Maintaining a Large Formalization Project – 2

Part II: Automation & AI-Specific Challenges

Damiano Testa

University of Warwick

Tsinghua University, March 16-20, 2026

Section 1

AI Formalisation Successes

Progression

When modern AIs first became widespread, they were terrible at maths.

Last summer, four AI companies won gold medals at the IMO.

Now, they are approaching university-level questions.

With varying amounts of supervision and feedback, they are getting better at converting informal proofs into formal ones.

So far, these have been small, incremental advances built on pre-existing foundations.

AI Successes and the Human Bottleneck

AI companies have developed amazing automation and reasoning tools.

Here, I want to focus on what I see as the "human bottleneck".

Modern AI proof agents succeed thanks to

- the carefully curated code-base provided by Mathlib
- the proof scaffolding bridging the implicit informal proofs and the formalised target
- the constant supervision and redirection towards the intended target

AI takes a few steps ahead by exploiting these strong foundations.

AI proofs

The recent autoformalisation successes produced

- verbose proofs
- narrowly targeted statements
- poorly maintainable code
- essentially unusable end results

This is an accumulation of technical debt.

Such code cannot scale to CFSG-level formalisation.

The next step should be producing valuable, reusable code.

Mathlib vs CFSG

Approximate size comparison (very rough estimates):

- Mathlib contains about 5,000–10,000 pages of informal mathematics
- the CFSG spans about 10,000–20,000 pages

Autoformalisation will have to build on top of autoformalised results.

The only data point is that Mathlib-level quality enables autoformalisation.

Conclusion: scaling requires Mathlib-level quality throughout.

Atomic Lemmas and Short Proofs

Atomic lemmas are essential for building usable code.

Short proofs are a *symptom* of good quality code.

Curated lemma statements provide (auto)formalisers with the "right lemma at the right time".

Mathlib requires such polished lemmas, hence proofs can be short.

Formulating "correct" lemmas and useful abstractions is still more of an art than an automated process.

Such lemmas are essential for sustainable development that leads to CFSG.

Forward Reasoning

Common mathematical reasoning

- starts from the hypotheses
- uses them iteratively to accumulate further usable facts
- concludes once enough facts are present.

Direct translation of this proof style to Lean leads to "walls of `have`s".

This is valid code, but

- it pollutes the context with single-use assumptions
- it hides proof structure
- it hinders the identification of reusable lemmas
- it conceals internal relations

```
theorem mathd_algebra_141 (a b : ℝ) (h₁ : a * b = 180) (h₂ : 2 *
  a ^ 2 + b ^ 2 = 369 := by
  have h3 : a + b = 27 := by
    field_simp [h₂]
  have h4 : (a + b) ^ 2 = 729 := by
    rw [h3]
    norm_num
  have expand : a ^ 2 + b ^ 2 = (a + b) ^ 2 - 2 * a * b := by
    ring
  have step1 : a ^ 2 + b ^ 2 = 729 - 2 * a * b := by
    rw [expand, h4]
  have step2 : 729 - 2 * a * b = 729 - 360 := by
    rw [h₁]
  have step3 : 729 - 360 = 369 := by
    norm_num
  exact step1.trans (step2.trans step3)
```

Backward Reasoning

Lean's tactic framework allows for more efficient and clear reasoning.

Tactics such as `apply` and `refine` isolate intermediate goals that are sufficient to prove the goal.

They do not clutter the local context, but rather create a tree-like dependency graph for the proof.

Each separate goal can then be attacked independently.

The corresponding informal analogue is arguing that "to prove what we want, it suffices to show ...".

This naturally yields maintainable, idiomatic proofs.

It identifies candidates for API lemmas, since each subtree is a separate target.

Section 2

AI Failure Modes

AI Errors

In general, AI-generated definitions and statements tend to be

- more prone to misformalisations
- more inefficient
- harder to maintain

than human-written ones.

Humans tend to spot quickly if a definition or statement diverges from its intended meaning.

AI agents tend to be less aware of context and expectations.

Key Point

Alignment with intended goals should be automatically enforced.

Exploits

AI might exploit Lean's flexible metaprogramming to

- bypass kernel checking
- tampering with build artifacts
- sneakily introduce new `axiom`s

Partial protection: external checkers that replay the environment

- lean4lean
- leanchecker
- nanoda
- comparator

See the Lean Kernel Arena for more information.

Disclaimer on the Lean Kernel

There is no known soundness issue in the Lean kernel.

But Lean metaprogramming can circumvent checks and create the illusion of progress.

In a setup where there is an expectation that humans will not be able to check all the AI-generated code, this issue needs to be addressed.

