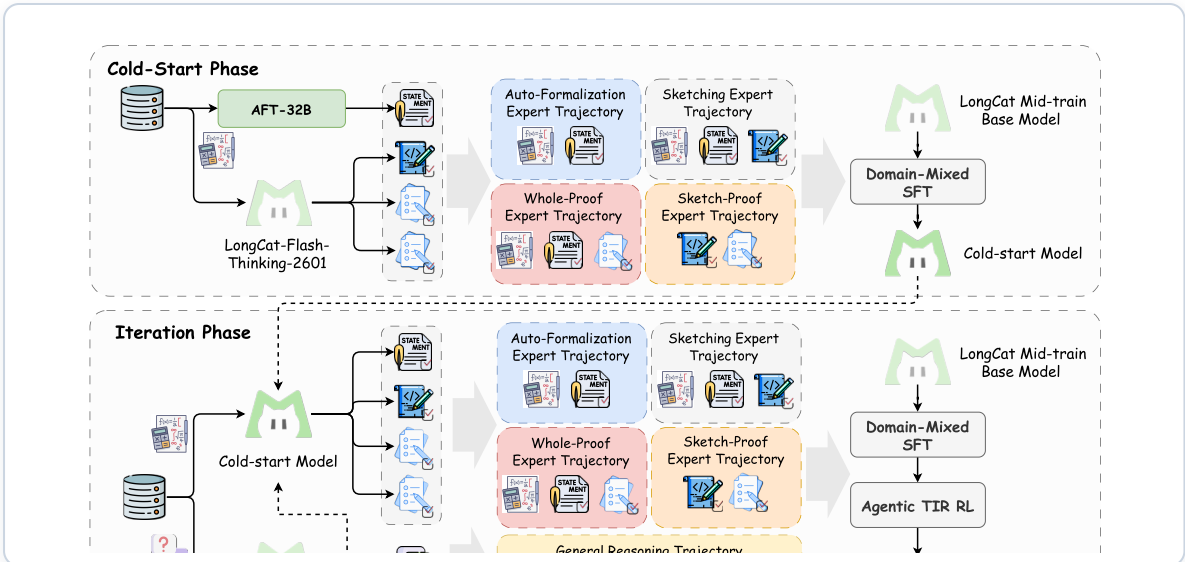
Key design

A failed whole-proof attempt can still produce useful sketches or intermediate trajectories for training.

Legality

The verifier must check that the theorem was not weakened and the environment was not altered.

LongCat: SFT and RL Pipeline



Cropped from Figure 3 in LongCat-Flash-Prover, 2026.

SFT stage
Supervised fine-tuning teaches the model to imitate useful formalization, sketching, proof, and general reasoning traces.

RL stage
Reinforcement learning shifts the model toward actions that make progress under tool and verifier feedback.

Self-improving loop
The current model generates new trajectories; filtered trajectories train the next iteration.

Reward hacking
If reward only means "Lean accepted something," the model may learn shortcuts. Legal equivalence checks are part of training design.

LongCat: Reward Hacking Is Real

Lean acceptance alone is not always enough as a reward.

