Lab: CS 194/294-280: Advanced LLM Agents, Spring 2025

Deliverable 3: Coding Agent

Implementation Details & Starter Code Description

Earlier, you explored the capabilities of Lean 4 by formulating algorithmic problems along with their specifications. In this deliverable, you will write an agentic workflow for Lean 4 that implements solutions to programming tasks and proves that those implementations satisfy the given specifications. To support multi-agent conversations, we allow extended thinking for up to 60 seconds per question. Additionally, we strongly encourage incorporating retrieval-augmented generation (RAG) techniques into your workflow.

Installation

This lab uses system commands to execute Lean 4 scripts and redirect the output to Python. We strongly recommend using a Unix-based system. To install all required packages and tools, please follow the instructions in the README file provided with the code.

Warning: Installation may take a significant amount of time (20+ minutes). We recommend starting this step early to de-risk potential issues. Also, please verify that the following command from the README completes within a reasonable time:

```
lake lean Lean4CodeGenerator.lean
```

If not, please make a private post on Ed. We will offer some workarounds!

Codebase Overview

Testing & Main Workflow Overview

This project is an automated theorem prover that generates Lean 4 code and proofs based on natural language descriptions. The testing environment follows a structured workflow for processing tasks and

evaluating solutions.

The project uses several key directories:

- `tasks/` : Contains theorem-proving tasks organized in folders named `task_id_*`
- `src/` : Contains implementation code
- `tests/` : Contains the testing framework
- `lean_playground/` : Used for temporary Lean code execution

Each task folder (e.g., `tasks/task_id_0/`) includes several important files:

- `description.txt` : Natural language problem description
- `task.lean` : Lean template file with placeholders for code and proof
- `tests.lean` : Unit tests to verify the implementation
- Additional metadata files

The main workflow begins when the testing framework (`tests/tests.py`) scans the `tasks/` directory for folders matching the `task_id_*` pattern. For each task, it reads the problem description from `description.txt` , parses the Lean code template from `task.lean` , and retrieves the unit tests from `tests.lean` .

The `main_workflow` function (which you need to implement) receives the problem description and Lean code template as strings, and must return a dictionary containing the implementation code and formal proof under the keys `"code"` and `"proof"` , respectively.

For solution evaluation, the system replaces the and placeholders in the template. It first tests only the implementation (setting proof to "sorry") to verify correctness, and then tests both the implementation and proof together. Code execution occurs through the `execute_lean_code` function, which writes the generated Lean code to `lean_playground/TempTest.lean` , runs it using the Lean compiler, and returns output or error messages.

Your primary task is to implement the `main_workflow` function in `src/main.py` . You can use the provided agent classes in `src/agents.py` (GPT-4o and O3-mini models), chain multiple agents together, utilize a RAG database for knowledge retrieval, and implement corrective feedback loops to improve solution quality.

When submitting your solution, include your implementation of `main_workflow` , any `.pkl` or `.npy` files used in your embedding database, and any additional utility functions or classes you've created. Do not modify the test harness or the core functionality of the provided framework.

Starter Code Summary

Workflow & Testing

- `src/main.py` – Entry point for your code. Implements the `main_workflow` function that receives problem descriptions and Lean templates, and returns generated code and proofs.
- `tests/tests.py` – Contains test cases that verify if your generated code compiles and meets requirements. Irrelevant or trivial theorems will receive a score of zero.
- `src/lean_runner.py` – Executes Lean code by writing to temporary files and running the Lean compiler.

Build Tools

- `Makefile` – Defines useful commands. `make test` runs the test suite; `make zip` prepares your submission.

Embedding & RAG

- `src/embedding_db.py` – Processes text files in `documents/` to generate `.npy` embeddings and `.pkl` chunks. Supports similarity search using cosine similarity. Documents are split using the `<EOC>` tag. You may implement your own solution as long as it requires no additional external dependencies and is self-contained within your submission.
- `src/embedding_models.py` – Provides OpenAI and lightweight embedding models for your RAG system.