Section 3

Reviewing AI Contributions

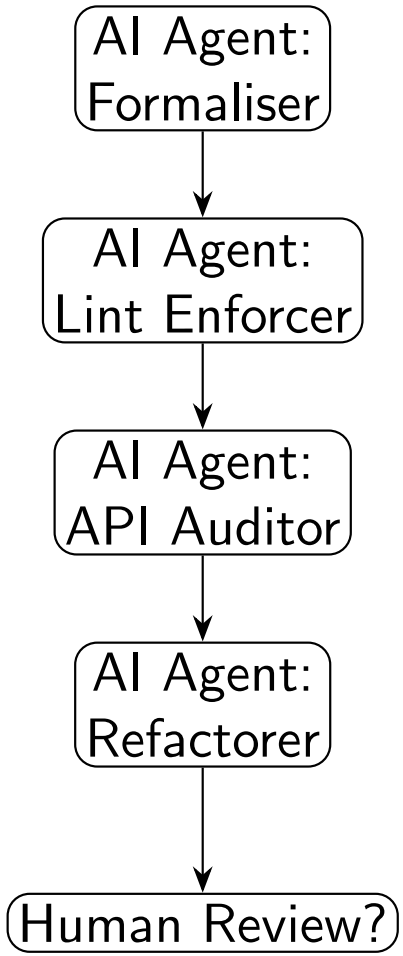
AI Reviewing AI

Different AI agents excel at:

- understanding human arguments
- formalisation
- enforcing guidelines
- suggesting improvements

Harness these different agents to review AI-generated code.

For a project of this scope, human reviews will be at a premium.



External Re-Checking

Require PRs to include provenance notes (prompt, model tag, seed/hash) for auditing.

Periodically:

- reassess reviewing criteria
- adapt to recent AI outputs
- look for common patterns, generalisations, further automation

Section 4

Automation Foundations

Automation

As we discussed so far, fully AI-driven projects need checks at all levels.

After all, what good is a proof spanning several million lines of code, if *anywhere* in the middle there is an undetected `sorry`?

This is a genuine and valid concern that must be addressed.

The "default" Lean settings prefer usability and flexibility over adversarial proof development.

This "good faith" principle can (and does) get violated by AI agents.

Section 5

Lean-Specific Automation

Lean-Specific Automation

Figure out reusable generalisations automatically

- identify common rewrites
- isolate subproofs and extract them to standalone lemmas
- explore new potential typeclasses to boost progress

Consider custom tactics — but AI-generated tactics need strict boundaries and tests.

Bad tactics are a major liability.

A more advanced version of `group` would be useful.

As far as I know, there is no example of any serious tactic having ever been written autonomously by an AI.

Abstraction and Predictability

When formalising, certain auxiliary concepts emerge.

These are often typeclasses or definitions that are not explicit in informal mathematics.

AIs should also have mechanisms for discovering such abstractions.

And they should also retain them in later iterations!

Developing new concepts benefits from predictability in:

- general definitions
- abstraction via typeclasses
- naming conventions
- careful documentation

Section 6

Environment Shaping

Helping Automation

Train agents to add `simp`, `aesop`, `grind` tags appropriately.

Some "discovery" tactics such as

- `try?`
- `exact?`
- `apply?`
- `rw?`

are more performant with smaller environments.

Environment reduction can be achieved enforcing

- consistent module structure
- semantic grouping of lemmas
- predictable API
- minimised `import`s

Section 7

Computation & Blueprinting

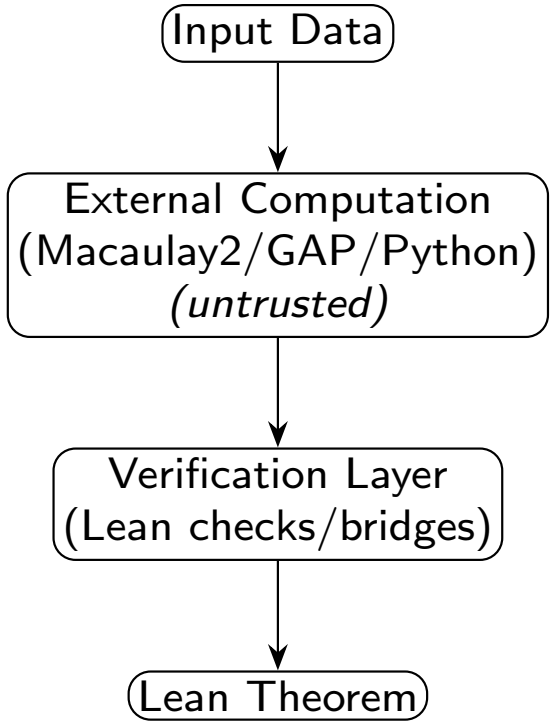
Proofs by Computation

Parts of the classification require explicit computations:

- finite case splits
- numeric inequalities
- small search spaces
- character computations

Mathlib has limited infrastructure for this currently.

