LeanArchitect: Automating Blueprint Generation for Humans and AI

Thomas Zhu
Carnegie Mellon University, Pittsburgh, PA, USA

Pietro Monticone
University of Trento, Italy

Sean Welleck
Carnegie Mellon University, Pittsburgh, PA, USA

Jeremy Avigad
Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Large-scale formalization projects in Lean rely on blueprints: structured dependency graphs linking informal mathematical exposition to formal declarations. While blueprints are central to human collaboration, existing tooling treats the informal (LaTeX) and formal (Lean) components as largely decoupled artifacts, leading to maintenance overhead and limiting integration with AI automation. We present LeanArchitect, a Lean package for extracting, managing, and exporting blueprint data directly from Lean code. LeanArchitect introduces a declarative annotation mechanism that associates formal declarations with blueprint metadata, automatically infers dependency information, and generates LaTeX blueprint content synchronized with the Lean development. This design eliminates duplication between formal and informal representations and eases fine-grained progress tracking for both human contributors and AI-based theorem provers. We demonstrate the practicality of LeanArchitect through the automated conversion of several large existing blueprint-driven projects, and through a human–AI collaboration case study formalizing a multivariate Taylor theorem. Our results show that LeanArchitect improves maintainability, exposes latent inconsistencies in existing blueprints, and provides an effective interface for integrating AI tools into real-world formalization workflows.

2012 ACM Subject Classification Theory of computation → Interactive proof systems; Software and its engineering → Software maintenance tools

Keywords and phrases Lean theorem prover, interactive theorem proving, proof assistants, formal methods, human–computer interface, software development tools

Digital Object Identifier 10.4230/LIPIcs.ITP.2026.25

Related Version *Full Version*: `https://arxiv.org/abs/2601.22554`

Supplementary Material *Software (Source Code)*: `https://github.com/hanwenzhu/LeanArchitect` [27], archived at `swh:1:dir:b5fd29736a2e6b7dc989266fe19d489e5f7c644d`

Funding Work partially supported by the NSF AIMing program through NSF Grant DMS-2434614, the DARPA expMath program through the DARPA CMO contract number HR0011262E028, and a gift from Convergent Research.

Acknowledgements We thank Patrick Massot for developing leanblueprint and its workflows. We thank the Lean and Mathlib communities for feedback, bug reports, and suggestions that improved the tool.

Declaration about GenAI/LLM use LLMs were used only for autoformalization experiments as described in Sec. 5.3 and for proofreading the final paper.

1 Introduction

A long-term goal of formal mathematics is to represent mathematical knowledge in a precise, machine-checked language that supports both verification and large-scale collaboration. Proof assistants such as Lean [7, 8] have made substantial progress toward this goal, with libraries like Mathlib [20] formalizing much of undergraduate mathematics and an increasing body of research-level material.

As formalization projects scale, managing structure becomes as important as managing proofs. Large developments are typically decomposed into many interdependent definitions and theorems, spread across multiple files and contributors. To support this process, many Lean projects adopt *blueprints* [16]: high-level dependency graphs that describe what remains to be formalized and how results depend on one another. Blueprints serve both as planning documents and as coordination tools for distributed teams.

Existing blueprint workflows, however, exhibit two fundamental limitations. First, blueprint information is typically duplicated across an informal LaTeX document and the formal Lean code. As proofs evolve, this duplication creates significant maintenance overhead and opportunities for divergence. Second, current tooling provides limited support for integrating AI tools, despite the fact that blueprints naturally decompose formalization tasks into small, well-scoped units suitable for automation. These limitations have become more pressing as AI-based theorem provers mature. While modern systems can solve many isolated problems, real-world formalization requires reasoning about dependencies, partial progress, and natural language proofs – precisely the information encoded in blueprints. Without a native connection between blueprints and Lean code, deploying such systems in large projects remains limited in scope.

This paper introduces *LeanArchitect*, a Lean-native framework for blueprint extraction and management. LeanArchitect allows developers to annotate Lean declarations with blueprint metadata, automatically infers dependency and proof-status information, and generates synchronized LaTeX blueprint content directly from the Lean environment. By unifying the formal and informal views of a project, LeanArchitect reduces duplication, improves consistency, and provides a structured interface for AI-assisted formalization.

The contributions of this paper are:

- An annotation and extraction mechanism for blueprint nodes in Lean,
- Automatic inference of dependencies and proof status in Lean,
- Integration with existing LaTeX blueprint workflows,
- Empirical validation through conversion of large existing projects and a human–AI formalization case study.

LeanArchitect does not replace the existing leanblueprint tool [16]; rather, it complements leanblueprint by synchronizing Lean data automatically to the blueprint.

2 Related Work

2.1 Blueprints in Lean

The term blueprint originates from Patrick Massot's leanblueprint project [16], a plasTeX-based system that generates dependency graphs from LaTeX documents using macros such as `\uses` and `\leanok`. Leanblueprint has been widely adopted in non-Mathlib projects due to its effectiveness in organizing large formalization efforts (such as [2, 4, 19, 13, 9, 3]). Although leanblueprint already has a `checkdecls` command for checking whether all formal declarations

in the blueprint have a corresponding Lean part, there is no support for syncing dependency and proof status information. LeanArchitect builds on this ecosystem by providing Lean-native extraction of blueprint data, reducing manual synchronization between LaTeX and Lean.

