Agentic Math

Michael Shomsky 4/2025

Brief System Overview

- Agentic System to solve math problems
  - Use the power of LLM
  - Numeric, Symbolic, Word
- Calculations can utilize Math Tools
  - A subset of all math tools are implemented as an Agent Tool
- Bridge the gap and communicate to Rust
  - One tool is written in rust and integrated into the python api
- Generate/Maintain virtual tools for faster computation
  - FAISS for pattern matching and local vector store

5 Agents

3 independent agents + 2 dependant agents

- **Solver Agent** - Uses math tools and LLM to solve
- **CAS Agent** - Uses LLM to frame the problem to be solved by a Computer Algebra System
- **Verification Agent** - Uses LLM to solve
- **Math Planner Agent**
  - plans how to compute
- **Plan Execution Agent**
  - computes each step

Tools

- **MathToolbox (Python)** sum, product, divide, subtract, power, sqrt, modulo, round_number
- **MathToolbox (Rust)** calculate_average
- **Virtual Tool Manager** - Create/Manage new functions

PLAN -> EXECUTE PATTERN

Plan: Create a plan on how to solve the problem and suggest tools to do so

Execute: Take the plan and use the available math toolkit resources to execute the plan

MVP - Majority Rules

- If any **two or three agents agree** on a solution, that solution is chosen as the majority solution.
- When there's no majority agreement:

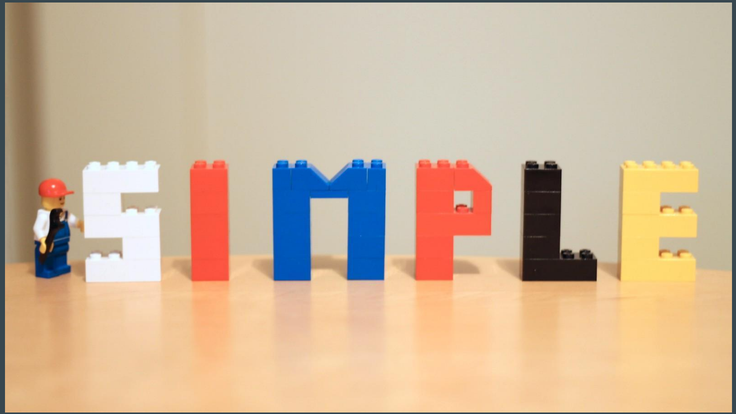
  - If the **Solver and CAS agents agree** (but validation disagrees), we use their solution.

  - If the **CAS and Verification agents agree** (but solver disagrees), we use their solution.

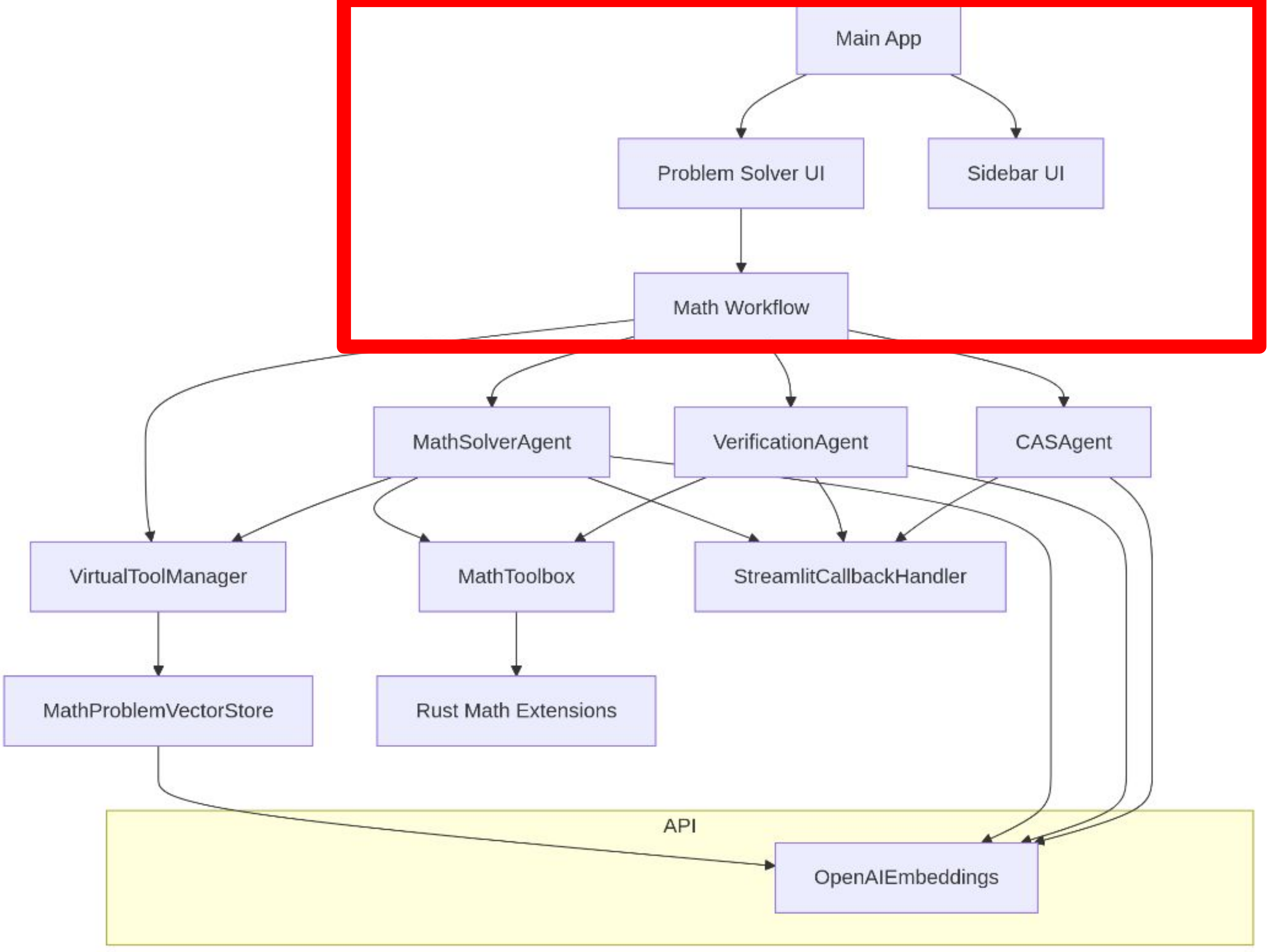
  - If the **Planner Execution Agent and Verification agents agree** (but solver disagrees), we use their solution.