The classification of finite simple groups will significantly benefit from trusted computations.



Mathematics: Blueprint

The blueprint is a bridge between the original proof and the final target.

It should ideally be part of

- the training data and
- the final formalisation.

Its history is a good indicator of successes and failures of the formalisation.

Training on Past Versions

Training AI agents to *write* good blueprints is probably a good intermediate target.

Keep the blueprint up-to-date and treat it as:

- a source of truth
- an intermediate representation mapping informal $\rightarrow$ formal goals

For AI-driven projects, past versions could still be used as training data.

Re-using code that worked well is a way of consolidating good practices.

Future iterations can then learn good practices and improve further.

Section 8

Reducing Human Burden

Support to Reduce Human Input

Linters can flag deviations from good practices, both

- locally (typically syntax linters) and
- globally (typically environment linters).

CI can integrate automatically the linter suggestions: it is important that the linter suggestions are actionable!

The "informalisation" process that goes from a maths paper intended for humans to a text that is adapted to formalisation is very laborious.

Dedicated agents that provide details, fill in gaps, suggest intermediate goals, state "formalisation-friendly" results are immensely valuable.

Human input and vetting will still be needed, but only for deeper issues.

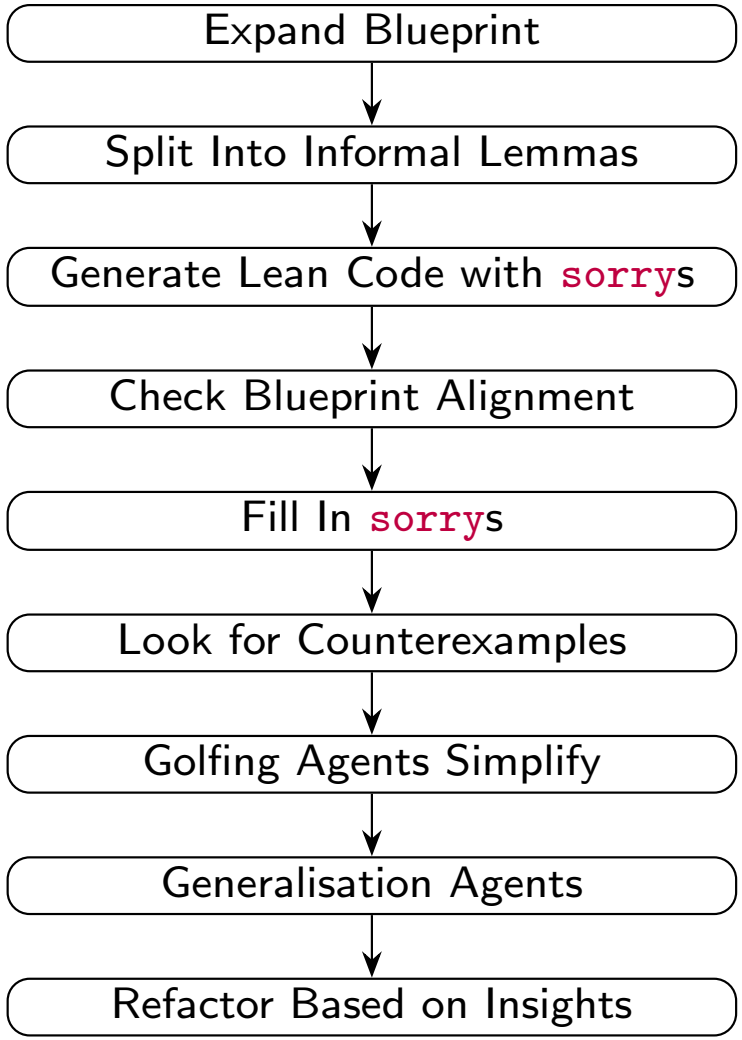
Section 9

AI Pipeline Architecture

Auto-Formalisation Skeleton Layout

Possible layered AI approach:

- Expand blueprint
- Split argument into informal lemmas
- Write formal statements with `sorry`s
- Verify alignment with blueprint
- Iterate until agreement
- Fill in `sorry`s
- Use agents to try to disprove claims
- Golfing agents simplify proofs
- Generalisation agents detect patterns
- Refactor based on insights



Thank You!

Questions?