丘成桐数学科学中心
YAU MATHEMATICAL SCIENCES CENTER

Lean based AI for Math

From Formal Proofs to Verifiable Discovery

Rongge Xu
Dec 15, 2025

Towards an automatically verifiable, distributed scientific ecosystem

Motivation : Researcher's Perspective

- Frustrations with Traditional Proofs: In complex research, keeping track of every detail in a proof is challenging.
- Lean as a "Math Compiler": Lean connects human-like reasoning with strict formal logic. You write proofs in a structured language, and Lean can ensure rigor and catch subtle errors.
- Lean's ecosystem is most welcoming to mathematicians among proof assistants (Coq, Isabelle, Agda, etc.)

However

- Lean has a steep learning curve. Formal proofs can take ~10× longer than informal (Tao's estimate)

Getting started resources:

- *Theorem Proving in Lean4 tutorial*https://leanprover.github.io/theorem_proving_in_lean4/ ,
- *Lean Zulip chat for Q&A* https://leanprover.zulipchat.com/ *, etc.*

Background: Three Paradigm Shifts in Mathematics

- **Axiomatization (circa 1900):** Hilbert's Program – formalize all math in axioms and prove consistency https://plato.stanford.edu/entries/hilbert-program/

This era sought absolute rigor from first principles – every theorem should, in principle, be traceable to basic axioms. (Though Gödel later showed limits to this vision, the axiomatic approach transformed math)

- **Structuralism (1940–1980):** Bourbaki & Category Theory– Mathematics underwent a unification via abstract structures

The Bourbaki group advocated treating math as a network of structures and mappings, and Category Theory (initiated by Eilenberg & Mac Lane in 1945) became a language for unifying different fields. This structural view tamed the explosion of knowledge by organizing it into coherent frameworks.

- **Formalization (1994–present):** QED Manifesto – called for all mathematical theorems to be formally proven and machine-verified https://www.cs.ru.nl/~freek/qed/qed.html

Facing ever more complex proofs (some spanning hundreds of pages or relying on extensive computer checks), mathematicians recognized the need for automated proof checking.

When knowledge grows too complex, the solution is to **reinvent the framework** (a paradigm shift).

Motivation: Complexity Breeds New Paradigms

Now, this formalization wave is coinciding with the AI revolution.

AI + formalization = more than just rigor

AI generating conjectures +a system like Lean verifies or refutes them – we glimpse the possibility of **automated mathematical discovery**.

While we're not there yet, the convergence of formal proof systems and advanced AI is moving us step-by-step toward that future.

So the focus of the talk is to explore **AI for Math with Lean.**
By the end of the talk, you should see how Lean provides the rigorous backbone for mathematics and how AI tools can dramatically amplify what's possible on that backbone.

Table of contents

Foundations of Mathematics

Math can be formalized as **“axioms + rules”** using different frameworks, two major paradigms are Set Theory and Type Theory.:

- **ZFC Set Theory:** All mathematical objects are sets; uses ZF(C) axioms in first-order logic as a unified foundation.
- **Type Theory :** Everything is built from *terms* and *types*; (Curry–Howard correspondence) propositions ⇔ types and proofs ⇔terms.

Lean’s Choice – Dependent Type Theory (DTT): types can depend on terms. This foundation naturally expresses mathematical structures with precision.

Other Foundations:

- **HOL (Higher-Order Logic):** Based on simple type theory but allows quantifying over functions/predicates (i.e. variables of higher types); usually assumes classical logic and Choice.
- **ETCS / Category Theory:** Axiomatizes mathematics via category theory (e.g. an elementary topos with a natural numbers object), treating “everything as objects and morphisms”.
- **Homotopy Type Theory (HoTT):** Extends dependent type theory with *Univalence* axiom and higher inductive types, interpreting equality as a homotopy (paths) rather than a binary truth

Lean's Core – A Tiny Trusted Kernel

Lean's kernel = a dependent type λ-calculus with universes and inductive types

- The kernel's job is to *check* proofs: it type-checks terms and confirms definitions are logically valid (ensuring no inconsistencies).
- It's intentionally small– Lean 4's kernel is just a few thousand lines of C++ code lean4/src/kernel at master · leanprover/lean4 · GitHub. (Someone even rewrote the Lean4 kernel in Lean itself GitHub - digama0/lean4lean: Lean 4 kernel / 'external checker' written in Lean 4)
- *Curry–Howard in action:* In Lean, a theorem is literally a type, and a proof is a program that **inhabits** that type. The kernel verifies this program-proofs correspondence.

Math Compressed into a Kernel

Philosophical Remark:

- Modern engineering is increasingly **black-box and metric-driven**: systems get stronger, but their **guarantees** get fuzzier. Formalization makes those guarantees explicit and mechanically checkable, grounded in a **small trusted core** for final verification—shifting us from black-box performance to verifiable commitments.

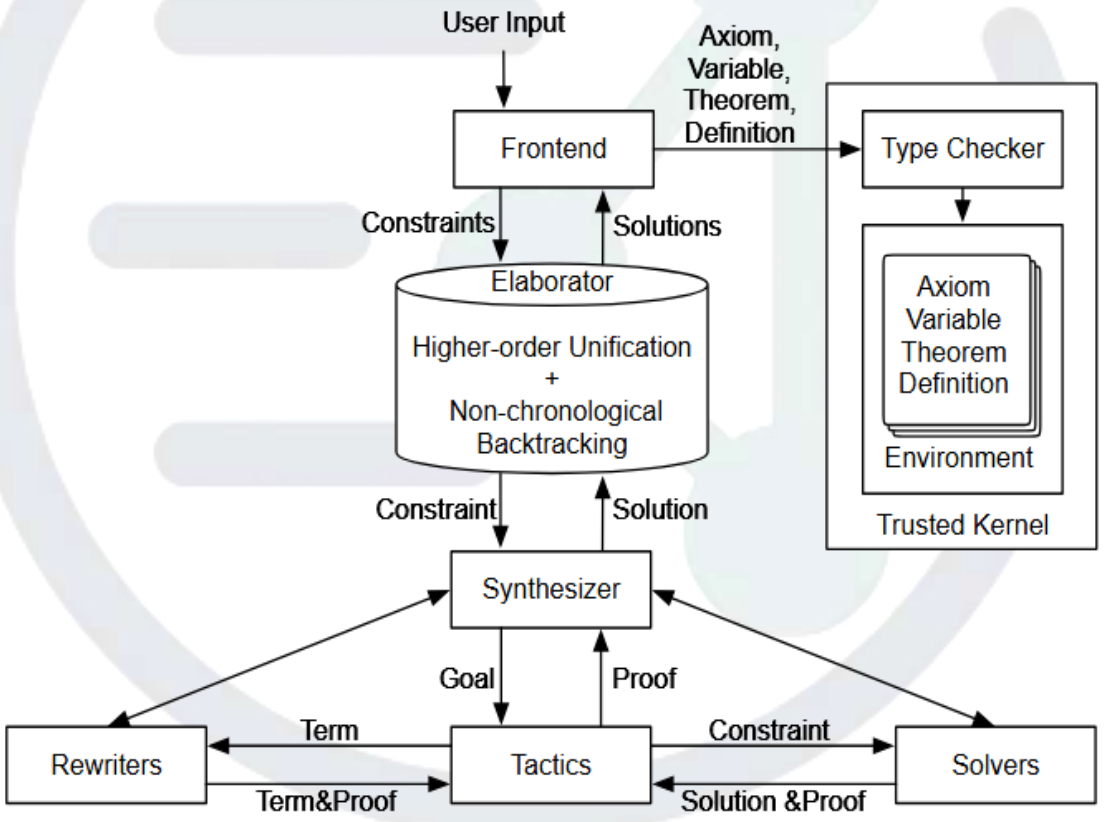
- Natural laws are written in mathematics; mathematics is enforced by that kernel. In this sense, a surprising amount of our universe's intelligibility is mediated by a small piece of logic. This deepens Wigner's insight on the "unreasonable effectiveness of mathematics" in the natural sciences: **the world obeys math, and math obeys a small formal core**.