  - *Note:* ***Solver + Verification agent*** *is the the garden path to acceptance*

- Only if we don't have any of the above cases, we fall back to the original behavior of using the solver solution with verification.



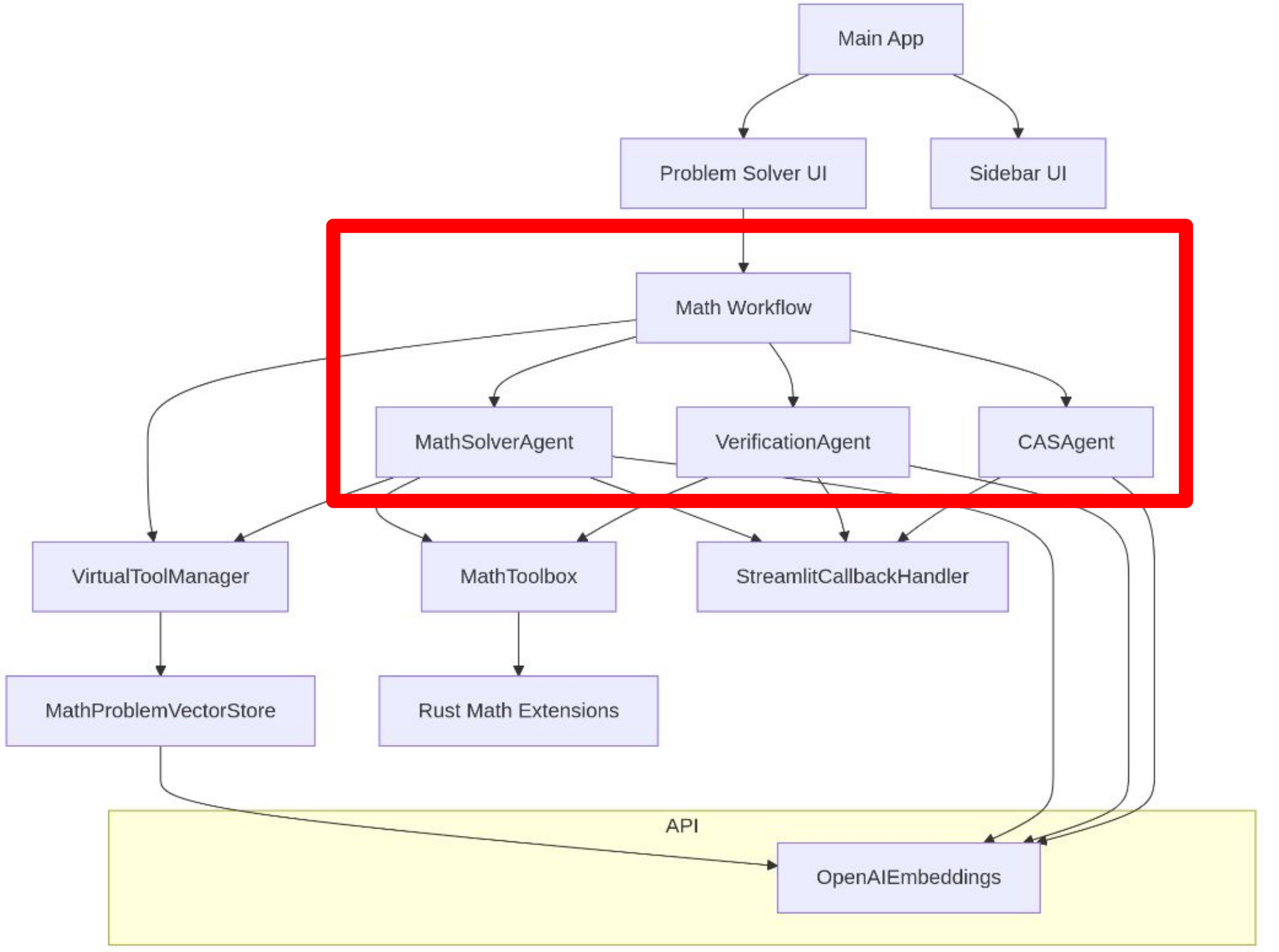
