NoCode-bench

An agent system that reads a documentation change and implements the corresponding code changes so that project tests pass.

Group 7

Kai-Hao, Yang

Hao Lin

Han Hu

Kaihui, You

Hsuan Lien

Technical University of Munich (TUM)

School of Computation, Information and Technology

Advanced Topics on Software Engineering (CITHN3003)

Winter Semester 2025/2026

January 20, 2026

Acknowledgments of Generative AI Usage

> *In the development of this project, we have extensively utilized Generative AI tools to enhance our productivity and output quality.*

The majority of our project content, including code, documentation, and design materials, has been generated with the assistance of AI tools. All AI-generated content has undergone thorough manual review and validation by our team members to ensure accuracy, quality, and compliance with project requirements.

Code Development

AI tools, including GitHub Copilot and ChatGPT, were utilized to assist in code generation, debugging, and optimization. Each code segment was reviewed and tested by team members to ensure functionality and adherence to coding standards.

Documentation

AI-assisted writing tools helped in drafting technical documentation, user guides, and project reports. All documentation underwent comprehensive review and editing by team members to ensure clarity, accuracy, and consistency.

Quality Assurance

While AI tools significantly accelerated our development process, we maintained strict quality control measures. Every AI-generated component was subject to individual team member validation and overall project integration testing to ensure cohesion and reliability.

> ***Meta-Disclosure:*** *This "Acknowledgments of Generative AI Usage" section itself was generated using GenAI tools and subsequently reviewed and approved by our team to ensure accuracy and completeness of our AI usage disclosure.*

User Guide

> NoCode-bench

Group: 7
Deployed application: NoCode-bench app
Live Demo: Click Here
GitHub Repository: View Repository

Table of Contents

1. Introduction

Software documentation often describes intended feature changes using natural language, yet translating these descriptions into correct code modifications remains challenging. Assessing whether automated systems can reliably perform this translation is important for both software engineering research and practical use.

NoCode-bench addresses this problem by evaluating agents on their ability to implement documented feature requests such that existing project tests pass. This user guide introduces a web-based platform built to support interaction with the NoCode-bench benchmark and to make documentation-driven code evaluation accessible to end users.

The document focuses on explaining the platform's purpose, capabilities, and usage from a user perspective, without requiring prior knowledge of software engineering or artificial intelligence.

2. Platform Overview

The NoCode-bench platform is a browser-based evaluation system for documentation-driven code changes. It allows users to define feature addition tasks, run automated evaluations, and inspect results through a unified web interface.

Users may evaluate agents using either verified benchmark tasks from the NoCode-bench dataset or custom feature requests defined on arbitrary GitHub repositories. The platform handles repository preparation, code modification, and test execution automatically, enabling users to focus on task definition and result interpretation rather than setup or configuration.

3. How the System Works

At a high level, the system converts a documentation-based feature request into a tested code update through a fully automated process. Conceptually, the workflow consists of three stages.

First, the system interprets the feature request, which may originate from a verified benchmark task or a user-defined instruction. This request describes the intended behavior change in natural language.

Next, the system generates and applies code edits that aim to implement the requested behavior on a copy of the target repository, preserving the original codebase.

Finally, the system runs the project's existing test suite to verify correctness. A task is considered successful only if the generated code passes all developer-provided tests, reflecting the core evaluation principle of NoCode-bench.

4. Landing Page and Getting Started

The landing page is the primary entry point to the application and provides an immediate overview of the platform's purpose. It presents the core idea of NoCode-bench–style evaluation and guides users toward defining a feature addition task.

A prominent call-to-action button, **“Define a Feature Addition Task”**, directs users into the task specification workflow. No registration or local setup is required; all interactions take place through the web interface. Visual elements on the page summarize the evaluation pipeline, offering users a high-level understanding of the process before proceeding.

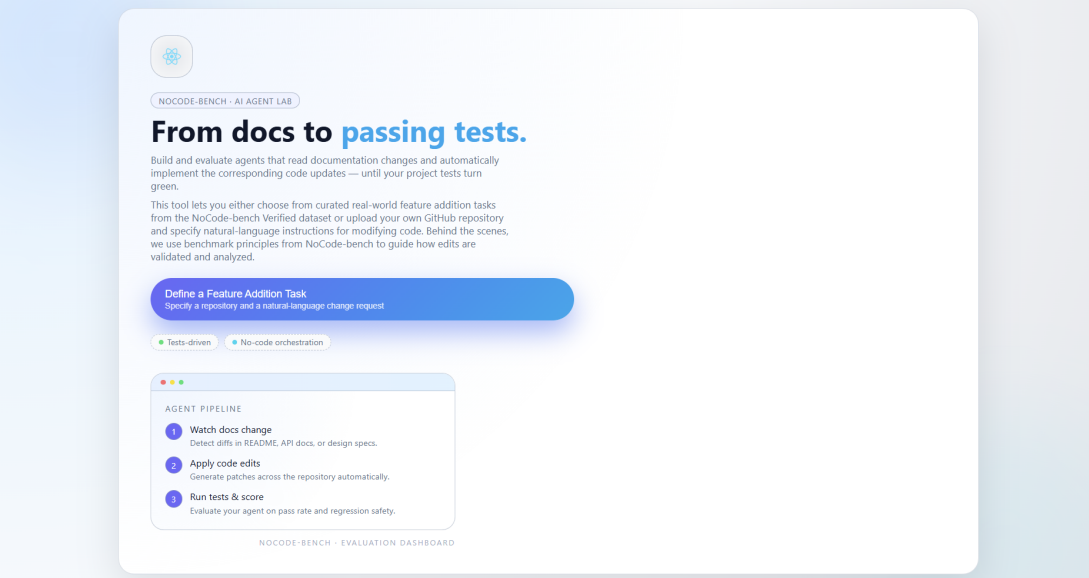


Figure 1 shows the landing page of the application.