Formalization isn't just a tool – it's a window into the logical structure of reason.

Lean's Architecture – Outer Layers

Lean = Kernel + everything else. Kernel is the trusted core. All other components are untrusted – they build proofs that the kernel later checks.

- **Tactics (Proof search):** Strategies or scripts to break down goals and build proofs automatically or semi-automatically.
- **Attributes & Hints:** Metadata on lemmas (e.g. @[simp], @[ext]) marking them for use in automation or special treatment.
- **Macros / Syntax Extensions:** Custom syntax and domain-specific languages that expand into core Lean terms.
- **Elaborator & Typeclass system:** Fills in implicit arguments, finds typeclass instances automatically.
- **Libraries and Plugins:** Organized libraries (like mathlib) with consistent conventions, plus performance tweaks
- …



https://www.cs.cmu.edu/~soonhok/talks/20131004/#/

Lean's Architecture

Lean = kernel +

parser/syntax + elaborator + tactics + libraries (mathlib, Physlean) + compiler, etc.

- **Metaprogramming:** We can extend or modify most of these outer parts freely: write new tactics, add syntactic sugar, plugins, attributes, etc., to make proof development smoother.
- **Lean + AI = natural fit:** This architectural choice is a big reason we can be confident in pushing the envelope with aggressive AI automation on top. As long as the kernel verifies the final proof, the result is 100% sound.

We mainly introduce AI on tactics (choose the next move) and libraries (supplies the moves).

Tactics: How Lean Constructs Proofs

In Lean, proving a goal is like solving a puzzle**. Goal = what you need to prove; tactics are moves that change the state.**

Typical proof process:

- Start with a goal.
- Apply a tactic (e.g. intro to assume an input, or apply to use an existing theorem) – this breaks or transforms the goal.
- Each tactic updates the tactic state: some goals are solved, new sub-goals may appear.
- Keep applying tactics to each sub-goal.
- When no goals remain, the proof is complete.

The process of finding a proof = **a sequential decision game** where you aim to reach the "goal achieved" state. This analogy suggests AI can be applied similarly to how it's applied in games.

Lean tactics cheat-sheet

`intro` / `intros`	Introduce assumptions / arguments into the context (optionally pattern-match them). Lean Language +1	core/std
`apply` / `refine`	Solve the goal by matching the conclusion of a lemma (with `refine` allowing placeholders / subgoals). Lean Language	core/std
`exact` / `assumption`	Close the goal with a term / an existing hypothesis of the right type. Lean Language	core/std
`cases` / `induction`	Case-split or do structural induction on an inductive term. Lean Language	core/std
`constructor`	Solve goals whose targets are constructors (e.g. $\wedge$, `Exists`, structures) by supplying fields. Lean Language	core/std
`simp` / `simp?` / `dsimp` / `simp_all` / `simpa`	Normalize by rewriting with simp-lemmas (with `?` to show what would happen; `dsimp` only unfolds defs; `simp_all` also uses context; `simpa` = `simp` + `exact`). Lean Language	core/std
`rw` / `rewrite`	Rewrite using equalities or equivalences (optionally at locations). Lean Language	core/std
`change`	Replace the goal with a definitional equal target to make it easier to attack. Lean Language	core/std
`revert` / `generalize`	Move hypotheses back into the goal / abstract a subterm by a fresh variable. [illegible] +1	core/std
`aesop` / `aesop?`	Rule-driven proof search; `?` prints a "try this" suggestion. Lean Prover Co... +1	mathlib
`linarith` (and `nlinarith` †)	Linear-arithmetic solver († `nlinarith` = nonlinear pre-processing). Lean Prover Co... +1	mathlib
`ring` / `ring_nf`	Normalise and solve equalities in commutative (semi)rings; `ring_nf` can normalize in subterms. Lean Prover Co... +1	mathlib
`norm_num`	Close numeric goals by normalising numerals (e.g. `2 + 2 = 4`, `3 < 15/2`). Lean Prover Co... +1	mathlib
`ext` (structure rule generator)	Extensionality support for structures (automates field-by-field equality). Lean Language	core/std
`gcongr`	Generalized congruence: reduce relational goals to sub-goals about inputs. Lean Prover Co...	mathlib
`simp_rw`	Like `rw`, but repeatedly and under binders, combining `simp`-style power with ordered rewrites. Lean Prover Co...	mathlib
`tauto`	Propositional tautology solver ($\wedge$, $\vee$, $\leftrightarrow$, $\exists$): splits/breaks and tries to finish. Lean Prover Co...	mathlib
`use`	Instantiate existentials / one-constructor inductives (friendlier than raw `exists`). Lean Prover Co...	mathlib

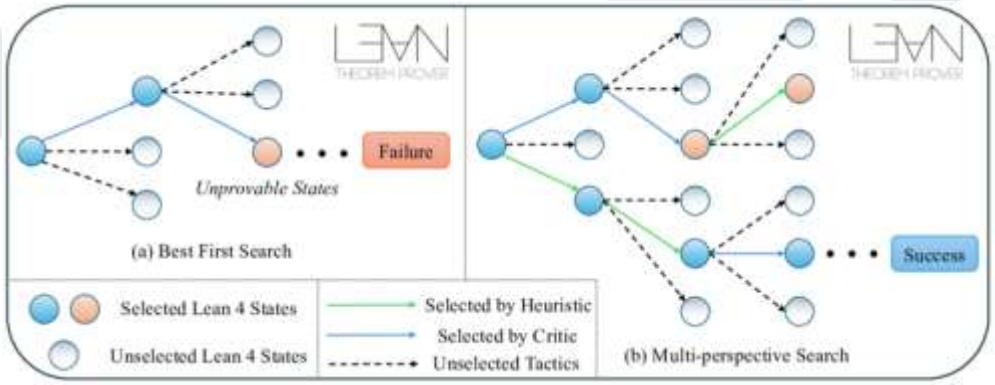
Automation Example – The aesop Tactic

Aesop is Lean 4's built-in heuristic prover (a tactic for automatic proof search). Think of it as an AI-lite inside Lean.

How aesop works:

- It treats proving like exploring a tree: **State =** current goals, **Action =** apply a rule (a lemma or introduction rule) to generate new subgoals.
- It scores each state using hand-crafted heuristics (some rules are considered more promising) and uses a **best-first search** – always expanding the most promising branch first.
- There's a budget (limits on steps or nodes) to prevent infinite search or explosion of possibilities.

Insight: Because Aesop's strategy is fixed, one improvement it to add an AI guiding those choices (e.g. an LLM suggests which rule to try next).



https://arxiv.org/pdf/2505.10962

Mathlib – Lean's Mathematical Library

Mathlib is Lean's community-driven library of formalized mathematics. It's the collective repository of proved theorems and definitions for Lean. https://github.com/leanprover-community/mathlib4

It covers a broad swath of math: from basics of set theory and algebra up through topology, geometry, analysis, number theory, and more. Many automation tactics and supporting tools live in mathlib too.

It's constantly growing, dozens of contributors worldwide add to it regularly.