UI Drives a Math Workflow



Math Workflow Orchestrates Agents

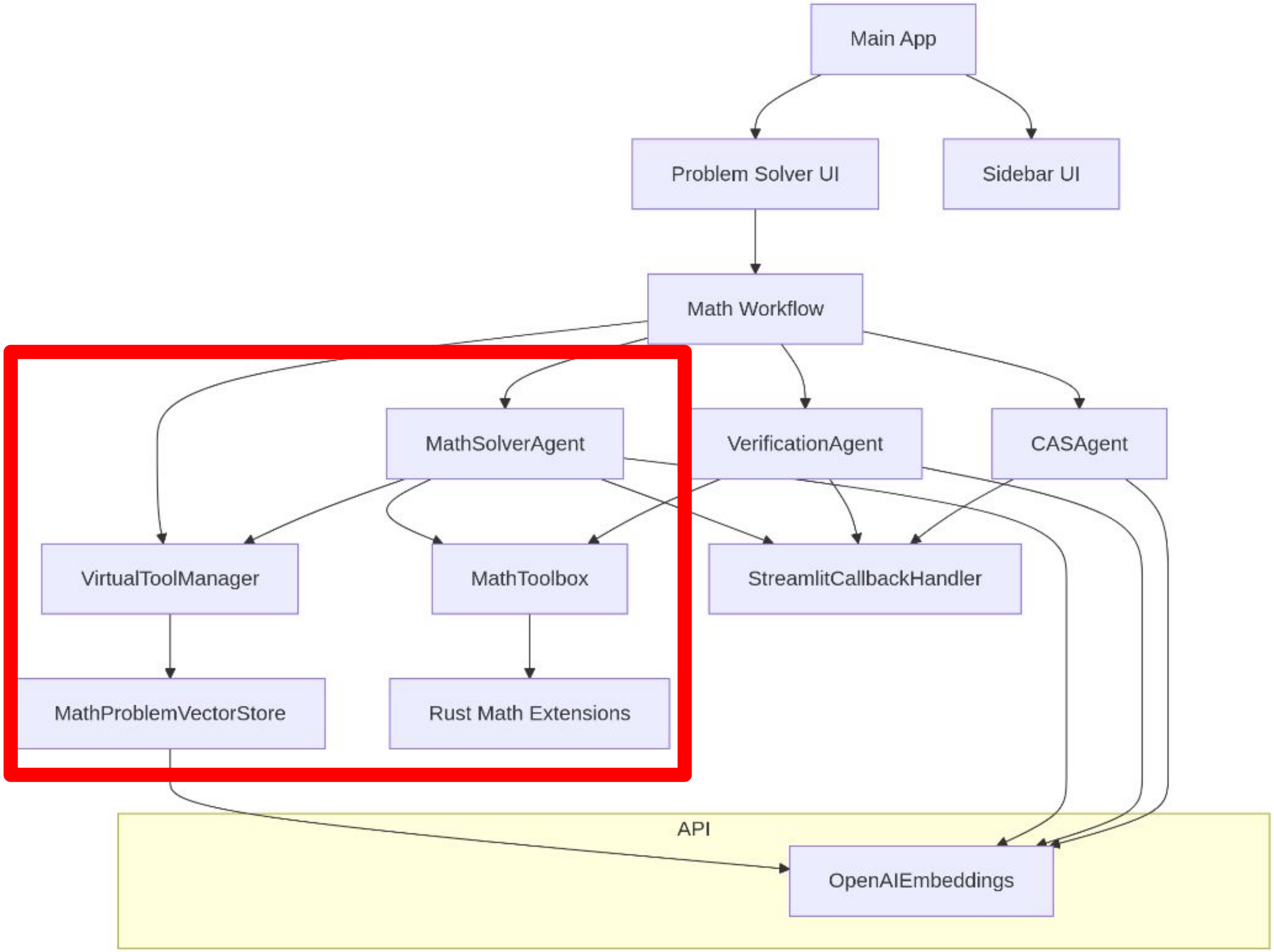
Or

Uses a Virtual Tool



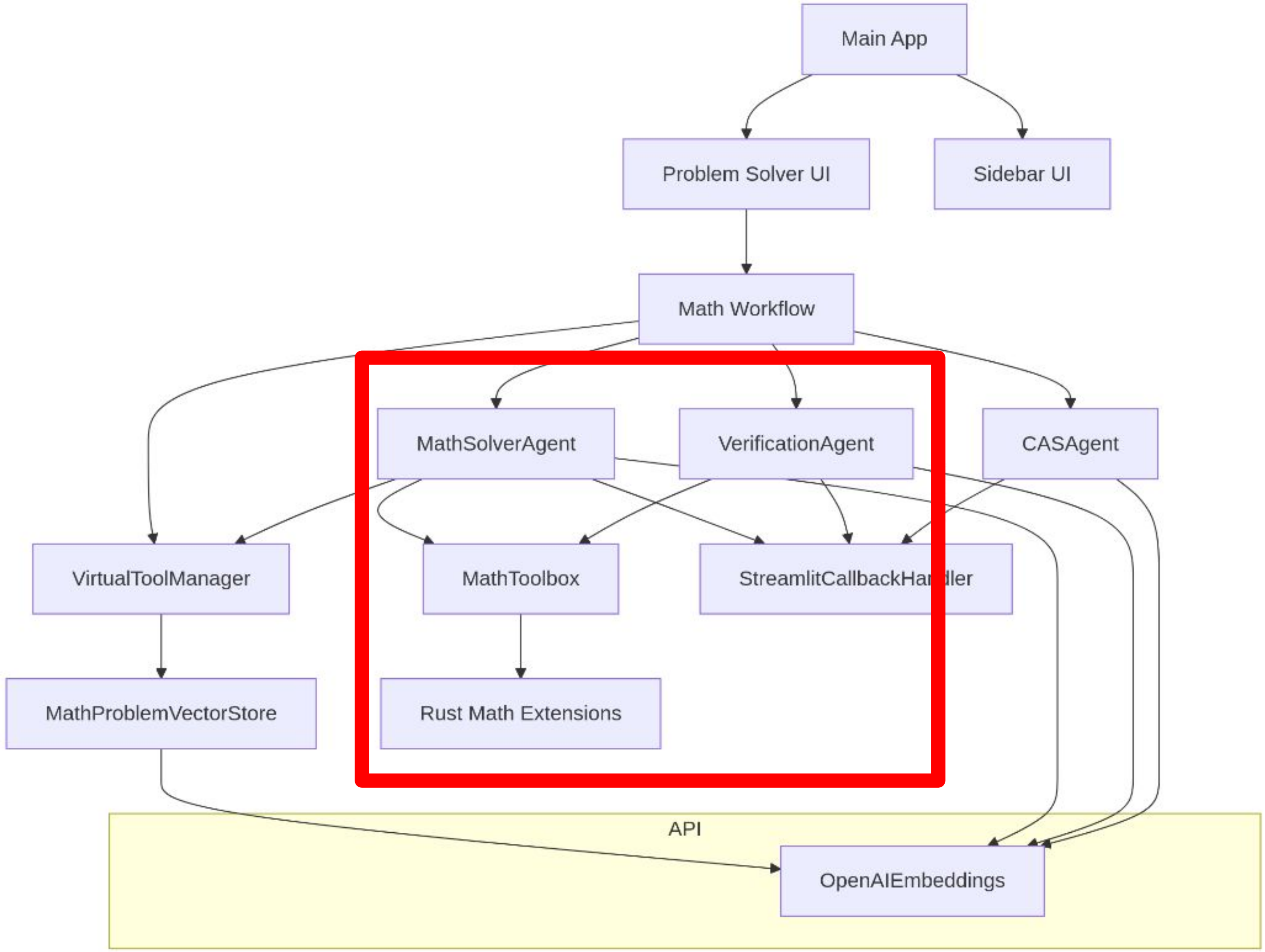
MathSolver Agents

uses Virtual Tool Manager to use an existing tool or record successful completion



