System Verification and Validation Plan for Nodify: The Graph-based Python Visualizer

Team 7

Full Name	Student Number	Email
Ahmet Erdem Dumlu	400227350	dumlua@mcmaster.ca
Anthony Hunt	400297564	hunta12@mcmaster.ca
Mutaz Helal	400323509	helala2@mcmaster.ca
Youcef Boumar	400366531	boumary@mcmaster.ca

April 4, 2025

1 Versions, Roles, Contributions

1.1 Revision History

Date	Version	Notes
February 2, 2025	1.0	Initial Version
April 4, 2025	2.0	Added Test Results

Project Description

Professional software developers use numerous tools to increase understanding and productivity when interacting with the details of a codebase. Debuggers, IDEs, syntax highlighting, and intellisense all provide great insights to program development at a low level. However, there is a distinct lack of simple, effective, and automatic visual aids for quickly grasping a high-level view of large programs.

Nodify is a codebase visualization tool that instantly generates interactable, up-to-date execution diagrams to better understand the various segments and services of any Python program. This project aims to simplify and streamline the process of understanding code through automating the creation of powerful visual aids.

The rest of this document will detail our plan to validate and verify Nodify's efficacy as a productivity-focused VSCode extension. Ensuring the correctness of components dealing with code analysis and automatic program summarization will be of particular interest. See the Design Document and SRS for more details on precise components and further project justification.

Component Test Plan

The below test plan enumerates all components from the design document. For convenience, a one-sentence summary of the component and its purpose will be provided.

Opened Python Project (VSCode Document Webview)

This frontend-facing component adds several useful buttons and interactions to the default VSCode text view.

Unit Tests

A clickable link above function/class definitions and callable objects will open the Nodify webview in a new VSCode tab. This component should send enough information to the VSCode

webview component that the displayed graph will center around a relevant location.
Semi-manual tests to ensure that clickable links only appear above desired code structures are necessary since VSCode does not provide an API for testing such interactions. Further unit tests regarding setting toggles will be necessary to ensure that links only appear above desired code structures. For example, users may want to disable clickable links for function definitions, but keep links for class definitions.

Performance Tests and Metrics

In VSCode, this component runs in the background to calculate where clickable links should appear before displaying all links on the document webview. Performance for this section is not vital to the operation of the project, but should still be reasonable (~1-2 seconds when first starting VSCode and ~0.5 seconds upon opening a Python file).

Python File Analysis

This component gathers all files and libraries required to analyze the project.

Unit Tests

When a user submits a request to analyze a Python project in VSCode, all files related to the currently open project must be findable. Our unit tests will primarily consist of a comparison of preconfigured folders, files, and import statements against a known list of expected file paths and imports. Integration tests will ensure that file paths are updated when files in the project folder change.

Performance Tests and Metrics

Although we do not anticipate any bottlenecks in searching for files, we will measure the wall time of all functions within this component. Ideally, something this small should take at most a few seconds, even for large Python projects with hundreds of files.

Abstract Syntax Tree (AST) Analysis

AST analysis converts plaintext Python files into a format more suitable for diagramming.

Unit Tests

The unit tests for this component will cover every piece of valid Python syntax valuable to Nodify. More specifically, we need to ensure that control structures (loops, if-statements, etc.), unique kinds of function definitions, class definitions, and expressions containing callable objects will produce a complete and correct AST. Irrelevant parts of Python expressions should be removed from the tree or completely ignored. The resulting AST structure should differ depending on the selected entry point. Further, we must catch invalid Python statements through use of VSCode's builtin Python Language Server extension. Unit tests on component functions through VSCode's builtin Mocha testing platform[1] and Vitest[2] will ensure we consider these kinds of inputs and return errors when necessary.

In order to increase performance of this component on large python projects, we will lazily evaluate files as they are needed. As such, we record references for each defined object in the analyzed Python file. We will include unit tests to ensure correct scoping and definedness of references for different file structures and definitions. It should be noted that we cannot reasonably catch every referenced value due to Python's sheer expressive flexibility in using objects through indirect means. For example, it is possible to override the setattr function, which can modify an object's attributes through use of strings. Or a developer may more commonly create function definitions by assigning a lambda expression to a variable.

Performance Tests and Metrics

The performance of this component primarily relies on an external library, AST-grep, and the size of the given Python project. To accommodate projects with hundreds of files, we will lazily evaluate files as needed, aiming for sub-second runtimes per file.

Code Partitioning and Summarization

Once our system creates a suitable AST from the given Python files, this component will make use of an LLM to summarize and partition segments of the code for the sake of detail abstraction.

Unit Tests

Since this component is primarily built on non-deterministic AI, the number of possible unit tests is limited. However, for the sake of performance, this component also caches completed code partitions. Therefore, we test the cache for consistency and speed (ie. duplicate chunks of code should not need to be re-analyzed by an LLM). Tests will be constructed from preconfigured Python files/modules.

We will include additional tests for establishing connections to LLM servers, especially for locally run setups. Any LLM output should also be well-formed and work with the JSON-format Graph Generator.

