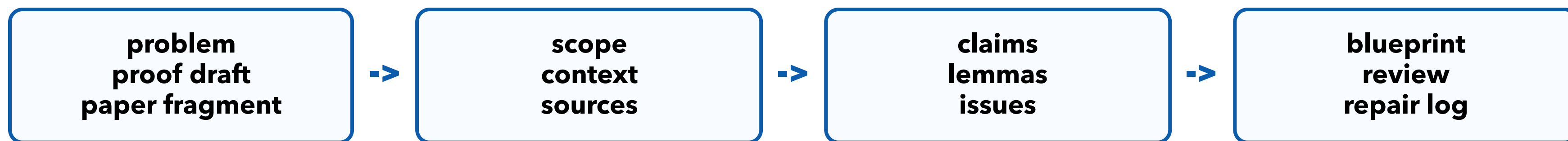
Natural-Language Proof Systems

From problem text to scoped claims, dependencies, checks, and repair records.

Objects In A Proof-Review Workflow



The intermediate records are part of the object: scope, sources, claims, checks, and repairs.

Access

definitions, source spans, theorem search, examples, counterexamples

Structure

claim nodes, dependency edges, issue records, repair history

Agents

generator, verifier, referee, librarian, coordinator

Proof Job Scope

Unscoped prompt

```
Review the proof of theorem T.
```

- which proof?
- which background?
- what level of detail?
- can external sources be used?
- is this generation, review, or learning?

Scoped proof job

```
Target: theorem T
Mode: blueprint + review
Proof route: supplied by source or selected by t
Sources: provided notes and references
Verifier: local informal checker
Output: DAG, gaps, repair log
```

- worker roles are listed
- context packets have explicit fields
- review standard is explicit

Single Chat And Agentic Calls

One transcript

- one growing transcript
- proof plan, sources, verification, and repair mix together
- implicit memory
- unclear tool boundary
- hard to reproduce exact state

Separate worker calls

- job packet
- explicit role
- specific working directory
- limited tools
- structured output and logs

A worker call can be logged as: prompt, files, retrieved context, tool policy, output schema, and result.

Context And Context Injection

Context

The information made available to a model or worker during one call.

- prompt text
- files in the working directory
- tool outputs
- memory records
- source spans and schemas

Context injection

The act of selecting and inserting specific information into a worker call.

- which statement is sent
- which proof text is sent
- which sources are included
- which previous failures are visible
- which output format is required

Rethlas Verification Context

In Rethlas original, the generator calls the MCP tool:

```
verify_proof_service(statement, proof)
```

The verification HTTP service injects those fields into the Codex verifier prompt:

```
Run_id: <run_id>.
Statement: <statement>.
Proof:
<proof>

Use AGENTS.md to verify the above proof for the statement.
```

Source: `agents/generation/mcp/server.py` and `agents/verification/api/server.py` in [frenzymath/Rethlas](https://github.com/frenzymath/Rethlas).

Rethlas Verifier Agent Contract

From `agents/verification/AGENTS.md`:

Input	`Run_id`, informal theorem `Statement`, and markdown `Proof`.
Reading order	Verify statements and subproofs sequentially in the order written.
Checks	logical validity, theorem application, missing assumptions, unjustified jumps.
External references	search theorem text first; compare definitions, terminology, and hypotheses.
Verdict rule	`correct` iff there are zero critical errors and zero gaps; otherwise `wrong`.
Output	write `results/{run_id}/verification.json` with report, verdict, and repair hints.

Source: `agents/verification/AGENTS.md` in [frenzymath/Rethlas](https://github.com/frenzymath/Rethlas).

Claude As Callable Worker

```
claude -p "review this proof"
claude --output-format json -p "return a JSON review"
claude --json-schema referee.schema.json -p "emit referee_report_v1"
claude --agent reviewer -p "check this proof"
claude --agents '{"reviewer":{"description":"Reviews proofs","prompt":"You are a proof reviewer"}}'
claude --add-dir /path/to/kb -p "inspect the node library"
```

Role: reviewer, retriever, repair worker

Boundary: directory access and tool policy

Output: text, JSON, stream JSON, schema-constrained JSON

Permissions: broad directory access changes the job boundary

Codex As Callable Worker

```
printf '%s\n' "$PROOF_PACKET" | codex exec --sandbox read-only -
codex exec -C /path/to/project "verify the proof in proof.md"
codex exec --json "emit structured events"
codex exec --output-schema referee.schema.json "produce review JSON"
codex exec -o verifier-report.md "write the final review"
codex review --uncommitted
codex mcp-server
```

Agent calls Codex

A coordinator can pass a proof packet through stdin or a prompt.

