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.

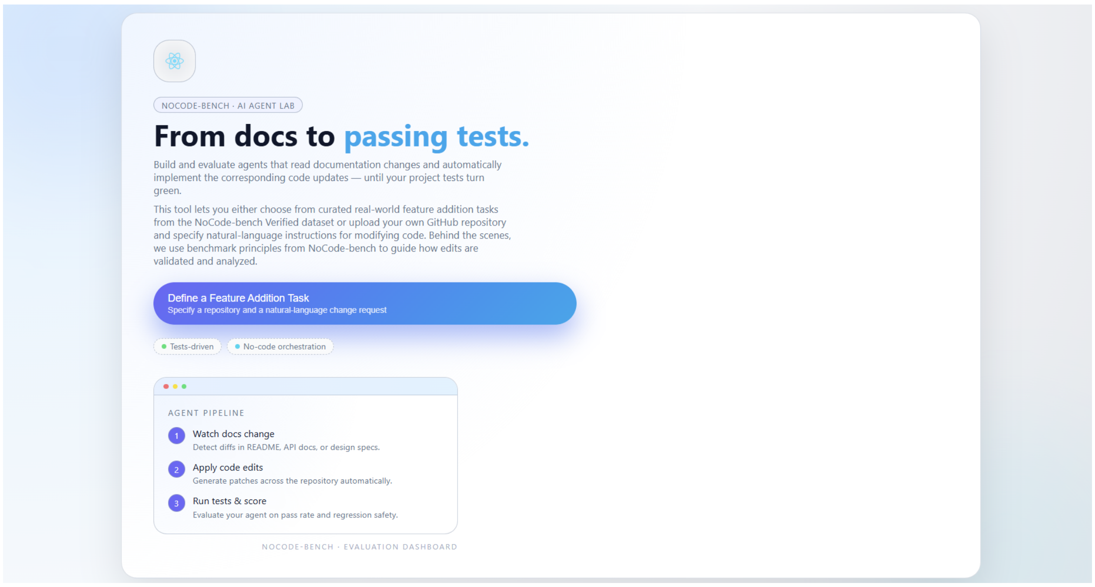
Next, the system generates and applies code edits that aim to implement the requested behavior on a copy of the target repository, preserving the original codebase.

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.



NOCODE-BENCH · AI AGENT LAB

From docs to passing tests.

Build and evaluate agents that read documentation changes and automatically implement the corresponding code updates — until your project tests turn green.

This tool lets you either choose from curated real-world feature addition tasks from the NoCode-bench Verified dataset or upload your own GitHub repository and specify natural-language instructions for modifying code. Behind the scenes, we use benchmark principles from NoCode-bench to guide how edits are validated and analyzed.

Define a Feature Addition Task
Specify a repository and a natural-language change request
Tests-driven
No-code orchestration

AGENT PIPELINE

1 Watch docs change
Detect diffs in README, API docs, or design specs.

2 Apply code edits
Generate patches across the repository automatically.

3 Run tests & score
Evaluate your agent on pass rate and regression safety.

NOCODE-BENCH · EVALUATION DASHBOARD

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.

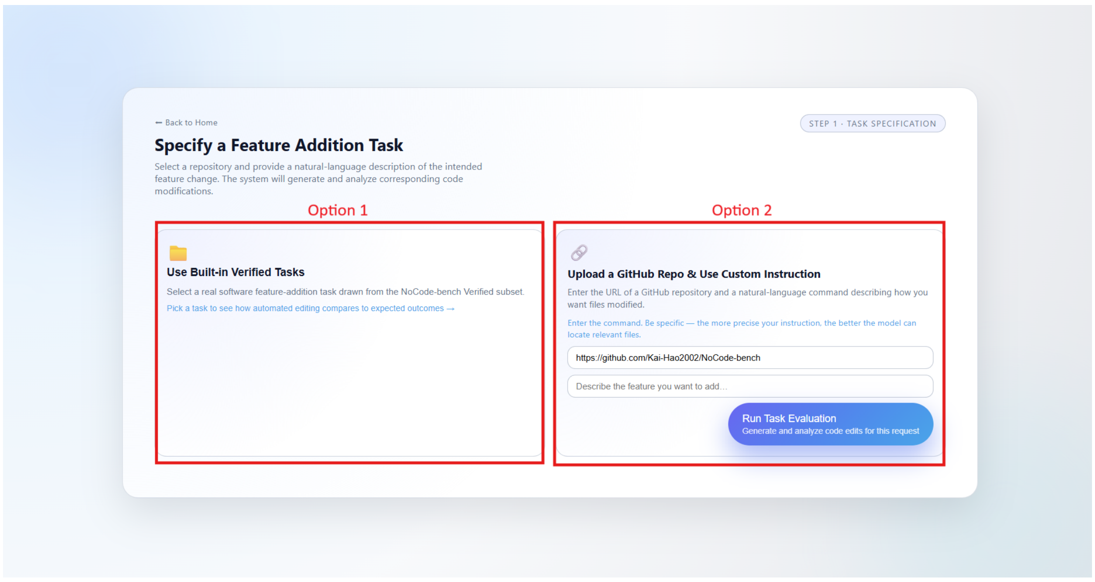
5. User roles and Available Features

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.