Task Structure

- `tasks/task_id_*/description.txt` – Natural language description of each theorem-proving problem.
- `tasks/task_id_*/task.lean` – Lean template with placeholders (and) for your solution.
- `tasks/task_id_*/tests.lean` – Unit tests that verify the correctness of your implementation.

Agents & Models

- `src/agents.py` – Defines two LLM agents: GPT-4o for high-quality generation, and GPT-o3-mini for lightweight reasoning. You may use either, or chain them together.

Your Agent's Task

Your LLM agent will be given a programming problem in Lean 4, consisting of three files (see the *Contribute to a Dataset* section for the exact format):

1. `description.txt` – A detailed description of the programming task
2. `signature.json` – The function signature in JSON format
3. `task.lean` – A partially completed Lean file with placeholders

The `task.lean` file includes:

1. A function signature matching that in `signature.json`
2. Placeholders marked with and
3. A specification defining the correctness condition to be proven
4. An example theorem connecting the implementation to the specification

Example `task.lean` file:

```
import Mathlib
import Aesop

def minOfThree (a : Int) (b : Int) (c : Int) : Int :=
  -- << CODE START >>

  -- << CODE END >>

def minOfThree_spec_isMin (a : Int) (b : Int) (c : Int) (result : Int) :
  -- << SPEC START >>
  (result <= a ∧ result <= b ∧ result <= c) ∧ (result = a ∨ result = b ∨
  -- << SPEC END >>

example (a : Int) (b : Int) (c : Int) : minOfThree_spec_isMin a b c (minC
  -- << PROOF START >>
  unfold minOfThree minOfThree_spec_isMin -- helper code

  -- << PROOF END >>
```

Your agent should:

- Generate code to replace
- Generate a proof to replace

Note:

- Your agent should output only the content for these placeholders—not the entire Lean file.
- Your agent must generate a non-trivial proof. Use of `sorry` or similar placeholders will be detected and disqualified.
- The following libraries will be imported and are available for use: `Mathlib`, `Aesop`.

Public Tests

Please look inside the `tasks/` directory for all public tests used to evaluate your implementation. Files prefixed with `problem` contain the specific coding challenges.

Do not hard code a solution. Hard-coded responses in prompts or the database will result in **no credit**. We will audit all submissions for hard coding.

Recommended Implementation

We recommend a three-agent architecture, as follows. While collapsing stages is allowed, we **strongly recommend** maintaining a separation between code generation and correction.

1. Planning Agent

Handles task decomposition and high-level strategy.

- Adjusts plans based on error messages or feedback from Lean execution
- Tracks previously generated code to avoid feedback loops of repeated errors

2. Generation Agent

- Generates Lean 4 code and proofs following the suggested plan
- Integrates with the RAG database to retrieve relevant examples

3. Verification/Feedback Agent

- Executes Lean 4 code to check for syntactic and logical correctness
- May use the execute_lean_code function call
- [Optional] Uses RAG to retrieve documentation or examples based on verification errors or known pitfalls

Warnings:

- Do not hard code solutions. Submissions will be audited and disqualified if violations are found
- Use the provided agent abstractions and models. Do not use unapproved models (e.g., o1-pro)

Submitting to Gradescope

Run the command `make zip` to package your entire project directory. Submit the resulting `.zip` file to Gradescope.

In addition to your code submission, you may optionally submit a **at most two-page PDF** that describes your agent architecture. While not required, this document is **strongly recommended**, especially if your solution is a strong attempt that may not fully pass all tests. It gives us valuable insight into your system design and can help you earn **partial credit** for thoughtful architecture and engineering. Strong

candidates for this will incorporate some multi-agent workflow, have corrective behavior, use advanced techniques that incorporate some state-of-the-art techniques in current research. Be sure to embed any relevant citations.

The document should briefly outline your agent's design, including:

- The number and role of each agent
- How your system handles planning, code generation, verification, and feedback
- How (if at all) RAG was used to support generation
- Any error-handling or retry logic you implemented
- Design choices and trade-offs
- Highlighting any advanced / state-of-the-art techniques inspired from any relevant research papers.

Note: This is optional, but submitting it may help recover points during regrade requests if your implementation demonstrates strong design, even if the final output does not fully pass all public tests.