LeanArchitect is also inspired by Ian Jauslin and Alex Kontorovich's leanblueprint-extract tool,[1] which is an extension of leanblueprint that extracts raw comments from Lean to generate the LaTeX blueprint. However, leanblueprint-extract is limited in that the user has to manually write metadata such as `\uses` dependencies, and the blueprint structure is restricted to follow the Lean code structure exactly.

2.2 Data extraction tools for Lean

The tool doc-gen4[2] generates HTML documentation directly from Lean code. While its purpose is different, LeanArchitect's code is inspired by doc-gen4. The tool ntp-toolkit [11] extracts declarations and unfinished proofs from Lean, primarily for training and testing AI tools like LeanHammer [26]. However, these systems focus on documentation or analysis rather than blueprint-driven project management, and they do not capture informal mathematical exposition or dependency structure at the level required for blueprint workflows.

2.3 AI-assisted theorem proving

Neural theorem provers have advanced rapidly, with capability increasing from simple high-school problems to complex undergraduate problems [25, 24, 22, 23, 18, 14, 15, 21, 6, 5, 1, 12]. They are increasingly tested on and integrated into real-world problems such as miniCTX [11] and formal-conjectures [10]. However, most setups consist largely of isolated problems and do not reflect the dependency-rich blueprint structure of real formalization projects. Most recently, RLMEval [17] incorporates blueprint-style information for evaluation. However, most prior work does not address the tooling required to deploy AI systems within existing blueprint-based Lean developments. LeanArchitect fills this gap by exposing blueprint structure directly in Lean, enabling AI systems to reason about partial progress, dependencies, and informal context within large projects.

3 Methods

3.1 Overview

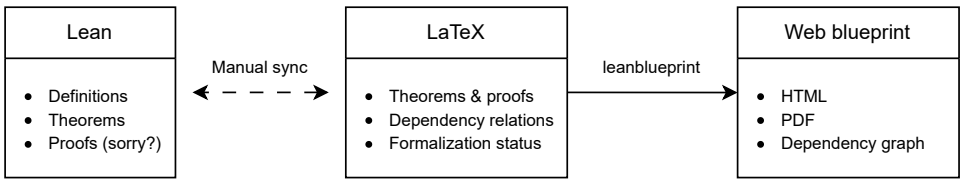
LeanArchitect is a tool for extracting blueprint information directly from Lean code. Its core design principle is to minimize duplication between LaTeX and Lean, by treating Lean as the authoritative source of information for any formalized blueprint node.

The system introduces a new attribute, `@[blueprint]`, which can be attached to Lean definitions and theorems. This attribute records metadata such as a LaTeX label, natural language statements and proofs, and project-management annotations. In addition, LeanArchitect automatically infers metadata such as:

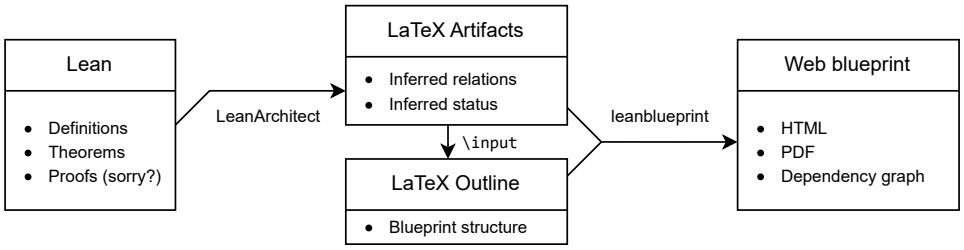
- Dependencies between definitions and theorems,
- Formalization status of the theorems (i.e., if they are `sorry`-free).

[1] See the GitHub repository.
[2] See the GitHub repository.

Blueprint data is stored in an environment extension and can be exported via a dedicated Lake facet. The exported artifacts consist of LaTeX fragments, which integrate seamlessly with existing leanblueprint documents through macros such as `\inputleannode`. The leanblueprint program then renders generated LaTeX content, including dependency relations specified by `\uses`, into a blueprint document and a dependency graph. See Figure 1 for a diagram.



(a) Blueprint generation workflow without LeanArchitect.



(b) Blueprint generation workflow with LeanArchitect.

Figure 1 Comparison of blueprint generation workflows with and without using LeanArchitect. (a) Without LeanArchitect, the entire LaTeX blueprint needs to be manually written and synchronized with the evolving formalization part. (b) With LeanArchitect, maintainers only need to manually write the structure of the LaTeX blueprint, whose dependency relations and formalization status are automatically synchronized from the corresponding Lean part.

This design supports a workflow in which informal exposition and formal proofs evolve together, while minimizing manual duplication and enabling structured interaction with AI automation.