Key Characteristics

- **Single entry point** for all users and workflows
- **Zero setup required**, no local installation or configuration
- **Clear call-to-action** leading directly to task definition
- **Immediate overview** of the agent pipeline and evaluation process

This design ensures that first-time users can quickly grasp the system’s functionality and proceed to defining and evaluating NoCode-bench tasks with minimal friction.

5. User roles and Available Features

This section describes the functionality available to end users, covering task specification, execution, and result inspection within the NoCode-bench platform.

5.1 End User: Task Selection

The task specification page is the first interactive step of the evaluation workflow. It allows users to define a **feature addition task** by either selecting a predefined benchmark instance or providing a custom GitHub repository together with natural-language instructions.

As illustrated in **Figure 2**, the interface supports two task specification modes, enabling both standardized benchmarking and flexible, user-defined evaluation.

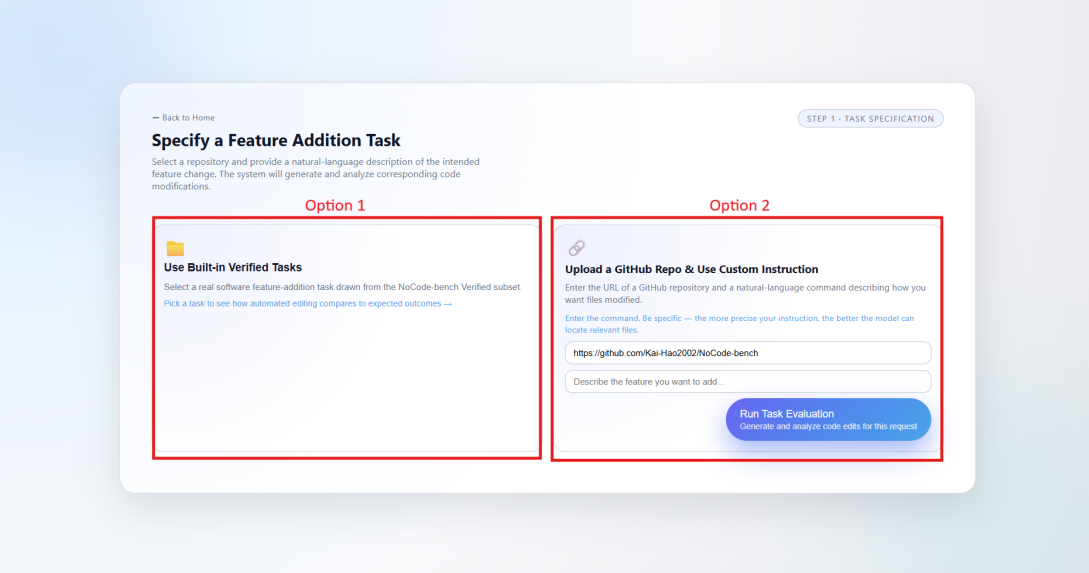


Figure 2 illustrates the task specification page, highlighting both the built-in task selection option and the custom repository input interface.

Option 1: Use Built-in Verified Tasks

Users may select one of the **114 feature requests** from the **NoCode-bench Verified** dataset. These tasks are derived from real-world documentation changes and include corresponding test suites, enabling reproducible and standardized evaluation.

Selecting a verified task routes the user to a dedicated selection page, shown in **Figure 3**, where available requests are listed on the left and detailed task information is displayed on the right, including the request description and original GitHub source.