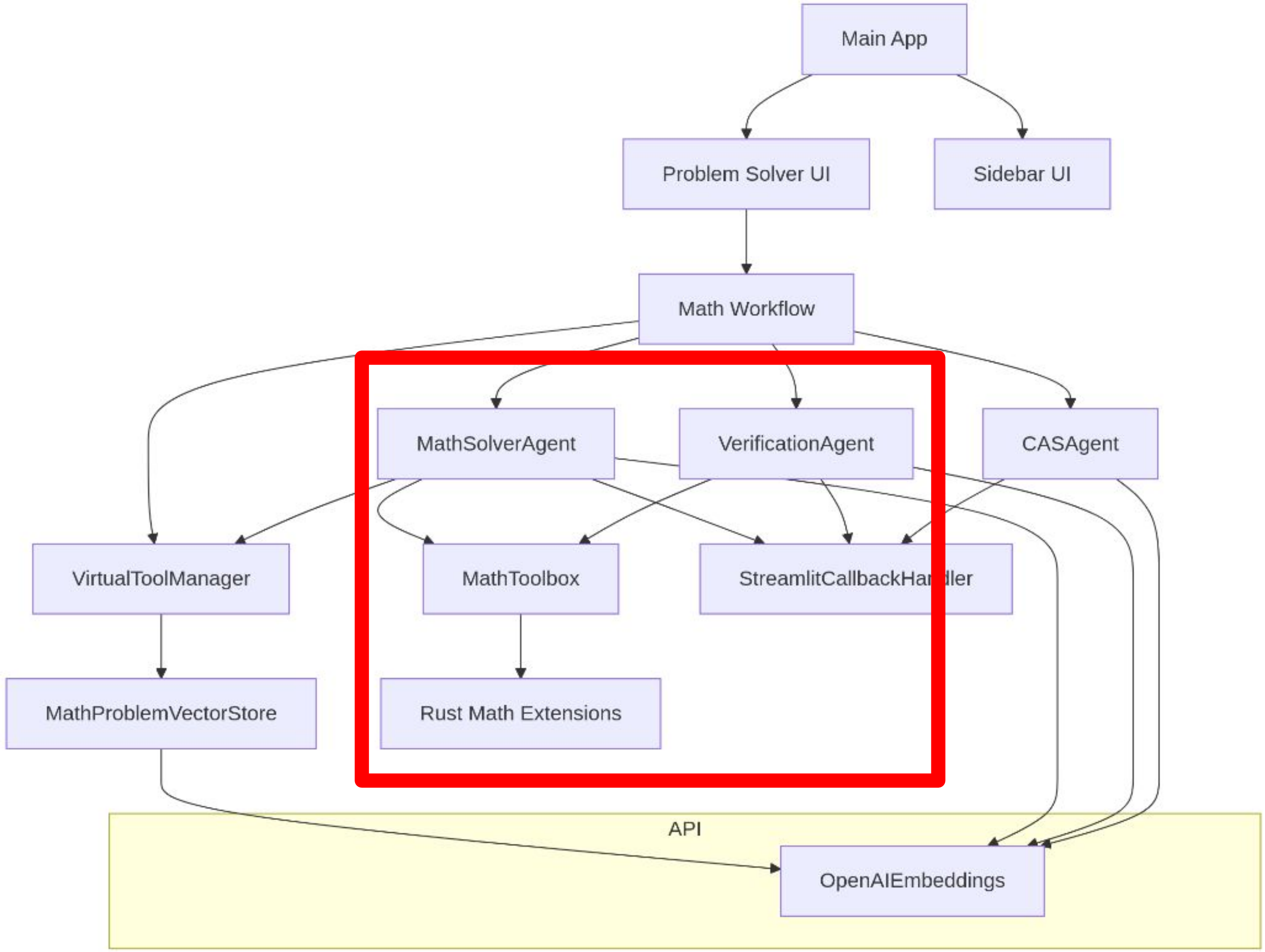
MathSolverAgent

Rely on MathToolbox For Math Functions



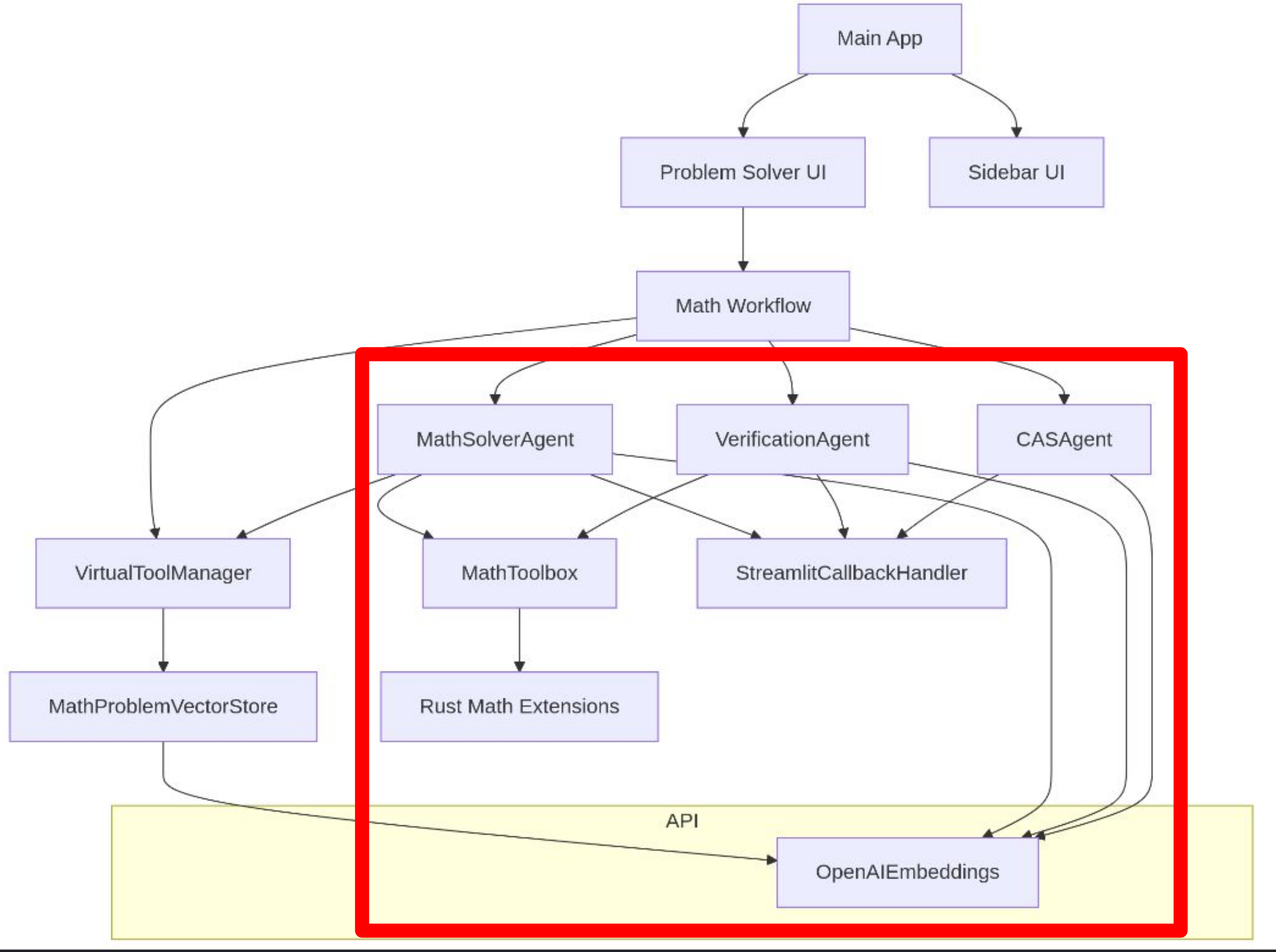
VerificationAgent

-LLM only for calculation