← Back to Home
STEP 1 · TASK SPECIFICATION
Option 1
Option 2
Run Task Evaluation
Generate and analyze code edits for this request

Specify a Feature Addition Task

Select a repository and provide a natural-language description of the intended feature change. The system will generate and analyze corresponding code modifications.

Use Built-in Verified Tasks

Select a real software feature-addition task drawn from the NoCode-bench Verified subset.

Pick a task to see how automated editing compares to expected outcomes →

Upload a GitHub Repo & Use Custom Instruction

Enter the URL of a GitHub repository and a natural-language command describing how you want files modified.

Enter the command. Be specific — the more precise your instruction, the better the model can locate relevant files.

https://github.com/Kai-Hao2002/NoCode-bench

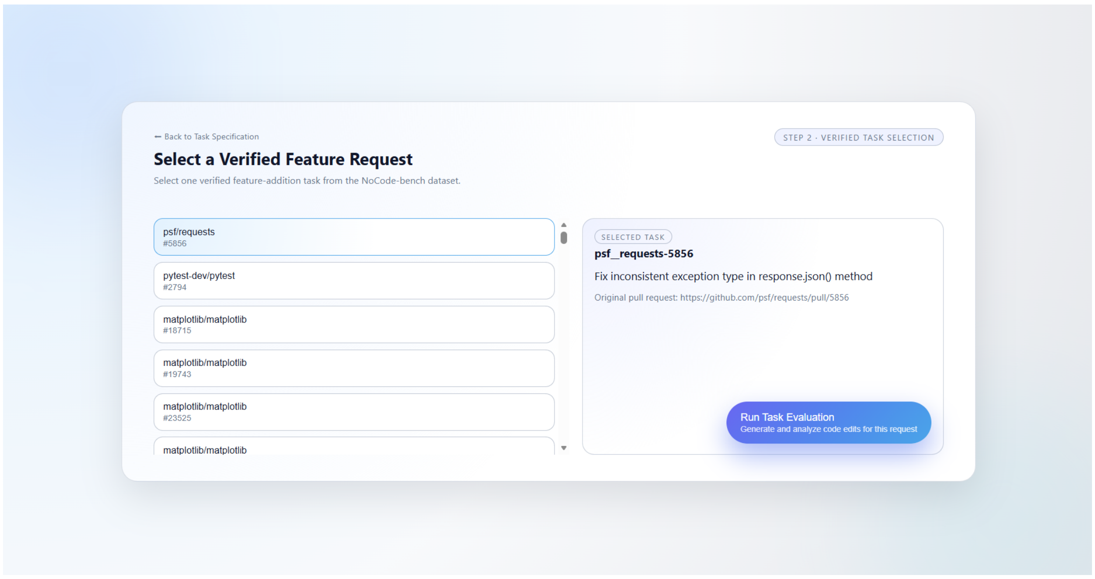
Describe the feature you want to add…

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.



← Back to Task Specification
STEP 2 · VERIFIED TASK SELECTION
SELECTED TASK
Run Task Evaluation
Generate and analyze code edits for this request

Select a Verified Feature Request

Select one verified feature-addition task from the NoCode-bench dataset.

psf/requests
#5856

pytest-dev/pytest
#2794

matplotlib/matplotlib
#18715

matplotlib/matplotlib
#19743

matplotlib/matplotlib
#23525

matplotlib/matplotlib

psf__requests-5856

Fix inconsistent exception type in response.json() method

Original pull request: https://github.com/psf/requests/pull/5856

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.

