

# Project Report: Multi-Agent Coding System Powered by DeepSeek-R1

---

**Student:** Lyu Linze 3036658619

**Course:** COMP7103 – Data Mining

**github link:** [https://github.com/1297656202/agent system](https://github.com/1297656202/agent_system)

---

## 1. Introduction

---

Large Language Models (LLMs) have demonstrated remarkable capabilities in software generation. However, using a single model to generate full applications often leads to structural inconsistencies, broken files, and unmaintainable code.

To address this, we designed a multi-agent software engineering framework that decomposes the coding workflow into modular autonomous agents.

This project presents a fully functional multi-agent coding system using DeepSeek-R1 as the code generator. The goal is to build a robust pipeline capable of reading natural-language requirements, planning a project, generating code file-by-file, fixing errors, and validating output.

This framework was applied to build an “arXiv CS Daily” web application, demonstrating the feasibility of using agents to automate complex full-stack software projects.

---

## 2. System Overview

---

The system consists of three major agents:

Agent	Responsibility
<b>PlannerAgent</b>	Interprets user requirements and generates architecture + tasks
<b>CoderAgent</b>	Creates code files one-by-one using DeepSeek
<b>EvaluatorAgent</b>	Validates the project structure and checks for file errors

All agents operate under **LLMClient**, which handles DeepSeek API calls.

---

## 3. System Architecture

---

```
1 user_requirement.txt
2 ↓
3 PlannerAgent
4 ↓
```

```
5   JSON plan
6       ↓
7 CoderAgent
8     (per file)
9       ↓
10  Base64 JSON → decode → write file
11      ↓
12 EvaluatorAgent
13      ↓
14 Final project folder
```

The system follows a strict **file-by-file generation** method, avoiding common LLM pitfalls such as mixed output, incomplete JSON, or truncated files.

## 4. Detailed Component Design

### 4.1 PlannerAgent

PlannerAgent receives natural-language requirements and converts them into:

- project architecture
- module structure
- ordered task list
- list of files to be generated

It ensures **top-down planning**, enabling downstream agents to operate deterministically.

#### Example Output (summarized):

```
1  {
2    "architecture": {
3      "language": "Python",
4      "framework": "Flask",
5      "project_root": "arxiv_cs_daily"
6    },
7    "tasks": [
8      {
9        "id": 1,
10       "name": "Project Setup",
11       "files": ["requirements.txt", "config.py"]
12     },
13     ...
14   ]
15 }
```

## 4.2 CoderAgent

CoderAgent is the core of the system.

### Key innovations:

#### ✓ Single-file generation

Instead of asking the LLM to generate an entire project at once, CoderAgent sends DeepSeek **one file at a time**, drastically reducing hallucination and formatting errors.

#### ✓ Strict JSON-only output

DeepSeek is instructed to output only:

```
1 | {
2 |   "path": "templates/index.html",
3 |   "content_b64": "..."
4 | }
```

#### ✓ Automatic JSON cleaning

DeepSeek often produces:

- explanations (“Let me think...”)
- fenced code blocks
- trailing commentary
- broken JSON

A custom `_extract_json()` loader ensures:

- remove ` ``json
- remove commentary
- extract between first `{` and last `}`

#### ✓ Base64 recovery

DeepSeek frequently outputs truncated Base64.

To fix it:

```
1 | missing = len(content_b64) % 4
2 | content_b64 += "=" * (4 - missing)
```

This eliminates:

- “Incorrect padding” errors
- Binary-safe decoding

## ✓ Binary file skipping

DeepSeek cannot reliably generate `.ico`, `.png`, `.pdf`, etc.

These are automatically skipped and replaced with empty placeholders.

---

## 4.3 EvaluatorAgent

EvaluatorAgent performs:

- file existence verification
- directory integrity checks
- Python syntax tests via `compileall`

This ensures generated code is structurally valid.

---

## 5. Experimental Results

The system successfully generated:

- Full Flask web app folder
- Navigation views
- Categories page
- Paper detail page
- Citation utilities
- JavaScript interactions
- Styling and layout
- API integration modules
- All templates

The agent handled:

- ✓ Complex multi-file HTML/CSS/JS
- ✓ Recursive task dependencies
- ✓ Structured API integration
- ✓ Automated JSON repair
- ✓ Per-file generation with zero crashes

This validates the reliability of the multi-agent architecture.

---

## 6. Key Challenges & Solutions

Challenge	Solution
LLM produces explanations before JSON	Implement <code>_extract_json()</code> cleaner

Challenge	Solution
Broken Base64 strings	Add padding recovery logic
Binary files crash DeepSeek	Skip <code>.ico/.png</code> generation
Long outputs cause connection reset	Single-file generation + retry loop
JSON hallucination	3-stage repair: retry → clean → repair agent
Path confusion in nested modules	Enforce <code>project_root/</code> prefix

The system is now robust against nearly all typical LLM failure modes.

---

## 7. Reflection

This project demonstrates that:

- LLMs need **strong constraints**, not freedom
- Multi-agent decomposition dramatically improves reliability
- File-by-file generation is the most stable approach
- Automated JSON fixing and Base64 padding are essential
- DeepSeek is powerful but requires aggressive output sanitization

Future work may include:

- Self-healing: evaluator sends error reports to coder
  - Streaming LLM calls for accelerated performance
  - Support for binary file download or template replacement
  - More advanced debugging agents
- 

## 8. Conclusion

We successfully built a fully functioning multi-agent coding system capable of generating complete software applications autonomously.

The system is stable, modular, extensible, and demonstrates real-world feasibility of LLM-driven autonomous software development.