- (initially used math toolbox)



3 Agents

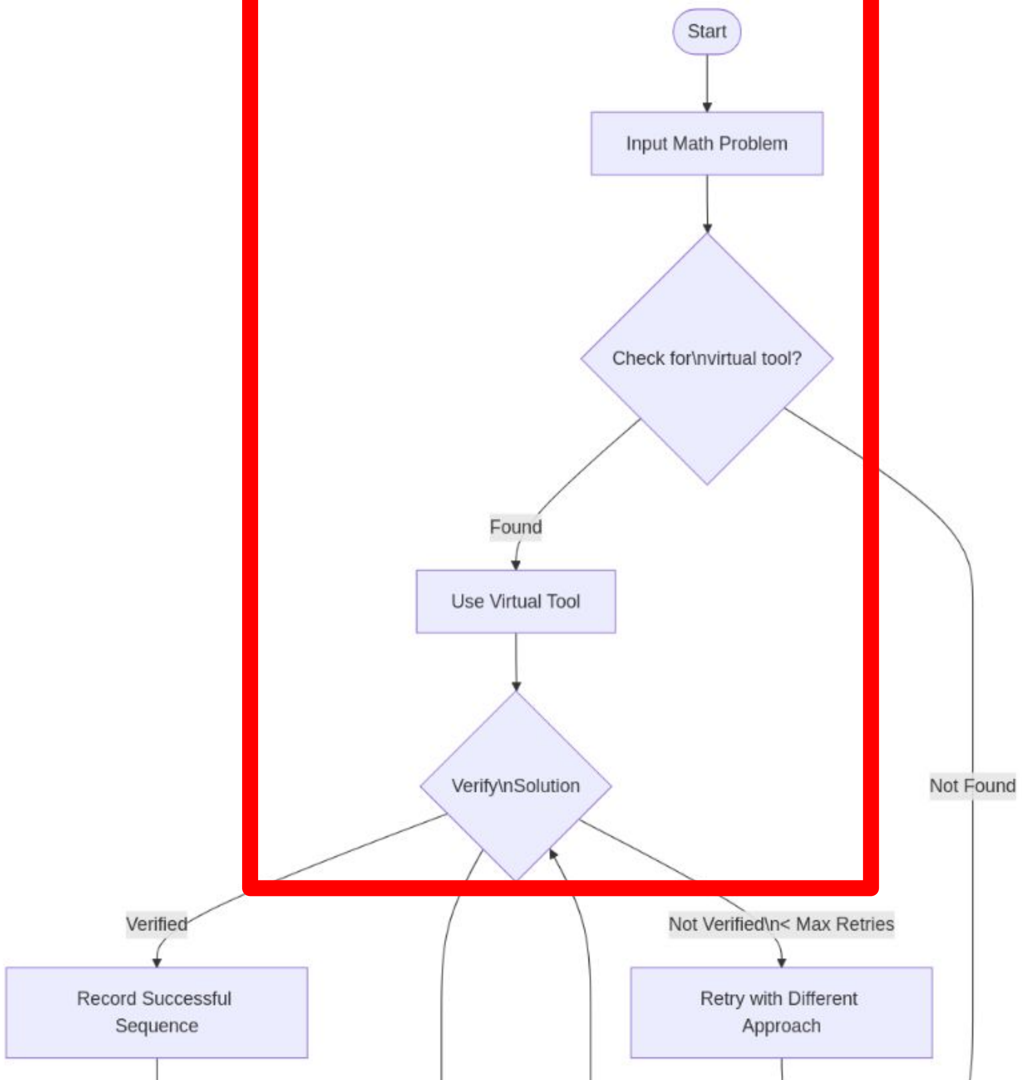
Rely on LLM integration to work with the math question



- Start by checking if there’s a virtual tool.

