Training Models to Prove Theorems

AI for Mathematics (General Elective) — Week 12

23 June 2026

Purpose-trained provers

A purpose-trained prover —DeepSeek-Prover, Goedel-Prover, Seed-Prover, Kimina-Prover, LongCat-Flash-Prover, Leanstral, OProver —is a language model whose training data is overwhelmingly Lean proofs, Lean tactic traces, and Lean error feedback. The proof discipline is built into the weights.

The model knows the syntax. It knows the standard library. It knows the elaborator's failure modes. In the most recent systems it also knows the `lean-lsp-mcp` tool catalog itself, because it was trained on rollouts of those tool calls.

What the model is trained on

Training data —overwhelmingly Lean, in three forms:

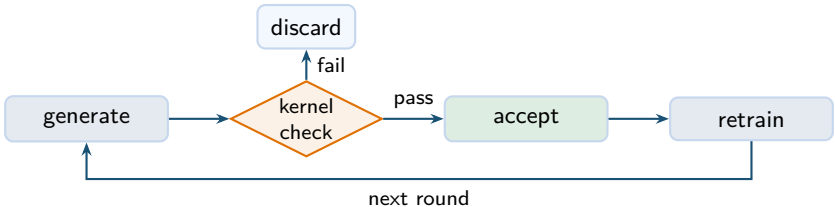
- complete kernel-checked proofs;
- tactic traces: (proof state, next tactic) pairs;
- kernel and elaborator error messages from failed attempts.

Recent systems add rollouts of `lean-lsp-mcp` tool calls.

Training signal —the Lean kernel verdict. A proof either passes the kernel or it does not; this binary, decidable result is the reward.

The expert-iteration loop

Every modern trained prover, GPT-f (2020) to OProver (2026), sits on one shape:



LM proposes a proof → Lean accepts or rejects → keep only kernel-verified → fine-tune on the accepted set.

Parts 1—6 each vary one or more of these stages —policy, search, problem pool, reward, decomposition.

Trained to call Lean tools

By 2026 LongCat-Flash-Prover, Leanstral, and OProver are trained to emit `lean-lsp-mcp` tool calls inside their rollouts: the model calls Lean tools because those trajectories were in its training distribution.

The tool-call discipline lives in the weights, not in an inference-time wrapper bolted on around the model.

These are the same `lean-lsp-mcp` tools covered in Week 11 Part G —here they are part of what the model was trained to use, rather than a catalog an external loop hands it.

Roadmap

Part 1 GPT-f $\rightarrow$ LeanDojo $\rightarrow$ ReProver: the expert-iteration loop

Part 2 DeepSeek-Prover v1 $\rightarrow$ v1.5 $\rightarrow$ v2: three open releases

Part 3 AlphaProof: IMO 2024, 28/42

Part 4 IMO 2025: Seed-Prover and Aristotle

Part 5 Trained to use Lean tools: Kimina-Prover, LongCat-Flash-Prover, Leanstral

Part 6 OProver, Goedel-V2, Aleph, and the variations survey

Part 7 The verifier question; open problems

Part 1 · The Origin

GPT-f (OpenAI, 2020)

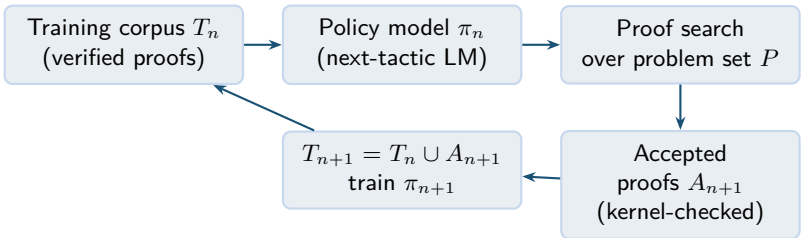
Polu and Sutskever, arXiv:2009.03393. Proof assistant: **Metamath** (not Lean —the choice mattered less than the loop).

The expert-iteration loop —four pieces:

1. Train a language model on *(goal state, next tactic)* pairs
2. Best-first search expands the proof tree: model proposes K candidate tactics; the proof assistant runs each
3. A path reaching `no goals` is a closed proof
4. Closed proofs become new training data $\rightarrow$ fine-tune $\rightarrow$ go to 2

Step 4 is what makes the loop self-reinforcing: the model proves what it can; the next model trains on those proofs; the next model proves a strictly larger set. Only proofs the verifier accepts become training data.

GPT-f —one round of expert iteration



Later provers differ at the boxes of this loop: which model generates the next tactic, how the proof tree is searched, which problems make up the pool, what counts as an accepted proof, and whether RL is added on top of supervised fine-tuning. A given system changes one or two of these and keeps the rest fixed.

GPT-f —the template later systems reuse

Every system in this lecture sits on top of the GPT-f template:

- **DeepSeek-Prover v1**: GPT-f scaled by two orders of magnitude
- **v1.5**: adds RL on top of expert iteration
- **v2**: adds subgoal decomposition (have-chain)
- **AlphaProof**: GPT-f with AlphaZero-style MCTS over the search
- **Seed-Prover**: whole-proof generation, keeps the iteration loop
- **Kimina-Prover**: scales the policy model, keeps the loop
- **LongCat-Flash**: trains tool-use into the policy
- **OProver**: trains the entire harness loop into the policy

What was missing in 2020: a shared Lean dataset, a shared interaction harness, a shared benchmark. The next system fixed all three.

LeanDojo + ReProver (Yang et al., 2023) —overview

Yang et al., *LeanDojo: Theorem Proving with Retrieval-Augmented Language Models*, arXiv:2306.15626, **NeurIPS 2023** (Datasets & Benchmarks, oral).

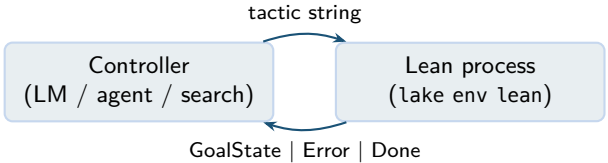
Three contributions, any one of which would have been a paper:

1. **The benchmark** —98,734 theorems from Mathlib; 217,776 tactic steps; 129,243 with resolved premise annotations; three splits
2. **The harness** —a Python `Dojo` class that wraps a Lean process and exposes typed tactic interactions
3. **ReProver** —a retrieval-augmented tactic generator trained on the data

LeanDojo
Yang et al., 2023

LeanDojo —the Dojo interaction harness

Architecture: a Python `Dojo` wraps a Lean process. The controller sends a tactic string; Lean returns a new goal state, an error, or `ProofFinished`.




```
with Dojo(theorem) as (dojo, init_state):
    state = dojo.run_tac(init_state, "intro h")
    state = dojo.run_tac(state, "exact h.left")
    assert isinstance(state, ProofFinished)
```

Conceptual ancestor of `lean-lsp-mcp` (Week 11 Part G). **Lean is a server; the prover is a client; every interaction is a typed message.**

LeanDojo —the benchmark and premise annotations

The benchmark (three splits):

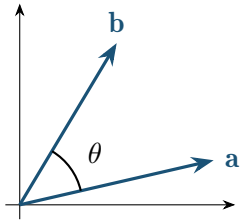
- **Random split** —interpolation; 98,734 theorems from 3,384 Mathlib files
- **Novel-premises split** —test theorems use at least one premise never seen in training (tests generalization, not recall)
- **miniF2F-test** —ported into the LeanDojo harness for cross-system comparison

The premise annotations: 129,243 tactic steps carry resolved premise names linking each tactic to the named lemmas it applied.

The annotation is load-bearing: it makes premise selection trainable as **supervised learning** rather than emergent behavior. Every Lean prover paper since 2023 reports on at least one of these splits.

What is cosine similarity?

An embedding turns a state or a premise into a **vector** — a list of numbers, i.e. a direction in a high-dimensional space.



Cosine similarity of two vectors $\mathbf{a}, \mathbf{b}$ is the cosine of the angle θ between them:

$$\cos\theta \;=\; \frac{\mathbf{a}\cdot\mathbf{b}}{\|\mathbf{a}\|\,\|\mathbf{b}\|} \;=\; \frac{\sum_i a_i b_i}{\sqrt{\sum_i a_i^2}\,\sqrt{\sum_i b_i^2}}.$$

- $= 1$: same direction — most similar.
- $= 0$: at right angles — unrelated.
- $= -1$: opposite directions.

It compares **direction, not length**: two vectors pointing the same way score 1 no matter how long they are. Ranking premises by cosine to the state embedding picks those whose embedding points most nearly the same way as the goal.