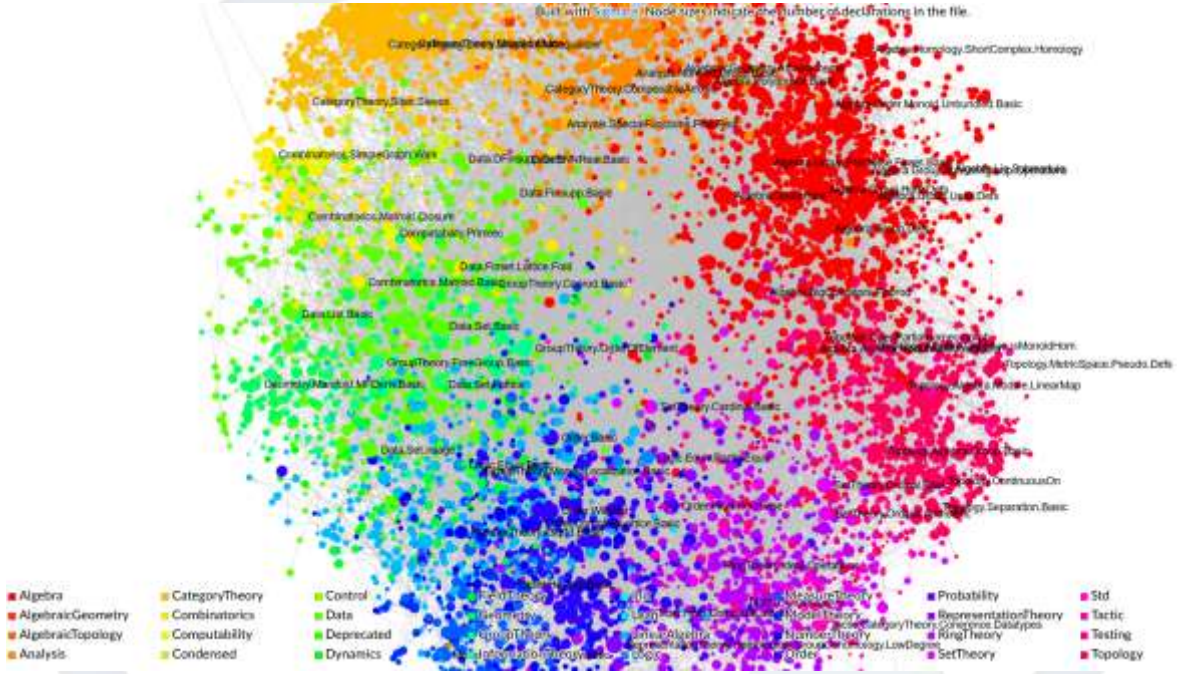
Mathlib statistics

https://github.com/leanprover-community/mathlib4/pulls

https://leanprover-community.github.io/undergrad_todo.html .

Insight: the data from this collaboration (like the history of proofs being fixed or refactored in PRs) is a rich resource. For example, the sequence of "fix proofs" in PRs provides real training data for AI on how to repair proofs and add lemmas.

Mathlib – A New Model for Collaboration



- Mathlib enforces a level of rigor and detail that traditional papers often gloss over. Every prerequisite must be explicit: no “it is obvious that…”

- Contributors can break big goals into many small lemmas, which can be tackled in parallel by different people. It’s like dividing a research project into bite-sized, verifiable pieces.

- Such an approach hints at a “distributed research” model: a large conjecture could be resolved by formalizing all its dependencies and splitting tasks among many mathematicians (or even AI agents).

Unlike traditional math research (where priority and personal credit are everything), mathlib is open-source: the shared goal is to build a common library of knowledge.

Interesting program: https://lean-lang.org/use-cases/flt/
https://www.cs.ru.nl/~freek/100/

Program: Classification of Finite Simple Groups (CFSG)

CFSG in Lean: value, difficulty, approach

- **Legend → auditable object:** CFSG is scattered across hundreds of papers; Lean would make it **one kernel-checked proof object**. It upgrades trust from expert consensus to mechanical verification

- **Wide web (CFSG) vs deep tower (FLT):** FLT stresses depth; CFSG stresses **scale**—a huge dependency graph plus massive finite-group infrastructure.
  Hard part: building missing background and a clean API in Lean.

- **How to do it:** Start with **infrastructure + milestones** (finite-group library, local analysis), then formalize major blocks modularly; **reuse/translate existing Coq/MathComp developments** (eg: **Feit–Thompson**), and use automation + project management to scale.

Mathlib's Growing Pains

With the growth of mathlib, **technical debt accumulates:**

.

- **Proof style variance:** Some proofs are "over-golfed" (made as short/clever as possible, but hard to read), while others are overly long and verbose.
- **Inconsistent naming & APIs:** Different contributors might name similar concepts differently. Sometimes multiple versions of essentially the same lemma exist with slight variations in naming or formulation.
- **Duplicate lemmas:** With many contributors, similar results get proved multiple times in slightly different forms.
- **Automation issues:** The automated tactics (like `simp`, `linarith`, `aesop`) have large rule sets which aren't always well-organized. Sometimes they try too hard or not enough:
- **Documentation gaps:** While mathlib has documentation, some lemmas have sparse descriptions ("only a name, no story"). Also, examples of how to use a theory may be lacking.
- **Style divergence:** Different subfields in mathlib were developed by different groups, leading to stylistic differences; etc.

PhysLean – A Library for Physics

PhysLean is an initiative to formalize physics using Lean. If mathlib is math's library, PhysLean aims to be the physics counterpart. https://physlean.com/

- The goal is to register physical theories and results so they can be checked and reused (made definitions, theorems, and derivations from physics formal, which also clarify the foundations of physics.
- PhysLean currently can't cover some complex or modern physics topics.

Directions:

- Even relatively straightforward models (e.g. the Kitaev chain in condensed matter) haven't been formalized – these could be good practices for interested contributors.
- Extending mathlib/PhysLean to cover more physics is a natural direction – e.g. category-theoretic physics, like topological quantum field theory, are completely untouched.

Vision: The boundary of formalization is clearly not "math only." Recent results such as https://pubs.rsc.org/en/content/articlelanding/2024/dd/d3dd00077j show that any scientific discipline can mechanize the part that is precisely stated: its assumptions, models, and derivations. What remains is experiment and engineering. This is what we call **Verifiable Discovery.**

AI in Formal Mathematics

Learn broadly, choose precisely; accumulate deeply, deliver rigorously.