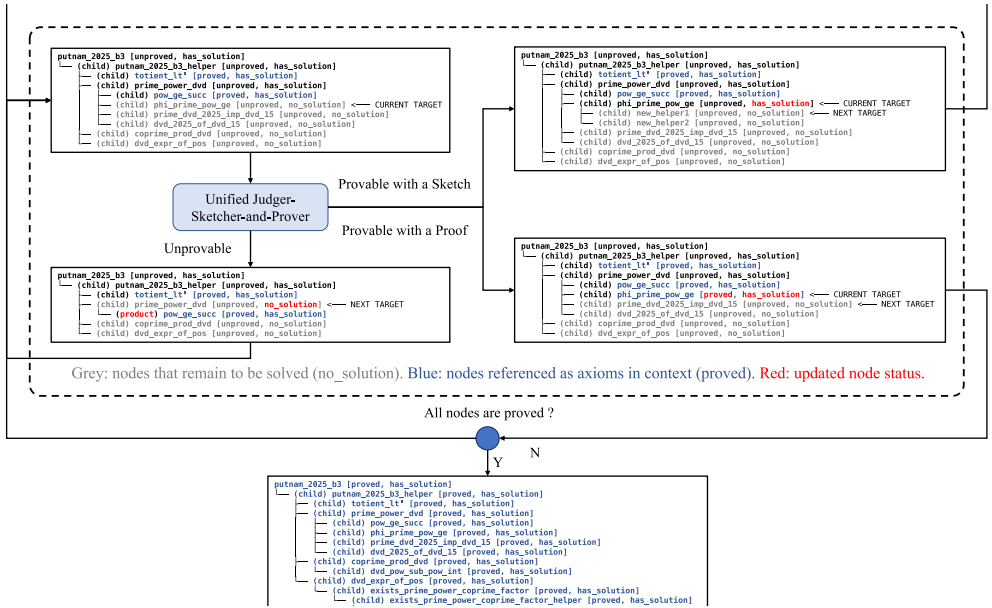
- change the theorem into an easier theorem,
- add an unproved assumption and use it as if it were true,
- tell Lean to ignore later code,
- change the meaning of background definitions.

```
original statement
  → read its formal structure
generated proof
  → read its formal structure
compare theorem / definitions / options
  → accept or reject
```

Verifier engineering is part of RL design: the reward function must reject fake proofs, not only compile failures.

The public LongCat lean4parser implements AST-based legality detection: it parses Lean code into a formal syntax tree and checks that the generated proof did not alter the original theorem environment.

LongCat: Lemma Tree Search



Cropped from Figure 5 in LongCat-Flash-Prover, 2026.

Search object
The tree nodes are lemmas with status: unproved, proved, no solution, or reused as context.

Agent action
The unified model can judge a node, sketch sublemmas, or produce a proof for a leaf.