Context boundary

`-C`, `--add-dir`, stdin, and sandbox mode specify what Codex can see and do.

Coordination surface

`--json`, `--output-schema`, `-o`, and `mcp-server` make results machine-readable.

OpenCode As Callable Worker

```
opencode run "review this proof"
opencode run --agent reviewer "check the proof DAG"
opencode run --format json "emit structured review"
opencode run --dir /path/to/project "inspect this workspace"
opencode serve
opencode attach http://localhost:4096
opencode agent list
opencode agent create
```

> For a proof system, the CLI is a worker interface. The system is the coordinator, memory, tools, graph state, and review protocol around it.

JSON Is Structured Text

RFC 8259 defines JSON as a text format for structured data exchange.

```json
{
  "verdict": "wrong",
  "critical_errors": [
    "uses a target claim before it has been established"
  ],
  "gaps": [
    "missing minimal counterexample argument"
  ],
  "repair_hints": [
    "isolate the missing lemma and recheck the dependent step"
  ]
}
```

Object: `{ "name": value }` pairs with string names.

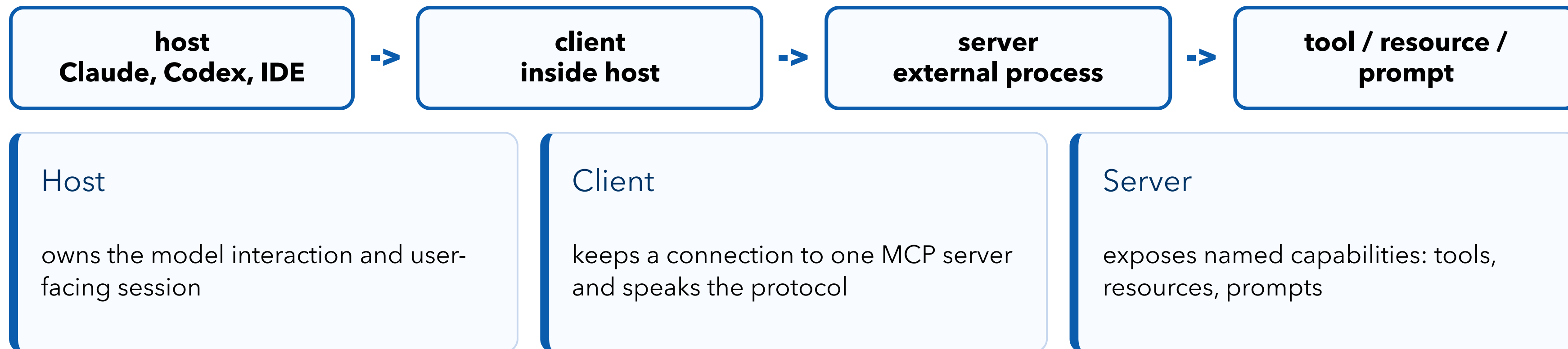
Array: ordered list: `[value, value, ...]`.

Values: string, number, object, array, `true`, `false`, or `null`.

Source: [RFC 8259: The JavaScript Object Notation Data Interchange Format](https://www.rfc-editor.org/rfc/rfc8259).

MCP Architecture

MCP means Model Context Protocol.



Source: [MCP architecture specification](https://modelcontextprotocol.io/specification/2025-06-18/architecture).

MCP Calls Are JSON-RPC Messages

The MCP architecture specification says MCP is built on JSON-RPC.

```
{
  "jsonrpc": "2.0",
  "id": 17,
  "method": "tools/call",
  "params": {
    "name": "verify_node",
    "arguments": {
      "statement": " ... ",
      "proof": " ... ",
      "allowed_dependencies": ["matching definitions"]
    }
  }
}
```

Method	which operation is being requested
Params	the structured input packet
Id	links the response to this request
Result / error	the server's structured answer

Sources: [MCP architecture specification](https://modelcontextprotocol.io/specification/2025-06-18/architecture), [JSON-RPC 2.0 specification](https://www.jsonrpc.org/specification).

MCP Makes Proof Tools Callable

For a proof system, an MCP server can expose proof-specific operations.