(AI)博观而(Lean)约取，(AI)厚积而(Lean)薄发

- AI's strength: generating guesses and patterns. Lean's strength: checking rigorously for correctness. They naturally complement each other.
- 50 awesome paper on AI for math in recent years https://seewoo5.github.io/awesome-ai-for-math/

Lean based AI for Math Explain

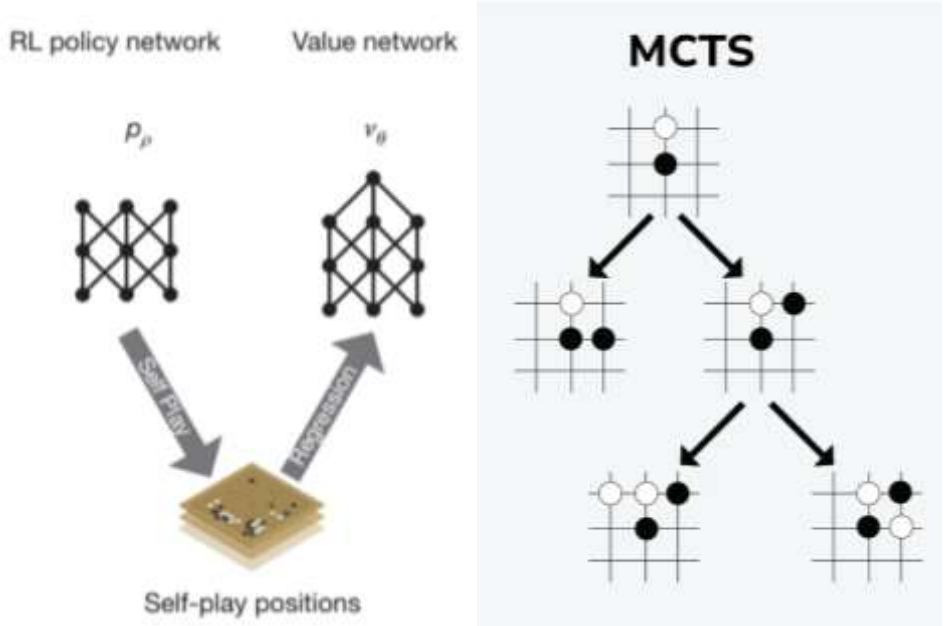
Lean based AI for Math ≈ LLM + RAG (retrieval) + search & planning + verification kernel.

- **LLMs** are trained for next-token prediction and minimizing cross-entropy. Most use Transformer architectures, where attention acts like context-aware retrieval to focus on relevant inputs. Their strength is absorbing vast textual patterns: language, facts, and rough problem-solving "intuition," enabling natural-language/code generation and tentative proof steps. The weakness is hallucination, especially in sparse domains (Lean proofs are far rarer than Python). Notably, some "hallucinations" can spark creative ideas—if later verified.

- **RAG**. Before an LLM answers or proposes a proof step, it first retrieves relevant facts from external sources (e.g., a theorem/definition index, mathlib docs). In formal math, this means looking up lemmas/definitions instead of expecting the model to memorize or reinvent them. RAG turns free-form generation into evidence-grounded generation: the LLM acts like a composer that weaves known facts, reducing hallucinations, improving reliability, and enabling citations and better correctness than from-scratch arguments.

Self-Play and Learning

- **Search/Planning:** A proof is a long, interdependent sequence of steps—so it's a **planning** problem. Model it as **state** = current goals/context, **action** = a tactic, **transition** = applying it to get new goals. This yields a huge **search tree**. We guide exploration with **rewards** and algorithms like **MCTS** or **RL**.



Silver et al. arXiv:1712.01815

- Eg: AlphaGo had: - a policy network (to suggest moves), - a value network (to estimate who's winning from a state), and MCTS to combine them with random exploration.
- We can analogously have: - a policy model (maybe an LLM fine-tuned to suggest next tactics given the Lean state), - a value model (maybe another model that predicts the probability a given goal is provable or how many steps left), - and then do a search that tries out tactic sequences, prunes ones that look bad, deepens ones that look good, etc.

Notably, AlphaGo started by learning from human games, then achieved superhuman play via self-play. Likewise, an AI prover might first learn from human proofs, then surpass the best trained mathematician by exploring proofs without human patterns.

Lean based AI – Emerging Results

Retrieval/Search (finding theorems, premises, semantic search)

- LeanSearch semantic search (Mathlib4 + API; there's a client integrated into Lean). This was developed by Professor Dong Bin's team at Peking University. The community blog also lists it as a mainstream search engine. (https://leansearch.net/)
- Loogle (finds lemmas by type signature/subformula, like Hoogle; has a website, VSCode/CLI/Neovim integration) (https://loogle.lean-lang.org/)
- Moogle (LLM-driven semantic search for Mathlib; one of the engines listed on the Lean community blog) (https://www.moogle.ai/)

LeanSearchClient (use LeanSearch/Moogle/LeanStateSearch/Loogle directly from Lean) VSCode integration

- LeanExplore (semantic search across multiple Lean4 packages [Lean/Std/Batteries/Mathlib/PhysLean]; provides a website, Python API, and MCP interface for easy integration with agents) (https://www.leanexplore.com/)
- Lean State Search (uses the "current proof state" as a query to search for premises and lemmas, for Lean4 users) (https://premise-search.com/)

"Next Tactic Suggestions" / Automated Proof (LLM + Search)

- LLMSTEP (displays "Next Tactic" suggestions in Lean4; initial release in 2023) — Clear principles and easy to use, but relatively outdated compared to the rapidly evolving mathlib4. (https://arxiv.org/abs/2310.18457)
- Lean Copilot (LeanDojo team): Suggest tactics, select premises, and search for proofs within Lean. It can use built-in models or connect your own local/cloud-based models. The paper also compares the `suggest_tactics` and `search_proof` modes. (https://github.com/lean-dojo/LeanCopilot)
- REAL-Prover (Retrieval Augmented Lean Prover): A step-by-step prover for Lean 4 that integrates semantic search (LeanSearch-PS). It is competitive on benchmarks such as ProofNet and miniF2F. It also introduces the FATE-M benchmark (see the next category). (https://arxiv.org/abs/2505.20613)
- ReProver (a "retrieval-enhanced prover" provided by LeanDojo, which retrieves a small number of highly relevant premises from the proof state and then regenerates tactics) (https://arxiv.org/abs/2306.15626)
- Seed-Prover (ByteDance Seed Team): A Lean 4 "lemma-to-whole proof" automated prover. It first breaks down reusable sub-lemmas, then iterates repeatedly using Lean feedback and self-reflection. It achieves strong results on benchmarks such as IMO, Putnam, and MiniF2F. (https://arxiv.org/pdf/2507.23726)
- ProvingPantograph (a Lean 4 "machine-to-machine" interface, supporting search and high-level reasoning such as MCTS; suitable as a search backend for AI provers) (https://arxiv.org/abs/2410.16429)
- GFLean (a Lean automatic formalization framework based on the Grammatical Framework: parses simplified English mathematical statements into Lean code; focuses on the "turning problems into Lean statements" aspect) (https://arxiv.org/abs/2404.01234)

Lean based AI – Emerging Results

Datasets/Benchmarks and Question Banks

LeanDojo (Data and Interaction Platform): Extracts fine-grained data such as ASTs, tactics, and prerequisites, and transforms Lean into a "gym-like" environment for programmable interaction; they used this data to train ReProver. (https://leandojo.org/leandojo.html)

FATE-M (Formal Algebra Theorem Evaluation-Medium, a benchmark for advanced algebra; introduced by the REAL-Prover paper) (https://arxiv.org/pdf/2505.20613)

DeepMind Formal-Conjectures (only formalizes the "statement of the conjecture," not the proof; an open benchmark and resource for automatic proofs/automatic formalization, the repository is still actively updated) (https://github.com/google-deepmind/formal-conjectures)

*Other commonly used:

ProofNet (an automatic formalization and proof benchmark for undergraduate mathematics), miniF2F (a cross-system benchmark for Olympiad/high school difficulty) (https://arxiv.org/abs/2302.12433)

We lack benchmarks that test definitions→lemmas→theorems pipelines in abstract algebraic/categorical settings.