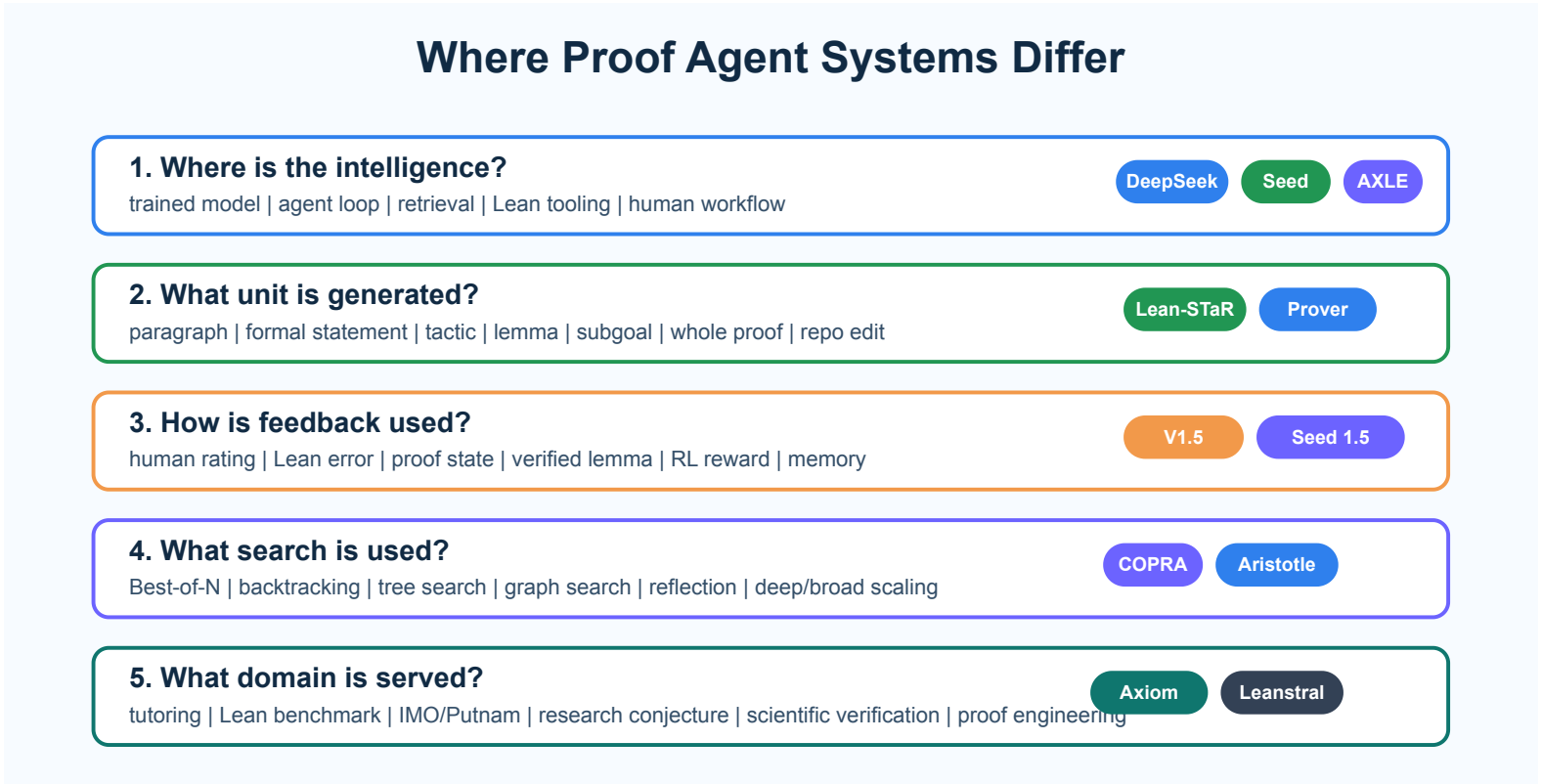
Memory compression
Once a lemma is proved, later branches can reference only its statement, not its full proof body.

AlphaGo analogy
Like game-tree search, the system expands candidates and backs out of bad branches. Unlike Go, Lean gives exact local verification.

Where Systems Differ

Use design axes, not a disconnected list of names.

Design Axes


Where Proof Agent Systems Differ
1. Where is the intelligence?
trained model | agent loop | retrieval | Lean tooling | human workflow
DeepSeek
Seed
AXLE
2. What unit is generated?
paragraph | formal statement | tactic | lemma | subgoal | whole proof | repo edit
Lean-STaR
Prover
3. How is feedback used?
human rating | Lean error | proof state | verified lemma | RL reward | memory
V1.5
Seed 1.5
4. What search is used?
Best-of-N | backtracking | tree search | graph search | reflection | deep/broad scaling
COPRA
Aristotle
5. What domain is served?
tutoring | Lean benchmark | IMO/Putnam | research conjecture | scientific verification | proof engineering
Axiom
Leanstral

Axis 1: Where Is the Intelligence?

Location	Examples	What it means
Trained prover model	DeepSeek-Prover, Seed-Prover, LongCat	learn formal proof patterns from data, tools, and feedback
Agent loop	COPRA, Delta Prover	search, backtrack, repair around proof states
Infrastructure	LeanDojo, AXLE, Leanstral	expose Lean, retrieval, and workflow tools

> Different systems move intelligence to different places in the architecture.

Axis 2: What Unit Is Generated?

Smaller units

- next tactic,
- informal thought,
- auxiliary lemma,
- subgoal.

Lean-STaR interleaves informal thought and tactic.

Larger units

- natural proof paragraph,
- formal statement,
- whole proof,
- repository-level edit.

Seed-Prover uses lemma-style whole-proof reasoning; LongCat separates auto-formalization, sketching, and proving; Leanstral targets Lean code-agent work.