3.2 Blueprint Attribute

LeanArchitect provides a new Lean attribute `@[blueprint]` that can be attached to definitions and theorems. Users can optionally supply metadata such as a LaTeX label, natural language statement and proof, and other project-management annotations. The attribute serves as the user interface between Lean code and the blueprint system. A typical example is:

```
@[blueprint "thm:add-comm"
  (statement := /-- Addition in $ℕ$ is commutative. -/)]
theorem MyNat.add_comm (a b : MyNat) : a + b = b + a := by
  /-- By induction and then \cref{lem:zero-add, lem:succ-add}. -/
  induction a with
  | zero => exact MyNat.zero_add b
  | succ a ih => sorry_using [MyNat.succ_add]
```

3.3 Environment Extension

After tagging a declaration, LeanArchitect constructs an internal representation of the corresponding blueprint node and stores it in an environment extension `blueprintExt`. For each tagged declaration, LeanArchitect automatically infers dependency information by recursively traversing the constants used in the type and value of the declaration, and determines proof status by checking if `sorry` is used. Then an internal `Node` is constructed; e.g. with the following data:

```
-- Identifiers
name := MyNat.add_comm, latexLabel := "thm:add-comm"
-- Statement status, dependencies, and text
statement := { leanOk := true, uses := ["def:nat", "def:nat-add"],
               text := "Addition in $ℕ$ is commutative." }
-- Proof status, dependencies, and text
proof := { leanOk := false, uses := ["lem:zero-add", "lem:succ-add"],
         text := "By induction and then \cref{lem:zero-add, lem:succ-add}." }
-- Miscellaneous metadata
notReady := false, discussion := none, title := none
```

3.4 LaTeX Export

To build the LaTeX blueprint from such data, LeanArchitect provides a build-time export mechanism integrated with Lean's Lake build system, inspired by doc-gen4. For each module, blueprint data is extracted and rendered into LaTeX fragments that can be imported into an existing blueprint document using the macro `\inputleannode{label}`. Multiple Lean declarations are allowed to correspond to a single blueprint node. The export process is deterministic and incremental. For example, when the user writes `\inputleannode{thm:add-comm}` in the LaTeX blueprint, it will be expanded to:

```
% Statement
\begin{theorem}
  \label{thm:add-comm} \lean{MyNat.add_comm}  % Identifiers
  \leanok \uses{def:nat, def:nat-add}         % Status and dependencies
  Addition in $ℕ$ is commutative.             % Text
\end{theorem}

% Proof
\begin{proof}
  \uses{lem:zero-add, lem:succ-add}           % Status and dependencies
  By induction and then                       % Text
  \cref{lem:zero-add, lem:succ-add}.
\end{proof}
```

Here `\cref` is the standard cross-reference command provided by the `cleveref` package; it does not interact with leanblueprint and belongs only to the informal proof text. In contrast, leanblueprint uses its own macro `\uses` to record dependency information, which LeanArchitect generates from the Lean development.

Finally, leanblueprint produces a PDF and web visualization of the blueprint and its dependency graph.

3.5 Project Management Using LeanArchitect

We envision the workflow of a large Lean project using LeanArchitect to be as follows.

1. Mathematicians write a detailed exposition of the formalization target in a LaTeX document (e.g. on Overleaf), following some source material.[3]
2. A new formalization project is set up (e.g. on GitHub) with the document as a blueprint, with manually labeled dependency relations using the `\uses` command.
3. Lean experts translate each theorem or definition into Lean, tagging each one with `@[blueprint]`. This translation can leave `sorry` placeholders in the code.
4. In the LaTeX blueprint, managers can then use `\inputleannode` to replace existing theorems with automatically inferred blueprint metadata.
5. Lean experts fill the `sorry`s in the code, while LeanArchitect automatically updates the metadata.

We also provide `blueprintConvert`, a script that replaces a theorem or definition in the LaTeX blueprint with `\inputleannode`, and inserts an appropriate `@[blueprint]` tag into the Lean code. This script is primarily used to automatically migrate existing leanblueprint-based projects to LeanArchitect format (see Section 5.2), but it can also automate steps 3–4 above.

4 Technical Details

Here we describe the mechanism of LeanArchitect in detail. We begin with an example Lean input and its generated blueprint, and then introduce the implementation of LeanArchitect.

4.1 Example

We begin with an example Lean snippet that defines natural numbers `MyNat`.

Listing 1 Example Lean input to LeanArchitect.

```
import Architect

@[blueprint "def:nat"]
inductive MyNat : Type where
  | zero : MyNat
  | succ : MyNat → MyNat

namespace MyNat

@[blueprint "def:nat-add"
  (statement := /-- Natural number addition. -/)]
def add (a b : MyNat) : MyNat :=
  match b with
  | zero => a
  | succ b => succ (add a b)

@[simp, blueprint "lem:zero-add"
  (statement := /-- For any natural number $a$, $0 + a = a$,
  where $+$ is \cref{def:nat-add}. -/)]
```

[3] Of course, one can also start with a Lean development, add `[blueprint]` annotations directly, and then create a LaTeX blueprint that inputs these annotations.

```
theorem zero_add (a : MyNat) : add zero a = a := by
  /-- The proof follows by induction. -/
  induction a <;> simp [*, add]

@[blueprint "lem:succ-add"
  (statement := /-- For any natural numbers $a, b$,
  $(a + 1) + b = (a + b) + 1$. -/)]
theorem succ_add (a b : MyNat) : add (succ a) b = succ (add a b) := by
  /-- Proof by induction on $b$. -/
  sorry

@[blueprint "thm:add-comm"
  (statement := /-- For any natural numbers $a, b$,
  $a + b = b + a$. -/)]
theorem add_comm (a b : MyNat) : add a b = add b a := by
  induction b with
  | zero =>
    /-- The base case follows from \cref{lem:zero-add}. -/
    -- MyNat.zero_add is used implicitly here as a [simp] lemma
    simp [add]
  | succ b ih =>
    /-- The inductive case follows from \cref{lem:succ-add}. -/
    sorry_using [succ_add]  -- the ‘sorry_using‘ tactic declares dependency

end MyNat
```

Running LeanArchitect on this example generates a blueprint. Its dependency graph[4] is visualized in Figure 2. Note the automatically inserted statements and proofs, and the relations between the nodes.