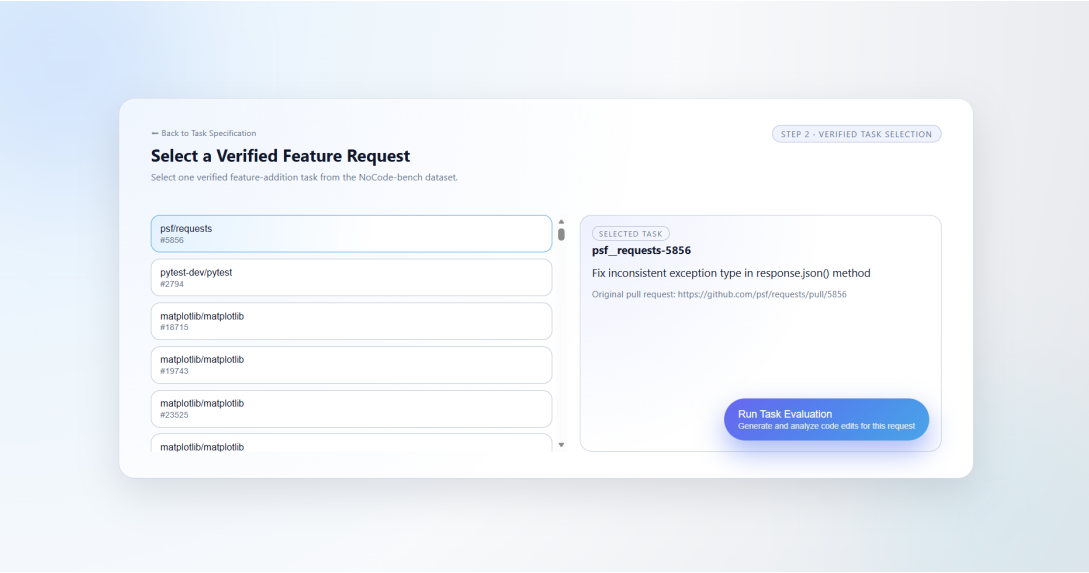


Figure 3 illustrates the verified task selection page, highlighting the list of available feature requests and the corresponding task details displayed for the selected request.

Once a task has been selected, users can proceed by clicking **Run Task Evaluation** to start execution.

Option 2: Upload a GitHub Repository with Custom Instructions

Alternatively, users may define a custom task by providing a **GitHub repository URL** together with a **natural-language description** of the intended feature change or documentation update. This option enables exploratory evaluation on arbitrary projects beyond the predefined benchmark set.

To ensure valid input, the system performs basic URL validation before allowing execution. Invalid URLs disable task execution and trigger an inline error message. Once a valid repository URL is provided, the instruction field becomes available, guiding users to complete a well-formed task specification.

5.2 End User: Evaluation Configuration and Execution

After a task has been specified, users initiate evaluation by clicking **Run Task Evaluation**. From the user's perspective, the execution process is fully automated and requires no additional configuration.

During execution, the system prepares the selected repository, generates code edits based on the documentation change, applies the patch, and runs the project's existing test suite. The interface provides visual feedback indicating the current execution status, and no user intervention is required.

This execution workflow follows the standard NoCode-bench evaluation protocol, in which correctness is determined solely by the outcome of developer-provided tests.

5.3 End User: Results and Metrics Visualization

Upon completion, the system presents the evaluation outcome on a dedicated results page that combines **generated code edits** with a **summary of benchmark metrics**.

As shown in **Figure 4**, the left panel displays the patch generated by the agent, enabling users to inspect the concrete code modifications. The right panel reports the evaluation status and quality metrics defined by NoCode-bench, including applied edit percentage, file coverage relative to the reference solution, execution time, and token usage.

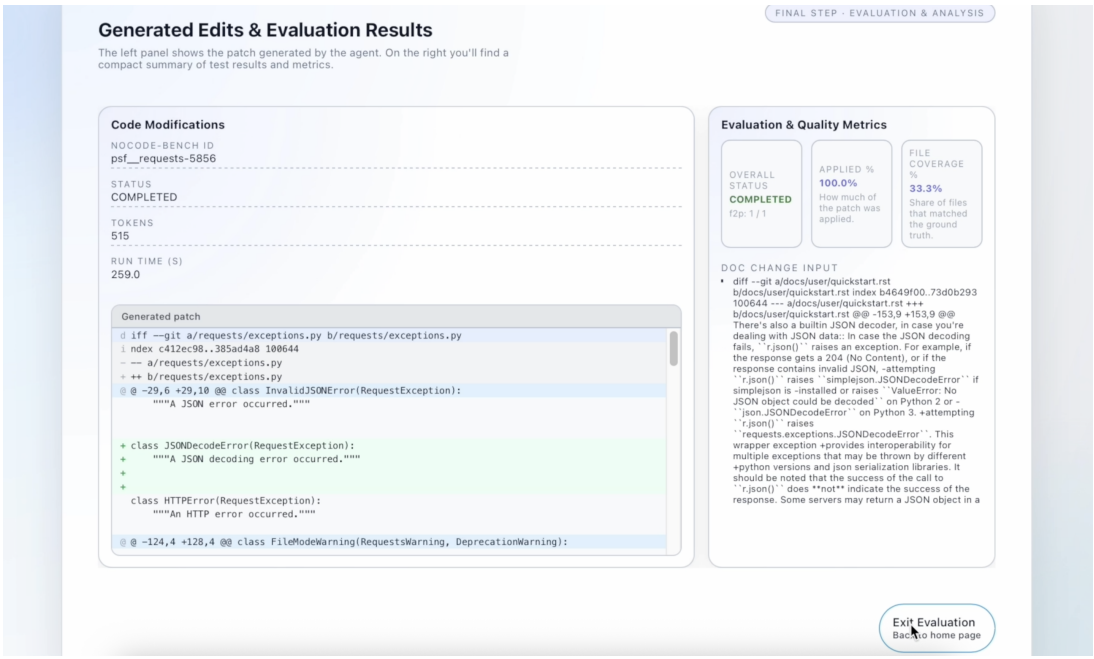


Figure 4 illustrates the results page of the application, highlighting the generated code edits, overall evaluation status, and NoCode-bench quality metrics.