Tools

`verify_proof`, `search_theorems`, `graph_query`

Resources

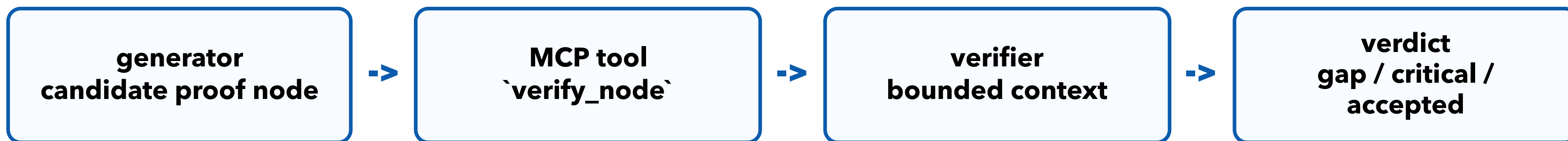
files, source spans, node documents, reports

Prompts

role contracts and reusable job templates

MCP is not a proof method. It is an interface layer for controlled tool and context access.

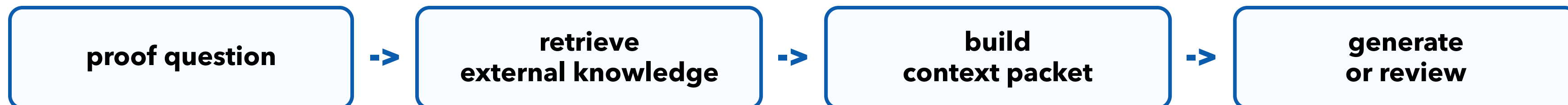
Verifier Through MCP



Generator sends	statement, proof attempt, allowed dependencies, source packet, context hash
Verifier sees	only the node to check and explicit supporting material
Coordinator records	job id, input hash, budget, output schema, log location

Retrieval-Augmented Generation

RAG means retrieval-augmented generation.



In this lecture, we use RAG for the part of a proof/review system that retrieves material before an agent writes or checks a response.

Mathematical RAG Retrieves Objects

Definitions

notation, hypotheses, local conventions

Theorems

complete statement, assumptions, source

Proof steps

where a construction or reduction is justified

Examples

small models that test a claim

Counterexamples

evidence that a proposed lemma is false

Reports

earlier gaps, failed paths, repair hints

For theorem use, the retrieved record should include the statement, hypotheses, local definitions, and source.

RAG Tool Layers

Layer	Role	Examples
Parsing	extract PDF, TeX, HTML, Markdown	PyMuPDF, GROBID, Unstructured
Sparse search	exact names, labels, notation	Elasticsearch, OpenSearch
Vector search	semantic similarity	FAISS, Qdrant, Milvus, Weaviate, pgvector
Theorem search	mathematical statement retrieval	LeanSearch-style systems
Reranking	choose actually relevant evidence	cross-encoder or LLM rerankers
Orchestration	loaders and context assembly	LlamaIndex, LangChain
Graph retrieval	dependency-aware retrieval	Kuzu, Neo4j, ArangoDB, GraphRAG

Graph retrieval reference: [Edge et al., From Local to Global: A Graph RAG Approach to Query-Focused Summarization, 2024](https://arxiv.org/abs/2404.16130).

RAG Failure Modes In Proof Review

Chunk-only retrieval

```
embed chunks retrieve top−k nearest
chunks paste into prompt ask the model
to answer
```

- text fragments replace mathematical objects
- hypotheses are separated from statements
- wrong assumptions pass unnoticed

Object-level retrieval

```
identify proof node retrieve needed
objects filter by assumptions and trust
assemble bounded packet audit evidence
use
```

- context is job-specific
- sources are recorded with the retrieved object
- verifier input includes the declared boundary

Directed Acyclic Graph

DAG means directed acyclic graph: edges have direction, and no directed cycle is allowed.

Examples outside mathematics: build systems, task dependencies, citation-like dependency graphs, and data pipelines.

DAG For Proof Dependencies

In our proof-review setting, a DAG can represent dependencies among definitions, lemmas, proof steps, sources, gaps, and repairs.

```
T_theorem
  depends_on D_definition
  depends_on L_key_lemma
  depends_on P_proof_step

L_key_lemma
  depends_on D_definition
  supported_by S_source_span

P_proof_step
  rejected_by I_gap
  revised_into P_repair
```

> If the graph says a theorem depends on a lemma that depends back on the theorem, the system has found either circular reasoning or a wrong dependency extraction.

Graph Operations For Review Records

Cycle detection

find circular proof dependencies

Topological order

check definitions and lemmas before the theorem

Dependency closure

list everything a claim relies on

Reverse dependency

find every theorem affected by a rejected lemma

> In proof review, "what does this gap affect?" is a graph query.