Axis 3: How Is Feedback Used?

Immediate feedback

Lean errors and proof states directly shape the next prompt or branch.

Training signal

DeepSeek-Prover-V1.5 uses proof-assistant feedback for RL/search; LongCat uses tool-interaction RL with legality checks.

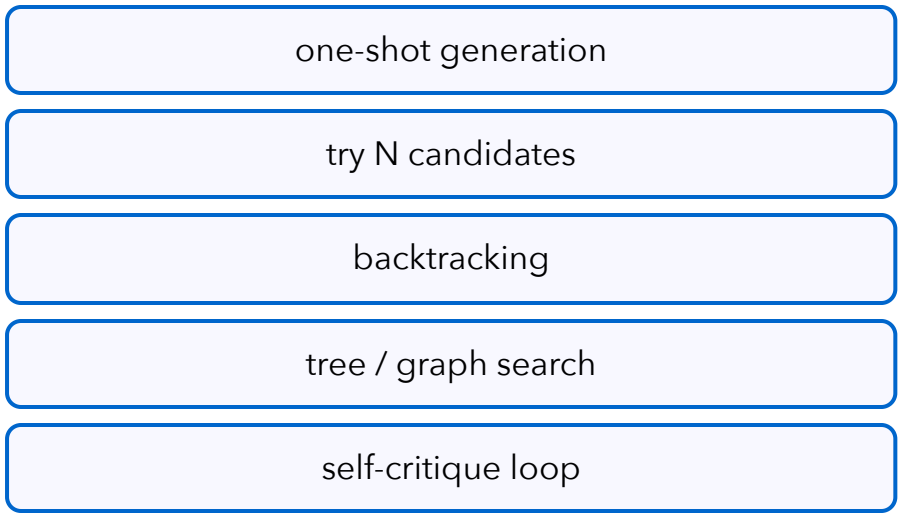
Experience memory

Seed-Prover 1.5 emphasizes learning from accumulated Lean/tool interaction experience.

Reusable engine operation

AXLE exposes checking, extraction, repair, and simplification as infrastructure.

Axis 4: What Kind of Search?



- COPRA: backtracking with proof-environment feedback.
- DeepSeek-Prover-V1.5: Monte Carlo tree-search variant.
- Aristotle: Monte Carlo graph search in a full system.
- Delta Prover: self-critique and spending more attempts at solving time.

> Search is how uncertain model proposals become systematic proof exploration.

Axis 5: What Domain Is Served?

Domain	Better examples to discuss
Natural-language proof and tutoring	NaturalProver, DeepTheorem, Mathstral
Lean test sets and competition math	DeepSeek-Prover-V2, Seed-Prover, LongCat, Aristotle, DAP
Research-level mathematics	Rethlas/Archon, AxiomProver Fel conjecture
Scientific verification	Ax-Prover
Proof engineering	Leanstral, Lean Copilot, AXLE

A Course-Scale Proof Agent

Build a small loop, measure it honestly.

Minimal Student Project

> Do not try to reproduce Seed-Prover or Aristotle. Build a small controlled proof agent and measure it.

Target

```
20 small Lean-friendly theorems
```

Method

```
retrieval + Lean-error repair loop
```

Baseline

```
one-shot Lean proof generation
```

Output

```
checked proofs or failure reports
```

What to Compare

one-shot **vs** retrieval **vs** repair **vs** retrieval + repair

Success

Does the final Lean proof type-check?

Cost

How many model calls and Lean calls were needed?

Failure

Was the problem formalization, retrieval, tactic generation, or repair?

A Minimal Proof-Agent Stack

LLM
proposer of plans, code, lemmas, and repairs

Retriever
mathematical memory outside the model

Lean
verifier and feedback source

Search
persistence under uncertainty

Logger
evidence, metrics, reproducibility

Human
intent, judgment, and repair of assumptions

A proof agent is a controlled loop, not a chatbot that writes proofs.