Definition 1. ✓

Natural numbers.

Definition 2. ✓

Natural number addition.

Theorem 3. ✓

For any natural number a, $0 + a = a$, where $+$ is definition 2.

Proof ▶

Theorem 4.

For any natural numbers a, b, $(a + 1) + b = (a + b) + 1$.

Proof ▼

Proof by induction on b.

Theorem 5.

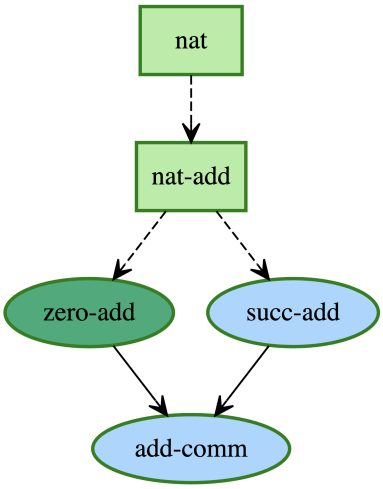
For any natural numbers a, b, $a + b = b + a$.

Proof ▼

The base case follows from theorem 3.

The inductive case follows from theorem 4.

(a) Blueprint document.



(b) Dependency graph.

Figure 2 The blueprint generated from the example Lean file (Section 4.1).

[4] See the hosted blueprint.

4.2 Implementation

We briefly introduce the implementation of LeanArchitect. At its foundation, LeanArchitect is built upon the following `Node` datatype, which represents a `@[blueprint]`-tagged Lean declaration with relevant metadata.

```
/-- The statement or proof of a node. -/
structure NodePart where
  /-- The natural language description of this part. -/
  text : String
  /-- The specified set of nodes that this node depends on, in addition to
    inferred ones. -/
  uses : Array Name
  /-- The set of nodes to exclude from ‘uses‘. -/
  excludes : Array Name
  /-- Additional LaTeX labels of nodes that this node depends on. -/
  usesLabels : Array String
  /-- The set of labels to exclude from ‘usesLabels‘. -/
  excludesLabels : Array String
  /-- The LaTeX environment to use for this part. -/
  latexEnv : String

/-- A theorem or definition in the blueprint graph. -/
structure Node where
  /-- The Lean name of the tagged constant. -/
  name : Name
  /-- The LaTeX label of the node. Multiple nodes can have the same label. -/
  latexLabel : String
  /-- The statement of this node. -/
  statement : NodePart
  /-- The proof of this node. -/
  proof : Option NodePart
  /-- The surrounding environment is not ready to be formalized, typically
    because it requires more blueprint work. -/
  notReady : Bool
  /-- A GitHub issue number where the surrounding definition or statement is
    discussed. -/
  discussion : Option Nat
  /-- The short title of the node in LaTeX. -/
  title : Option String
```

Nodes are stored in a persistent environment extension, mapping the name of each `@[blueprint]`-tagged constant to the underlying node.

```
/-- Environment extension that stores the nodes of the blueprint. -/
initialize blueprintExt : NameMapExtension Node ←
  registerNameMapExtension Node
```

A useful feature is for a blueprint “node” to correspond to multiple Lean declarations `a` and `b`. In the output, the node would have `\lean{a, b}`. To enable this feature, the user would specify the same LaTeX label for different Lean declarations (which still technically correspond to different Lean `Nodes`). We maintain a mapping from a LaTeX label to the set of nodes with such a label, as an environment extension alongside `blueprintExt`. During output, the nodes with the same label are merged to a single LaTeX environment. This allows uses such as interaction with Mathlib’s `to_additive`, an attribute that generates an additive version of a multiplicative declaration:

```
@[to_additive (attr := blueprint "comm") add_comm]
theorem mul_comm {G} [CommMagma G] (m n : G) : m * n = n * m := sorry
```

which defines single blueprint node (`comm`) corresponding to commutativity of both multiplication (`mul_comm`) and addition (`add_comm`).

We then define an attribute `@[blueprint]`. Upon tagging a Lean declaration, the attribute constructs a new `Node` with fields populated by the configuration given to `@[blueprint]`. The `@[blueprint]` attribute specifically has the following configuration options:

```
@[blueprint
  "latex-label"             -- The LaTeX label to use for the node (default: Lean
    name)
  (statement := /-- ... -/) -- The statement of the node in LaTeX
  (hasProof := true)        -- If the node has a proof part (default: true if the
    node is a theorem)
  (proof := /-- ... -/)     -- The proof of the node in LaTeX (default: the
    docstrings in proof tactics)
  (uses := [a, "b"])        -- The dependencies of the node, as Lean constants or
    LaTeX labels (default: inferred)
  (proofUses := [a, "b"])   -- The dependencies of the proof of the node, as Lean
    constants or LaTeX labels (default: inferred)
  (title := /-- Title -/)   -- The title of the node in LaTeX
  (notReady := true)        -- Whether the node is not ready
  (discussion := 123)       -- The discussion issue number of the node
  (latexEnv := "lemma")     -- The LaTeX environment to use for the node
    (default: "theorem" or "definition")
]
```

Then we add the node with the above data to `blueprintExt`.