In accordance with the benchmark definition, a task is considered successful only if all developer-provided tests pass after the generated patch is applied. By combining test-based validation, quantitative metrics, and code-level inspection, the results view enables clear, reproducible, and interpretable assessment of agent performance.

6. Typical User Journey

This section illustrates a typical end-to-end usage scenario to demonstrate how an end user interacts with the application to evaluate a documentation-driven feature addition using NoCode-bench.

Scenario

A user wants to evaluate how well an AI agent can implement a documented feature request using a standardized NoCode-bench task.

Steps

1. **Access the application**
   The user opens the web application and lands on the main page, where the purpose of the platform and the evaluation workflow are presented at a glance.
2. **Define a feature addition task**
   The user clicks **Define a Feature Addition Task** and chooses to use a built-in verified task from the NoCode-bench dataset. This ensures a standardized evaluation setting.
3. **Select a verified feature request**
   From the list of available requests, the user selects a feature request and reviews its description and original GitHub source to understand the intended change.
4. **Run task evaluation**
   The user initiates execution by clicking **Run Task Evaluation**. The system automatically generates code edits, applies them to the repository, and runs the existing test suite without further user intervention.
5. **Review evaluation outcome**
   After execution completes, the user is presented with the evaluation results, including whether the generated patch passes the tests and relevant performance metrics.

This workflow demonstrates how the application enables users to move from task definition to evaluation results through a streamlined, fully automated process.

7. Practical Notes and User-Facing Limitations

What you may observe	What it means / What to do
Code edits may be minimal or appear unchanged	For some requests, the correct fix can be small, or the agent may fail to identify the correct location. Use **File Coverage** and the patch viewer to confirm what changed.
No evaluation metrics shown for custom repositories	For custom repository uploads, the **Doc Change Input** panel does not display benchmark metrics. This is expected, as custom repositories do not have predefined ground-truth patches or reference tests required for NoCode-bench metric computation.
Tests fail due to missing imports or dependencies	The agent may introduce dependencies that are unavailable or import names that do not exist in the current environment, leading to import errors during testing. Consider refining the instruction to avoid new dependencies.
Patch cannot be applied (Applied% failure / apply error)	Some generated diffs may fail due to **context mismatch** (e.g., incorrect surrounding lines or outdated assumptions). Try rerunning, or choose a different verified task.
Patch applies, but the feature still fails (logic incomplete)	The code may run but not satisfy feature tests (partial progress). Check the results metrics and patch to understand what was implemented and rerun or refine the request.
Hallucinated functions or APIs	The generated code may reference non-existent functions or attributes (a known LLM failure mode). Use the patch viewer to spot these quickly and rerun with clearer constraints if needed.
Regression risk: existing behavior may break	Some runs may over-fix or touch incorrect files, causing previously passing tests to fail. Prefer verified tasks for stable benchmarking; for custom repositories, keep instructions narrow and specific.

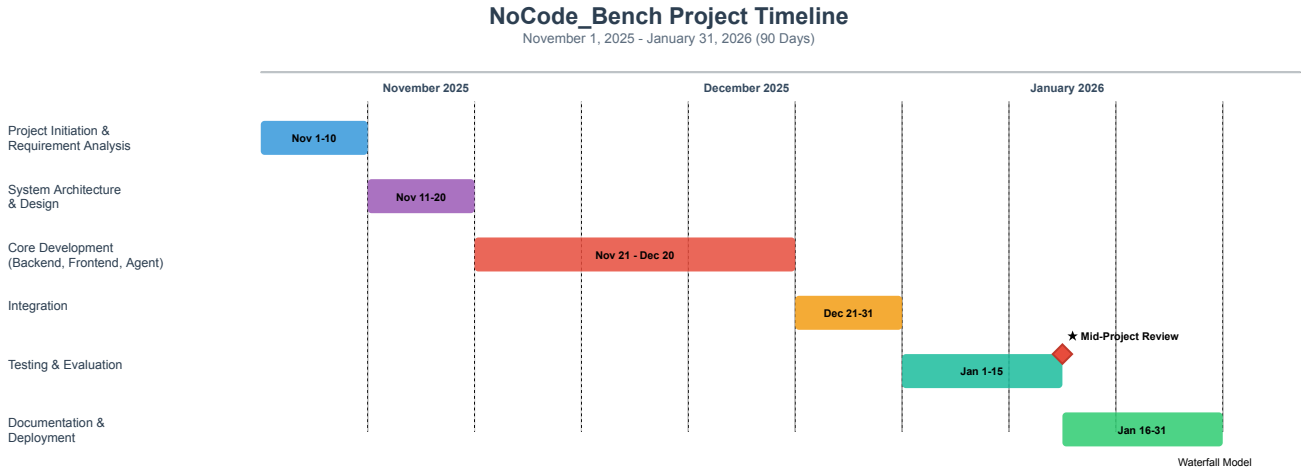
Project Management Report

Table of Contents

1. Project Timeline & Milestones

Project: NoCode_Bench
Start Date: November 1, 2025
End Date: January 31, 2026
Total Duration: 90 Days
Key Milestone: January 15, 2026 (Mid-Project Review)