Other

Lean4trace (Data extraction and augmentation for training neural provers) (https://openreview.net/pdf)

LeanInteract (A Python package for interacting with the Lean REPL, making it easy to run agent/scripted experiments) (https://github.com/augustepoiroux/LeanInteract)

Lean-STaR (A training paradigm: Generates "informal ideas" before each tactic, significantly improving the pass rate of Lean proofs; can be combined with the above tools) (https://arxiv.org/abs/2407.10040)

Lean benchmark on 1-Cat

1-Cat Benchmark

LeanBridge

December 10, 2025

1 Base Benchmark

[ ] easy
[*] medium
[**] hard

1.1 Basic

Problem 1. — *Let $\mathcal{C}$ and $\mathcal{D}$ be two categories. Let $F : \mathcal{C} \to \mathcal{D}$ be a functor. Then for any $x, y \in \mathcal{C}$ and any $f \in \mathrm{Mor}_{\mathcal{C}}(x, y)$ such that f is an isomorphism, $F(f) \in \mathrm{Mor}_{\mathcal{D}}(F(x), F(y))$ is an isomorphism.*

```
/--
Let $\EuScript{C}$ and $\EuScript{D}$ be two categories. Let
$F:\EuScript{C}\to\EuScript{D}$ be a functor. Then for any
$x,y\in\EuScript{C}$ and any $f\in\mathrm{Mor}_{\EuScript{C}}(x,y)$ such
that $f$ is an isomorphism, $F(f)\in\mathrm{Mor}_{\EuScript{D}}(F(x),F(y))$
is an isomorphism.
-/

-- Model: gpt-5.1

import Mathlib

open CategoryTheory

variable {C D : Type*} [Category.{v} C] [Category.{v} D]

/-- If `f` is an isomorphism in `C`, then `F.map f` is an isomorphism in `D`.
 -/
lemma functor_map_isIso_new (F : C ⥤ D) {X Y : C} (f : X ⟶ Y) [IsIso f] :
    IsIso (F.map f) := by
  -- This is exactly `Functor.map_isIso`.
  simpa using (F.map_isIso f)
```

Problem 2. — *The forgetful functor $\mathbf{Top} \to \mathbf{Set}$, $\mathbf{Grp} \to \mathbf{Set}$, $\mathbf{Ring} \to \mathbf{Ab}$, $\mathbf{Top}_* \to \mathbf{Top}$ are faithful but not full.*

```
/--
The forgetful functor $\EuScript{T}\mathrm{op}\to\EuScript{S}\mathrm{et}$,
$\EuScript{G}\mathrm{rp}\to\EuScript{S}\mathrm{et}$,
```

LeanBridge – A Multi-Agent Proof System

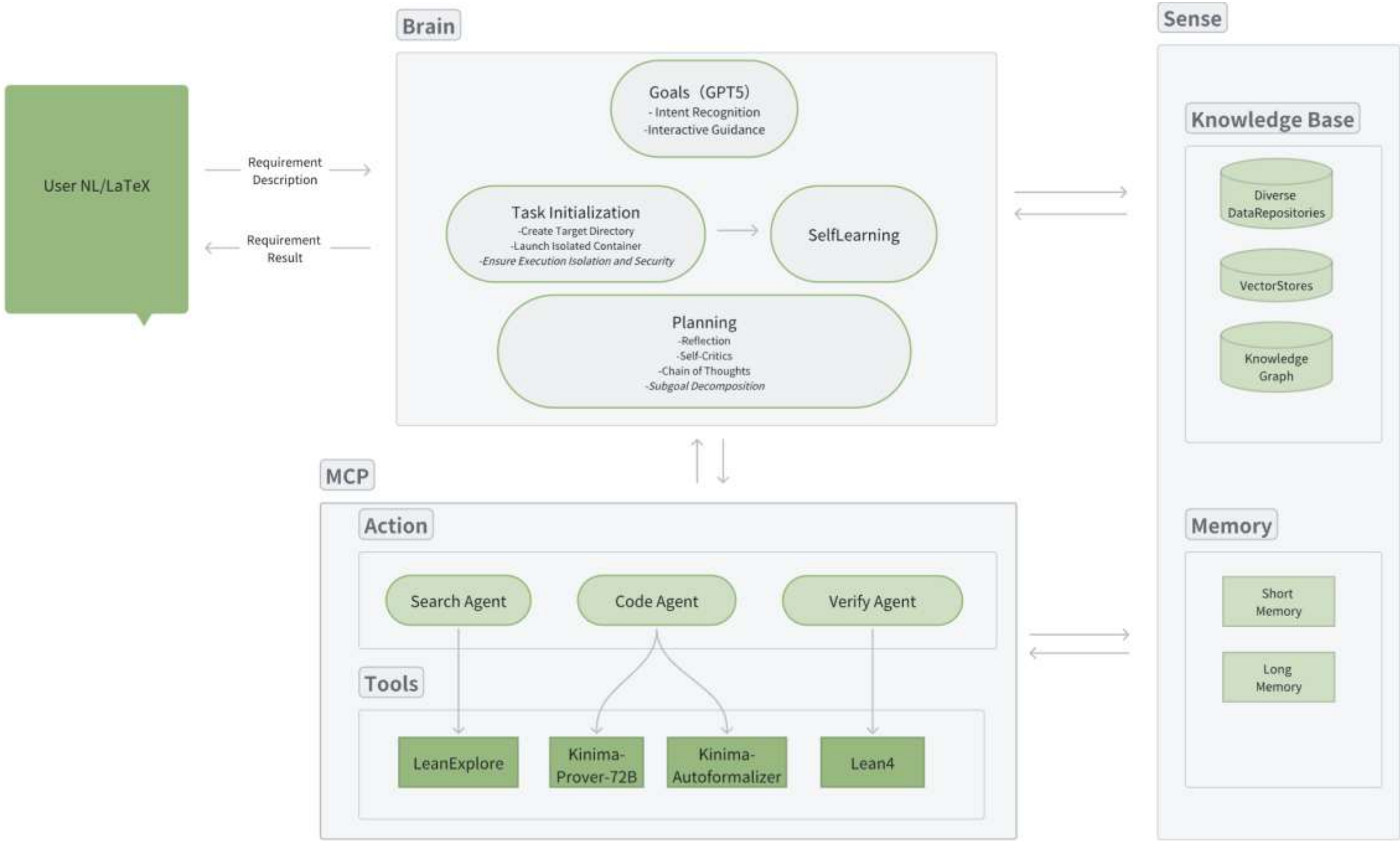
LeanBridge is a closed-loop: **Planner -> Retriever -> Coder -> Verifier,** and back around if something fails.

A demo system our team built that won a recent AI4Science competition https://ai4.science/events/beijing-hackathon#. It uses multiple specialized **agents** working together to solve proof tasks.

Movitation:

- Problem: General LLMs lack Lean-specific data—drafts often fail.
- Pain: I was the "courier," copy-pasting compiler errors between models.
- Solution: **MCP**-driven auto-feedback, multi-model orchestration, iterative repair—close the loop end-to-end.