We also define two auxiliary tactics for writing proofs that more elegantly integrate with LeanArchitect. The first is allowing docstrings to prepend a tactic, such as `/-- Proof by induction on $b$. -/ induction b with ...`. Such docstrings will be concatenated and become the "proof text" of the node. The second is `sorry_using [a, b]`, which acts like `sorry` as a proof placeholder, but also allows for specifying constants used by the would-be proof (achieving the same effect as `proofUses`).

The *blueprint content* of a module is the content of a Lean module to be converted to LaTeX, and it is defined as the ordered list of `Nodes` in the module. Users may also use `blueprint_comment /-- ... -/` to manually write raw LaTeX to the blueprint content. Suppose Lean declaration `MyNat.add_comm` is tagged with `@[blueprint "thm:add-comm"]`. We specify the rules for converting this node into LaTeX as follows.

- `\label` is `thm:add-comm`
- `\lean` is `MyNat.add_comm`
- `\uses` is the automatically inferred dependencies of the declaration, plus or minus any manually specified dependencies or exclusions. Automatic collection of used constants uses the same logic as `#print axioms` in Lean core, but it collects either dependencies tagged with `@[blueprint]` or axioms. The `\uses` of the statement are the dependencies of the declaration type (and if a definition, its value), while the `\uses` of the proof (if a theorem) are the dependencies of the declaration value.

- `\leanok` of the statement (resp. proof) is added if `sorryAx` is not in the inferred Lean dependencies above. For example, if the proof of `MyNat.add_comm` depends on `sorry`, `\leanok` is not added to the proof; but if its proof *only* depends on `MyNat.succ_add` which in turn depends on `sorry`, then `\leanok` *is* added to denote a completed proof formalization pending on an earlier unformalized lemma.
- `\mathlibok` is added to the statement (signifying the declaration is already in Mathlib) if `MyNat.add_comm` is defined in a module under `Init`, `Std`, `Batteries`, or `Mathlib`. To tag such a theorem, the user can write `attribute [blueprint] MyNat.add_comm`. This automatically differentiates if a theorem is already in Mathlib, easing management work for humans upstreaming results.
- The text of the statement is given by `(statement := /-- ... -/)` in the attribute, and the proof by `(proof := /-- ... -/)` or the proof docstrings.
- The statement title (`\begin{theorem}[title] ... \end{theorem}`) is given by `(title := /-- ... -/)`.
- Similarly for other metadata such as `\discussion` and `\notready`, which are utility features in leanblueprint.

The output LaTeX defines two macros:

- `\inputleannode{thm:add-comm}`, which expands to a node in the LaTeX blueprint with the above information, as shown in Section 3.4;
- `\inputleanmodule{MyModule}`, a higher-level command which expands to the LaTeX blueprint content of the entire module `MyModule`.

These macro definitions are saved to separate files in the LaTeX build artifacts directory `.lake/build/blueprint`. In the LaTeX blueprint, the user may import the macros and write `\inputleannode{...}` to enter a theorem in the blueprint.

Parallel to doc-gen4, LeanArchitect provides an internal `lean_exe` executable `extract_blueprint` that outputs the LaTeX artifacts of a single module to a file in `.lake/build/blueprint`. LeanArchitect then specifies a module facet[5] `blueprint` (`lake build My.Module:blueprint`), that calls this executable and registers the output file, allowing Lake's incremental build mechanism to only build the file if it is not up-to-date. We define a library facet `blueprint` (`lake build my-library:blueprint`) that outputs the main header file, and a package facet `blueprint` (`lake build :blueprint`) that runs the library facet on all libraries.

The `blueprintConvert` script primarily uses Python regular expression parsing to read the nodes in the existing blueprint, and calls Lean to obtain the locations of each node in Lean. Then it inserts `@[blueprint]` tags at appropriate locations in the code (while respecting existing attributes like `to_additive`). For a Lean declaration imported from another project (usually, from Mathlib), the script inserts `attribute [blueprint] mathlib_node` at an appropriate location: immediately before the first node (in topological order) that depends on this node, or the root file if there is no such location. Finally, the script replaces the existing `\begin{theorem}\end{theorem}` nodes in LaTeX with `\inputleannode{}`. The conversion script is invoked by a Lake script called by `lake script run blueprintConvert`. Among the customizable options are: whether to convert only the nodes with `\lean` (default) or all nodes, whether to remove the `\uses` information for `\leanok` nodes and let LeanArchitect infer it (default) or not, and formatting options for docstrings.

[5] See the Lake documentation.

5 Case Studies

5.1 PrimeNumberTheoremAnd

PrimeNumberTheoremAnd [13] is a large community-driven project aiming to formalize the prime number theorem and related results in analytic number theory in Lean. It is the first external project to adopt LeanArchitect as its primary blueprint infrastructure. The project began in 2023 and was migrated to LeanArchitect in January 2026.

Prior to migration, PrimeNumberTheoremAnd used a custom tool (*leanblueprint-extract*) that embedded blueprint information as specially formatted comments inside Lean files and extracted them into LaTeX for blueprint generation. This workflow enabled co-locating formal code and informal exposition, but required manual maintenance of metadata such as `\leanok` and `\uses`. (The design of LeanArchitect, in particular the `\inputleanmodule` mechanism, was inspired by this approach.)