ReProver —retrieval-augmented tactic generation

Two ByT5 models on the same backbone:

Retriever (encoder-only):

- Embeds goal state s
- Embeds all Mathlib premises p_i (pre-indexed)
- Returns top-K premises by cosine ($K \approx 100$)

Hard negatives (key novelty): premises *accessible* at the current proof position but not used by the human annotator. Only LeanDojo's program analysis makes this possible.

Tactic generator (encoder-decoder):

- Input: goal state + top-K retrieved premises
- Output: next tactic string

Training cost: ≈one GPU-week on a single NVIDIA A100. The paper's framing was deliberate: **anyone with a single GPU can reproduce and extend.**

ReProver —benchmark numbers

Pass@k: a theorem counts as solved if any of k sampled attempts is kernel-verified. Here Pass@1, 10-minute budget per theorem:

System	Random	Novel-premises	miniF2F
`tidy` (Mathlib hammer-ish)	23.8%	5.3%	—
GPT-4 zero-shot	29.0%	7.4%	—
ReProver **without** retrieval	47.6%	23.2%	—
ReProver (full, retrieval)	**51.2%**	**26.3%**	**26.5%**

Three readings:

- Retrieval ablation: +3.6 pp random, +3.1 pp novel-premises —exactly on the hardest generalization test, retrieval recovers proofs the generator alone cannot find
- A 250M-parameter ByT5 beats GPT-4 zero-shot by ≈22 pp (random) —the right data and harness matter more than raw model scale
- 26.5% miniF2F-test was state-of-the-art for any open system in 2023; best open systems in 2025–2026 sit above 90%

Lean State Search — querying Mathlib by proof state

Tao, Liu, Wang, Xu (Renmin University of China, RUC AI4Math), arXiv:2501.13959, 2025. Deployed at `premise-search.com`; code `ruc-ai4math/Premise-Retrieval`.

What it does: input a Lean proof state, output a ranked list of Mathlib lemmas likely to apply. The search engine lets a user paste a goal and see candidate premises — aimed at users who do not already know Mathlib's names.

Model — a BERT bi-encoder (not ReProver's ByT5):

- Two encoders embed the state and each premise into a shared 768-dim space; relevance is the similarity of the two embeddings.
- BERT (6 layers, hidden 768) with a tokenizer trained on formal-math text, not a general-English tokenizer.
- Premises are embedded once and indexed; a query embeds the state and ranks by similarity.

Lean State Search — training and results

Training data (LeanDojo's extraction script, Mathlib v4.10.0):

- 65,567 training theorems; 149,549 Mathlib premises indexed.
- Each tactic step gives a state and the premises that tactic used — a positive pair is (state, a premise actually used there).

Objective — in-batch contrastive learning (batch size 32): for each (state, premise) pair the model scores the true premise against the other premises in the batch as negatives; the loss pulls the matching pair together and pushes mismatched pairs apart.

Results vs ReProver's retriever (random split):

Retriever	R@1	R@5	R@10	Proving pass@1
ReProver	11.79	28.78	36.69	28.28%
Lean State Search	**15.17**	**38.20**	**46.53**	**30.74%**

Higher recall at every k, and a higher end-to-end proof rate: a retriever trained from scratch on Lean text, with a Lean-specific tokenizer, beats one adapted from a general byte-level model.

Where the retrieval pattern reappears

ReProver's retriever is now everywhere:

- **Lean Copilot** `select_premises`: ReProver retriever exposed inside Lean
- **Leandex** (Week 11 Part D5): dense retrieval over Mathlib, productionized as a microservice for numina-lean-agent
- **Lean State Search** (`lean_state_search`, premise-search.com): a BERT bi-encoder trained by contrastive learning on (state, premise) pairs
- **`lean_hammer_premise`** in lean-lsp-mcp (Lean Hammer server, `leanpremise.net`): premise retrieval as one of the 22 MCP tools
- **Kimina-Prover**: retrieval-augmented generation
- **LongCat-Flash**: trains tool use including premise retrieval via HisPO

The structural fact behind this: a fixed-context language model cannot fit all of Mathlib in its context window. **Mathlib was ≈130,000 declarations in 2023; over 200,000 in 2026.** This constraint is not going away.

LeanDojo Benchmark 4 and LeanDojo-v2

Benchmark 4 (July 2024) — ported to Lean 4 / mathlib4 and grown to ≈122.5k theorems, ≈259.6k tactic steps, ≈167.8k premises (from the Lean-3 numbers above); extracts from any Lean 3 or Lean 4 repo; it is the Lean-4 successor to the 2023 dataset.

LeanDojo-v2 (2025, NeurIPS MATH-AI) — the original LeanDojo is now deprecated. v2 is an end-to-end platform, not just a dataset:

- trace a repo into a lifelong dataset database
- train: supervised fine-tuning, reinforcement learning, or retrieval objectives
- retrieval-augmented agents, proof search, autocompletion, deploy
- absorbs LeanAgent's lifelong-learning loop (a 2024 prover); ships Lean4Code, a small IDE

The arc: from "benchmark + harness + retriever" to a prover-development platform. The 2023 Lean-as-a-server harness is the ancestor of the `lean-lsp-mcp` layer the rest of the course runs on.

Design axis 1: whole-proof or stepwise

Once GPT-f fixes the loop shape, design choices collapse onto **two axes**.

Axis 1 —whole-proof vs proof-step + search

- *Stepwise*: model emits one tactic at a time; external search (BFS / MCTS / MCGS) expands the proof tree. Systems: GPT-f, ReProver, AlphaProof, Aristotle, BFS-Prover, Lean-STaR
- *Whole-proof*: model emits the entire proof in one rollout; kernel either accepts or returns an error. Systems: DeepSeek-Prover v1/v2, Goedel-Prover, Kimina-Prover, Seed-Prover, OProver
- Neither mode dominates in 2026 —Seed-Prover and Aristotle both reached 5/6 on IMO 2025 (Seed-Prover 4/6 in-contest, P1 after the deadline)

Design axis 2: where the search runs

Axis 2 —where the search runs

- *Search-in-tree*: external harness expands nodes. Systems: BFS-Prover, AlphaProof (MCTS), DeepSeek-Prover v1.5 (RMaxTS)
- *Search-in-CoT*: model plans a decomposition in its chain-of-thought. Systems: DeepSeek-Prover v2 (have lemmas), Seed-Prover (refinement), Goedel-Prover (self-correction), Lean-STaR (think-before-tactic)
- *Search-in-agentic-loop*: model emits tool calls to interrogate Lean mid-proof. Systems: LongCat-Flash (TIR — Tool-Integrated Reasoning), OProver (retrieval + feedback)

Trend (2024 $\rightarrow$ 2026): search migrates from external machinery into the policy. v1.5 ran search-in-tree; v2 moved most of it into CoT; LongCat and OProver moved it into the agentic loop the model itself drives.

Each system on the two axes

System	Axis 1	Axis 2 (search)
DeepSeek-Prover v1	whole-proof	none
DeepSeek-Prover v1.5	both (shared weights)	tree (RMaxTS)
DeepSeek-Prover v2	whole structured proof	CoT (have decomp)
Kimina-Prover	whole-proof	CoT (minimal)
Goedel-Prover v1/v2	whole-proof	CoT (self-correction)
AlphaProof	stepwise	tree (AlphaZero MCTS)
Seed-Prover	whole-proof (lemma-style)	CoT (refinement)
Aristotle	stepwise	tree (MCGS + informal CoT)
BFS-Prover	stepwise	tree (best-first)
Lean-STaR	stepwise	CoT (think-before-tactic)
LongCat-Flash	both	CoT (TIR tool-calls)
OProver	whole-proof	CoT (retrieval + feedback loop)

Each later part of Session 08 refers back to this table.

Part 2 · DeepSeek-Prover

Three open releases on the DeepSeekMath base

The trilogy in one year (2024–2025):

- **v1** (May 2024, 7B) — synthetic data + expert iteration
- **v1.5** (Aug 2024, 7B) — RL + tree search (RMaxTS)
- **v2** (Apr 2025, 7B & 671B) — subgoal decomposition + RL

Base-model line: v1/v1.5 on DeepSeekMath-Base 7B; v2 on DeepSeek-V3-Base 671B. All weights on Hugging Face under permissive licences.

miniF2F-test climbs: 46.3% → 63.5% → 88.9% (v1 → v1.5 → v2) — one piece per release: scale, search, decomposition.

