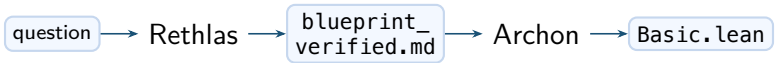
Using Rethlas and Archon End to End

Euler's partition theorem, from a natural-language proof to a Lean proof

June 2026

One theorem, two tools

Euler's partition theorem runs through two tools in sequence. Rethlas writes a natural-language proof; Archon turns that same proof into Lean code the Lean kernel checks.



- **Rethlas** — an LLM verifier reads the proof and judges it.
- **Archon** — the Lean kernel checks the proof and cannot be fooled.

Same loop — generate → verify → repair → accept — different verifier.

Euler's partition theorem

A **partition** of n writes n as a sum of positive integers, order ignored.

- **distinct parts:** all summands different. $D(n)$ counts them.
- **odd parts:** every summand odd. $O(n)$ counts them.

Theorem (Euler). For every n, $D(n) = O(n)$.

Worked case $n = 6$ (both counts equal 4):

distinct	6	$5{+}1$	$4{+}2$	$3{+}2{+}1$
odd	$5{+}1$	$3{+}3$	$3{+}1{+}1{+}1$	$1{+}1{+}1{+}1{+}1{+}1$

Mathlib proves $D(n) = O(n)$, but through generating functions — not a bijection and not computable. The goal here is the **constructive** proof.

The bijection: binary multiplicities and 2-adic factorization

Write a partition into odd parts by the multiplicity $m(a)$ of each odd part a.

- Φ (odd $\to$ distinct): expand each multiplicity in binary, $m(a) = \sum_{r \geq 0} \varepsilon_r 2^r$; for every set bit r, emit the distinct part $2^r \cdot a$.
- Ψ (distinct $\to$ odd): every positive x factors uniquely as $x = 2^r \cdot a$ with a odd; replace x by 2^r copies of a.

Example ($n = 8$): odd part 1 with multiplicity $5 = 101_2 \to$ parts $2^0 \cdot 1 = 1$ and $2^2 \cdot 1 = 4$; odd part 3 with multiplicity $1 \to$ part 3. So Φ gives $\{1, 4, 3\}$ (sum 8), and Ψ inverts it.

Φ and Ψ are inverse: that is the whole proof.

Stage 1 · Rethlas

Rethlas writes `blueprint_verified.md`

You drive a coding agent with prompts: the prompt poses the question; Rethlas writes the problem notes and runs its own generate $\rightarrow$ verify $\rightarrow$ repair loop.

Prompt — paste to Claude:

```
Investigate this and produce a self-contained proof: Euler's partition
theorem -- the number of partitions of n into DISTINCT parts equals the
number into ODD parts. Set up and run Rethlas with external search
disabled; then report the verified blueprint and the verdict.
```

```
results/<id>/blueprint_verified.md   the accepted natural-language proof
results/<id>/verification.json       the verifier's verdict
```

`blueprint_verified.md` holds the bijection above: the 2-adic lemma, Φ, Ψ, well-definedness, and the two round trips.

Rethlas's verdict is an AI judgment

The verifier is an LLM reading the proof. The verdict in `verification.json` is its judgment.

- It is **not** a Lean kernel certificate.
- A plausible-but-wrong step can slip through.

This is the gap Stage 2 closes: take the same proof and make the Lean kernel check it.

Stage 2 · Archon

Archon: Claude Code writing Lean, checked by the Lean kernel

Archon is a loop that launches Claude Code to write Lean, then runs the **Lean kernel** — Lean's small trusted proof checker — to check it.

- a **plan agent** decides what to prove and how to split it;
- **prover agents** write the tactics, one `.lean` file each;
- the Lean kernel (run via `lake env lean`) is the verifier — it cannot be talked into accepting a wrong proof.

