Project Report

Student Name: Erfei YU

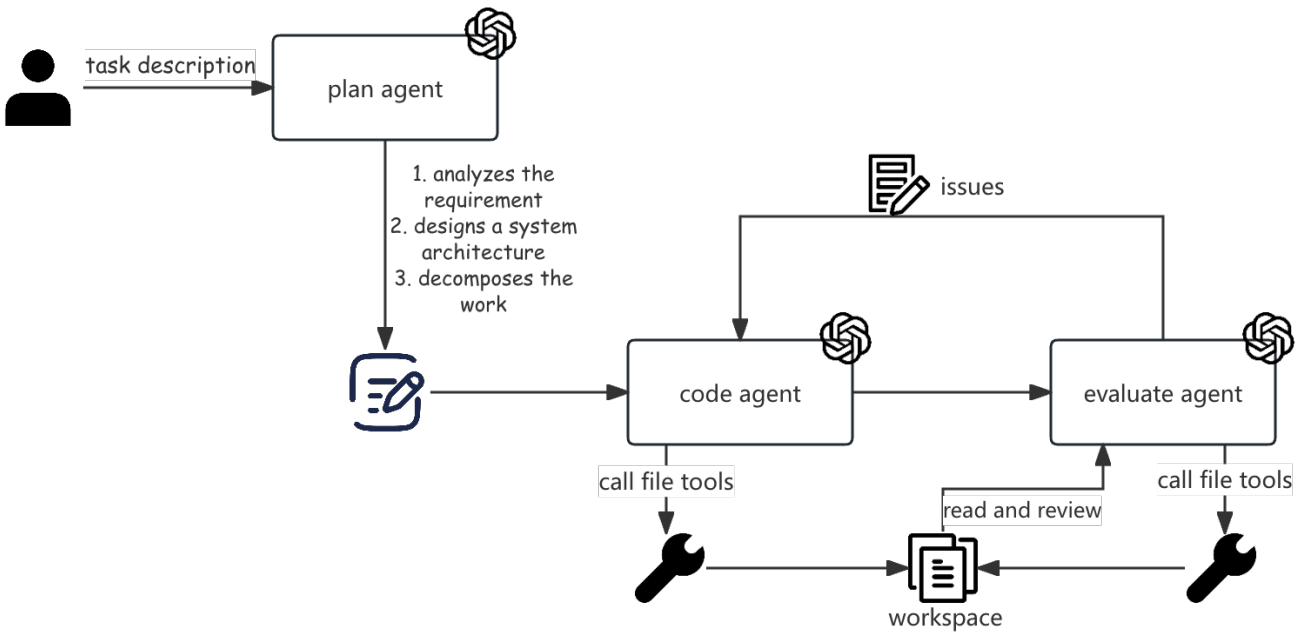
University No.: 3036657732

1. Overview

This report details the development of a **Multi-Agent Code Generation System**, an academic project that automates the translation of natural language requirements into executable code. The system employs a pipeline of four specialized AI agents—Planning, Coding, Evaluation, and Fix—orchestrated by LangGraph. A central innovation is its sophisticated **context management architecture**, which intelligently overcomes LLM context window limitations through stateful workflow management and strategic data compression, enabling the coherent generation of multi-file software projects.

2. System Architecture

The system operates as an automated pipeline. A user provides a natural language task description via a command-line interface built with the Rich library. This input triggers a sequential workflow managed by LangGraph. A Planning Agent first analyzes the requirement, designs a system architecture, selects a technology stack, and decomposes the work into 2-5 concrete subtasks. A Coding Agent then executes these subtasks sequentially, generating or modifying code files within a confined `workspace` directory. Following generation, an Evaluation Agent reviews the code for syntax, functionality, and integration issues. If problems are identified, a Fix Agent analyzes the feedback, groups issues by file, and creates targeted repair subtasks, sending the workflow back to the Coding Agent for iteration. This loop continues for a maximum of three cycles or until the code meets quality standards, culminating in the output of a runnable codebase.



3. Technical Implementation

The system is built using a modern Python stack. The core agent logic and multi-step reasoning are implemented using the LangChain and LangChain-OpenAI frameworks. LangGraph is the critical component for orchestrating the cyclical workflow between the four agents. The OpenAI API serves as the underlying LLM provider. Configuration and data validation are handled by Pydantic and python-dotenv, while the user interacts with a visually styled terminal interface powered by the Rich library.

The project structure is modular, separating agent logic (`src/agents/`), graph orchestration (`src/graph/`), utilities (`src/utils/`), and tools (`src/tools/`). All generated code is safely written to a dedicated `workspace/` directory, ensuring a secure sandbox for file operations.