Who builds on the checkpoints:

- **Goedel-Prover v1** — fine-tune of v1.5-Base
- **Leanabell-Prover** — continues from v1.5
- **Kimina-Prover** — v1.5 ckpts in comparators
- **Goedel-Prover-V2** — v2 decomp. at 8B/32B

The arc, as questions: v1 = "what does scale alone do?", v1.5 = "what does search add?", v2 = "what does decomposition add?". Every later prover (Parts 5–6) implicitly answers "what did v2 leave on the table?".

v1 (May 2024): synthetic data + expert iteration

The recipe: GPT-f expert-iteration, scaled $\times 100$ in data, on Lean 4, with open weights.

Inputs:

- 869,659 NL math problems (competitions, high school)
- DeepSeekMath-Base 7B, fine-tuned on MMA dataset

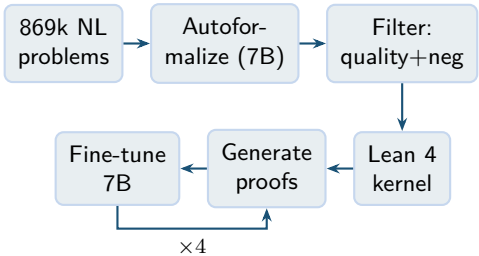
Key trick —autoformalization tolerance:
Accept proofs of the original statement *or its negation*. Many autoformalized statements are subtly wrong; proving the negation still teaches proof skill.

Outcome:

- 712,073 statements survive filtering
- 8 million verified (statement, proof) pairs

Iteration: 4 rounds of
generate $\rightarrow$ Lean-verify $\rightarrow$ fine-tune.

v1 pipeline



- Filter: (a) CoT quality score, (b) hypothesis-rejection check (drop statements whose hypotheses prove `False`)
- Only **kernel-verified** proofs enter training
- Final dataset: **8 million** verified pairs

v1 numbers

Benchmark	Result	Notes
miniF2F-test	46.3%	@64 samples
miniF2F-test	52.0%	cumulative
miniF2F-valid	30.0%	single sample
miniF2F-valid	60.2%	@65,536 samples
FIMO	5/148	GPT-4: 0

Context:

- Prior tree-search RL SOTA: 41.0%
- GPT-4 whole-proof (64 samples): 23.0%
- v1 beats both *without any tree search*

What v1 settled: supervised fine-tuning on massive verified synthetic data beats tree-search RL baselines —whole-proof generation is viable at scale.

What v1 left open: no RL on SFT (supervised fine-tuning); no efficient search (65k i.i.d. samples is wasteful); still 7B.

v1.5 (Aug 2024): RL + RMaxTS

Three training stages on top of DeepSeekMath-Base:

1. **SFT** on a larger, cleaner synthetic dataset. Both whole-proof *and* proof-step-with-state demonstrations —one checkpoint serves either inference mode.
2. **RLPAF** (RL from Proof Assistant Feedback).
   Reward: binary 1/0 from Lean's kernel.
   Algorithm: GRPO (Group Relative Policy Optimization).
   Curriculum: moderately difficult theorems (signal non-trivially $\neq 0$ or 1).
3. **Final stage** for Lean 4 tactic formatting.

Headline contribution: a new inference-time tree search —**RMaxTS**.

Monte Carlo Tree Search (MCTS)

MCTS grows a search tree by repeating four steps many times, then plays the move it visited most.

1. **Select** from the root, walk down by a rule that balances *exploit* (high value so far) against *explore* (few visits) — the UCB / PUCT score.
2. **Expand** add a new child node —an untried move (here: a tactic).
3. **Evaluate** estimate that node's value: a random rollout to the end (classic MCTS), or a **value network** (AlphaZero —no rollout).
4. **Back up** send the value to every node on the path, updating its visit count N and mean value Q.

Key design choices: the exploration constant in step 1, a policy-network prior $P(s, a)$ on moves (PUCT), and whether step 3 uses a rollout or a value network.

For a prover: state = proof state, move = a Lean tactic, terminal = the kernel verdict. **Data:** self-play, kernel-verified trajectories train the *policy* (which tactic) and the *value* (will this state close?).

The selection rule: UCB

At a node, which child do we descend into? Each child i has a visit count N_i and a mean value Q_i (its average result so far).

- Always take the largest Q_i: a child tried once with a lucky result looks best and is never questioned.
- Always take the least-visited child: the budget is wasted on bad moves.

UCB (Upper Confidence Bound) scores each child and descends into the highest:

$$\mathrm{UCB}_i = \underbrace{Q_i}_{\text{exploit}} + \underbrace{c\sqrt{\frac{\ln N}{N_i}}}_{\text{explore}}, \qquad N = \text{parent visits}, \quad c = \text{exploration constant}.$$

- First term: how good the child has looked so far.
- Second term: a bonus, large when N_i is small and shrinking as the child is visited — a rarely-tried child gets the benefit of the doubt.

Q_i plus the bonus is an optimistic upper estimate of the child's true value: you pick the option that *could* be best. UCB applied to a tree is called UCT.

Two variants of UCB used here: PUCT and DUCB

Plain UCB explores every child on the same footing. Two refinements appear in this lecture.

PUCT (AlphaZero, and the PUCT-style provers): weight the explore term by a policy network's prior $P(s, a)$ — its guess of how promising a move is.

$$\text{score}(s, a) = Q(s, a) + c\, P(s, a)\, \frac{\sqrt{N}}{1 + N(s, a)}.$$

Moves the policy likes are explored first, so the budget is not spent uniformly.

DUCB (discounted UCB, used by DeepSeek-Prover's RMaxTS): the same rule, but past results decay by a factor γ each step ($\gamma = 0.99$ here). An early branch that looked good cannot dominate forever — its Q fades unless revisited, freeing the search to explore elsewhere.

RMaxTS —the components, broken down

Why plain MCTS stalls here: the kernel reward is 0 on almost every attempt, so every node scores 0 —the search cannot tell branches apart. RMaxTS adds an *exploration* reward instead.

Nodes a node is a partial-proof tactic state; a child is one more tactic block sampled from the prover (MCTS *expand*).

Intrinsic reward a rollout scores $\mathbf{1}$ if it reaches a tactic state never seen before, $\mathbf{0}$ if all its states already exist —the search is paid to find new states, not to close the proof (it usually cannot yet). This replaces the all-zero leaf value in MCTS *evaluate*.

Selection —DUCB discounted UCB, $\gamma = 0.99$: explore/exploit, but old Q-estimates decay so an early-good branch cannot dominate.

Why "R-Max" R-Max treats any unvisited state as maximally rewarding until visited; RMaxTS lifts that to a tree of generation rollouts.

Result: 63.5% miniF2F-test (RMaxTS) vs 60.2% single-pass —same model; the gain is how the search budget is spent.

Truncate-and-resume

The other v1.5 idea: convert a failed whole-proof into tree nodes.

1. Send generated proof to Lean.
2. Read back the **first error position**.
3. **Truncate** everything from that point onward.
4. Promote the verified prefix to a tree node, with the current tactic state appended as a comment.
5. **Re-sample** from that node.

One failed generation becomes a sequence of partial proofs —each revisitable, branchable, or discardable based on how much progress it made.

Combined loop: RMaxTS over nodes produced by truncate-and-resume.

v1.5 numbers

Benchmark	Whole-proof	With RMaxTS	Notes
miniF2F-test	60.2%	63.5%	+10.2 pp over v1
ProofNet	23.7%	25.3%	SOTA at the time

- Same RL checkpoint serves both inference modes (no retraining to swap)
- First open prover to cross 60% miniF2F-test with a fixed model

What v1.5 settled: the search axis is real —given a fixed model, *how* you spend the search budget materially changes pass rate.

What v1.5 left open: search is post-hoc (harness, not model); model does not know about decomposition; still 7B.

v2 (Apr 2025): subgoal decomposition + RL

The shift: the model itself becomes the planner.

v1.5 model's role:
Generate proofs; harness cleans up failures (truncate-and-resume).

v2 model's role:

1. Write a list of intermediate have lemmas (the decomposition)
2. Prove each piece separately
3. Harness assembles the verified pieces

Harness can still search, but the architecturally hard part —planning the proof structure —is now inside the model's chain-of-thought.

Key reward shift: in CoT mode, each subgoal correctly closed is a verifiable unit —subgoal-level signal arrives *before* the full theorem closes, unlike v1.5's all-or-nothing whole-proof reward.

Two modes, one checkpoint: a prompt toggle switches between CoT decomposition and direct whole-proof.