You do not run it by hand. You give Claude one prompt; Claude installs Archon, sets up the project, and runs the loop (`archon init`, `archon loop`) for you — the same way you drove Rethlas in Stage 1.

Prompt: formalize the blueprint with Archon

You drive Stage 2 with one prompt; Claude does the installs, the setup, and the run.

Prompt — give to Claude:

```
Formalize this natural-language proof (blueprint_verified.md) in Lean 4
using Archon. Set up a Lean project with Mathlib, install and initialize
Archon pointed at the blueprint, then run the loop until the Lean kernel
accepts (no sorry). Report progress and the final result, and check
#print axioms at the end.
```

Claude then runs `archon setup` / `archon init .` / `archon loop .` and monitors the run.

Prompt: the constructive constraint

Archon's default rule is "prefer existing Mathlib" — left alone it would call the generating-function theorem and finish in one line. So the prompt must pin the constraint (Claude passes it to Archon via `USER_HINTS.md`):

Prompt — add to the instruction:

```
Constraints:
1. Constructive bijection only: build Phi/Psi (binary multiplicities <->
   2-adic factorization) and prove them mutually inverse.
2. FORBIDDEN: Mathlib's generating-function proof (Glaisher /
   card_odds_eq_card_distincts), PowerSeries, any lemma stating #distinct=#odd.
3. Computable: every map is a plain `def` (no noncomputable, no Classical
   inside the maps); Phi, Psi, D, O must #eval.
```

Claude runs the loop: plan → prove → review (the Euler run)

Each iteration:

- **plan** — reads `PROGRESS.md` + hints, writes objectives per file
- **prove** — one prover per file: draft tactics, compile, read the Lean kernel via `lean-lsp-mcp`, repair
- **review** — writes the journal; the loop advances the stage

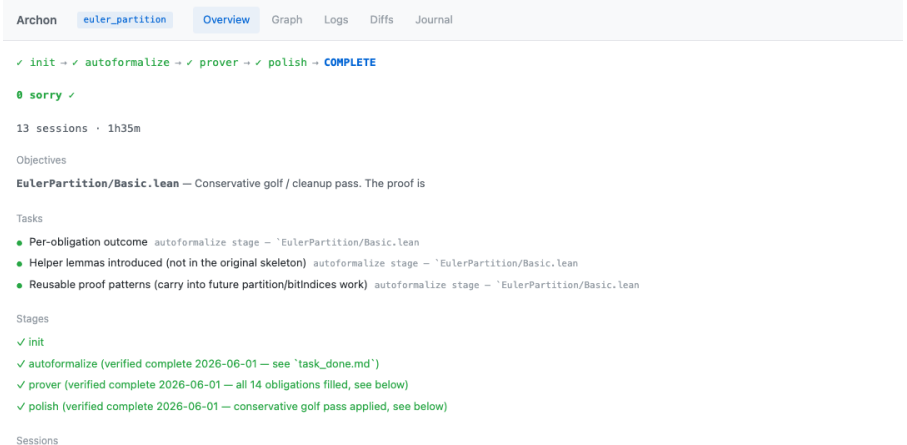
Stops when `PROGRESS.md` reaches `COMPLETE` (no `sorry` left).

```
iter 1  autoformalize:
   skeleton, 14 obligations
   left as sorry; compiles
   D 6, O 6 already eval to 4
iter 2  prove:
   filled all 14 in one pass
iter 3–4  polish:
   golf; stage -> COMPLETE

4 iters, ~61 min, ~$29 (Opus)
sorry: 14 -> 0
```

The dashboard shows the proof progress

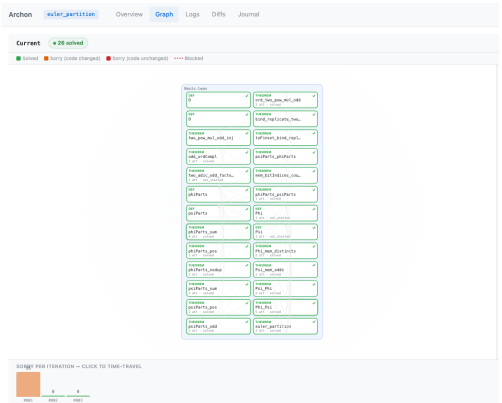
Archon serves a web dashboard while it runs — you watch it instead of typing.