- Use it if there is one

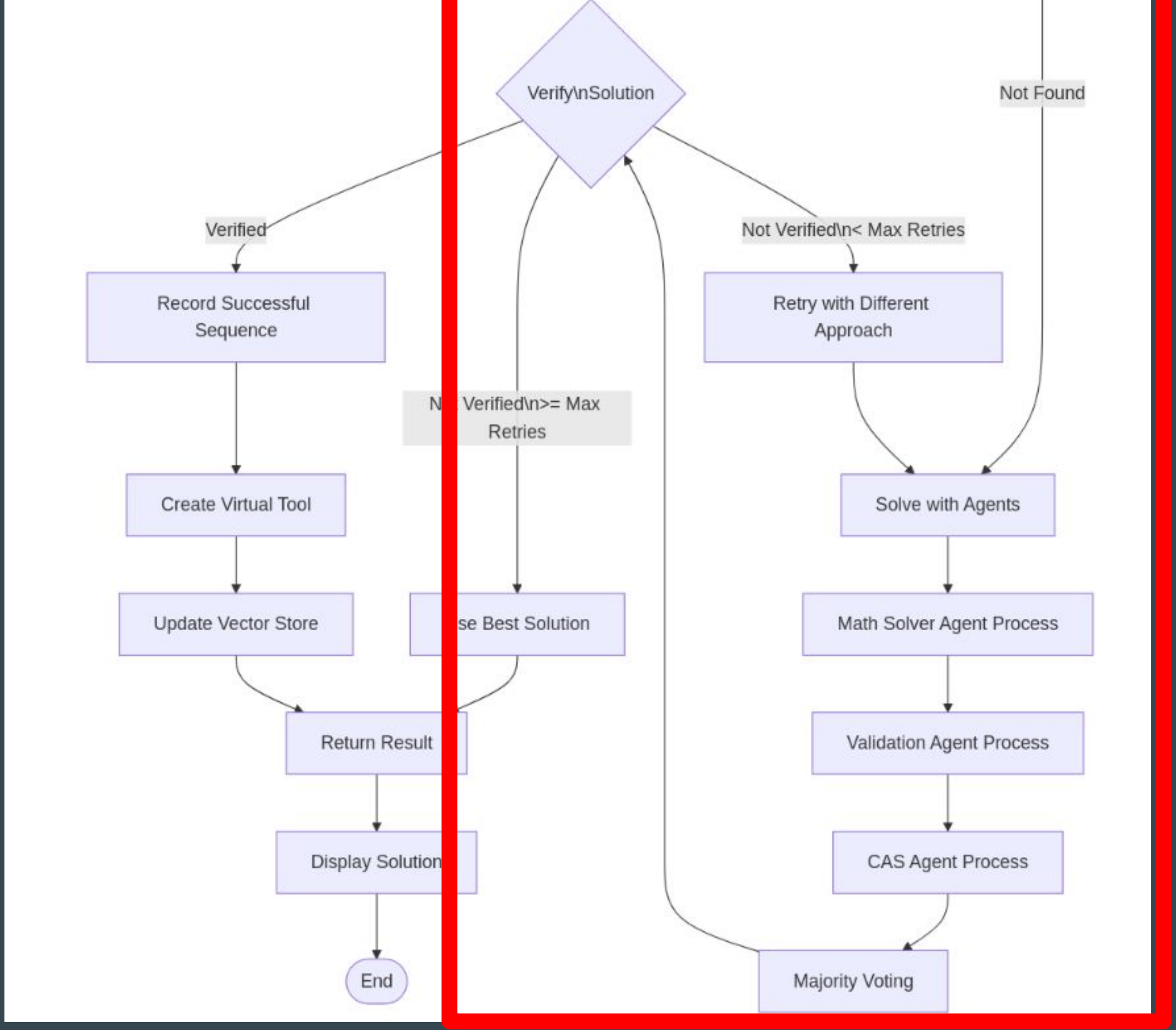
- Verify the Solution



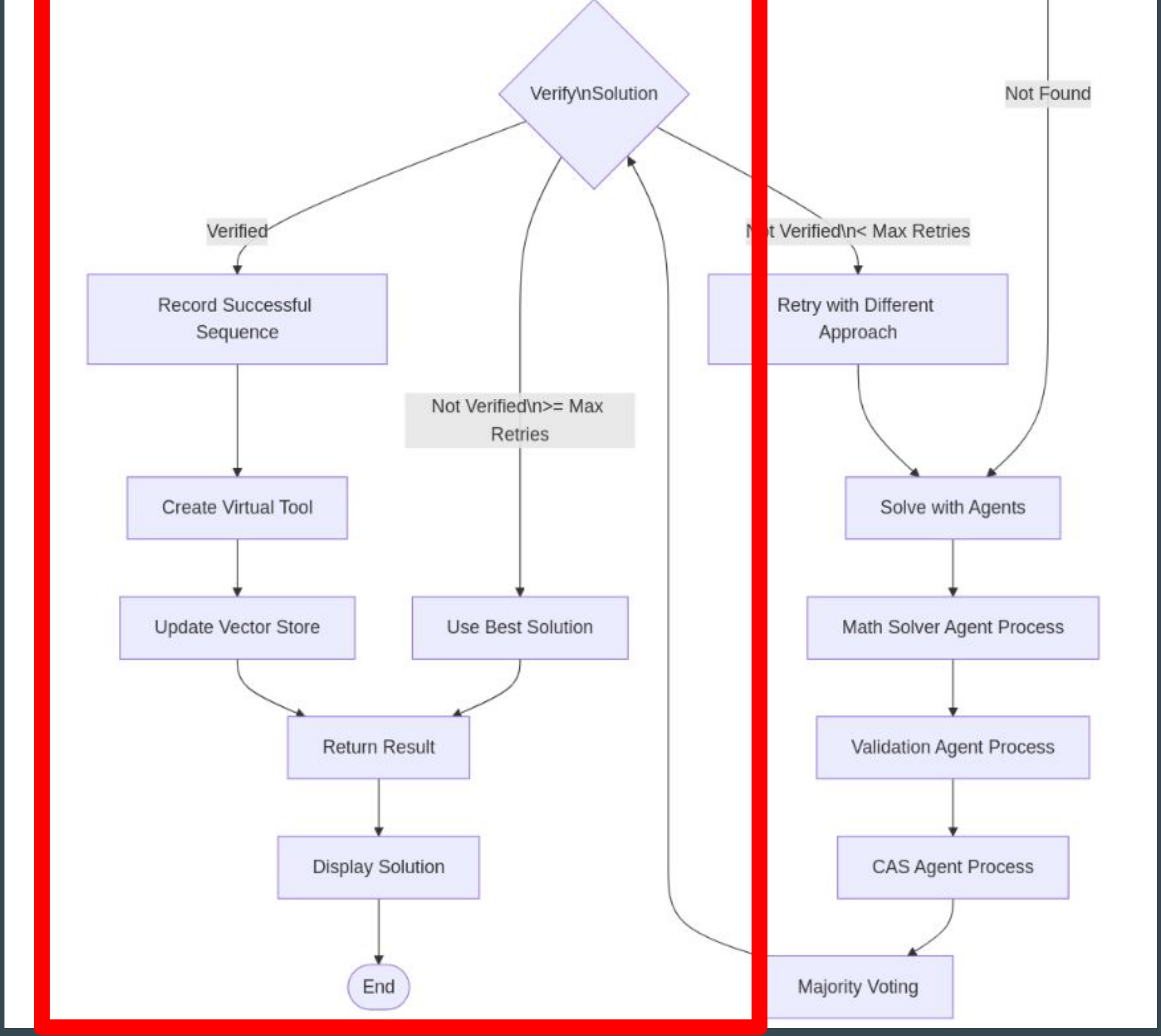
- If verification fails

- Solve with Agents

- Use Majority Voting to verify answer



- Record successful Sequence
- Create a new virtual tool
- Update Vector Store
- Return/Display result