Decomposition: the counting argument

Direct generation of an n-step proof:

$$\Pr[\text{success}] \approx p^n \quad \text{(every tactic must fire)}$$

Decomposition into m subgoals of k steps each ($n = mk$):

$$\Pr[\text{lemma}_i \text{ closes}] \approx p^k \qquad k \ll n$$

Subgoals are **independently retryable** —and recursively decomposable.

The model is rewarded for decompositions whose pieces *actually close*, not for decompositions that look well-structured.

Same structural argument as classical proof planning (Bundy 1990s); v2 turns it into a training objective.

v2 numbers (671B unless noted)

What v2

Benchmark	Result	Notes
miniF2F-test	88.9%	Pass@8192, CoT mode; +25 pp over v1.5
PutnamBench	49/658	First open prover in double digits on Putnam
AIME 24+25	6/15	Formal; DeepSeek-V3 informal: 8/15
ProofNet-test	37.1%	Pass@1024, CoT mode
7B variant	—	Same pipeline; runs on single GPU

settled: subgoal decomposition is trainable; the general-LLM bootstrap seeds the corpus; the model itself becomes the planner.

What v2 left open: 671B is expensive; 7B is weak; harness still externalises some search.

Three design axes: v1 → v1.5 → v2

Axis	v1	v1.5	v2
Proof generation	Whole-proof, one shot	Both modes, one ckpt	Structured whole-proof via have decomp.
Search	i.i.d. samples, no search	RMaxTS in tree (harness searches)	Search-in-CoT; harness parallelises subgoals
RL reward	None (SFT only)	Per-proof binary (GRPO, RL-PAF)	Per-subgoal binary (GRPO) —signal arrives before full theorem closes

Most 2025–2026 provers can be located on these three axes: Goedel-V2 = v2 axis at 8B/32B (Part 6); Kimina = v1.5 axis at scale (Part 5); OProver = v2 axis with an agentic repair loop trained into the weights (Part 6).

Part 3 · AlphaProof

Reinforcement learning —the setup

SFT copies proofs already known to be correct. RL instead lets the model learn from *its own* attempts, each scored by a reward.

Classic RL trains **two** networks (actor–critic):

Policy π_θ (actor) the model —given a statement, it samples a proof attempt.

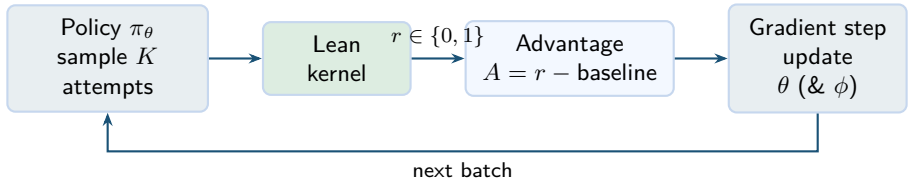
Value V_ϕ (critic) a second network estimating the expected reward of a state —"how likely is this attempt to end in a passing proof?" It is the *baseline* the policy is scored against.

Reward r the Lean kernel verdict —**1** if it compiles, **0** if not. From the verifier, so it cannot be gamed.

Input: an SFT'd model, a pool of Lean statements, the Lean kernel. **Output:** updated weights θ (and ϕ, if a critic is used).

Reinforcement learning —the training loop

One step, repeated over batches of statements:



The baseline is the key design choice:

- **Learned value network** (critic): PPO, VAPO (Seed-Prover, OProver), AlphaZero value net (AlphaProof). A second network to train —and it can hurt (Seed-Geometry dropped its value model).
- **Group mean** (GRPO): the baseline is the mean reward of the K samples —no critic. DeepSeek-Prover; LongCat's HisPO is a variant.

On-policy (learns from its own samples); the sparse $0/1$ reward needs a curriculum —statements where the K attempts are not all-0 or all-1.

Reward is sparse —what each team does about it

The kernel reward is binary and terminal: on a hard problem almost every sample scores 0, so plain RL has no gradient. The recurring fixes:

Partial credit reward closed subgoals (DeepSeek v2) or independently-compiling lemmas (Seed-Prover) —signal before the whole proof closes.

Intrinsic reward pay a rollout for reaching a *new* proof state even if nothing closes (DeepSeek v1.5, RMaxTS).

Value network turn the terminal 0/1 into per-state estimates (AlphaProof; VAPO in Seed-Prover, OProver).

Curriculum keep solvable-but-hard problems —self-play ladder (AlphaProof); group not all-0/all-1 (DeepSeek, GRPO).

Refinement / repair turn one failure into many partial-progress signals (Seed-Prover lemma pool; DeepSeek truncate-and-resume).

Throughput + scale keep the sparse reward, just run far more verified rollouts (Kimina's fast Lean server).

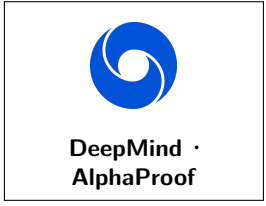
AlphaProof —AlphaZero-style RL on Lean

What AlphaProof is

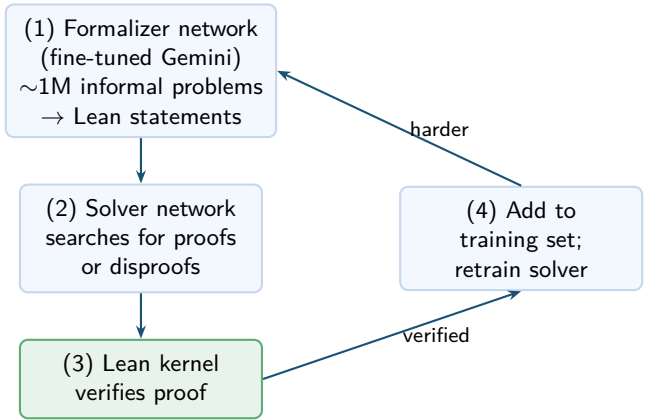
- DeepMind's RL prover operating inside **Lean 4**
- Pre-trained language model + AlphaZero search loop
- **Lean kernel** is the ground-truth reward signal

Axes (Week 12 coordinate system)

- Axis 1: **stepwise** prover (tactic-by-tactic)
- Axis 2: **MCTS / tree search** (AlphaZero-style)
- Axis 3: **value estimates** from AlphaZero value network



The AlphaProof self-play loop



Same structural move as AlphaZero on Go: self-generated data verified by an oracle; bootstrapped from a weak base model. The actions are Lean tactics; the proof state is what the search explores. Loop ran for **weeks** before IMO 2024.

How AlphaProof's reward is designed

The only ground-truth reward is the **Lean kernel**: a complete proof (or disproof) the kernel accepts scores **1**; everything else scores **0**. The reward is terminal —no partial credit for a half-finished proof —and on a hard problem almost every attempt scores 0.

A 0/1 signal that is almost always 0 is hard to learn from. AlphaProof answers the sparsity three ways:

Value network AlphaZero-style: it estimates, at each proof state, how likely that branch ends in a verified proof. The tree search spends effort where a proof looks plausible —turning the terminal 0/1 into per-step guidance.

Self-play curriculum the formalizer makes ∼1M statements across a range of difficulty; the solver retrains on its own verified proofs and is pushed to harder ones, so some problems are always solvable and the reward is never uniformly 0.

Proof or disproof the solver may settle a statement either way, so a false statement still produces a reward.

IMO 2024: 28/42 —score and problems

Combined result (AlphaProof + AlphaGeometry 2): 28/42 Gold-medal cutoff: **29** —one point below gold.

Problem	Topic	Solver
P1	Algebra / inequalities	AlphaProof
P2	Number theory	AlphaProof
P4	Geometry	AlphaGeometry 2 (19 seconds)
P6	Functional equation (algebra)	AlphaProof —*hardest problem*
P3	Combinatorics	Unsolved
P5	Combinatorics	Unsolved

Caveats on the IMO-2024 result

Silver-level result, with three real caveats: time, formalization, combinatorics

1. **Time.** Solution times ranged from **minutes to three days** per problem. The IMO human session is **4.5 hours**. AlphaProof was not under contest time.
2. **Formalization.** Problems were **manually formalized** into Lean statements before the system saw them.
3. **Combinatorics gap.** P3 and P5 (both combinatorics) were not solved. The system excelled at algebra and number theory.

These are real design tradeoffs, not faults —closed timing is part of how AlphaProof was built. Stating them honestly is what lets the IMO 2025 pair (Part 4) be read against AlphaProof at the right altitude.