[1] https://code.visualstudio.com/api/working-with-extensions/testing-extension
[2] https://vitest.dev/

Performance Tests and Metrics

As part of the crowdsourcing techniques mentioned in the Design Document, we will ask users to evaluate the partition performance on legibility, logical consistency, and correctness. Due to the subjectivity of summarization, our best methods of evaluation will follow Human-Computer Interface methodology, namely, surveying for interpreted success on a scale of 1 to 5. Moreover, relying on LLMs to complete tasks with large amounts of information in this manner runs the risk of some hallucinations, so our team will perform several manual tests on the reasonability of the model. The manual tests include at minimum the following:

- Examining the effect of small changes in the code on resulting partitioned graphs
- Re-running this component with tweaked prompts on the same code
- Crowdsourced A-B testing, wherein users select the better of two different partition graphs

These tests will also be used in the model's fine-tuning process until the above surveys return satisfactory results (4/5 interpreted success rate or higher).

JSON-format Graph Generator

This component transforms a partitioned AST in JSON format into nodes for use in the graph webview.

Unit Tests

Preliminary tests involve assurance that common AST structures will result in proper JSON graphing objects. Similar to the AST analysis component, we must test for all possible control flow structures, function/class definitions, and callable objects. Additional tests will involve combining the output from multiple ASTs when objects are referenced in multiple files and importing/exporting generated graphs in JSON format.

Performance Tests and Metrics

The functions in this component should be small and direct mappings, so long as the code partition is well formed. We will still test the service time of this component, although we do not anticipate any performance issues.

VSCode Graph Webview/Website Webview

The primary view and interactions with Nodify will be from this component, containing the generated diagram on an infinite canvas.

Unit Tests

Unit tests for this component will be used to ensure that any type of data received from the backend can be transformed into expected React Flow components. Many of the tests regarding AST types (control flow, definitions, called objects) used in the JSON-format Graph Generator will be similar, translating JSON-formatted graph nodes into display components.

Integration and unit tests involving user interaction are as follows. Some of these may require semi-manual testing:

- When opening Nodify from a specific file or line of code, the graph should open in the corresponding spot.
- After a piece of code has been modified and saved, the graph should refresh with updated data. The performance of this hot reload process should be efficient and fast, ensuring use of cached data where appropriate.
- The graph should refresh with the currently opened file when the “refresh” button is clicked.
- The graph should be readable given a sufficient zoom level. Different project sizes and structures will need suitable initial layouts.
- User-modified layouts should be remembered until the code related to the nodes are changed.
- Well-formed JSON files should be imported and displayed correctly, without error.

Performance Tests and Metrics

Although the initial passing of code to the LLM partition component will likely take around 10 seconds for a medium-sized project, hot reload updates from small code modifications should be near-instant, even on extremely large projects. Graph interactions like scrolling, dragging, and zooming should be reasonably responsive. The usability of the UI will also be measured through conventional Human-Computer Interface means, like surveys, interviews, and user observation.

Test Results

Backend component unit tests can be found throughout the repository in *.test.ts files, as specified in the above test plan. Among these, parsing plaintext code into Nodify ASTs proved crucial to the correctness of translation functions and optimization of syntactic chunking for LLM input. Programming language constructs that affect the flow of execution, like function definitions and conditional statements, were of particular importance. Toggles for VSCode related features like codelensing, using different Python interpreters, and using different LLM models are also available in Nodify’s VSCode settings. Performance of all backend components combined (except for LLM calls) were under 2 seconds for even large source code files, thanks to lazy evaluation of object references.

Comparatively, since Nodify's underlying data structures were well-formed for direct graphical transformation, most UI-related tests were performed by inspection. Generally, this process involved sending well-known test files/repositories through Nodify and manually checking that the resulting graph was sensible. A few short code excerpts used for this form of testing are located in the test-workspace folder. Large repositories like mypy and the black formatter performed reasonably, with around 5 seconds to load for the first layer (main.py file), and around 30-60 seconds for the entire repository[3]. Further, the graphs for all inputs start entirely collapsed, so users will not be overwhelmed by a tangled mess of nodes. Instead, keyboard navigation expands desired paths intuitively and only when necessary.

The performance of Nodify largely depends on the LLM and underlying hardware used to calculate node partitions and summaries. Larger models running on industrial servers, like GPT-4o and Claude, naturally performed far better than smaller or locally-run models, both in terms of speed and semantic output quality. However, our custom, fine-tuned version of Microsoft's Phi model gave decent summarizations on consumer-grade hardware. With an RTX 4080 GPU, the fine tuned model took about 1-3 minutes on large projects like mypy, but performed fairly quickly (<30 seconds) on medium sized projects[4] and near-instantly on small scripts. Most of the output from all tested models with over 8 billion parameters gave accurate summaries. While we may not be able to control the quality of the LLM used within Nodify, we have successfully mitigated most of the drawbacks of summarizing large projects in low VRAM environments by way of chunking, caching, and lazy reference following.

[3] These results are taken with a depth limit of 5 layers, since most of the useful architectural information is contained within the first few layers of function calls. Broadly, a higher depth limit could take possibly exponentially longer to render, although a settings option is available to change this depth limit if desired.

[4] For example, an RNN project from one of our group members in a previous class: https://github.com/Ant13731/Chess-Prediction-Model