The `euler_partition` run: `init` → `autoformalize` → `prover` → `polish` → `COMPLETE`, **0 sorry**, all 14 obligations filled (13 sessions, ∼1h35m).

The proof dependency graph

The dashboard's Graph view: each declaration is a node, edges are dependencies.



26 declarations, all solved (green) — from the 2-adic lemma up to `euler_partition`; bars = `sorry` per iteration, ending at 0.

The result: `euler_partition` via `Finset.card_nbij'`

```
def D (n : ℕ) : ℕ := (Nat.Partition.distincts n).card   -- Mathlib's defs
def O (n : ℕ) : ℕ := (Nat.Partition.odds n).card

theorem euler_partition (n : ℕ) : D n = O n := by
  unfold D O
  exact Finset.card_nbij' Psi Phi
    (fun p hp => Psi_mem_odds hp)        -- Ψ : distinct → odd
    (fun p hp => Phi_mem_distincts hp)   -- Φ : odd → distinct
    (fun p hp => Phi_Psi hp)             -- Φ ∘ Ψ = id
    (fun p hp => Psi_Phi hp)             -- Ψ ∘ Φ = id
```

`Finset.card_nbij'` = equal size from a forward map, an explicit inverse, and proofs each lands in the other set — the four obligations, no counting. The statement uses Mathlib's own `distincts`/`odds`, so it is genuinely Euler's theorem; only the proof is new.

Verification: `#print axioms` and `#eval`

`Basic.lean`'s own `#eval` lines:

```
#eval D 6, O 6                -> 4, 4
#eval phiParts {1,1,1,1,1,3}  -> {1, 4, 3}
#eval psiParts {4,2}          -> {1, 1, 1, 1, 1, 1}
```

Independent checks (a scratch file importing the proof):

```
lake build                      -> clean (0 sorry / 0 admit)
#print axioms euler_partition  -> {propext, Classical.choice, Quot.sound}
#eval (List.range 13).map (n=>(D n,O n)) -> D n = O n, n = 0..12;
      D = 1,1,1,2,2,3,4,5,6,8,10,12,15   (OEIS A000009)
```

No `sorryAx`. `Classical.choice` is from the *proof* (as in most Mathlib lemmas); the *maps* are plain `def`s and `#eval`. Constructive and computable, not just existence.

The same theorem, two verdict artifacts

Rethlas

- `blueprint_verified.md` + `verification.json`
- a natural-language proof an LLM verifier accepted
- trust = the verifier's judgment (can be wrong)

Archon

- `Basic.lean` + `#print axioms`
- a Lean proof the Lean kernel checked, with maps that compute
- trust = the Lean kernel + three standard axioms

Same loop, different verifier. Rethlas tells you a proof probably exists; Archon hands you one the Lean kernel certifies.

The whole pipeline is two prompts

You never run a tool by hand; you send Claude two prompts.

```
1. (Stage 1, Rethlas) Investigate Euler's partition theorem and produce a
   self-contained proof; run Rethlas with external search disabled.
      -> blueprint_verified.md   (an LLM verifier accepted it)

2. (Stage 2, Archon) Formalize blueprint_verified.md in Lean 4 with Archon:
   constructive bijection, no generating functions, computable; run until
   the Lean kernel accepts.
      -> EulerPartition/Basic.lean   (0 sorry; Lean-kernel-checked)
```

Claude does the installs, the setup, the runs, and the monitoring; you read the dashboard and the final `#print axioms`.