Project 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

October 25, 2024

Team Meeting Plan

Our team will meet at least once a week on Teams to discuss task progress and any open issues, following an agile methodology. In addition to regular scrum meeting content, collaborative documents (like the SRS, Project Plan, etc.) will also be written and reviewed during this time.

Our default meeting time will be on Mondays from 10 am - noon, but this time is flexible. Additional meetings may be scheduled as needed.

We will track progress, issues, and project milestones through Linear (a tool similar to Jira).

Team Communication Plan

Our team will use WhatsApp for text-based communication and Teams for meetings. A 24 hour response time is expected from all members, extenuating circumstances should be informed ahead of time.

Team Member Roles

Ahmet Erdem Dumlu: QA/Dev/Writer

- Frontend
    - UI Components: Canvas, Nodes, Edges
    - Graph customization (Settings, Storing/fetching placements)
- Telemetry Integration
    - Injection Program
    - Display Collected data

Anthony Hunt: QA/Dev/Writer/Product Manager

- Create execution flow abstractions for each lexical structure in Python
- Human-computer interaction
    - Frontend sketching
    - Define user interaction
    - Node placement of the graph
- Backend
    - Python AST scanning/parsing
    - Abstraction implementation
    - Code partition infrastructure
    - Import/Export JSON data with the graph
- Integration with VSCode

Mutaz Helal: QA/Dev/Writer,

- Frontend
    - Swapping abstraction levels
    - Developer interaction testing
    - User feedback
- Telemetry Integration
    - Injection program
    - Display collected data
    - Collect and organize data from injection program

Youcef Boumar: QA/Dev/Writer,

- Frontend Node Canvas
    - Node placement
    - Edge pathing
    - Data transformation
    - Graph controls
    - General graph user interaction QA
- LLM Management
    - Inference code
    - Prompt Engineer (writing the prompts for our different LLM use cases)
    - LLM testing
    - Base model selection
    - Fine-tuning code if needed

Workflow Plan

We list a set of guidelines to govern the workflow of our team:

Version Control and Task Management

- We will use a feature-branch workflow where each new feature or bugfix is developed in its own branch.
- The main branch will be the up-to-date working version of our application
- Once a feature or bugfix is completed, the developer will create a pull request (PR), which requires at least one approval from another team member before merging into the main branch.
- We will not be using Github Issues for task management, instead, we will use Linear to track all tasks and bugs. Each PR will be linked to the corresponding Linear ticket.
- Linear tickets will be tagged according to the type of issue. For example, tags may include “feature”, “bug”, “P0”, “P1”, etc.
- At a minimum, issues will have two main sections: Problem and Description. Other sections may be included as needed.

Agile Process

- We will have weekly scrum meetings to monitor the progress.

- We will have biweekly sprints (as discussed in the Team Meeting Plan).

Data Storage

- During development and in the local version of the application, we will use Postgres to store labeled data locally. In this case, the Python repository source files will also be on the local user's machine.
- For the hosted application (that collects telemetry data), we will use Neon - a serverless Postgres database. We will connect our backend with GitHub to enable analysis of repositories without the need for a local component. This will require read-only permissions from the repository.

Model Execution and Training

- Since we will be using LLMs, model execution will exist only through OpenAPI calls. We will test local model execution to see if it is a viable alternative.
- We will be creating a tailored LLM for the purpose of partitioning chunks of code using available OpenAPI tools.

User Interface Requirements

- We will utilize React libraries, specifically React Flow for graph generation. Usability will be measured by conducting interviews with developers to gather feedback.

Proof of Concept Demonstration Plan

One of the major factors we must consider when working through this project is the reliability and correctness of the LLM in terms of grouping together similar pieces of code. Without proper partitioning for level 1 abstractions, the graph risks becoming a tangled mess of nodes and code. Therefore, for the proof of concept, we will focus on creating the backend infrastructure for partitioning and organizing source code. Further, the backend will necessarily contain functionality of the AST parser/scanner to work with different lexical structures in Python. The overall operation of the PoC is as follows: inputs for this system will be any Python program/repository, and the output should be a JSON-formatted list of nodes along with correct partitioning. We will manually assess the generated JSON for semantic correctness and relevance to the summarized code on a small custom-made Python script.

Although creating the frontend graph will consume a great deal of time, we feel that the accuracy of the LLM is a far greater risk to the project's progress. At a minimum, we will present sketches/ideas for the frontend portion of Nodify. If time permits, we will create a skeleton webapp to present the node information gathered in the backend for Level 1 abstractions.