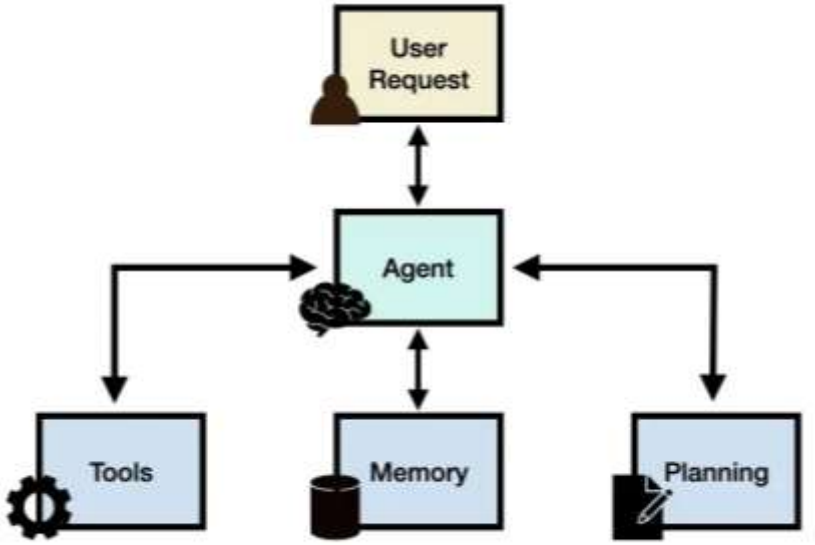
- Planner: Takes a user's request (in natural language or high-level terms) and breaks it into a structured plan or tasks.
- Retriever: Gathers relevant definitions, lemmas, or data from mathlib/PhysLean or even academic papers to support the tasks.
- Coder: Generates Lean proof code for the tasks (essentially an AI proving the subgoals).
- Verifier: Checks the code by compiling/running in Lean, and provides feedback (errors, counterexamples).


Brain
Sense
User NL/LaTeX
Requirement Description
Requirement Result
Goals（GPT5）
- Intent Recognition
-Interactive Guidance
Task Initialization
-Create Target Directory
-Launch Isolated Container
-Ensure Execution Isolation and Security
SelfLearning
Planning
-Reflection
-Self-Critics
-Chain of Thoughts
-Subgoal Decomposition
Knowledge Base
Diverse DataRepositories
VectorStores
Knowledge Graph
MCP
Action
Search Agent
Code Agent
Verify Agent
Tools
LeanExplore
Kinima-Prover-72B
Kinima-Autoformalizer
Lean4
Memory
Short Memory
Long Memory

LeanBridge – Agent

An **agent** is an autonomous loop around a goal, with:

- **Policy/Brain:** rules, LLMs, or policy nets to decide the next step.
- **Perception: reads the environment/context** (e.g., proof state, errors).
- **Action: invokes tools** (retrieve, propose tactics, call Lean).
- **Memory:** writes outcomes to **short-/long-term memory** for the next round.
- **Closed Loop: uses feedback** (success/failure/reward) to continue, backtrack, or stop.

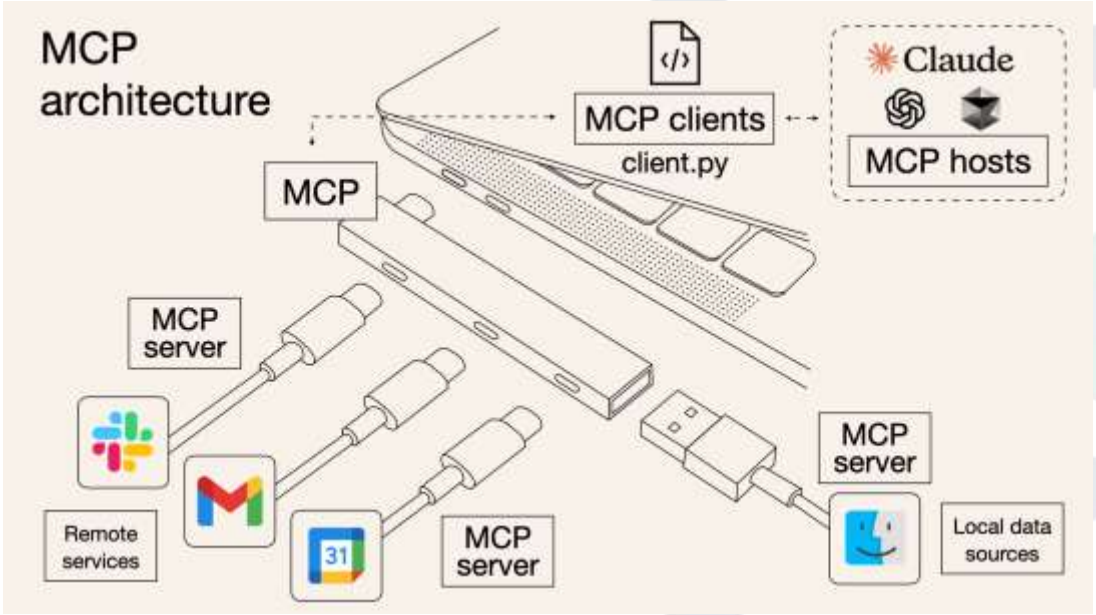


https://blog.gopenai.com/5-phases-of-llm-agents-involves-research-points-application-scenarios-and-development-directions-10ae4deec4e2

Proofs are long-horizon tasks: decompose subgoals, **repeat retrieval + attempts + backtracking**. A **single LLM call isn't enough**—you need **persistent state and memory**. That's the agent pattern.

LeanBridge – MCP

All these communicate via MCP (Model Context Protocol) – a standardized way for agents/tools to share state and results.



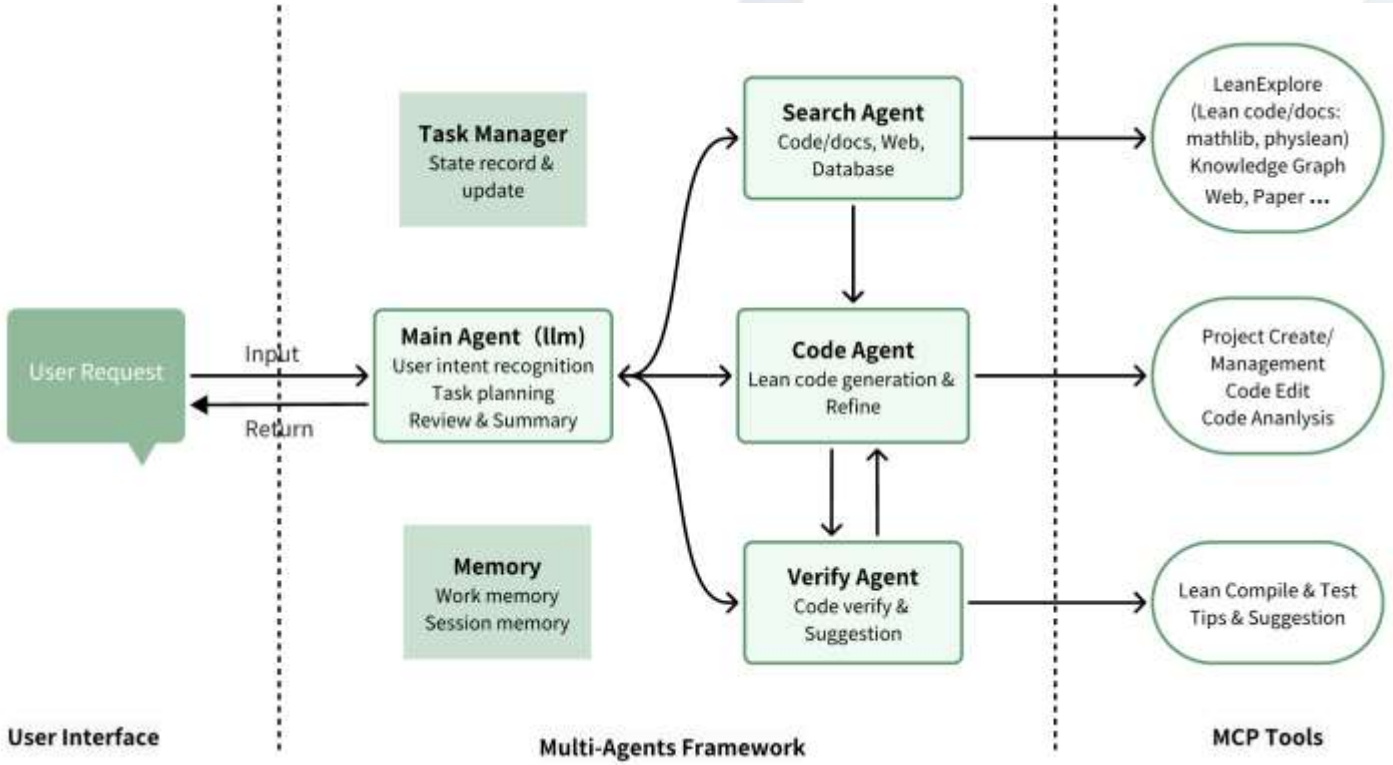
https://dev.to/pavanbelagatti/a-hands-on-guide-to-model-context-protocol-mcp-5hfo