Tool	Use
NetworkX / `graphlib`	in-memory DAG algorithms
Mermaid / Graphviz	classroom and report visualization
Kuzu / Neo4j / ArangoDB	graph database storage and multi-hop queries
PostgreSQL recursive CTE	relational baseline for dependency closure

RAG And DAG In One Review System

Question	RAG	DAG
What it does here	retrieve relevant material	represent dependency structure
Core object	source, chunk, theorem, evidence	node and dependency edge
Operation	locating definitions, lemmas, examples	exposing gaps, cycles, stale dependencies
Error case	misleading context	false node marked trusted
Review question	What evidence should be inspected?	What remains to prove?

> The coordinator can retrieve evidence with RAG, then attach the accepted evidence to nodes and edges in a proof DAG.

Proof Blueprint

Polished proof

Compressed exposition for a mathematical reader.

Proof blueprint

Expanded construction plan with dependencies, lemmas, sources, gaps, failed paths, and repair state.

> A blueprint is the artifact a natural-language proof system can review, repair, accumulate, and later hand to formalization work.

Blueprint Schema

Statement

complete theorem statement, hypotheses, notation

Dependencies

definitions, lemmas, external results, source spans

Strategy

route, construction, induction or contradiction structure

DAG

proof obligations and dependency edges

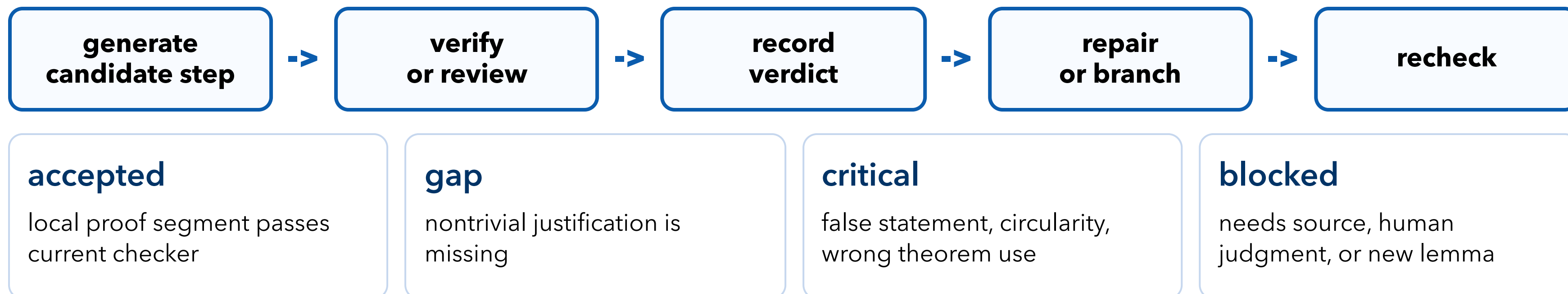
Review

accepted claims, gaps, critical errors, citation checks

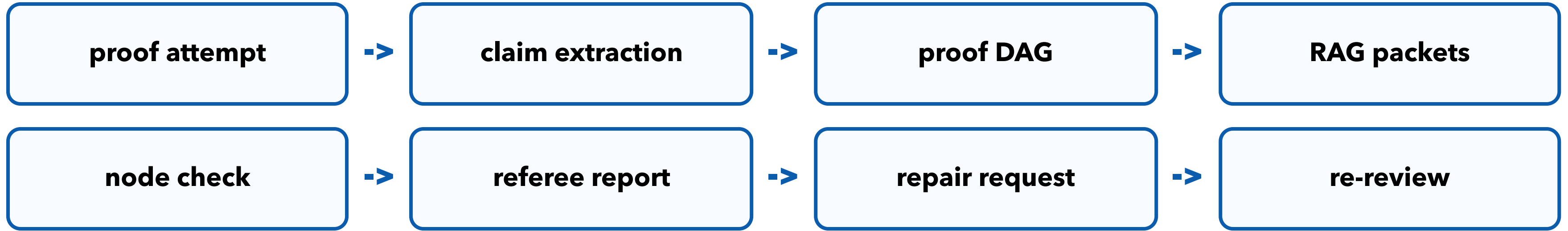
Repair

failed attempts, revised nodes, rechecked proof segments

Generation-Verification Loop



Review-System Loop



The failed attempt remains in the artifact and can be retrieved during later repair.

Rethlas Original: Inspectable System

Rethlas original is a natural-language reasoning system built around two Codex agents.