AlphaProof on the two design axes

System	Design choice	Result (2024)
AlphaProof	Per-step MCTS + RL, Lean kernel reward	28/42 IMO 2024 (silver)
DeepSeek-Prover v1.5	RMaxTS (whole-proof rollouts with MCTS)	published after AlphaProof
DeepSeek-Prover v2	Subgoal decomposition, sketch-based	moves away from per-step MCTS

Takeaway: whole-proof generation became competitive once model size and data depth caught up —not because MCTS failed.

AlphaProof sits at the far end of the "search + RL" axis in 2024. The DeepSeek trilogy followed, moving toward whole-proof generation.

AlphaProof is closed-source

What is not released

- No code, no weights, no training corpus
- No reproducible artifact of any kind
- Analysis stays at the **paper level**

Sources available

- DeepMind blog: *AI achieves silver-medal standard* (25 Jul 2024)
- *Nature* methodology paper (12 Nov 2025) —structural detail sufficient to inspire replicas, not to reproduce

Open replicas (Part 6, K3)

- **BFS-Prover**: replicates the search side
- **STP** (Self-play Theorem Prover)
- Closest one can get to a reproducible AlphaProof

Descendants (Part 4)

- **Aristotle**: closed-source, step+tree-search line
- **Seed-Prover**: open, whole-proof line

AlphaProof is the common ancestor of both Part 4 systems —explicit for Aristotle, implicit-via-template for Seed-Prover.

Part 4 ·IMO 2025

IMO 2025 —the shared problem set

Two systems. Same six problems. Lean-kernel-verified proofs.

Seed-Prover (ByteDance Seed)

arXiv 2507.23726 (Aug 2025)

Apache-2.0 artifacts, closed weights

Aristotle (Harmonic)

arXiv 2510.01346 (Oct 2025)

Fully closed —paper only

Both crossed the formal gold-equivalent boundary: **5/6 IMO 2025 problems** proved in Lean, verified by the Lean kernel.

IMO 2025 —the scorecard

Seed-Prover (v1) timeline

- **During contest window** (July 18 deadline):
  P2 geometry —*Seed-Geometry, 2 s*
  P3 number theory —∼2000-line proof, 3 days
  P4 number theory —∼4000-line proof, 3 days
  P5 combinatorics/algebra —1 day
  Score inside window: 4/6
- **Post-deadline**: P1 (combinatorics) proved
  Public headline: 5/6
- P6 combinatorics: unsolved (v1 and v1.5)

Aristotle result

- Formally proved **5/6** IMO 2025 in Lean
- Unsolved: P6 (same as Seed-Prover)
- Closed compute budget
- Test-time training during inference

For context: AlphaProof 2024

28/42 on IMO 2024 (one point below gold). 2025 is the first formal cycle at gold-equivalent.

Formal and informal IMO-2025 golds

- **Seed-Prover and Aristotle**: formal gold —"accept" is the Lean kernel (deterministic, decidable, incorruptible).
- **Gemini Deep Think, OpenAI reasoning models**: informal gold on the same problem set —"accept" is a stochastic AI judge.

Both kinds of gold are real, on the same problem set. The within-Session-08 distinction to keep: Seed-Prover and Aristotle proofs are **Lean-kernel-verified**; Gemini Deep Think and OpenAI posted **informal** IMO-2025 golds graded by humans/AI.

Seed-Prover —the whole-proof design choice

Seed-Prover is a **whole-proof** prover (Axis 1): it generates the entire Lean 4 proof in one structured artifact rather than stepping tactic by tactic. Within that family it adds two things:

- **Lemma-style structuring** —emit named `lemma`s, then a `theorem` that applies them.
- **Iterative refinement** —rewrite the proof across rounds, driven by Lean compiler feedback.

Both target the known weakness of plain whole-proof generation: no Lean interaction during the one-shot generation.

Lemma-style and have-style proofs

The model is trained to emit named `lemma` definitions before the main theorem, not anonymous have steps inside it.

```
-- Whole proof (standard have-style)
theorem eg : 1 + 2 = 3 := by
  have h0 : 1 + 1 = 2 := by ring
  have h1 : 1 + (1+1) = 3 := by ring
  linarith [h0, h1]
```

```
-- Lemma-style (Seed-Prover)
lemma round1_h0 : 1 + 1 = 2 := by ring
lemma round1_h1 : 1 + (1+1) = 3 := by ring
theorem eg : 1 + 2 = 3 := by
  linarith [round1_h0, round1_h1]
```

Named lemmas compile independently —Lean reports which passed and which failed. Progress is observable, not binary.

Lemma-style proofs: modular compilation, reuse, difficulty sorting

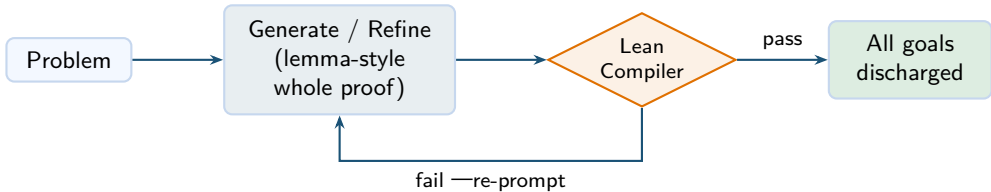
Modular compilation Each lemma compiles independently. Lean tells you exactly which lemmas passed and which failed. Progress is measurable.

Cross-trajectory reuse A lemma proved by one trajectory can be dropped into another trajectory's prompt. Lemmas accumulate into a per-problem **lemma pool**.

Difficulty sorting Lemmas are scored by proof rate, semantic relevance, and length. Hard-to-prove lemmas often turn out to be the load-bearing ones.

Iterative Refinement Loop

One generator runs every round. Round 1's prompt is just the problem; on failure it is re-prompted with what the previous attempt produced.



Next-round prompt = original problem + three streams:

- **Lean error feedback** —from the compiler.
- **Lemma pool** —lemmas that compiled, accumulated across rounds.
- **Self-summary** —the model's own post-mortem of the failed attempt.

How Seed-Prover's reward is designed

Seed-Prover is trained with RL (the VAPO algorithm — value-augmented proximal policy optimization). The reward is the familiar binary kernel verdict, plus one shaping term:

Outcome reward 1 if the proof compiles, 0 if not.

Formatting penalty pushes the model to emit named `lemmas` before the main `theorem` —the lemma-style structure is rewarded directly, not just hoped for.

The other design choice is the **training prompt**, not the reward: prompts randomly include NL hints, proved lemmas, *failed* lemmas, past attempts, summaries, and Lean feedback —so the inference-time refinement loop sees prompt shapes it was trained on.

v1.5's Agentic Lean Prover uses agentic RL (VAPO + tool use), reward $+1$ if the final proof compiles, -1 otherwise.

Three Test-Time Settings: Light, Medium, Heavy

Light Single trajectory with refinement. Each whole-proof attempt refined 8–16 times against Lean feedback. Budget $\approx$ Pass@64–256 single-shot. Wall clock 1–2 h/problem. *Proves IMO 2022 P2 (MOHS=15), which raw Pass@8192 cannot.*

Medium Outer + inner refinement. Adds an inner Light loop (8×8 budget) on each failed lemma. Final proofs routinely exceed 1000 lines of Lean. *Wins: IMO 2025 P5, IMO 2020 P5, IMO 2024 P1.*

Heavy Open-ended conjecture $\times$ refinement. A proposer generates 5000 candidate conjectures; each proved under Light; proved ones enter the lemma pool; Medium then attacks the main problem using the pool. Runs for days. *Wins: IMO 2025 P3, P4 (2000- and 4000-line proofs).*

Seed-Geometry —Companion Geometry Engine

Lean's synthetic geometry reasoning is poor. ByteDance ships a separate engine.

DSL Geometric diagrams via ruler-and-compass with composite primitives (isogonal conjugate, exsimilitude/insimilitude centers). Each composite expands to plain ruler-and-compass; keeps sequences short for LLM tokenization and the symbolic backend.

C++ backend Forward-chaining reasoning engine ($\approx 100\times$ speedup over the prior Python implementation). Pybind11 binding.

Policy-only Actor + critic was initial design; the value model harmed performance under extensive search. Final: single policy + beam search, each beam evaluated by the C++ engine to closure.

Training data 230 million unique geometry problems, 38B tokens.

IMO 2025 P2: solved in **2 seconds** (after human formalization).