MCP is a general-purpose protocol that gained momentum late last year, originating from an article published by Anthropic: Introducing the Model Context Protocol https://www.anthropic.com/news/model-context-protocol . Anthropic uses it to connect LLM to external data and tools (think of it as a "USB-C interface for AI").
In LeanBridge, MCP acts as a common language or bus.

Multi-Agent Architecture

We scale from a single loop to a coordinated swarm: specialized agents share memory and a task queue over MCP.

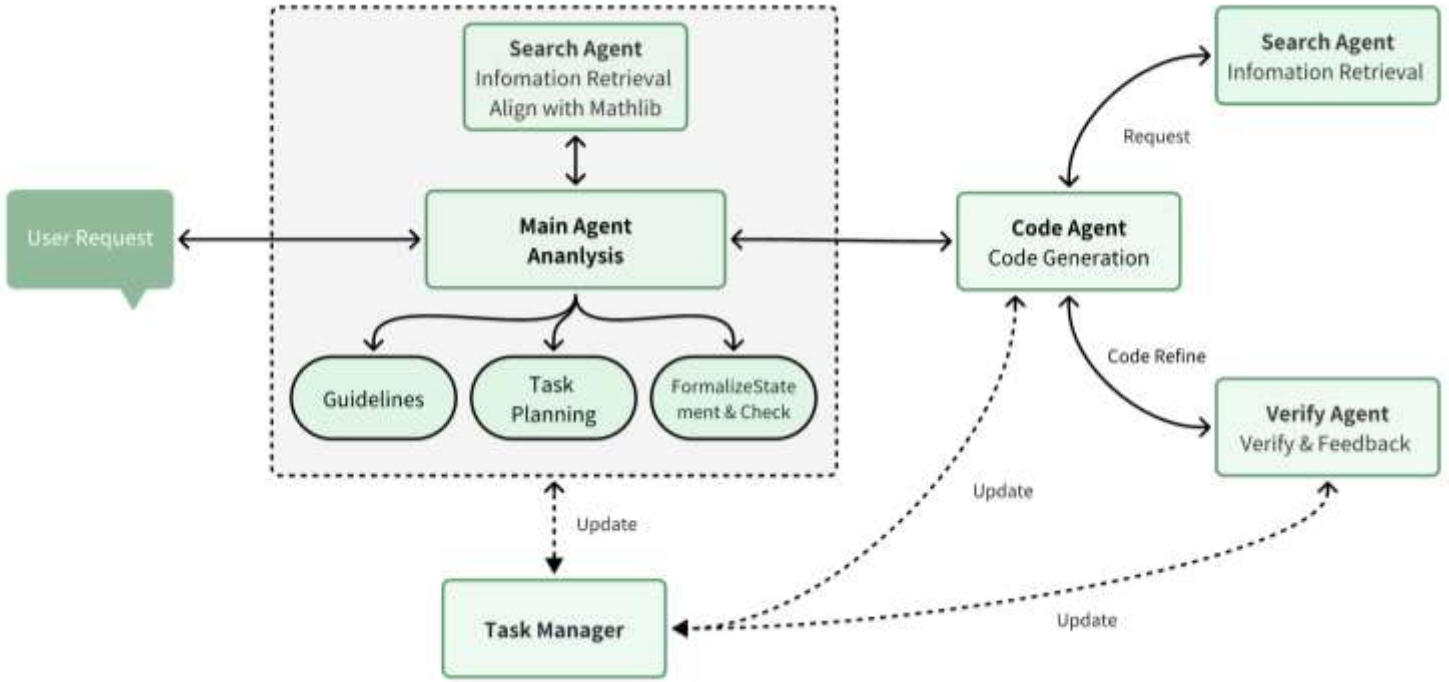
Multi-agent Parallelism:
Separation of concerns, auditability (Task Manager trace), reliability (kernel-gated outputs).



Statement Check and Search Aligned

Statement verification (LeanBridge): We back-translate the generated formal statement into natural language and run multi-model voting to check semantic alignment with the original intent.

Search aligned with mathlib: Our retrieval and proof search are biased toward existing mathlib lemmas and canonical definitions, minimizing new ad-hoc primitives and maximizing reuse.



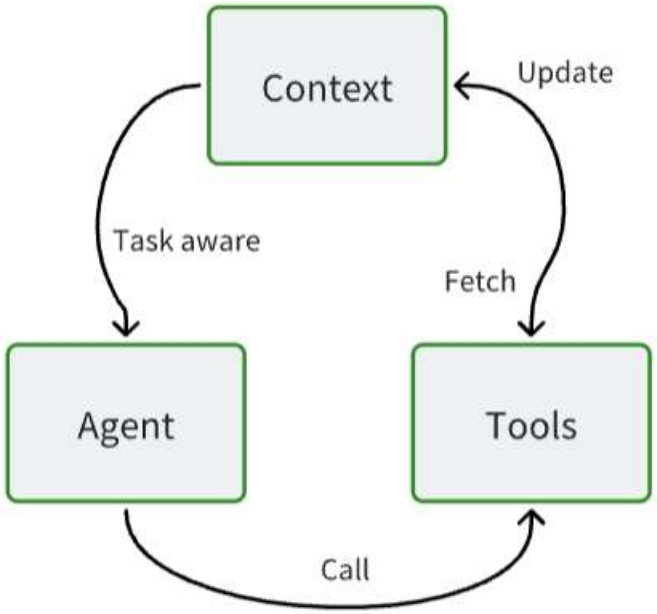
Agent Context

```
agent_context = {
    'statement': {'statement_id': {'user_statement':'', 'formalized_statement':''}},
    'search': {'search_id':{'info_type':'', 'search_results':''}},
    'current_task': 'statement_id',
    'task':{}
}
```

```
'task':{
    'statement_id': {
        'strategy':'',
        'search_id':'search_id',
        'solved':False,
        'max_attempts':3,
        'attempt':0,

'current_code_id':'code_id',
        'code':{}
        }
    }
```

```
'code':{
    'code_id':{
        'code':'',
        'has_exec_verification':False,
        'pass_verification':False,
        'verification_message':'',
        'verification_advise':''}
    }
```



	Statement				Proof					Difficulty		
	GPT5.1	Gemini3pro	Deepseek	Kimi	GPT5.1	Gemini3pro	Deepseek	Kimi	Grok4.1	LLM	Human	Average
Weight	0.4	0.4	0.4	0.4	2	2	2	2	2	0.5	0.5	
Prb.1										10	5	7.5
Prb.2										9.6	3	6.3
Prb.3										8.8	7	7.9
Prb.4										7.2	5	6.1
Prb.5										3.6	1	2.3
Prb.6										9.6	2	5.8
Prb.7										10	3	6.5
Prb.8										8	4	6
Prb.9										5.2	3	4.1
Prb.10										9.2	5	7.1
Prb.11										4.8	3	3.9
Prb.12										10	7	8.5
Prb.13										3.6	4	3.8
Prb.14										9.6	6	7.8
Prb.15										10	7	8.5
Prb.16										9.6	6	7.8
Prb.17										5.2	5	5.1
Prb.18										9.2	4	6.6
Prb.19										8.4	6	7.2
Prb.20										8.8	6	7.4
Prb.21										5.6	4	4.8
Prb.22										9.6	7	8.3
Prb.23										10	5	7.5
Prb.24										5.2	3	4.1
Prb.25										8.4	2	5.2
Prb.26										9.2	4	6.6
Prb.27										8	4	6
Prb.28										7.2	3	5.1
Prb.29										9.6	7	8.3
Prb.30										5.6	4	4.8

A bridge language for portable formal assets

- **Problem: formal assets are siloed**
  Coq/MathComp, Lean/mathlib, Isabelle… each has huge verified libraries, but they don't compose.