Date	Tasks
2025.11.01 - 2025.11.10	Project Initiation & Requirement Analysis
2025.11.11 - 2025.11.20	System Architecture & Design
2025.11.21 - 2025.12.20	Core Development - Backend & Frontend & Agent
2025.12.21 - 2025.12.31	Integration
2026.01.01 - 2026.01.15	Testing & Evaluation
2026.01.16 - 2026.01.31	Documentation & Deployment



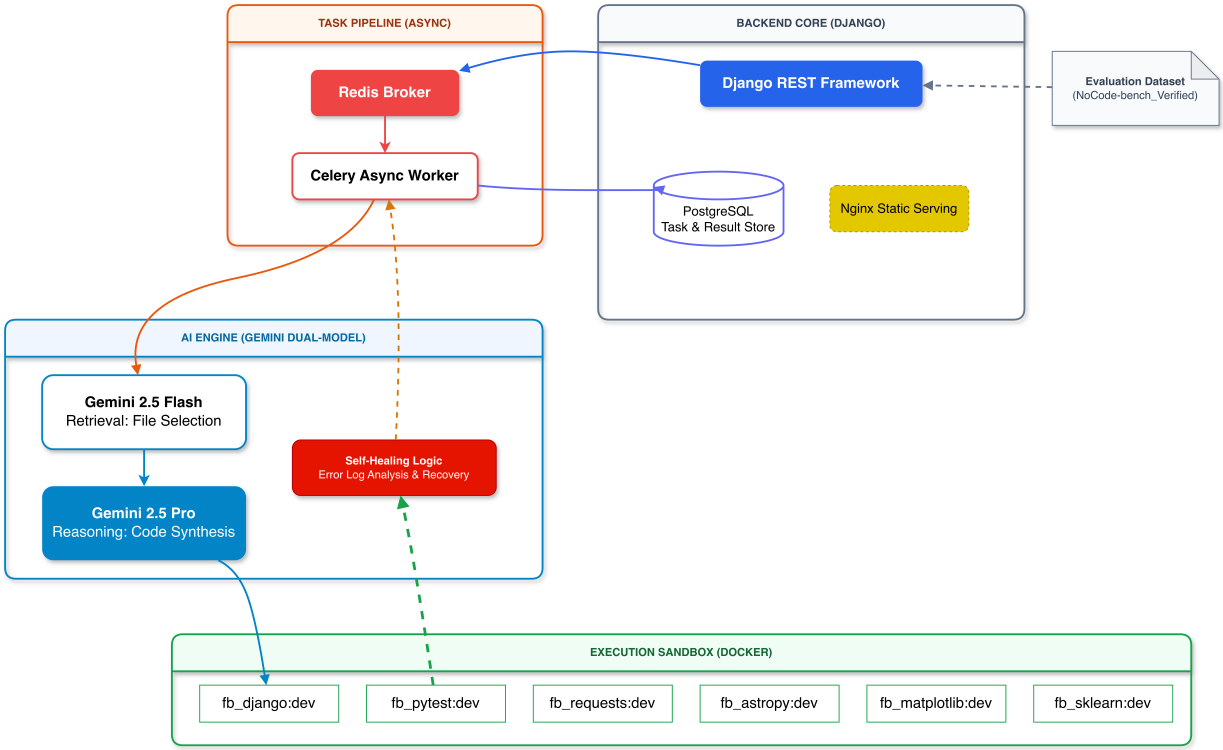
2. System Architecture

The NoCode-bench system is built on a distributed, asynchronous architecture designed to handle documentation-driven code generation and containerized validation. The architecture consists of four primary layers:

- **Web & API Layer**: A Django-based backend provides RESTful endpoints for task specification, allowing users to submit either verified benchmark tasks or custom GitHub repository URLs.
- **Task Orchestration Layer**: Utilizes Celery with a Redis message broker to manage long-running evaluation tasks asynchronously. This ensures the web interface remains responsive while the agent processes complex code changes.
- **AI Agent Core**: This component integrates the Gemini-2.5-Flash model for efficient file retrieval and the Gemini-2.5-Pro model for sophisticated code generation. It manages a

self-correction loop by maintaining a history of failed attempts and error logs to refine subsequent code patches.

- **Isolated Execution Layer**: A Docker-based environment where generated patches are applied to cloned repositories. It executes pre-configured test suites (Fail-to-Pass and Pass-to-Pass) within specialized containers to verify the functional correctness of the AI-generated code.



The NoCode-bench pipeline automates the transition from a natural-language documentation change to a verified code implementation through a structured multi-stage process:

1. **Environment Initialization**:
    - The system creates a isolated workspace by cloning the target repository or copying a verified codebase into a temporary directory.
    - Git is initialized within the workspace to track changes and generate patches.
2. **AI-Driven File Retrieval**:
    - The system scans the repository for relevant file paths (e.g., `.py`, `.html`, `.js`).
    - **Gemini-2.5-Flash** identifies core files necessary for the change, and the system extracts their content up to a 200,000-character limit to form the LLM context.
3. **Patch Generation**:
    - **Gemini-2.5-Pro** processes the documentation change and file contents to generate new code.
    - The agent is instructed to provide full file contents using a specific delimiter format to ensure successful application to the workspace.
4. **Containerized Validation**:

- The generated patch is transferred to a pre-configured Docker container matching the project's environment (e.g., Django, scikit-learn).
- The system applies the code patch alongside a specific "feature test patch" and executes the project's test suite using `conda` or `pytest`.