Seed-Geometry Benchmarks

Benchmark	Seed-Geometry	AlphaGeometry 2
IMO-AG-50 (IMO 2000–2024 geometry)	**43**/50	42/50
IMOSL-AG-30 (IMO shortlist geo, hard back)	**22**	19
IMO 2025 P2	2 seconds	—

Division of labor at IMO 2025:

- Seed-Geometry: P2 (instantly)
- Seed-Prover: P1, P3, P4, P5 (multi-day refinement)
- P6: unsolved

Seed-Prover v1.5 —Three-Agent Architecture (Dec 2025)

v1.5 replaces the monolithic refinement loop with three specialized agents.

NL Prover Generates rigorous lemma-style natural-language proofs from the formal statement. Initialized from Doubao-Seed-1.6.

Sketch Model Translates an NL proof into a lemma-style Lean sketch (with `sorry` placeholders for hard lemmas). Trained via **Rubric RL**: reward $+1$ if ≥ 3 lemmas, Lean sketch valid, NL-proof score ≥ 0.7.

Agentic Lean Prover Tool-using agent that proves each individual lemma in the sketch. Cold-started from Seed-Prover 1.0 + agentic SFT + Agentic RL (VAPO). 64K-token budget, 28 tool calls.

v1.5 —Agent Tools and Adaptive Behavior

The Agentic Lean Prover can call three tools any number of times:

Lean verification LooKeng REPL —verifies one lemma at a time; compiled lemmas are cached and axiomatized for downstream steps.

Mathlib search Embedding-based retrieval over Mathlib v4.22.0. Returns matching theorems and definitions by semantic similarity.

Python execution Numerical checks, computational experiments, search for small counterexamples.

Adaptive behavior: on Fate-H, Mathlib search is invoked $\approx 10\times$/trajectory; on PutnamBench, only 1–2$\times$. Average tool calls drop from $\sim$15 to $\sim$10 during training —the model learns when to search and when to prove directly.

Seed-Prover v1.5 Benchmarks

Benchmark	v1.0	**v1.5**
PutnamBench (660 problems, Pass@8×8)	50.4%	**87.9%**
Fate-H (100 honors / grad problems)	35%	**80%**
Fate-X (100 PhD-qual level)	9%	**33%**
CombiBench (100 combinatorics)	39%	**48%**
Putnam 2025 (12 problems)	—	**11/12** in 9 h

IMO 2025 (v1.5 Medium, no Heavy): P1: 16.5 h ·P2: 0.01 h (Seed-Geometry) · P3: 5 h ·P4: 8 h ·P5: 1 h ·P6: unsolved

Difficulty cliff visible: Fate-X (PhD-qual) stalls at 33% despite large gains elsewhere.

Aristotle (Harmonic) —four components

Opposite design: step-by-step tactic search, not whole-proof generation.

1. **Lean proof-search system** 200B+ parameter transformer serving as both policy (next-tactic distribution) and value function (estimated probability of closing the goal).
2. **Informal reasoning system** A separate informal LLM proposes auxiliary lemmas in natural language; a formalizer converts them to Lean; the prover treats them as intermediate targets.
3. **Yuclid** Dedicated C++ geometry engine combining deductive databases and algebraic reasoning. Geometry is NOT done in Lean —Yuclid solves, then the result is formalized.
4. **Test-time training (TTT)** During inference, the system retrains the policy/value network on its own discovered traces, specializing to the current problem.

Monte Carlo Graph Search (MCGS)

The headline algorithmic choice.

In standard MCTS, every node is a distinct state —the search tree grows even when many paths reach equivalent proof states.

Aristotle identifies equivalent states and actions, collapses the search **tree** into a search **hypergraph**, and runs dynamic programming over the graph.

MCTS (tree)

Equivalent states duplicated.

Cost grows with branching factor.

MCGS (hypergraph)

Equivalent states merged.

Per-step cost similar; global cost amortized over shared substructures.

Same intuition as AlphaGo's transposition tables, lifted to formal proof search.

Seed-Prover and Aristotle — where they differ

	Seed-Prover	**Aristotle**
IMO 2025 result	5/6 (4 during contest, P1 after deadline)	5/6
Lean interaction	Per-attempt (per-lemma in v1.5)	Per-tactic, tight feedback loop
Geometry engine	Seed-Geometry — separate C++ ruler-and-compass	Yuclid — C++ deductive/algebraic
Informal guidance	Self-summarization between refinement rounds	Informal LLM proposes lemmas; formalizer converts
Lemma reuse	Explicit pool, scored by proof rate $\times$ relevance $\times$ length	Implicit through MCGS graph nodes
Test-time adaptation	Lemma pool grows; v1.5 caches + axiomatizes proved lemmas	TTT retrains policy/value on per-problem traces
Model scale	Undisclosed (v1.5 on Doubao-Seed-1.6)	>200B parameters
Openness	Apache-2.0 artifacts; weights closed	Closed (paper only)

Proof-unit and search-structure rows are on the Part 1 canonical Axes table.

Seed and Aristotle on the two design axes

Where the planning lives:

- **Seed-Prover:** in the model's training distribution (lemma-style whole-proof is what it was trained to emit); plus the conjecture proposer at Heavy. Refinement adjusts the lemmas.
- **Aristotle:** in the MCGS graph and the informal LLM. The plan emerges from tree exploration.

Where the test-time signal comes from:

- **Seed-Prover:** the lemma pool grows across rounds —it accumulates *content*. Weights stay fixed.
- **Aristotle:** TTT —it accumulates *weights*. The memory stays implicit in the graph.

Both are forms of test-time learning. They differ in what gets updated.

Part 5 · Trained to use Lean tools

Kimina-Prover (Numina × Moonshot, 72B)

What it is. Lean 4 whole-proof generator (72B, fine-tuned from Qwen2.5-72B-Base).
Trained by **Project Numina × Moonshot AI** using the Kimi-k1.5 large-scale RL recipe.

Reward signal: binary Lean compiler verdict only.
No tactic-level intermediate reward; no learned reward model.

Numina × Moonshot

Releases.

- Preview (Apr 2025, arXiv 2504.11354): 72B trained; 1.5B & 7B distilled weights released; miniF2F Pass@8192 = 80.7%
- Full release (Jul—Aug 2025): 72B weights (MIT); 1.7B & 8B distilled; training pipeline (`kimina-prover-rl`) open-sourced

The scaling-law claim

Before Kimina, doubling parameters did not reliably improve proof rates —search and data dominated. Kimina-Prover Preview broke that pattern.

Same RL recipe, same 200k-problem corpus, three sizes:

Model	Pass@1	Pass@32	Pass@1024	Pass@8192
1.5B	42.6%	56.2%	61.9%	—
7B	52.5%	63.1%	70.8%	—
72B	52.9%	68.9%	77.9%	**80.7%**
72B + TTRL	—	—	—	**92.2%**

Source: Kimina-Prover Preview, Figure 3. miniF2F-test, Lean 4. *"Clear performance scaling with model size, a trend previously unobserved for neural theorem provers."*

Kimina training architecture

Base: Qwen2.5-72B-Base. All proving ability from fine-tuning.

Data: 200k Lean 4 problems. Roughly 1:1 mix:

- ≈100k auto-formalised by Kimina-Autoformalizer-7B (90% one-shot compile, 66% semantic accuracy)
- ≈100k human-curated Lean statements

RL recipe: Kimi-k1.5 style online RL on the 72B. Context window 32k tokens. Binary reward only.

Kimina Lean Server (in the loop):

- ≈100 Lean verifications/sec on 64 CPU cores
- Every RL rollout gets a real Lean accept/reject
- Makes online RL on whole proofs tractable
- Also released as the Numina Lean Server —reused at inference

Not in the loop: `lean-lsp-mcp` is a *separate artifact*. Kimina generates whole proofs; no interactive tool calls during training.

How Kimina's reward is designed

Where AlphaProof adds a value network and tree search, Kimina keeps the reward itself as simple as possible.

One binary signal reward $= \mathbf{1}$ if the Lean compiler accepts the whole-proof rollout, $\mathbf{0}$ otherwise —external, given by the verifier.

Nothing learned, nothing intermediate no process reward, no learned reward model, no tactic-level signal. The model is judged only on whether the finished proof compiles.

Whole-proof, not per-step the unit of reward is the entire rollout, matching Kimina's whole-proof generation.

The price of such a sparse reward is paid by engineering and scale:

- The Kimina Lean Server verifies ≈100 proofs/sec (64 CPU cores), so every rollout gets a real accept/reject inside the RL step.
- Under this same reward, proof rate scales cleanly with model size (1.5B → 7B → 72B).

TTRL —test-time reinforcement learning

Training-time RL changes the weights. Kimina's TTRL changes nothing in the weights —it reuses the same three ideas (policy, reward, explore/exploit) to steer the search the trained model runs on a *single* problem.