3.1 Agent Toolkit

The agents' capabilities are extended through a set of practical tools that bridge the gap between AI reasoning and the physical file and execution environment.

- **File System Tools:** These are the most critical tools, primarily used by the Coding and Evaluation Agents.
    - Reads and returns the content of a specified file within the `workspace` directory. Essential for the Coding Agent's repair mode and for the Evaluation Agent's analysis.
    - Creates or overwrites a file at the given path with the provided content. This is the primary mechanism through which the Coding Agent materializes its generated code.
    - Lists all files in a specified subdirectory of `workspace`, helping agents understand the current project structure. All file paths are strictly sandboxed to the `workspace` root for security.
- **Bash Execution Tools:**
    - Executes a shell command in a controlled subprocess. This tool is vital for the **Evaluation Agent** to run static analysis tools (e.g., `python -m py_compile`, `node -c`, `eslint`) on the generated code to catch syntax and style errors programmatically. The tool captures and returns `stdout`, `stderr`, and the return code, allowing the agent to interpret the success or failure of the check.
- **Data Fetching Tools:**
    - Performs a HTTP GET request to fetch data from a public API or URL. This tool enables the **Planning or Coding Agents** to incorporate external data or boilerplate code snippets based on the chosen technology stack. For instance, an agent could fetch the latest CSS reset from a CDN or retrieve mock data from a public API to include in the generated application.

---

4. Context Management Strategy

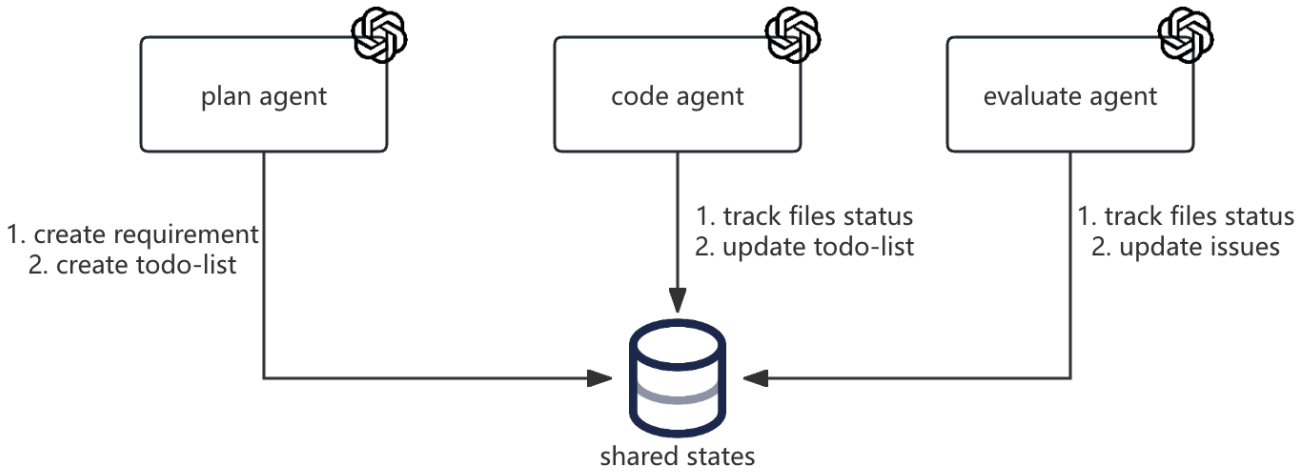
4.1 The Key Challenge and Strategic Approach

A fundamental constraint in multi-step, multi-file code generation is the finite context window of LLMs. Passing the complete content of all previously generated files to each agent would quickly exhaust token limits and degrade performance. Our system solves this through a dual-strategy approach: **centralized state management** for workflow coherence and **intelligent context compression** for efficient information transfer.

4.2 Centralized State Management with LangGraph

The system maintains a single source of truth through a shared `AgentState`, a TypedDict defined in `state.py`. This state object is passed through the entire LangGraph workflow, accumulating outputs from each phase.

- **Key State Fields:** It contains the original `task_description`, planning outputs (`architecture_plan`, `subtasks`), coding outputs (`generated_files`), evaluation results, and control variables like `iteration_count`.



4.3 Intelligent Context Compression and Summarization

Instead of passing raw file content, the system provides agents with tailored, highly compressed summaries, dramatically reducing token usage.