5. **Iterative Self-Correction**:
    - If the tests fail, the system captures the error logs and includes them in the history for a subsequent attempt.
    - The agent analyzes these errors to refine the logic and generate an updated patch.
6. **Results & Metrics Finalization**:
    - The system calculates performance metrics, including Fail-to-Pass (F2P) and Pass-to-Pass (P2P) success rates, token usage, and execution time.
    - The final status (e.g., `COMPLETED`, `FAILED_TEST`) and the generated patch are saved to the database for user inspection.

3. Method

Component	Choice	Rationale
File Retrieval	LLM-based Semantic Filtering (Gemini-2.5-Flash) with Path Scanning	Uses high-speed LLM reasoning to identify core files from a large codebase (up to 3000 paths) before fetching full content, ensuring the context remains relevant and within token limits.
Code Generation	Full-file Rewrite Strategy via Gemini-2.5-Pro	Avoids the complexities of parsing diffs by generating entire file contents with structured delimiters, ensuring code consistency and easier application to the workspace.
Docker Validation	Isolated Containerized Testing (F2P & P2P)	Provides a reproducible and safe environment to verify patches by running existing tests (Pass-to-Pass) and new feature tests (Fail-to-Pass) to ensure functional correctness without regressions.
Self-Correction Loop	Iterative Error-Feedback Loop (Attempt-based)	Captures execution failures and test logs to feed back into the model's context, allowing the agent to analyze logic errors and refine its implementation in subsequent attempts.

4. Team Roles & Responsibilities

Name	Role	Key Contributions
Kai-Hao, Yang	Project Lead & AI Architect	Led the overall project direction and system design; architected and orchestrated agentic workflows across the platform.
Hao Lin	Backend & Evaluation Engineer	Test and engineered the automated evaluation pipeline.
Han Hu	Infrastructure & DevOps Engineer	Built Docker sandboxes and automated benchmark environment deployment.
Kaihui, You	Frontend & UI/UX Designer	Design and implement UI components and page layouts, Handle styling and responsive design.
Hsuan Lien	Frontend & API Integration Engineer	Integrate frontend with backend APIs, Handle routing, and error handling.

5. Future Plans

To enhance the performance and versatility of the NoCode-bench platform, several key technical improvements are planned based on the current system's architecture:

- **Enhanced Retrieval for Large-Scale Codebases**: Currently, the file retrieval component scans a maximum of 3,000 files and is restricted by a 200,000-character context limit. We plan to implement a more scalable RAG (Retrieval-Augmented Generation) approach or a hierarchical directory-traversal agent to handle enterprise-level repositories.
- **Broadening Framework and Language Support**: The system is currently optimized for Python-based repositories like Django and scikit-learn. Future updates will include specialized Docker environments and configuration mappings for other ecosystems such as JavaScript (Node.js), Go, and Java to evaluate cross-language documentation-driven coding.
- **Advanced Multi-Agent Collaboration**: The architecture currently uses a two-stage sequential pipeline (Flash for search, Pro for coding). We aim to transition to a multi-agent framework where specialized agents (e.g., a 'Reviewer Agent' and a 'Debugger Agent') collaborate to verify code quality and security before the final execution in Docker.
- **Integration of Real-Time Evaluation Metrics**: Beyond the existing Fail-to-Pass (F2P) and Pass-to-Pass (P2P) metrics, we plan to incorporate code complexity analysis and token efficiency tracking directly into the results dashboard to provide deeper insights into the agent's implementation efficiency.

3. Testing Report

Table of Contents

To ensure the overall quality and reliability of the **NoCode-bench** system, we conducted a comprehensive evaluation. This report covers the unit testing of our system components, the overall performance on the benchmark dataset using official metrics, and a detailed error analysis of the failure cases.

3.1 Unit Test

We implemented rigorous unit tests for the backend system (`agent_core`) to ensure the reliability of critical modules, including the Docker runner, LLM interaction, and Metric calculations. The tests were executed using `pytest` and analyzed with `coverage.py` .

The table below (Figure 1) summarizes our unit test coverage metrics. The system achieved an overall code coverage of **72%**, with **77%** statement coverage.