- **Idea: an intermediate representation (IR) above provers**
  A *proof portability layer* that stores **statements + definitions + dependency DAG + proof skeletons** (not a new kernel).

- **Plausible Coq → Lean :**
  - **Align statements & structure**
  - **Map APIs, not scripts**
  - **Translate proof skeletons selectively**
  - **Iterate with CI**

Train a stronger model

Combine Kimina's internal exploration with retrieval-augmented search (REAL-Prover) 2504.11354 and robust LeanDojo [2306.15626] LeanDojo: Theorem Proving with Retrieval-Augmented Language Models tooling to push beyond Olympiad math into research-grade formalization.

1. **Data & tooling**
2. **Stage A — Minimal SFT**
3. **Stage B — Verifier-in-the-loop RL** (terminal win (+1 on QED), shaped signals for subgoal closure, penalize kernel errors/timeouts; optionally bonus for shorter proofs.)
4. **Add retrieval & tools**
5. **Distill & scale smartly**
6. **Evaluation**

AI mining Formal Theorems

Building knowledge graph for Mathlib

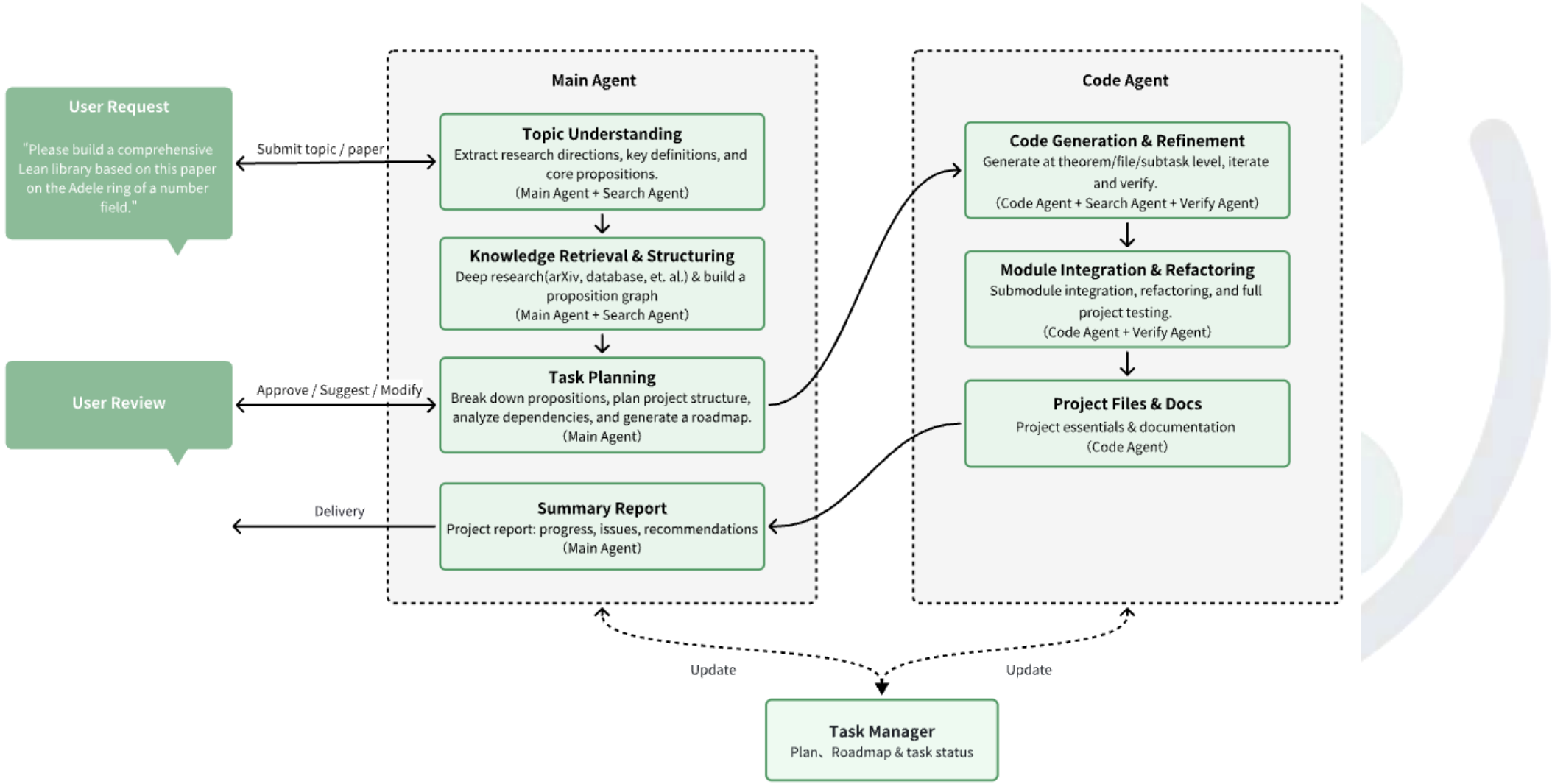
Scale need: mathlib already has **200k+ theorems**—navigation is the bottleneck.
Theorem→Theorem Graph for mathlib:

- **Precise retrieval** during interactive proving (follow *uses/used-by* edges, not just keywords).
- **Long-range composition** & alternative formulations (surface distant bridges in the graph).
- **De-dup & governance** (spot redundant lemmas via clusters/near-dupes). leanprover-community.github.io

Then Use the graph to drive AI mining

•**Find "gaps"** = pairs of nodes that should connect but don't; propose the missing lemma and **target it with autoformalization pipelines**.
•**Let agents read the literature**: retrieve candidate results from papers, align them to gap edges, and formalize—closing the loop at scale. Surveys and recent systems outline the workflow.
•**Evaluate end-to-end** on Lean environments (retrieval → code gen → kernel check), aim AI at to grow the library.

BridgeLib： A Lean Library Builder Example



Multimodal, verifiable research stack

Use Sage to explore and compute; use Lean to certify and integrate.

Lean × SageMath (and friends): a multimodal, verifiable research stack

For homology/spectral sequences: plug in **Kenzo** (constructive alg. topology, spectral sequences), **GAP+HAP** (group cohomology, LHS spectral sequence), **Macaulay2: SpectralSequences** (AG/CA workflows). Sage can orchestrate these; Lean verifies.

Why this helps physicists & higher math: pragmatic **compute→formalize** loop; covers most **discrete/algebraic** tasks out-of-the-box; scales to topology/SS via Kenzo/HAP/M2.