Migration to LeanArchitect required only surface-level syntactic changes, primarily replacing custom comments with `@[blueprint]` attributes and `blueprint_comment` commands.[6] The initial conversion took approximately one day of work by us. After conversion, blueprint metadata – including `\lean`, `\leanok`, `\uses`, and `\mathlibok` – is now inferred automatically from Lean, eliminating a significant source of duplication and reducing maintenance overhead.

A minor source of disruption was that existing pull requests had to be manually updated by their authors to conform to the new annotation format. From the project maintainer's perspective, the impact was immediately positive. In particular, the automatic coloring of nodes and management of `\leanok` tags substantially improved the usability of the blueprint as a progress-tracking and coordination tool. As Terence Tao commented publicly on Zulip:[7]

> The auto-coloring of the blueprint and management of the \leanok tags is very pleasant from the project maintainer side of things!

Another advantage of LeanArchitect is *dependency debugging*. Tao observed that although the statement of a lemma (Erdős 392) was syntactically correct, its statement was semantically incorrect, as its AI-generated proof did not depend on surrounding lemmas. This problem was surfaced by the visualization of dependency relations automatically inferred by LeanArchitect.

Overall, this case study demonstrates that LeanArchitect can be adopted in an active, large-scale project, preserve existing authoring workflows, and improve synchronization and maintainability of blueprint metadata with small migration cost.

5.2 Converting Existing Projects to LeanArchitect Format

We tested the conversion script on the following projects into LeanArchitect, at Lean version `v4.25.0`:

1. Carleson [2]
2. Brownian Motion [9]
3. Infinity Cosmos [19]
4. Fermat's Last Theorem [4]
5. Prime Number Theorem And [13]

[6] See the GitHub pull request.
[7] See the Zulip comment.

For each project, we first added LeanArchitect to the lakefile, and then ran the conversion script. For the most part, the script is able to automatically convert blueprint nodes into Lean `@[blueprint]` attributes.

During conversion, we found some discrepancies between the converted blueprint and the original one. This revealed some issues with the original manually written blueprint that could be uncovered and automatically fixed by LeanArchitect, such as:

- Isolated nodes that were actually not used, such as a layer-cake theorem `eLpNorm_pow_eq_distribution` not used in Carleson (a related theorem `eLpNorm_eq_distribution` was used instead)
- Missing dependency edges, such as `internalCoveringNumber_eq_one_of_diam_le` incorrectly marked as unused in Brownian Motion
- Theorems in Mathlib that should have been marked `\mathlibok`
- Oversimplified setups, such as Infinity Cosmos only applying `\leanok` and `\uses` on statements and not proofs of theorems.

In general, the conversion script successfully converts projects into LeanArchitect format. As LeanArchitect automates the previously manual specification of `\leanok` and `\uses`, it ensures the blueprint is in sync with the Lean formalization status.

5.3 Blueprint-Based Autoformalization

We evaluate LeanArchitect as an interface for human–AI collaboration by semi-automatically formalizing a theorem that is both technically nontrivial and naturally decomposable: multivariate Taylor's theorem in integral form. At the time of writing, Mathlib contains Taylor's theorem only in one-dimensional settings and does not have the multivariate Fréchet-derivative formulation in integral form.

5.3.1 Target theorem

Let E, F be normed vector spaces over $\mathbb{R}$ with F complete. Let $U \subseteq E$ be open and let $f : U \to F$ be C^n with $n > 0$. For $x, h \in E$ such that $[x, x + h] \subseteq U$, we aim to formalize:

$$f(x+h) = \sum_{k=0}^{n-1} \frac{1}{k!} D^k f(x)[h, \ldots, h] + \frac{1}{(n-1)!} \int_0^1 (1-s)^{n-1} D^n f(x+sh)[h, \ldots, h] \, \mathrm{d}s,$$

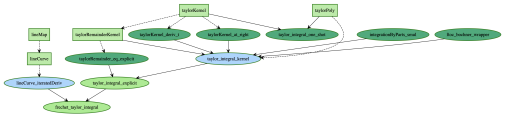
where $D^k f$ denotes the k-th Fréchet derivative.

5.3.2 Pipeline

Our pipeline uses two AI systems with complementary roles: GPT-5 Pro drafts a natural-language proof and a corresponding Lean *blueprint file* (a single Lean module annotated with `@[blueprint]`), and Aristotle operates in *sorry-filling* mode to attempt to complete the resulting proof obligations. Concretely:

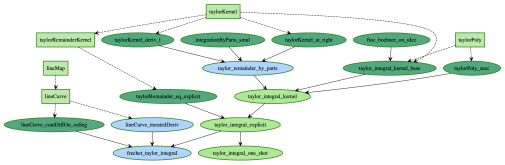
1. GPT-5 Pro produces a structured proof outline and translates it into a Lean file whose intermediate lemmas are tagged with `@[blueprint]` and initially proved with `sorry`.
2. We visualize progress using the generated blueprint dependency graph.
3. Aristotle (sorry-filling mode) attempts to discharge the `sorry` nodes automatically.
4. Remaining failures are iteratively addressed by refining the blueprint using GPT-5 Pro.
5. Finally, a human manually proves remaining parts that could not be automatically proved.

(a) GPT-5 Pro: initial blueprint.

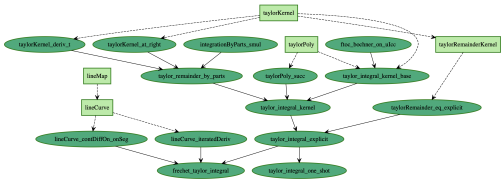


(b) Aristotle: fill proofs in initial blueprint.