Generation agent

reads a markdown problem, explores proof strategies, writes a proof blueprint

Verification agent

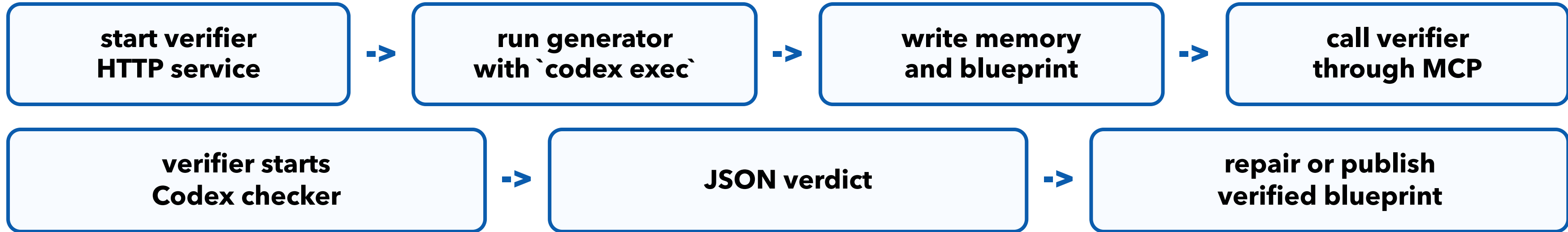
checks the blueprint and returns structured verdict plus repair hints

Proof loop

generation calls verification during repair and stops only after acceptance

Source: [frenzymath/Rethlas README](https://github.com/frenzymath/Rethlas).

Rethlas Original Runtime



Runtime state is stored in files: memory JSONL, blueprint markdown, verification JSON, and logs.

Rethlas Generation Prompt Contract

The generation agent's `AGENTS.md` is the main prompt contract.

Input	markdown problem filepath, problem id, and optional \`reference_dir\`
Boundary	read only inside the working directory
Local context	read the problem markdown first; read problem-specific references before external search
Memory policy	initialize memory, then append every intermediate artifact to named channels
Verification rule	call \`verify_proof_service\` only after a full proof is assembled in \`blueprint.md\`
Final output	write \`results/{problem_id}/blueprint.md\`, then \`blueprint_verified.md\` after verification passes

Source: \`agents/generation/AGENTS.md\` in [frenzymath/Rethlas](https://github.com/frenzymath/Rethlas).

Rethlas Generation Memory Channels

The generation prompt requires append-only memory files under `memory/{problem_id}/`.

Reasoning records

`immediate_conclusions`, `subgoals`, `proof_steps`, `big_decisions`

Examples

`toy_examples`, `counterexamples`

Failure records

`failed_paths`, `verification_reports`

Control records

`branch_states`, `events`

Source: `CHANNEL_FILES` and `memory_append` in `agents/generation/mcp/server.py`.

Rethlas Original MCP Surface

```
search_arxiv_theorems(query, num_results)
verify_proof_service(statement, proof)
memory_init(problem_id, meta)
memory_append(problem_id, channel, record)
memory_search(problem_id, query, channels)
branch_update(problem_id, branch_id, state)
```

> The generator's tools define what kinds of mathematical memory and verification are available.

Source: `agents/generation/mcp/server.py` in [frenzymath/Rethlas](https://github.com/frenzymath/Rethlas).

Rethlas MCP Tool Schemas

`search_arxiv_theorems`	input: `query`, `num_results`; output: theorem texts, arXiv id, theorem id, endpoint
`verify_proof_service`	input: `statement`, `proof`; HTTP POST to `/verify`; output: report, verdict, repair hints
`memory_append`	input: `problem_id`, `channel`, JSON object `record`; output: file path and timestamped entry
`memory_search`	input: `problem_id`, query, optional channels; output: BM25-ranked JSONL records
`branch_update`	input: `problem_id`, `branch_id`, JSON object state; stored through `memory_append`

Source: `build_mcp_app()` in `agents/generation/mcp/server.py`.

Rethlas Original Context Model

Workspace boundary	The generation agent is instructed not to read outside its working directory.
Problem id	Memory, logs, results, and branch states are keyed by the data-relative problem id.
Reference dir	Problem-specific references are read before external search.
Memory channels	Conclusions, examples, counterexamples, subgoals, proof steps, failed paths, verification reports, branch states.