- **For the Coding Agent:** The `format_existing_files()` function generates intelligent digests. For example, an HTML file is summarized by listing its critical `<link>` and `<script>` tags; a CSS file by its key selectors; a JavaScript file by its imports/exports and function names. This compresses files by **90-95%**, allowing the coder to maintain project-wide awareness without context overflow.
- **For the Evaluation Agent:** To check for integration issues (e.g., DOM element mismatches), the agent receives only the first 100 lines of each generated file, paired with separate syntax check results. This provides sufficient insight into structure without the burden of full files.
- **Dual-Mode Context for the Coder:** The strategy adapts based on the task.
    - In **Generation Mode**, the agent receives summaries of all existing files to ensure consistency.
    - In **Modification Mode** (triggered by the Fix Node), it receives the *full content* of the problematic file alongside summaries of others, enabling precise edits with necessary context.

4.4 Context Flow Through the Workflow

The strategy ensures each agent receives precisely the context it needs:

1. The **Planning Agent** requires only the original user input.
2. The **Coding Agent** receives the full plan, the current subtask, and—most importantly—the compressed summaries of the existing codebase.
3. The **Evaluation Agent** receives truncated file previews and syntax reports.
4. The **Fix Agent** works on the structured `evaluation_results` to create new, focused subtasks.

This orchestrated flow allows the system to manage complex projects, simulating a developer's growing understanding of the codebase while strictly adhering to LLM context limits. It is the foundational mechanism that enables the system's practical scalability beyond trivial single-file examples.

5. Performance

test case 1 - simple web app

User input:

```
Build a calculator with add, sub, mul, divide opration in purple background
```

Intermediate status：

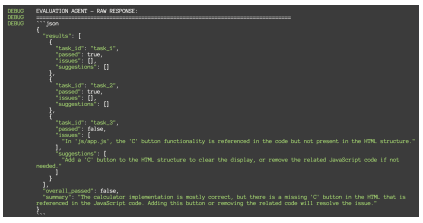
1. todo list：

```
**All Subtasks Status**: 1. [→] Create HTML structure for calculator (正在进行 ← 你现在的任务)
   文件: index.html
2. [ ] Style the calculator with CSS (待处理)
   文件: styles/main.css
3. [ ] Implement calculator functionality with JavaScript (待处理)
   文件: js/app.js
```

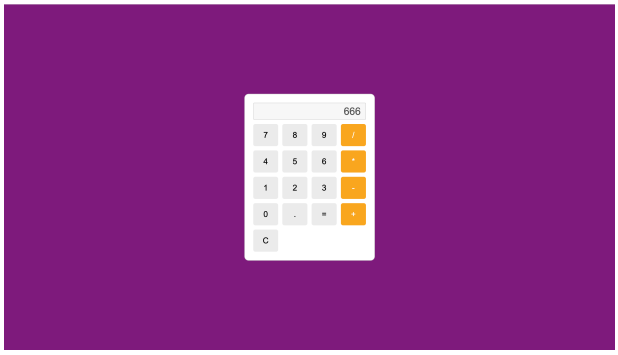
2. files context:

```
**Other Files Summary**: 📄 index.html
HTML 文件 (共 33 行)
引用:
  <link rel="stylesheet" href="styles/main.css">
  <script src="js/app.js" defer></script>

📄 styles/main.css
CSS 文件 (共 65 行)
主要选择器:
  body
  .calculator
  .display
  .buttons
  button
  button:hover
  button:active
  button.operation
  button.operation:hover
  button.operation:active
```

3. code review results:



Final Results:



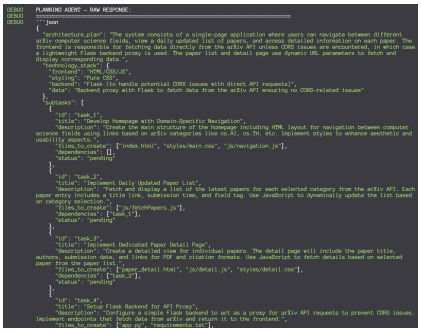
test case 2 -- arXiv webpage

User input:

```
Build an "arXiv CS Daily" webpage with three core functionalities to deliver a streamlined
experience for tracking daily computer science preprints:

1. Domain-Specific Navigation System
Implement categorized navigation based on arXiv's primary CS fields (cs.AI, cs.TH, cs.SY,
etc.).
This enables users to quickly filter and switch between major subfields, ensuring easy
access to their areas of interest.

2. Daily Updated Paper List
Create a daily updated list displaying the latest papers with essential details only. Each
entry may include the paper title (hyperlinked to its detail page), submission time, and
the specific arXiv field tag (e.g., [cs.CV]).

3. Dedicated Paper Detail Page
Design a comprehensive detail page that centralizes critical resources: direct PDF link
(hosted on arXiv), core metadata (title, authors with affiliations, submission date), and
citation generation tools supporting common formats (BibTeX, standard academic citation)
with one-click copy functionality.
```