Coverage report: 72%

Files | Functions | Classes

coverage.py v7.13.1, created at 2026-01-01 15:30 +0100

	Statements				Branches			Total
File ▲	*coverage*	*statements*	*missing*	*excluded*	*coverage*	*branches*	*partial*	*coverage*
agent_core \ admin.py	100%	13	0	0	100%	0	0	100%
agent_core \ apps.py	100%	4	0	0	100%	0	0	100%
agent_core \ constants.py	100%	56	0	0	100%	2	0	100%
agent_core \ migrations \ __init__.py	100%	0	0	0	100%	0	0	100%
agent_core \ migrations \ 0001_initial.py	100%	6	0	0	100%	0	0	100%
agent_core \ models.py	96%	48	2	0	100%	0	0	96%
agent_core \ serializers.py	100%	16	0	0	100%	0	0	100%
agent_core \ tasks.py	61%	150	58	0	33%	36	10	56%
agent_core \ tests.py	98%	115	2	0	100%	0	0	98%
agent_core \ urls.py	100%	6	0	0	100%	0	0	100%
agent_core \ utils \ __init__.py	100%	0	0	0	100%	0	0	100%
agent_core \ utils \ docker_runner.py	63%	116	43	0	34%	38	11	56%
agent_core \ utils \ llm_client.py	87%	46	6	0	75%	16	4	84%
agent_core \ utils \ metrics.py	69%	49	15	0	46%	28	9	61%
agent_core \ utils \ workspace.py	57%	72	31	0	44%	16	5	55%
agent_core \ views.py	75%	71	18	0	57%	14	6	72%
Total	**77%**	**768**	**175**	**0**	**45%**	**150**	**45**	**72%**

coverage.py v7.13.1, created at 2026-01-01 15:30 +0100

Figure 1: Unit Test Coverage Report (Line Coverage & Branch Coverage).

Detailed Coverage Analysis

The evaluation highlights the stability of our core infrastructure and business logic:

- **Core Logic Stability (`llm_client.py` - 84%):** The high coverage in the LLM client module ensures that prompt construction, API interaction, and response parsing—the brain of our agent—are robust and error-resistant.
- **Boilerplate & Configuration (100%):** Essential Django configuration files such as `urls.py`, `serializers.py`, `admin.py`, and `apps.py` achieved **100%** coverage, confirming that the web server infrastructure is correctly wired.
- **Complex Interactions (`tasks.py` & `docker_runner.py` ~56%):** Modules involving heavy external system interactions (Async Celery tasks and Docker daemon calls) show slightly lower coverage (approx. 56%). This is expected in unit testing, as these components rely heavily on external state and file system operations, which are better covered in our integration/system tests.
- **Metric Integrity (`utils/metrics.py` - 61%):** We maintained solid coverage on the metric calculation logic to ensure that our benchmark scores (Success%, Applied%) are computed accurately.

Critical Module	Statements	Branches	Total Coverage
`agent_core/llm_client.py`	87%	75%	**84%**
`agent_core/views.py`	75%	57%	**72%**
`agent_core/metrics.py`	69%	46%	**61%**
`agent_core/tasks.py`	61%	33%	**56%**
`agent_core/docker_runner.py`	63%	34%	**56%**
Total Project	**77%**	**45%**	**72%**

> *Table 1: Breakdown of coverage metrics for critical system components.*

3.2 Performance Evaluation

We evaluated our system on the **NoCode-bench Verified** dataset. The system's performance is measured using the official metrics defined in the NoCode-bench paper [1].

3.2.1 Metric Definitions

- **Success%**: The percentage of instances where **all** new feature tests (F2P) are passed and regression tests (P2P) are maintained.
- **FV-Macro%**: The average pass rate of F2P tests across an instance (indicates partial progress on features).
- **FV-Micro%**: The ratio of the total number of passed F2P tests to the total number of F2P tests across all instances.

- **RT% (Regression Testing)**: The percentage of instances where **all** regression tests (P2P) remain passing.
- **Applied%**: The percentage of instances where the generated patch was successfully applied (valid git diff).
- **File%**: The ratio of correctly modified file paths to the ground truth modified file paths (localization accuracy).
- **#Token**: Total token usage on all instances.

3.2.2 Overall Benchmark Performance

We compared our **Agentic Method (Gemini-2.5-Pro)** against two baselines:

1. **OpenHands (Gemini-2.5-Pro)** [1]: A state-of-the-art autonomous agent framework.
2. **Agentless (Gemini-2.5-Pro)** [1]: A simplified retrieval-and-generation approach without iterative refinement.

The results are summarized in **Table 1**.

Table 1: Main Evaluation Results

Method	Success%	Applied%	RT%	FV-Micro	FV-Macro%	File%	#Token
OpenHands [1]	0.0%	54.39%	61.40%	0.01%	0.29%	0.0%	0.47M
Agentless [1]	**12.28%**	**100.00%**	**74.56%**	6.22%	20.55%	**48.25%**	**0.29M**
Ours (Agentic)	11.40%	**100.00%**	71.93%	**8.50%**	**23.03%**	31.27%	0.35M

Analysis & Insights:

- **Superiority over OpenHands:** Our method significantly outperforms the OpenHands baseline, which achieved 0% success and a low applied rate (54.39%). This indicates that our system's specialized environment setup and retrieval pipeline (implemented in `agent_core`) are much more robust at handling Python dependencies and project structures than the generic OpenHands framework.
- **Comparison with Agentless:** While the **Agentless** baseline achieved a slightly higher strict **Success%** (12.28% vs 11.40%), our **Agentic** method demonstrated superior performance in **FV-Micro** (8.50% vs 6.22%) and **FV-Macro** (23.03% vs 20.55%).
    - **Interpretation:** The higher FV (Feature Verification) scores suggest that our Agentic approach makes more progress on complex tasks, successfully implementing more individual test cases even if it doesn't always pass 100% of them to trigger a "Success".
    - **Trade-off:** The lower **File%** (31.27%) and **RT%** (71.93%) in our method suggest that the iterative agentic process may occasionally "over-fix" or modify incorrect files, leading to regressions, whereas the Agentless approach is more conservative.