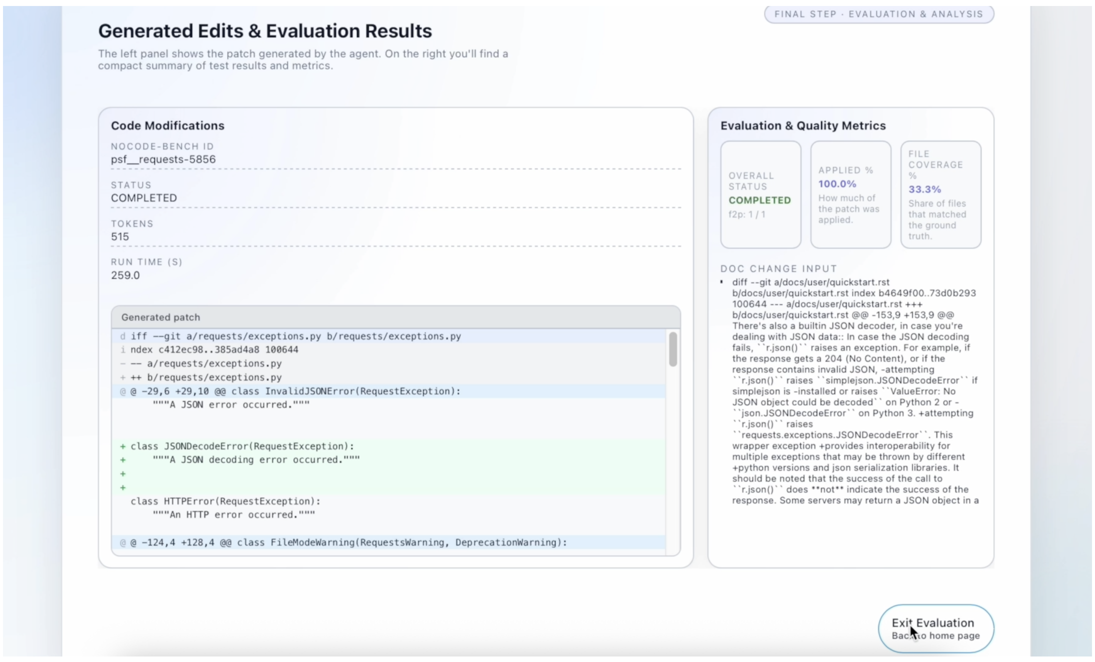
During execution, the system prepares the selected repository, generates code edits based on the documentation change, applies the patch, and runs the project's existing test suite. The interface provides visual feedback indicating the current execution status, and no user intervention is required.

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.



Generated Edits & Evaluation Results

The left panel shows the patch generated by the agent. On the right you'll find a compact summary of test results and metrics.

Code Modifications

NOCODE-BENCH ID
psf__requests-5856

STATUS
COMPLETED

TOKENS
515

RUN TIME (S)
259.0

Generated patch

```
d iff --git a/requests/exceptions.py b/requests/exceptions.py
i ndex c412ec98..385ad4a8 100644
- -- a/requests/exceptions.py
+ ++ b/requests/exceptions.py
@ @ -29,6 +29,10 @@ class InvalidJSONError(RequestException):
      """A JSON error occurred."""


+ class JSONDecodeError(RequestException):
+     """A JSON decoding error occurred."""
+
+
  class HTTPError(RequestException):
      """An HTTP error occurred."""

@ @ -124,4 +128,4 @@ class FileModeWarning(RequestsWarning, DeprecationWarning):
```

Evaluation & Quality Metrics

OVERALL STATUS
COMPLETED
f2p: 1 / 1

APPLIED %
100.0%
How much of the patch was applied.

FILE COVERAGE %
33.3%
Share of files that matched the ground truth.

DOC CHANGE INPUT

- diff --git a/docs/user/quickstart.rst b/docs/user/quickstart.rst index b4649f00..73d0b293 100644 --- a/docs/user/quickstart.rst +++ b/docs/user/quickstart.rst @@ -153,9 +153,9 @@ There's also a builtin JSON decoder, in case you're dealing with JSON data:: In case the JSON decoding fails, ``r.json()`` raises an exception. For example, if the response gets a 204 (No Content), or if the response contains invalid JSON, -attempting ``r.json()`` raises ``simplejson.JSONDecodeError`` if simplejson is -installed or raises ``ValueError: No JSON object could be decoded`` on Python 2 or -``json.JSONDecodeError`` on Python 3. +attempting ``r.json()`` raises ``requests.exceptions.JSONDecodeError``. This wrapper exception +provides interoperability for multiple exceptions that may be thrown by different +python versions and json serialization libraries. It should be noted that the success of the call to ``r.json()`` does **not** indicate the success of the response. Some servers may return a JSON object in a

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
“Completed” does not guarantee a correct feature	Completion indicates the run finished; the feature is only correct if the **test outcome** is fully passing. Always treat tests as the final criterion.
Patch cannot be applied (Applied% failure / apply error)	Some generated diffs may fail due to **context mismatch** (e.g., wrong surrounding lines or outdated assumptions). Try rerunning, or choose a different verified task / provide more precise instructions for custom tasks.
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.
Hallucinated functions or APIs	The generated code may reference non-existent functions/attributes (a known LLM failure mode). Use the patch viewer to spot these quickly and rerun with clearer constraints if needed.
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.
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 repos, keep instructions narrow and specific.
Code edits may be minimal or appear unchanged	For some requests, the best fix can be small (or the model may fail to identify the right location). Use **File Coverage** and the patch viewer to confirm what changed.