Intermediate status：

1. task decomposition:



2. todo list:

```
**All Subtasks Status**: 1. [→] Implement Domain-Specific Navigation System (正在进行 ← 你现在的任务)
   文件: index.html, styles/main.css, js/navigation.js
2. [ ] Develop Daily Updated Paper List (待处理)
   文件: index.html, js/paperList.js, backend/app.py
3. [ ] Create Dedicated Paper Detail Page (待处理)
   文件: paperDetail.html, js/paperDetail.js, backend/app.py
```

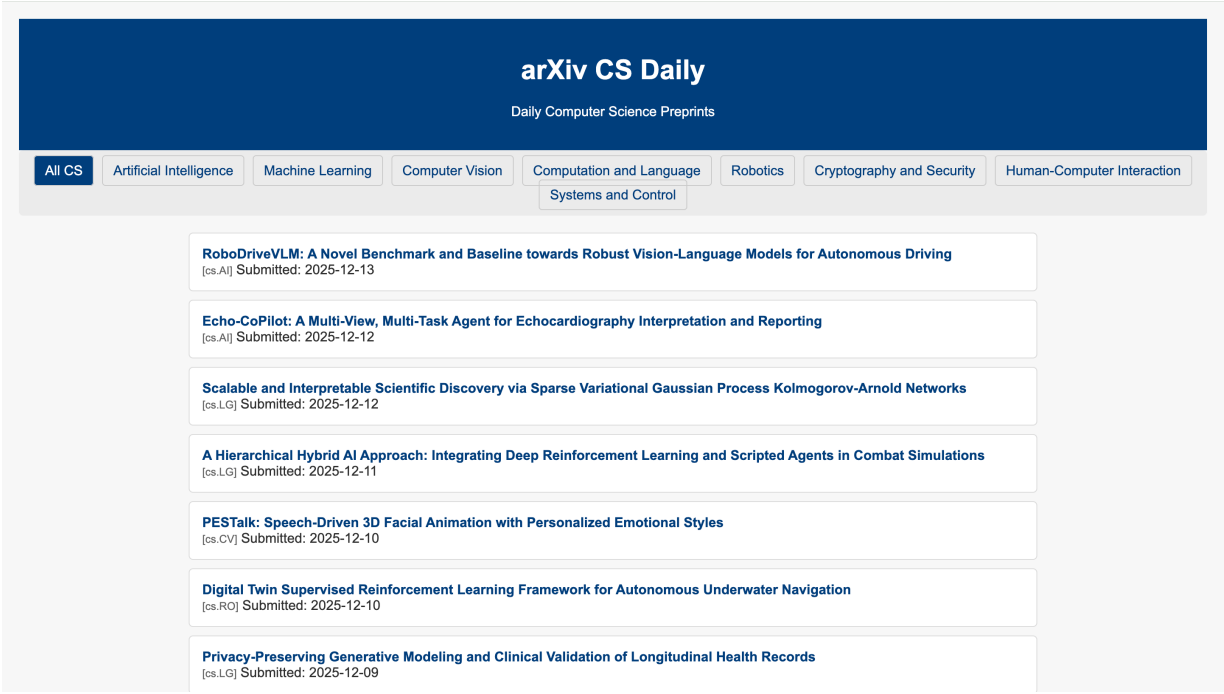
3. files context:

```
**Existing Files (reference only)**: 📄 index.html
HTML 文件 (共 24 行)
引用:
  <link rel="stylesheet" href="styles/main.css">
  <script src="js/navigation.js" defer></script>

📄 styles/main.css
CSS 文件 (共 43 行)
主要选择器:
  body
  header
  nav ul
  nav ul li
  nav ul li a
  nav ul li a:hover
  main
  #paper-list

📄 js/navigation.js
JavaScript 文件 (共 24 行)
函数:
  function updatePaperList()
  function to update the paper list()
```

4. code review:


```
    {
      "task_id": "fix_0_4",
      "passed": false,
      "issues": [
        "CORS handling: Ensure CORS is properly configured in the Flask app to allow requests from the frontend."
      ],
      "suggestions": [
        "Verify that CORS is enabled correctly in the Flask app using flask_cors. Ensure the frontend is calling the local backend and not directly accessing external resources."
      ]
    }
  ],
  "overall_passed": false,
  "summary": "Most tasks passed without issues. However, the backend requires proper CORS configuration to ensure frontend can communicate with it correctly."
```


Final results



Appendix

GitHub Repo: https://github.com/454270186/coding-agent