In the event that the LLM partitioning does not work, we will opt to use comment tagging as a means of manually identifying different sections of the code. While simpler to implement, this method requires much more manual effort on the user side, so we would much prefer an automatic partitioning of code via LLM.

Technology

Programming languages, frameworks, and libraries

- Typescript
    - React for frontend
    - NodeJS for backend
    - ReactFlow for creating nodes and graphs
    - Ast-grep for AST scanning
- OpenAI libraries

Linters

- Eslint (or biome)

Testing

- We will use Vitest, a fast Jest compatible unit testing, code coverage, and performance profiling framework for Typescript. Unit tests will be focused on AST parsing, abstraction level, and generated JSON file correctness. Additional tests may be added for importing/exporting data, automatic node placement/organization, and telemetry-related features. Since this project must be precise in its representation of code, unit tests will ensure that fundamental components of the project are correct.

Continuous Integration

- GitHub Actions will be used to run unit tests before pull requests can be merged

Project Scheduling

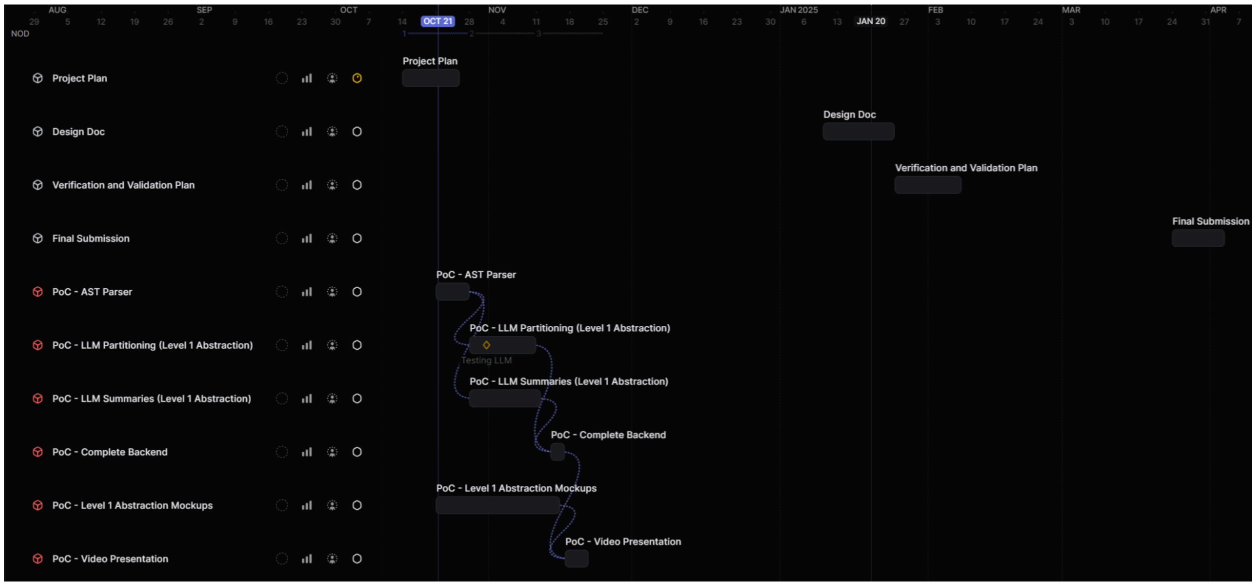

Figure 1: Gantt Chart for PoC and document submissions. All tasks are high priority.