Demo

Demo the using the agents as a service locally

Star the API:

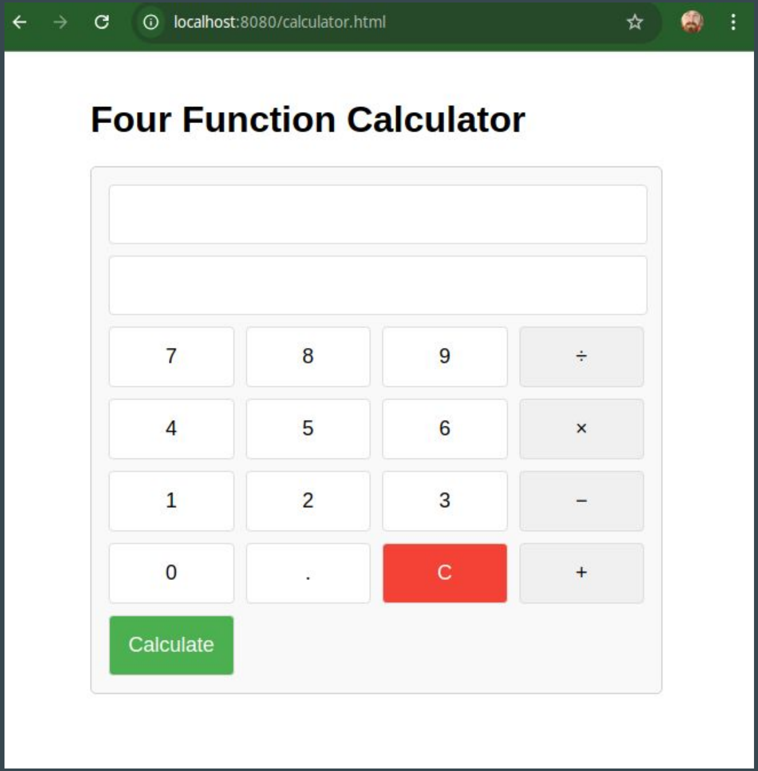
- Note: this is on the branch “add-plan-exe”
- Run the api (./api/run.sh)
- Run or load the page. (./api/serve_calculator.sh)

Key details:

- Agents potentially as endpoints
- Shared instance of VirtualToolManager
- math_workflow on /api/tribunal for results

Demonstrates:

- Agent / Workflow Reusability
- Removes some of the streamlit clutter



Thanks

- Next Steps
  - Add Kubernetes
  - Connect the virtual tool manager to postgres for distributed functions
  - Port more to Rust begin auto-testing to choose fastest functions
    - Create a run-harness with history. If the rust version runs faster than python, use the rust version.
    - Log/Tally resulting decisions for review
-

Appendix

Additional slides

Requirements

https://spindle.notion.site/Coding-Project-Option-A-for-15757291437d804b87edf816a4212cdb

This mission represents a stripped-down but realistic "toy version" of the kind of multi-agent system Spindle AI is engineering (including some actual challenges we've already faced):

1. **The Setup:** First, create ≥5 distinct, simple, deterministic tools that an LLM-based agent could call to help solve user-provided math problems (*e.g.* `SUM`, `DELTA`, `PRODUCT`, `QUOTIENT`, `MODULO`, `POWER`, `ABS`, `LOG`, `TRIG`, `SQRT`, `AVG`, `MODE`, `ROUND`, `UNION`, `INTERSECT`, `DIFFERENTIATE`, `INTEGRATE`, `FACTORIZE`, … — *the specific tools are entirely up to you*).
    1. **Modify 1-2 of the most basic tools to *intentionally but silently throw errors (and/or silently give incorrect answers) 30%-50% of the time the tool is called*.** You *may* also want to include a basic `GET_USER_INPUT` tool for requesting input/clarification from a human user. (You can organize all tools in some form of "toolbox" if you want, but we'd prefer you do **not** hardcode a string listing all the tools, their docs, and their usage examples in a *single* prompt file or prompt mega-string anywhere in the project.)
2. **The Architecture:** Prototype a multi-agent system with ***at least* 2 agents** and *at most* 5 agents (for whatever definition of "agent" you believe makes sense in this context), that discovers which tools are available and sequences tool calls to **reliably** solve basic user-provided math problems (or if you prefer, mathy word problems). The agents can *only* ****use the available tools ***(including the unreliable tool[s])***, *i.e.* no LLM-hallucinated arithmetic should be used for user-facing answers (*even* if that arithmetic is correct, as is increasingly the case among frontier models).
    1. You might well choose to include a lightweight planning, reasoning, and/or task decomposition layer in your prototype — but unless you have a compelling justification, all *user-facing* outputs (and most intermediate outputs) should be structured or semistructured, not unstructured.
    2. **Don't hesitate to ask us for an OpenAI API key or Anthropic API key.** Otherwise, we're happy to reimburse these costs after submission *(within reason/at Spindle's discretion)*.
3. **The Twist:** When your prototype identifies a sequence of tool calls that reliably *or fairly reliably* solves a certain class of math problem(s) **based on successful execution(s)**, it should learn to do something like *(e.g.)* **memoize or semantically cache that sequence of tool calls as a single, idempotent new** `VirtualTool` (*i.e.* some learning behavior akin to **"bundling" the tool calls into a *single* new idempotent tool, to which a *single* call can be made, which can be reliably invoked *next time a math problem of the same or similar form is encountered*).
4. **The Finish Line:** Prove programmatically that your prototype works reasonably well (or at least that it could be *completed* to work reasonably well, if short on time).
    1. **Bonus points for using actual evals to show this.**
        1. ***(If you're are an "evals-focused" candidate, consider reframing/approaching the entire task through the lens of an evals system instead, i.e. evals-driven development. Just tell us to judge your quality vs. emphasis vs. completion accordingly.)***
5. **Bonus Points:**
    1. Create the math toolbox/interfaces in a non-Python language (ideally Rust, Go, or Typescript).
    2. *If* you decide to use a vector database anywhere, consider prototyping your *own* vector DB or VDB-like utility. (Not if this takes up all your time, though. It's not the most important part.)