Rethlas Verification API Prompt

The generator calls `verify_proof_service(statement, proof)`. The HTTP service allocates a run id and sends Codex this prompt:

```
Run_id: {run_id}. Statement: {statement}. Proof:
{proof}

Use AGENTS.md to verify the above proof for the statement.
```

Injected fields	run id, theorem statement, full proof markdown
Verifier context	working directory plus `agents/verification/AGENTS.md`
Command	`codex exec -C {WORK_DIR} -m {MODEL} ... {prompt}`

Source: `build_prompt()` and `build_codex_command()` in `agents/verification/api/server.py`.

Rethlas Original Verifier

Verifier contract

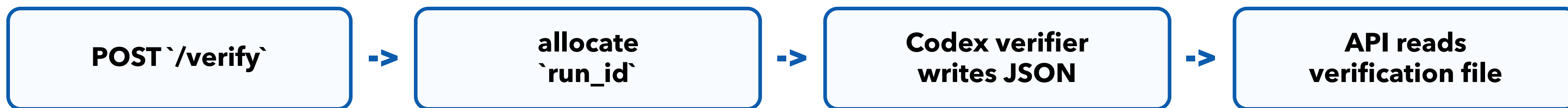
- read proof in textual order
- check logic, theorem use, missing assumptions
- record every critical error and gap
- accept iff no errors and no gaps remain

Output shape

```
{ "verification_report": {
"critical_errors": [], "gaps": [] },
"verdict": "correct", "repair_hints": ""
}
```

Source: `agents/verification/AGENTS.md` in [frenzymath/Rethlas](https://github.com/frenzymath/Rethlas).

Rethlas Verification Output Path



Request body	`{"statement": "...", "proof": "..."}`
Run directory	`results/{run_id}/`
Log file	`results/{run_id}/log.md` records command and Codex output
Expected JSON	`results/{run_id}/verification.json`

Source: `verify()`, `run_codex_verification()`, and `_verification_path()` in `agents/verification/api/server.py`.

Rethlas Original: Implemented Pieces

Loop

problem -> blueprint -> verifier -> repair -> checked blueprint

MCP tools

memory, theorem search, and verification are callable

Persistent memory

failed paths and branch states survive beyond one transcript

Verifier split

generation and checking are separate roles

Blueprint artifact

output is a structured mathematical construction

Repository scale

the relevant agents and MCP tools are visible in the repo

Rethlas Original: Missing Pieces For Review

Whole-proof verifier

checks full blueprint rather than one durable proof node at a time

DAG store

dependency graph is not stored as a separate durable object

Graph database

no Kuzu/Neo4j-style projection for dependency queries

Admission layer

checked facts do not pass through a separate schema/hash/provenance gate

Generator orchestration

the generator owns much of the repair loop

Review artifacts

reports, issue nodes, and repair records are not the main output format

AI Co-Mathematician: Design Principles

Reference: Zheng et al., AI Co-Mathematician: Accelerating Mathematicians with Agentic AI, 2026.

Refine questions

the user and coordinator clarify definitions, goals, and conjectures before dispatch

Native artifacts

the system centers work around a living working paper, not only chat

Asynchronous work

multiple specialized agents run in parallel while the user can still steer

Progressive disclosure

high-level project state is separated from low-level agent logs

Uncertainty records

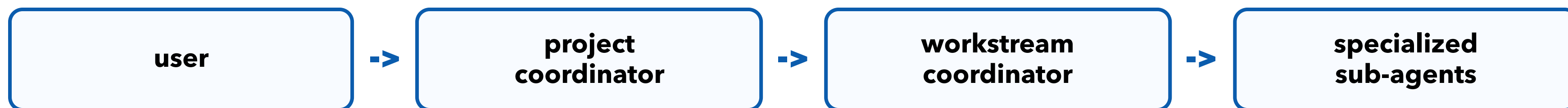
version history, reviews, simulations, citation checks, margin notes

Failed explorations

dead ends are preserved as project history rather than erased

Source: Section 2, "Core design principles", in [arXiv:2605.06651](https://arxiv.org/abs/2605.06651).

Co-Mathematician Agent Hierarchy



Project coordinator	asks clarifying questions, proposes goals, schedules workstreams
Workstream coordinator	receives an approved goal, runs a sequence of actions, updates a report
Specialized agents	literature review, coding, proof search, simulation, review
User channel	the user can intervene while background work continues

Source: Section 3 and Figures 1-4 in [arXiv:2605.06651](https://arxiv.org/abs/2605.06651).

Co-Mathematician Workstreams

The paper’s walkthrough uses moving-sofa variants to show workstream structure.