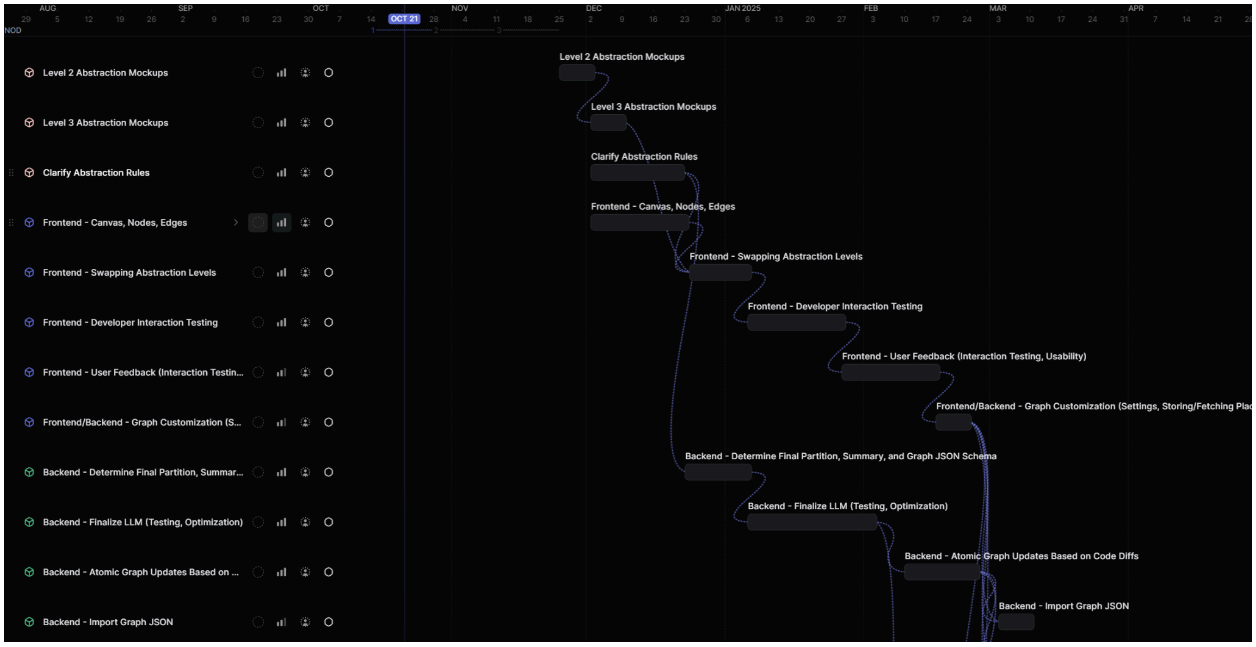

Figure 2: Gantt Chart for design, frontend, and beginning of backend tasks. Date ranges from the end of November to the beginning of March.

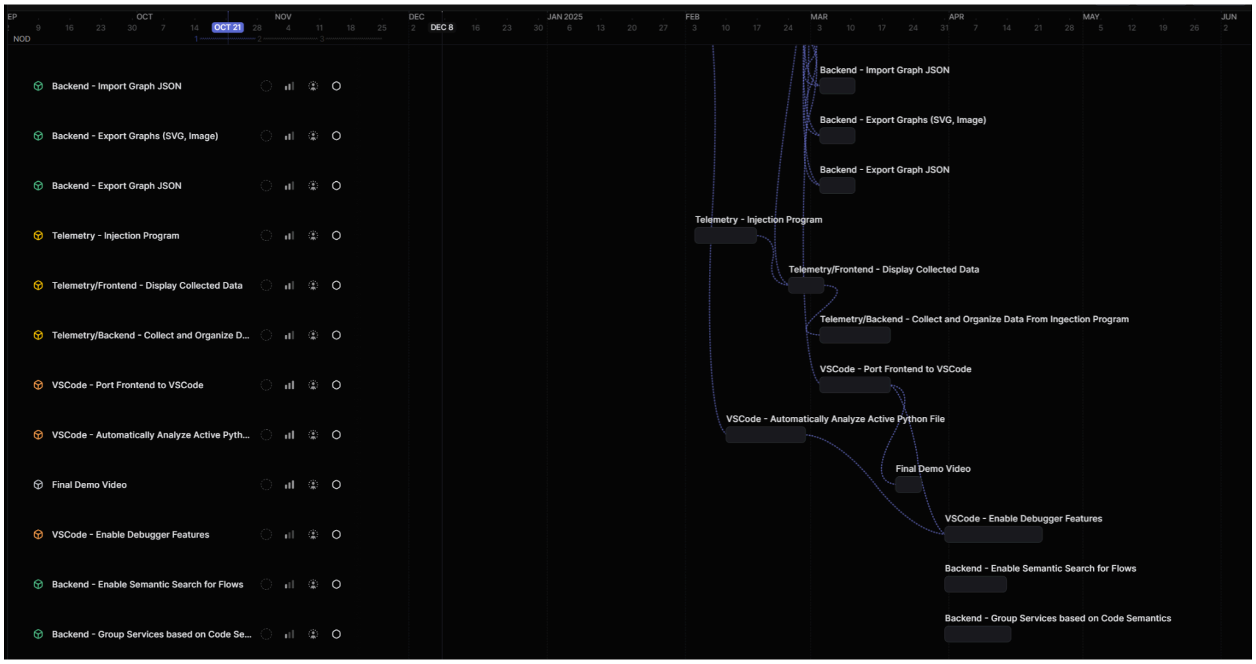

Figure 3: Gantt Chart for backend, telemetry, and VSCode integration. Date ranges from the end of February to the end of April. Low priority tasks are scheduled past the Final Demo Video date.