- **If you don't have enough time for a project like this, or have alternate ideas, please let us know so we can find a path forward that we all feel good about!** Either way, we really look forward to seeing you through these next steps.

Agent Framework (LangChain)

- **LangChain-Based**: Structured, consistent agent implementation
- **Model Selection**: gpt-4o-mini with temperature=0 for deterministic results
- **Tool Integration**: Streamlined interface between LLM and toolbox
- **Memory Management**: ConversationBufferMemory for stateful interactions when using tools
- **User Input Support**: Async interruption for gathering required information

MathToolbox

- **Core Operations**: sum, product, divide, subtract, power, sqrt, modulo, round_number
- **Reliability Control**: Tools can be set reliable/unreliable for testing
- **Error Simulation**: sum/product deliberately fail ~30-40% in unreliable mode
- **Complex Number Support**: Properly handles sqrt of negative numbers
- **Statistics Tracking**: Monitors usage and error rates per operation
- **Rust Integration**: Optimized avg function implemented in Rust
- **Testing**: Comprehensive test suite ensures correct math behavior

Rust Interoperability

- **High-Performance Extension**: Python calls native Rust function for avg operation
- **Fallback Mechanism**: Uses Python implementation if Rust unavailable
- **Dynamic Detection**: Auto-detects Rust extensions at runtime
- **Error Handling**: Graceful degradation if Rust function fails
- **Performance Benefits**: Faster calculations for repeated operations
- **Zero-Copy Interface**: Optimized data exchange between languages
- **Future-Ready**: Foundation for moving more compute-intensive operations to Rust

VirtualToolManager

- Creates reusable tools from successful math problem solutions
- Uses vector similarity to match new problems with existing tools
- Optimizes execution sequences & caches results for faster solutions
- Manages reliability through failure tracking & tool removal
- Provides pattern recognition for common math expressions

Walkthrough

1. **Problem Arrives**: Hash structure, normalize expression
2. **Tool Matching**: Seek similar structure via vector store lookup
3. **Execution**: Maps variables from new problem to original pattern
4. **Optimization**: Streamlines tool sequences (ex: sqrt of negative numbers)
5. **Tool Lifecycle**: Tracks performance, removes unreliable tools
6. **Vector Storage**: Employs embeddings for problem similarity
7. **Pattern Recognition**: Special handling for common forms (a+b)/(c+d)

Creating a New Math Agent: ex: MathTextSolutionAgent

Component	Explanation
Agent	math_text_solution_agent.py - Should be stored in /agents/ directory alongside the other agent files like cas_agent.py and solver_agent.py.
Adding to Workflow	In math_workflow.py, you would need to add the agent to the function parameters: def math_workflow(..., math_text_solution_agent, ...) [STREAMLIT SOLUTION ONLY] Because of the UI, you must initialize it in the initialize_session_state() function in main_app.py: if "math_text_solution_agent" not in st.session_state: st.session_state.math_text_solution_agent = MathTextSolutionAgent()
Adding to Tribunal In Workflow	To include the agent's results in the tribunal voting system, modify the voting logic in math_workflow.py by: 1. Adding the solution to the normalized_solutions dictionary: normalized_solutions["text_solution"] = _normalize_solution(text_solution) 2. Adding it to the "agent_solutions" in the result dictionary:"agent_solutions": { ... "text_solution": text_solution, ... }

Failsafe Mechanisms

When a Tool in the Solver Agent Fails

- **Error Detection and Reporting**
  - Tools track errors in `tool_stats` dictionary: `self.tool_stats["sum"]["errors"] += 1`
  - All tools return formatted error messages: `return f"Error occurred: {str(e)}"`
  - Example: In `math_toolbox.py`, division function explicitly checks for division by zero

When Solver Agent and Verification Agent Disagree

- **Retry Mechanism**
  - System attempts solution up to `MAX_VERIFICATION_RETRIES` (set to 5)
  - After disagreement, workflow tries alternative approaches

When Solver and CAS Agree but Verification Disagrees

- **Tribunal Voting System**
  - Implements a "majority rules" approach to resolve discrepancies:
  - CAS agent's symbolic verification becomes a third opinion to resolve deadlocks
  - If solver and CAS agree, their solution typically becomes the majority winner

Virtual Tool Deprecation - When Virtual Tools Fail

1. **Failure Counter System**
   - Each virtual tool has an associated failure counter tracked in `tool_failure_counts` dictionary
   - Failures are recorded via `record_tool_failure(problem_hash)` method
   - Maximum allowed failures controlled by `max_failures` parameter (default: 3)
2. **Verification and Removal Process**
   - When a virtual function fails verification or raises an exception during execution
   - The failure counter is incremented: `self.tool_failure_counts[problem_hash] += 1`
   - When threshold is reached: `if self.tool_failure_counts[problem_hash] >= self.max_failures:`

Tool is deleted from multiple locations:

```
del self.virtual_tools[problem_hash]          # Remove

functionself.vector_store.remove_problem(problem_hash)  # Remove from vector store

del self.successful_sequences[problem_hash]  # Prevent recreation
```

3. **Implementation in Workflow**
   - Used in `math_workflow.py` to manage unreliable virtual tools
   - If a virtual tool fails verification, the system falls back to standard solver
   - Logs notification: "Virtual tool has been removed due to reaching max failures"

This mechanism ensures unreliable virtual functions are automatically purged, maintaining solution quality while allowing the system to learn and create new, more reliable tools over time.