(c) GPT-5 Pro: refined blueprint.



(d) Aristotle: fill proofs in refined blueprint.



(e) Completed formalization by human.

Figure 3 Progress of blueprint-based autoformalization for multivariate Taylor's theorem. Blue nodes are unproved lemmas; green nodes are proved lemmas. Rectangles denote definitions; ellipses denote theorems.

5.3.3 Progress visualization

Figure 3a shows the initial blueprint produced from GPT-5 Pro's Lean draft, and Figure 3b shows the result after Aristotle's first pass. We then refined the blueprint based on the remaining unproved nodes (Figures 3c and 3d). After refinement, Aristotle proved all but three nodes.

Inspection of the remaining failures revealed a systematic issue in the initial formalization: one should use `derivWithin` rather than `deriv` for a function $f : \mathbb{R} \to E$ differentiable on $[a, b]$. We corrected the affected statements manually, making intermediate lemmas true, and then proved the remaining lemmas by hand.

5.3.4 Outcome

This proves the target theorem in Lean:

```
import Mathlib

open scoped Nat

variable {E : Type*} [NormedAddCommGroup E] [NormedSpace ℝ E]
variable {F : Type*} [NormedAddCommGroup F] [NormedSpace ℝ F] [CompleteSpace F]

theorem frechet_taylor_integral
  {U : Set E} {f : E → F} {n : ℕ} (hn : 0 < n)
  (hCn : ContDiffOn ℝ n f U) (hU : IsOpen U)
  {x h : E} (hseg : ∀ s ∈ Set.Icc (0 : ℝ) 1, x + s · h ∈ U) :
  f (x + h) =
  (Σ k ∈ Finset.range n, ((k)! : ℝ)⁻¹ · iteratedFDeriv ℝ k f x fun _ ↦ h) +
  ((n - 1)! : ℝ)⁻¹ · ∫ s in (0 : ℝ)..1,
    (1 - s) ^ (n - 1) · iteratedFDeriv ℝ n f (x + s · h) fun _ ↦ h
```

This case study highlights two advantages of blueprint-structured autoformalization. First, failures are localized: rather than treating the attempt as an all-or-nothing proof search, we can focus human attention on a small set of remaining nodes identified in the dependency graph. Second, progress is transparent and compositional: partial success yields reusable intermediate lemmas, and the dependency structure provides a natural unit of work for automated provers. More broadly, the blueprint acts as a shared interface between humans and automation: humans refine the decomposition and hypotheses, while automated systems attempt the resulting subgoals.

We also observed a limitation of the current human–AI loop: simply asking the language model to "refine" the blueprint did not reliably address the underlying causes of failure, partly because the prover provides limited diagnostic feedback beyond a list of unsolved nodes. Closing this loop will likely require richer feedback signals (e.g. counterexamples, missing lemma suggestions, or structured failure traces) that can be consumed by the planner model.

6 Limitations

LeanArchitect introduces a new integration point between Lean and blueprint workflows and has several limitations. First, authoring LaTeX inside Lean files is currently unergonomic: mainstream Lean IDE support provides neither syntax highlighting nor interactive feedback for embedded LaTeX in `.lean` files, and existing keybindings interfere with natural LaTeX input. Second, LeanArchitect supports a many-to-one correspondence from Lean declarations to blueprint nodes: multiple Lean declarations may correspond to the same blueprint node, but a given Lean declaration cannot correspond to more than one blueprint node, which limits expressiveness in cases where the same lemma appears in the blueprint multiple times. Third, the additional build steps of LeanArchitect deepen the build pipeline and increase the possibility for build and CI failures. Finally, blueprint nodes that exist only in the informal LaTeX layer must still be managed manually until a corresponding Lean declaration is introduced. These limitations primarily reflect tooling and infrastructure gaps rather than fundamental design constraints.

References

1 Tudor Achim, Alex Best, Alberto Bietti, Kevin Der, Mathïs Fédérico, Sergei Gukov, Daniel Halpern-Leistner, Kirsten Henningsgard, Yury Kudryashov, Alexander Meiburg, et al. Aristotle: IMO-Level Automated Theorem Proving. *arXiv preprint*, 2025. doi:10.48550/arXiv.2510.01346.

2 Lars Becker, María Inés de Frutos-Fernández, Leo Diedering, Floris van Doorn, Sébastien Gouëzel, Asgar Jamneshan, Evgenia Karunus, Edward van de Meent, Pietro Monticone, Jasper Mulder-Sohn, et al. A Blueprint for the Formalization of Carleson's Theorem on Convergence of Fourier Series. *arXiv preprint*, 2025. doi:10.48550/arXiv.2405.06423.

3 Matthew Bolan, Joachim Breitner, Jose Brox, Nicholas Carlini, Mario Carneiro, Floris van Doorn, Martin Dvorak, Andrés Goens, Aaron Hill, Harald Husum, Hernán Ibarra Mejia, Zoltan A. Kocsis, Bruno Le Floch, Amir Livne Bar-on, Lorenzo Luccioli, Douglas McNeil, Alex Meiburg, Pietro Monticone, Pace P. Nielsen, Emmanuel Osalotioman Osazuwa, Giovanni Paolini, Marco Petracci, Bernhard Reinke, David Renshaw, Marcus Rossel, Cody Roux, Jérémy Scanvic, Shreyas Srinivas, Anand Rao Tadipatri, Terence Tao, Vlad Tsyrklevich, Fernando Vaquerizo-Villar, Daniel Weber, and Fan Zheng. The Equational Theories Project: Advancing Collaborative Mathematical Research at Scale. *arXiv preprint*, 2025. doi:10.48550/arXiv.2512.07087.