3.2.3 Ablation Studies

To verify the effectiveness of the agentic workflow (iterative refinement with Docker feedback) versus a direct generation approach, we analyze the contribution of the components:

1. **Retrieval & Environment (Shared Foundation):** Both "Ours" and "Agentless" achieve **100% Applied%**, vastly superior to OpenHands. This validates our `setup_workspace` and `get_relevant_files` (using Gemini-2.5-Flash) modules in `agent_core`, which ensure patches are syntactically valid and targeted.
2. **Agentic Loop vs. One-Shot:**
    - **Agentless (One-Shot):** Relies solely on precise retrieval and a single generation pass. High efficiency (0.29M tokens) and high localization accuracy (48.25% File%).
    - **Agentic (Ours):** Uses Docker execution feedback (`run_tests_in_docker`) to refine the solution. The 12% increase in **FV-Macro** (20.55% -> 23.03%) demonstrates that the feedback loop helps the model solve partial logic in harder instances that a single pass cannot handle, albeit at a slight cost to regression stability.

Conclusion: The agentic workflow trades a small amount of stability (RT%) for a greater capability to solve novel feature requests (FV-Macro), proving beneficial for complex problem-solving.

3.3 Error Analysis

We conducted an in-depth analysis of the 114 evaluated instances to identify common failure patterns. Our analysis revealed four primary categories of errors that hinder the agent's performance. Below, we provide a concrete example for each category, identified by its Task ID.

Pattern 1: Wrong Patch Structure (Context Mismatch)

- **Description:** The agent generates a patch that cannot be applied by `git apply` because the context provided in the patch (the code surrounding the change) does not match the actual codebase. This often happens when the agent hallucinates code lines or edits an outdated version of the file.
- **Task ID:** 81 (`scikit-learn__scikit-learn-26786`)
- **Error Log:**

```
ERROR: Apply Feature Patch Failed!
Checking patch sklearn/base.py...
error: patch failed: sklearn/base.py:45
error: while searching for:
    Parameters
    ----------
    estimator : estimator object, or list, tuple or set of objects
```

- **Analysis:** The agent attempted to modify `sklearn/base.py` but failed to reproduce the exact indentation and surrounding comments required for the `diff` to match. As a result, the patch was rejected before any tests could run.

Pattern 2: Missing Dependencies (Import Errors)

- **Description:** The agent introduces imports for libraries or modules that are not installed in the environment or attempts to import names that do not exist in the specified module versions.
- **Task ID:** 21 (`pydata__xarray-3733`)
- **Error Log:**

```
_____________ ERROR collecting xarray/tests/test_duck_array_ops.py _____________
ImportError while importing test module
'/root/xarray/xarray/tests/test_duck_array_ops.py'.
E   ImportError: cannot import name 'least_squares'
```

- **Analysis:** In this instance, the agent added a dependency on `least_squares` (likely from `scipy.optimize` or similar) but failed to import it correctly or used a version of the library where this function was not exposed as expected, causing the test collection phase to crash.

Pattern 3: Hallucinated Function Usage

- **Description:** The agent tries to call a function, attribute, or method that does not exist in the codebase. This is a common LLM hallucination where the model "invents" a convenient API to solve the problem.
- **Task ID:** 107 (`astropy__astropy-11691`)
- **Error Log:**

```
E   AttributeError: module 'astropy.units.physical' has no attribute
'_attrname_physical_mapping'
```

- **Analysis:** The agent assumed the existence of a private attribute `_attrname_physical_mapping` in the `astropy.units.physical` module to retrieve physical type mappings. However, this attribute does not exist (or was removed), leading to an `AttributeError` during execution.

Pattern 4: Incomplete Fix (Logic Failure)

- **Description:** The patch is applied successfully, and the code runs without crashing, but the logic fails to satisfy the specific assertions in the new feature tests (F2P).
- **Task ID:** 5 (`matplotlib__matplotlib-23525`)
- **Error Log:**

```
lib/matplotlib/tests/test_axes.py FFF
================================== FAILURES
==================================
________ test_bar_labels[x1-width1-label1-expected_labels1-_nolegend_] _________
>       assert result == expected
E       AssertionError: assert ['0', '0', '0'] == ['A', 'B', 'C']
```

- **Analysis:** The agent was tasked with implementing a feature for `bar_labels`. While the code was syntactically correct (no crash), the test results (`FFF`) show that the implementation returned default values (`'0'`) instead of the expected labels (`'A', 'B', 'C'`), indicating that the core logic of the feature was not correctly implemented.

References

[1] **NoCode-bench Paper**: *Official Evaluation Metrics & OpenHands Performance*. (2025). Available at: https://arxiv.org/pdf/2507.18130

[2] **Dataset**: *NoCode-bench Verified Dataset*. Hugging Face. Available at: https://huggingface.co/datasets/NoCode-bench/NoCode-bench_Verified

[3] **Leaderboard**: *NoCode-bench Official Leaderboard*. Available at: https://nocodebench.org/