The test-time loop:

1. The model proposes **lemmas** and recursively tries to prove each one, up to 4 levels deep.
2. A **scoring policy** picks which lemma to work on next: 60% on the current best lemma (exploit), 40% on others (explore).
3. The **reward** is again the Lean kernel —a lemma counts only once it actually compiles.
4. A **negation filter** drops sub-goals that are logically inconsistent, so the search does not chase impossible lemmas.

So it is search-over-proofs, not sample-and-check: Pass@32 = 84.0%, Pass@1024 = 87.7%, with TTRL = **92.2%** miniF2F (at a ≈42k Lean-pass budget).

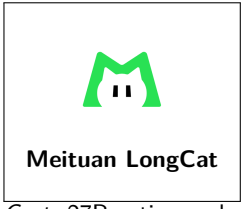
LongCat-Flash-Prover (Meituan, 2026)

arXiv 2603.21065 (Wang et al.), MIT.

Architecture:

- 560B total MoE, ≈**27B active** per token
- 128k context (long agentic trajectories)

Its role in Part 5: tool use is trained *into the weights* via agentic RL — the model drives the Lean tools itself, with no inference-time scaffold inserting the calls. **Best result:** 97.1% Pass@72 miniF2F (sketch + TIR + tree search); sample-efficient — Goedel needs 1024 attempts to reach 92.6%.



Cost: 27B active ≈ dense 30B; ≈1–3 min/problem on 8×H100.

LongCat method: three specialists + HisPO

Proving is split into three specialists, iterated and then unified into one policy:

Autoformalize NL statement $\rightarrow$ verified Lean statement (tools: V_{syn} syntax, V_{con} semantic judge).

Sketch split the theorem into helper lemmas (`:= by sorry`), then prove the body (V_{leg} AST-legality guard against reward-hacking).

Prove whole-proof, or fill the `sorry`-holes of a sketch.

At inference it runs **TIR** (tool-integrated reasoning): the model calls these tools mid-proof and re-attempts on their feedback.

Trained with **HisPO** (Hierarchical Importance Sampling Policy Optimization): a GRPO-style RL that drops high-variance sequences and masks noisy tokens, so many-round agentic RL stays stable on a 560B MoE.

LongCat results (Pass@32)

Mode	Params	Math-Oly	MiniF2F	ProverBench	Putnam
Whole-proof	560B/27B	16.9	84.4	49.9	—
Whole + TIR	560B/27B	27.5	90.2	57.9	—
Sketch + TIR	560B/27B	**35.8**	**93.9**	**66.5**	**28.9**

With tree search at model-specific budgets: MiniF2F 97.1%/@72 ·Putnam 41.5%/@118 ·Math-Oly 46.7%/@180. Sample-efficient: 97.1% at 72 attempts vs Goedel's 92.6% at 1,024. Cross-prover Pass@32 ranking: Part 7.

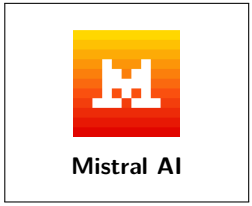
Leanstral (Mistral, March 2026)

System.

- 119B total / **6.5B active** (sparse MoE, 128 experts, 4 active per token)
- 256k context; Apache-2.0 weights
- Free `labs-leanstral-2603` endpoint: Mistral API key, no Claude account
- Native tool surface: `lean-lsp-mcp` (same 22 tools as Week 11 Part G)

Benchmark: FLTEval (Fermat's Last Theorem formalization tasks, *not* olympiad).

Cost claim:
Leanstral Pass@2 $\approx$ \$36/task
Claude Sonnet 4.6 same task $\approx$ \$549
(1/15 cost, 2.6 pts higher on FLTEval)



Leanstral is also covered in Week 11 Part C2. Here the focus is the **training** side: tool use in the weights.

Trained native to lean-lsp-mcp

What "native" means: the training distribution included `lean-lsp-mcp` tool-call trajectories. The model learned the tool surface, not just the proof distribution.

At inference, Leanstral emits tool calls *itself*:

- Unfamiliar Mathlib name → emits `lean_local_search` before guessing
- Kernel rejects a tactic → reads `lean_diagnostic_messages` before re-attempting
- Needs a type signature → calls `lean_hover_info`

No outer scaffold inserts these —the policy emits them because that is the distribution it was trained on.

Training data shape: fill-a-sorry and repair-a-broken-proof trajectories (research flavour), not competition-style whole-proof rollouts.

FLTEval, not olympiad: Mistral's positioning is verification engineering and Mathlib contribution, not olympiad scoreboards.

Leanstral and LongCat at different scales

	LongCat-Flash-Prover	**Leanstral**
Scale	560B / 27B active	119B / 6.5B active
Tool set	Custom $V_{syn}/V_{con}/V_{leg}$	Open `lean-lsp-mcp`
Training	HisPO agentic TIR RL	Post-training on tool-call traj.
Capability	AF + SKETCH + PROVE	Fill-sorry, repair-proof
Benchmark	miniF2F, Putnam, Math-Oly	FLTEval (research tasks)
Deploy	Self-hosted (vLLM / SGLang)	Free endpoint; single GPU

Both are trained to call Lean tools natively —the model emits the tool calls itself. The same pattern holds from a 6.5B active model (Leanstral) to a 27B active MoE (LongCat); the engineering cost is not size-dependent.

Summary: tool calls in the weights

Part 5 surveyed three trained provers on one axis: whether the model is trained to call Lean tools natively, or generates whole proofs and leaves tool use to a separate inference-time layer.

- **Kimina-Prover** —whole-proof rollouts; no tool calls in training. Tool use, if any, is a separate layer.
- **LongCat** and **Leanstral** —trained to emit Lean tool calls inside their rollouts; tool use in the weights.

The strongest 2026 trained provers —LongCat, Leanstral, and OProver (Part 6) —call Lean tools natively: the tool-call discipline lives in the weights, not in an external scaffold inserted at inference time.

Part 6 · Variations and the open frontier

OProver-32B —the repair loop trained into the weights

OProver-32B (Multimodal Art Projection, M-A-P; arXiv 2605.17283) makes one design move worth isolating.

Most agentic provers wrap a non-agentic model in an *inference-time* loop: retrieve similar proofs → attempt → read Lean feedback → repair. OProver instead makes that loop the training task itself: each training example is a round-level repair step with state $(s,\ \mathcal{R}_t,\ p_{t-1},\ f_{t-1}) \to p_t$ — statement, retrieved context, previous attempt, Lean compiler feedback, revised attempt.

So the policy learns to **use compiler feedback and retrieved proofs** natively, instead of running that loop in an external scaffold. (Weights unreleased at submission.)

Goedel-Prover v1 + v2 (Princeton)

Headline: Goedel-V2-8B: **83.3% Pass@32 on miniF2F**, matching DeepSeek-Prover-V2-671B at $\approx 100\times$ fewer parameters.

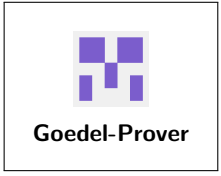
v1 (Feb 2025, arXiv 2502.07640):

SFT on Goedel-Pset-v1 (1.64M formal statements); expert iteration; 57.6% miniF2F; 7 PutnamBench problems.

v2 (Aug 2025, arXiv 2508.03613):

Princeton + Tsinghua, Amazon, Meta FAIR, Numina. Lead: Yong Lin, Shange Tang. 8B and 32B; MIT code, CC0 data.

Counter-evidence to "scale is all you need": at miniF2F, adding parameters beyond a point gives little gain once the data and feedback loop are right.



Goedel-V2: three innovations

1. **Scaffolded data synthesis** Progressive-difficulty curriculum: easier autoformalized problems first, then harder variants conditioned on what the current prover can solve. Problems at the model's frontier are amplified. Data-side analogue of expert iteration.
2. **Verifier-guided self-correction** Trains the model to revise proofs using Lean compiler feedback. When a proof is rejected, the error message is fed back; the model is supervised to produce a corrected proof. "Read the error, fix the tactic" is in the *weights*, not an outer scaffold.
3. **Model averaging** Late-stage checkpoints weight-averaged to preserve output diversity that would otherwise collapse under heavy expert iteration. Protects Pass@k from collapsing as Pass@1 climbs.

Goedel-V2: self-correction loop + benchmark numbers

Self-correction loop: two revision rounds per attempt. Token budget: 32K $\rightarrow$ 40K. Lean kernel is the only verifier —no LLM-judge, no external critic.