4 Kevin Buzzard and Richard Taylor. FLT: An Ongoing Lean Formalisation of the Proof of Fermat's Last Theorem, 2023. URL: https://github.com/ImperialCollegeLondon/FLT.

5 Jiangjie Chen, Wenxiang Chen, Jiacheng Du, Jinyi Hu, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Wenlei Shi, et al. Seed-Prover 1.5: Mastering Undergraduate-Level Theorem Proving via Learning from Experience. *arXiv preprint*, 2025. doi:10.48550/arXiv.2512.17260.

6 Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, et al. Seed-Prover: Deep and Broad Reasoning for Automated Theorem Proving. *arXiv preprint*, 2025. doi:10.48550/arXiv.2507.23726.

7 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In *Automated Deduction – CADE-25*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015. doi:10.1007/978-3-319-21401-6_26.

8 Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi:10.1007/978-3-030-79876-5_37.

9 Rémy Degenne, David Ledvinka, Etienne Marion, and Peter Pfaffelhuber. Formalization of Brownian Motion in Lean. *arXiv preprint*, 2025. doi:10.48550/arXiv.2511.20118.

10 Google DeepMind. Formal Conjectures: A Collection of Formalized Statements of Conjectures in Lean, 2025. URL: https://github.com/google-deepmind/formal-conjectures.

11 Jiewen Hu, Thomas Zhu, and Sean Welleck. miniCTX: Neural Theorem Proving with (Long-)Contexts. In *The Thirteenth International Conference on Learning Representations*, 2025. URL: https://openreview.net/forum?id=KIgaAqEFHW.

12 Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z. Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, et al. Olympiad-Level Formal Mathematical Reasoning with Reinforcement Learning. *Nature*, 2025. doi:10.1038/s41586-025-09833-y.

13 Alex Kontorovich and Terence Tao. Prime Number Theorem and More, 2024. URL: https://github.com/AlexKontorovich/PrimeNumberTheoremAnd.

14 Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, et al. Goedel-Prover: A Frontier Model for Open-Source Automated Theorem Proving. *arXiv preprint*, 2025. doi:10.48550/arXiv.2502.07640.

15 Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, et al. Goedel-Prover-V2: Scaling Formal Theorem Proving with Scaffolded Data Synthesis and Self-Correction. *arXiv preprint*, 2025. doi:10.48550/arXiv.2508.03613.

16 Patrick Massot. leanblueprint: plasTeX Plugin to Build Formalization Blueprints, 2020. URL: https://github.com/PatrickMassot/leanblueprint.

17 Auguste Poiroux, Antoine Bosselut, and Viktor Kunčak. RLMEval: Evaluating Research-Level Neural Theorem Proving. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, 2025. doi:10.48550/arXiv.2510.25427.

18 Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, et al. DeepSeek-Prover-V2: Advancing Formal Mathematical Reasoning via Reinforcement Learning for Subgoal Decomposition. *arXiv preprint*, 2025. doi:10.48550/arXiv.2504.21801.

19 Emily Riehl and Dominic Verity. Infinity Cosmos: A Blueprint for a Formalization of Infinity-Cosmos Theory in Lean, 2024. URL: https://github.com/emilyriehl/infinity-cosmos.

20 The mathlib Community. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP '20, pages 367–381. ACM, 2020. doi:10.1145/3372885.3373824.

21 Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, et al. Kimina-Prover Preview: Towards Large Formal Reasoning Models with Reinforcement Learning. *arXiv preprint*, 2025. doi:10.48550/arXiv.2504.11354.

22 Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. DeepSeek-Prover: Advancing Theorem Proving in LLMs through Large-Scale Synthetic Data. *arXiv preprint*, 2024. doi:10.48550/arXiv.2405.14333.

23 Huajian Xin, Z. Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, et al. DeepSeek-Prover-V1.5: Harnessing Proof Assistant Feedback for Reinforcement Learning and Monte-Carlo Tree Search. *arXiv preprint*, 2024. doi:10.48550/arXiv.2408.08152.

24 Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J. Prenger, and Animashree Anandkumar. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. *Advances in Neural Information Processing Systems*, 36:21573–21612, 2023.

25 Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. miniF2F: A Cross-System Benchmark for Formal Olympiad-Level Mathematics. *arXiv preprint*, 2021. doi:10.48550/arXiv.2109.00110.

26 Thomas Zhu, Joshua Clune, Jeremy Avigad, Albert Qiaochu Jiang, and Sean Welleck. Premise Selection for a Lean Hammer. In *The Fourteenth International Conference on Learning Representations*, 2026. URL: https://openreview.net/forum?id=m04JJNeRK6.

27 Thomas Zhu, Pietro Monticone, Sean Welleck, and Jeremy Avigad. Lean Architect supplementary material. Software, swhId: swh:1:dir:b5fd29736a2e6b7dc989266fe19d489e5f7c644d (visited on 2026-07-13). URL: https://github.com/hanwenzhu/LeanArchitect, doi:10.4230/artifacts.27122.