Literature review

search for papers, then query exact statements and proofs

Computational framework

design a framework, create coding agents, add tests and demonstrations

Execute search

run branch-and-bound work after the framework is available

Dependency between workstreams

later workstreams can import files produced by earlier workstreams

Incremental reports

each workstream updates a user-facing report during execution

Unfinished status

failed or blocked workstreams remain visible rather than disappearing

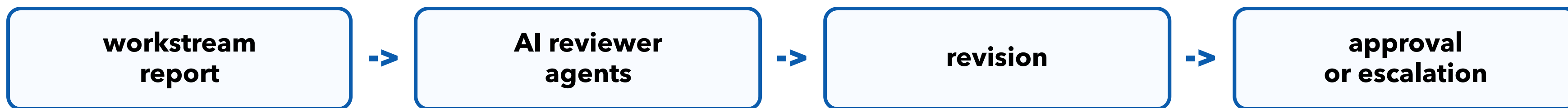
Source: Section 3.2, "Branching the Research", in [arXiv:2605.06651](https://arxiv.org/abs/2605.06651).

Co-Mathematician Working Paper

Main artifact	compiled and reviewed LaTeX write-up for each workstream
Exposition	the write-up includes process, not only final result
Margin annotations	notes link claims to workspace provenance, user suggestions, or sources
Internal linking	references point to internal documents produced by agents
Shared filesystem	agents write work to files that other workstreams can read

Source: Sections 2 and 3.4 in [arXiv:2605.06651](https://arxiv.org/abs/2605.06651).

Co-Mathematician Review Loop



Reviewer checks	content, style, references, code outputs, logical correctness
Persistence	reviewer agents persist between review rounds
Programmatic constraint	a coding sub-agent cannot mark code finished until tests and reviewer acceptance pass
Escalation	if review cannot pass, the workstream can be marked unfinished and escalated

Source: Sections 3.3-3.4 in [arXiv:2605.06651](https://arxiv.org/abs/2605.06651).

Co-Mathematician Case Studies

Kourovka problem

two workstreams attempted proof and disproof; a flawed proof plus reviewer critique helped the user fill a gap

Agent actions

coding search, literature search, conjecture sharpening, reviewer discussion, pivot to a proof method

Stirling coefficients

uploaded primer, separate workstreams for conjectures, computation plus proof attempts

User interaction

users read marginal comments, ask about highlighted insights, and revise final documents

Source: Section 5 in [arXiv:2605.06651](https://arxiv.org/abs/2605.06651).

Co-Mathematician Benchmark Mode

Evaluation	Setup described in the paper	Reported result
Internal research math	100 unleaked research-level problems with code-checkable answers	outperforms single Gemini 3.1 Pro and Gemini Deep Think calls
FrontierMath Tier 4	Epoch AI blind evaluation through the UI	23 / 48 excluding two public samples, reported as 48%
Harness difference	own tools, no model-call or token limit stated	higher inference cost than standard harness evaluations is noted

Source: Section 6 in [arXiv:2605.06651](https://arxiv.org/abs/2605.06651).

Co-Mathematician Limitations

Reviewer-pleasing bias	the revision loop can converge to an argument reviewers no longer catch, while errors remain
Non-termination	review and revision can cycle without consensus
Autonomy and control	long autonomous runs require ceding some control; model judgment remains limited
Typeset document risk	clean LaTeX can look more rigorous than the underlying argument
Literature signal	AI-written mathematical documents can increase review burden for the community

Source: Section 7, "Challenges and Limitations", in [arXiv:2605.06651](https://arxiv.org/abs/2605.06651).

Rethlas And Co-Mathematician: Source Objects

Object	Material in hand	Use in this lecture
Rethlas original	repo, agents, MCP tools, verifier service	inspect prompt contracts and tool calls
AI Co-Mathematician	paper-level system description	inspect workstreams, review loops, working-paper interface
Rethlas-Plus	proposed design sketch	discuss coordinator, graph state, and review artifacts

FrontierMath Is A Benchmark

Reference: Glazer et al., FrontierMath: A Benchmark for Evaluating Advanced Mathematical Reasoning in AI, 2024.

FrontierMath asks

- Can the system solve difficult expert-crafted problems?
- Can the answer be evaluated robustly?
- How far are current systems from advanced mathematical reasoning?

Review systems ask

- Is this proof correct?
- Which step is unsupported?
- Which source justifies a claim?
- Which repair is next in the queue?

Benchmark context: [Epoch AI FrontierMath](https://epoch.ai/benchmarks/frontiermath/).