8B vs 671B (miniF2F); 32B vs 671B (PutnamBench):

System	Params	miniF2F Pass@32	PutnamBench
DS-Prover-V2-671B	671B	$\approx$83%	47 @ Pass@1024
Goedel-V2-8B	**8B**	**83.3%**	—
Goedel-V2-32B (std)	32B	88.1%	—
Goedel-V2-32B	**32B**	**90.4%** (self-corr.)	**64 @ Pass@64 (#1 open)**

32B beats 671B on PutnamBench using $16\times$ fewer samples. Self-correction does the work that brute Pass@k did before.

MathOlympiadBench: 360 IMO-level problems (158 IMO 1959—2024, 131 shortlist, 68 regional). Introduced because miniF2F is approaching saturation —a benchmark-design move as much as a benchmark result.

Variations survey —seven systems

System	Key idea
BFS-Prover	Best-first search + kernel-labeled DPO; 71.3% miniF2F; MCTS not required
STP	Conjecturer + prover in one model; 2× LeanWorkbook; ICML 2025
Lean-STaR	Retrospective tactic rationale; +2.9 pt miniF2F Pass@64; ICLR 2025
Leanabell v1	Outcome RL (kernel 0/1 reward); 59.8% Pass@32 miniF2F
Leanabell v2	CoT interleaved with live verifier calls; 78.2% Pass@128 miniF2F
HILBERT (Apple)	Informal LLM + prover LLM + retriever + kernel; 99.2% miniF2F; 70.0% PutnamBench
APOLLO	Error-localized sub-lemma repair; 84.9% miniF2F sub-8B; ∼100× sample-efficient
AutoformBot	30K Claude agents; 130K Lean lines; git as coordinator

BFS-Prover ·STP ·Lean-STaR

BFS-Prover (ByteDance, arXiv 2502.03438). Plain best-first search + DPO on kernel-labeled nodes (positive = proof closes, negative = rejected). No reward model needed: the kernel is the reward. *miniF2F 71.3%.* Foil to MCTS: the gap between MCTS and BFS is smaller than the algorithm complexity difference would suggest.

STP —Self-play Theorem Prover (Dong & Ma, arXiv 2502.00212, ICML 2025). Two roles in one model: a **conjecturer** generating frontier problems (almost but not trivially provable) and a **prover** closing them. Each role provides training signal to the other. *LeanWorkbook 28.5% ($\approx 2\times$ prior 13.2%); miniF2F 65.0%; PutnamBench 8/644* (Pass@3200).

Lean-STaR (Lin et al., arXiv 2407.10040, ICLR 2025). Retrospectively generates an informal rationale explaining each tactic choice; trains the model to write that thought before emitting the tactic. Informal information omitted by mathematicians is recoverable post-hoc and useful. *miniF2F Pass@64: 43.4% $\rightarrow$ 46.3%.*

Leanabell ·HILBERT ·APOLLO

Leanabell v1 (arXiv 2504.06122). Outcome RL with Lean kernel as reward (1/0) atop existing generators. Induces self-correction and hypothesis-refinement in the weights. *59.8% Pass@32 miniF2F.*

Leanabell v2 (arXiv 2507.08649). 7B model interleaves CoT with **live verifier calls** during a single proof; feedback-token masking stabilizes RL across human-text / verifier-output boundary. *78.2% Pass@128 miniF2F; +6.4 pt ProofNet.* "Post-training scaling": better recipe, not bigger base model.

HILBERT (Apple, arXiv 2509.22819, NeurIPS 2025 MATH-AI). Four components: informal LLM (subgoal sketches) + prover LLM (Lean tactics) + formal verifier (kernel) + semantic retriever (Mathlib lemmas). Recursive sketching: decompose → dispatch → verify → revise. *miniF2F 99.2%; PutnamBench 462/660 = 70.0%* — strongest *published* result at time of writing.

APOLLO (arXiv 2505.05758). Error-localized repair: isolate the failing sub-lemma; constrained sampling on that goal alone; verify the patch. Repair where the failure is, not across the whole proof. *miniF2F 84.9% sub-8B,* ≤ 100 *samples; o3-mini 3–7% → 40%+ with APOLLO loop.*

AutoformBot

AutoformBot (yesnoerror, arXiv 2604.03071). **30,000 Claude Opus 4.5 agents** running in parallel. Coordination layer: a shared codebase mediated by **git** —merges, branches, conflict resolution. No central planner.

- 130,000 lines of Lean; 5,900 declarations; one week wall-clock
- Inference cost reportedly comparable to a human expert team
- **Caveat:** headline "26 graduate textbooks" is unverified; the arXiv paper documents *one* textbook in detail

Aleph Prover —closed and unpublished

Headline: #1 on PutnamBench leaderboard — **668/672 = 99.4%** (January 2026). Ahead of Seed-Prover (ByteDance) and HILBERT (Apple).

Company: Logical Intelligence (`logicalintelligence.com`)

What is published (blog post only):

- **Planning** —decompose into subproblems
- **Proving** —candidate Lean proofs + kernel verify
- **Refining** —adjust strategy on subproblem outcomes
- Orchestration decoupled from any single reasoning model (paired with GPT-5.2 for PutnamBench runs)

Aleph: closed-unpublished and closed-with-paper

What is NOT published:

- No paper, no preprint, no technical report
- No model weights, no orchestrator source
- No description of search algorithm, reward model (if any), training procedure (if any), or prompt structure
- No way to verify that the 99.4% follows the same evaluation protocol as other leaderboard systems

Two distinct categories:

Closed-with-paper AlphaProof (Nature), Aristotle (arXiv). Enough structural detail that BFS-Prover, STP, HILBERT can re-derive ideas. *Inspirable.*

Closed-unpublished Aleph. A result, not a method. Leaderboard position $\neq$ reproducibility. A benchmark number with no paper cannot be reproduced or compared until a method is published.

Aleph's 99.4% is real —the kernel accepts a proof or it doesn't. Whether the methodology is replicable or scientifically informative is a different question. Right now the public answer is: not yet.

Reward design across the provers

In every system the ground-truth reward is the Lean kernel verdict. They differ in the **unit** rewarded and any **shaping** to fight sparsity.

System	Reward unit	Shaping / extra signal
AlphaProof	terminal (whole proof)	value network gives per-step estimates
DeepSeek v1.5	whole proof (GRPO)	RMaxTS intrinsic exploration reward
DeepSeek v2	per subgoal (GRPO)	earlier signal via have decomposition
Kimina-Prover	whole proof	none —pure binary verdict
Seed-Prover	whole proof (VAPO)	formatting penalty (lemma-style)
BFS-Prover	search node	DPO preference pairs, no reward model

The kernel is a sparse, binary reward; the design work is turning that into a signal the policy can learn from.

Where the trained provers stand (Pass@32)

Pass@32 (%) from the OProver paper's cross-prover sweep; rows † use those papers' own protocols. A snapshot, not a controlled comparison.

System	Params	miniF2F	MathOly	ProofNet	Putnam
Kimina-Prover-72B	72B	83.4	11.5	16.3	4.0
DeepSeek-Prover-V2-671B †	671B	82.4	13.9	30.5	3.3
Goedel-V2-32B	32B	85.8	16.0	22.0	5.0
LongCat-Flash †	∼560B	90.2	27.5	36.1	10.4
OProver-32B	**32B**	**93.3**	22.8	33.2	**11.3**

Several of the higher miniF2F entries here come from smaller models, not the biggest ones. With tree search at larger budgets, LongCat reaches 97.1% @72 and Goedel-V2-32B 92.6% @1024.

Part 7 · Synthesis

Discussion: is a dedicated prover model necessary?

Every system here is kernel-grounded. The 2027 question: should the proving ability live in a *trained* model, or in a general model driving Lean tools?

Yes — train a dedicated prover

- Clean size scaling under RL + kernel reward (Kimina 1.5B $\rightarrow$ 7B $\rightarrow$ 72B).
- Small trained models match or beat far larger ones (Goedel-V2-32B matches V2-671B).
- Tool use and repair loop trained *into the weights* (LongCat, Leanstral, OProver).

No — a general model + harness

- `numina-lean-agent` (Claude + `lean-lsp-mcp` + Leandex) proved 12/12 Putnam 2025 — no dedicated prover.
- Rides the frontier-model curve; no prover-specific RL run.
- Tooling — retrieval, verifier feedback — reusable across any base model.

Both sides have olympiad- and Putnam-scale wins in 2026, so the question is open. The harder one underneath: **autoformalization at research scale** — a paper into Lean in days, not person-years.