What FrontierMath Does Not Measure

Proof exposition

whether the reasoning is communicable as a proof

Source grounding

whether each external theorem is correctly cited and applicable

Dependency tracking

whether the proof state exposes all obligations

Repair reliability

whether gaps become targeted fixes rather than rewrites

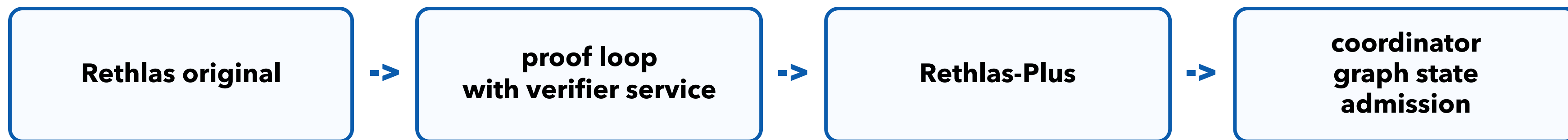
Review quality

whether a proof checker finds the right mathematical defect

Formal correctness

whether a proof artifact is machine checked

Rethlas-Plus: Research Design Sketch



Rethlas-Plus is the proposed review architecture: make context, dependencies, verifier calls, and durable state explicit.

Local design reference: [Rethlas-plus System Design](/Users/hoxide/mycodes/Rethlas/docs/RETHLAS_PLUS_SYSTEM_DESIGN.md).

Rethlas-Plus Components

Coordinator

job queue, budgets, context packets, logs, scheduling

MCP tool layer

`verify_node`, `retrieve_context`, `graph_query`, `request_repair`

Generator

proposes proof steps, lemmas, repairs, blueprint text

Verifier

checks one node or bounded segment under explicit dependencies

Referee

produces proof review reports and issue records

Librarian

admits accepted objects after schema, hash, and provenance checks

Coordinator Context Responsibilities

Before dispatch	choose target node, retrieve evidence, freeze dependencies, compute context hash
During worker call	give only the job packet, tool policy, output schema, and budget
After result	validate schema, record event, update graph, route repair or admission
Coordinator records	truth status, dependency changes, and promotion of repaired nodes

Node-Level Verifier

Whole-proof verifier

- large context
- hard to localize failure
- repair may rewrite too much
- dependency changes are hard to audit

Node-level verifier

- one statement
- one proof segment
- explicit dependencies
- accepted / gap / critical verdict

> The verifier report can be attached to a single node and its declared dependencies.

Librarian Admission

Generated text, retrieved evidence, and referee recommendations are not theorem-library truth until admitted.

Rethlas-Plus Review Pipeline

Small Review System Schema

Input

proof draft, theorem/proof source, or mathematical learning target

State

claims, dependencies, sources, issues, repairs

Workers

reviewer/referee, retriever, optional verifier, optional generator

Review

exact gaps, critical errors, unsupported theorem uses

Repair

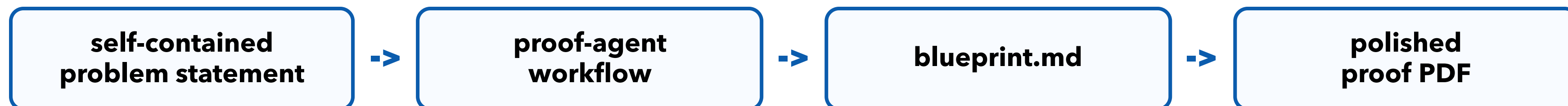
one repaired step and a recheck record

Artifact

proof blueprint, review report, repair log

Assignment 1: Proof-Agent Proof

Use a coding assistant to obtain, inspect, and launch a proof/proving agent.



Rethlas is one good option. Students may use another suitable proof agent if they know how to run it.

Assignment 1 Submission

Problem

precise enough for a complete proof, with definitions and hypotheses

Assistant

Codex, Claude, opencode, openclaw, or another agentic coding assistant

Agent

Rethlas or another suitable proof/proving agent

blueprint.md

proof blueprint or proof-search artifact, exported or renamed if needed

PDF

clear theorem statement, definitions, assumptions, and complete proof

Email

send both files to hoxide@gmail.com

Subject: `[AI4Math XMUM GE] Assignment 1 - <Student Name> - <Student ID>`

Task file: [Proof-Agent Natural-Language Proof](/Users/hoxide/mydoc/ai4mathcourse/xmum-ge/assignment01-natural-language-proof-learning.md).