Software Requirements Specification for Nodify: The Graph-based Python Visualizer

Team 7

Full Name	**Student Number**	**Email**	**Roles (as needed)**
Ahmet Erdem Dumlu	400227350	dumlua@mcmaster.ca	QA/Dev/Writer Frontend, Telemetry Integration
Anthony Hunt	400297564	hunta12@mcmaster.ca	QA/Dev/Writer Product Manager, Backend (AST scanning/parsing), User interaction and testing
Mutaz Helal	400323509	helala2@mcmaster.ca	QA/Dev/Writer Frontend, Telemetry Integration
Youcef Boumar	400366531	boumary@mcmaster.ca	QA/Dev/Writer Fullstack, LLM organization

October 11, 2024

Table of Contents

1 Versions, Roles, Contributions

1.1 Revision History

Date	Version	Notes
October 11, 2024	1.0	Initial Version

1.2 Team Member Contributions

Group Member	Contributions
Ahmet Erdem Dumlu	1, 2.2, 2.4, 2.6, 2.7, 4.1, 7.4-7.6, 8.1, 8.2
Anthony Hunt	1, 2, 2.1 - 2.7, 3, 4, 5, 6, 7.1, 7.4-7.6, 8
Mutaz Helal	1, 3, 4.1, 7.1, 7.2, 7.3, 7.6
Youcef Boumar	1, 2.4 - 2.6, 4, 5, 6, 7

1.3 Glossary

Term	**Definition**
Abstraction	In the context of this project, abstractions will be used as a measurement of the amount of detail contained in a particular part of the graph. For example, plaintext code would be the lowest level of abstraction, while summarized code blocks would be a higher level of abstraction (see section 2.5 for more details).
AST	An Abstract Syntax Tree (AST) is the representation of the structure of a parsed program.
Code blocks	A code block is a lexical structure of code arranged in a group. (source)
Edge	An edge is a connection between the nodes on the graph. (source)
Graph	A graph is a set of lines that connect to a set of nodes. (source)
LLM	A Large Language Model (LLM) is a specific type of Machine Learning model that specializes in understanding and predicting human language text.
Apache 2.0	Permissive free software license created by Apache software foundation.

	(source)
Node	A node is part of a graph and is connected to other nodes with edges. It represents atomic units of code
React Flow	A React component that simplifies the creation of Node-based user interfaces. (source)
Service	A collection of code to serve a single purpose or general function. For example, a module that handles a specific piece of business logic.
Telemetry	The automated process of gathering data and metrics to be used for monitoring and analytics. (source)
VS Code	An open source text editor developed by Microsoft.

2 Purpose of the Project

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, we found that there is a distinct lack of automatic, simple, and effective visual aids for quickly grasping a high-level view of large programs. This capstone project will explore a method of automatically generating graphs to present the various segments and services of a program. We aim to simplify and streamline the process of understanding code through automating the creation of powerful visual aids.

The rest of this document will discuss requirements and high-level design decisions for a graph-based program visualizer targeted at code written in Python.

2.1 Background

Good documentation is crucial for effective interaction with large codebases, whether as a user, contributor, or maintainer. Without documentation, users would need to read the source code to figure out how to use the project, and maintainers would need to remember every detail of every component they work on. At some point in a software project's lifetime, maintaining good documentation becomes a task just as challenging and time-consuming as maintaining the project itself. Tools like API documentation generators help, but still rely on active contributions and following specific styles.

As with all complex collaborative systems, large open-source projects can suffer from an extreme lack of communication on the intended use of the project. Even well-maintained projects may have some niche components that are not clearly well defined in documentation but are absolutely vital to effective use of the library[1]. Further, documentation of a program's intended flow and usage can change rapidly following new requirements and key features, rendering hand-written documentation outdated. If the obsolete pieces of documentation accumulate,

[1] For example, Pydantic provides generally excellent user documentation documentation but leaves out details for the process of representing data internally. The Core Schema docs only state the available APIs, but not how those APIs are used.

unsuspecting contributors or users may even be worse-off than if there was no documentation in the first place.

Insufficient documentation directly leads to reduced productivity, especially for new developers. In cases like this, programmers must waste time searching through implementation details and reading the source code to get an idea of the purpose and usage of a module or function. Without the aid of concise and informative documentation, it is extremely difficult to jump into a project and fix the parts that require the most attention. Even with common IDE tools like tracking references to a function or running a debugger to see how data changes over time, contributors can never be certain that changes to one part of a project will not have unintended consequences for other portions of the codebase.

2.2 Current Situation

The current landscape of documentation in many open source projects consist of two main parts: documentation written specifically for users/contributors, and API documentation generated automatically from the source code, intended as reference for experienced contributors. The former form of documentation, while objectively the most useful, currently requires a large team of people and is a process unlikely to be automated anytime soon, even with current LLMs. However, reference-type documentation can and should closely reflect the present state of the codebase, ideally without any additional effort from developers. Pydantic is one such open-source Python library that clearly separates its user-facing documentation from automatically generated API docs[2].

For Python in particular, the most popular documentation generation method is through a tool called Sphinx. Although Sphinx is a powerful tool with many community-contributed extensions, themes, formats, and guides, it generates documentation by loading program files into a Python instance, importing every module, and examining the loaded Python objects. Similar to a built-in IDE debugger, this approach to generating documentation is not practical in many real-world settings, where code imports only properly run on a remote server or a “fake” production setting. Even if Sphinx does execute as expected in such settings, updated documentation can be cumbersome for large projects and will not be freely available in a contributor’s local development environment.

2.3 Objectives

The main objective of this project is to provide dynamically updated visual documentation of large codebases.Supporting objectives of this project are as follows:

- Automate the generation of code diagrams for consistently up-to-date documentation
- Provide a tool to clearly communicate code execution flow
- Within the generated graph, allow for high level view of complex programs, with good abstractions where necessary
- Ability to interact with the flow of the program and delve into the details when needed
- Statically analyze project code structure without needing to execute it

All of these tasks should be done with minimal interruption to the current workflow of programmers, and should serve as an additional non-invasive tool to better understand and document code.

[2] User docs, API Reference docs.

2.4 Expected Benefits

By creating a project that can instantly generate interactive up-to-date flowchart documentation from any Python codebase, we hope to effectively eliminate the drawbacks of manually creating visual aids for communicating programs. Our tool will ensure that any execution-flow based documentation will never be left unmaintained or missing, allowing contributors and users alike to instantly grasp the high-level functionality of any program. Additionally, in approaching this problem with a non-invasive stance, we enable our project to run anywhere, anytime. In other words, even if the creators of a software project do not provide good documentation, users can still generate flow charts and easily conceptualize different components of a program. As a result, we expect much less frustration and wasted time when working with foreign (or even familiar) projects that have minimal documentation.

2.5 Functionality Overview

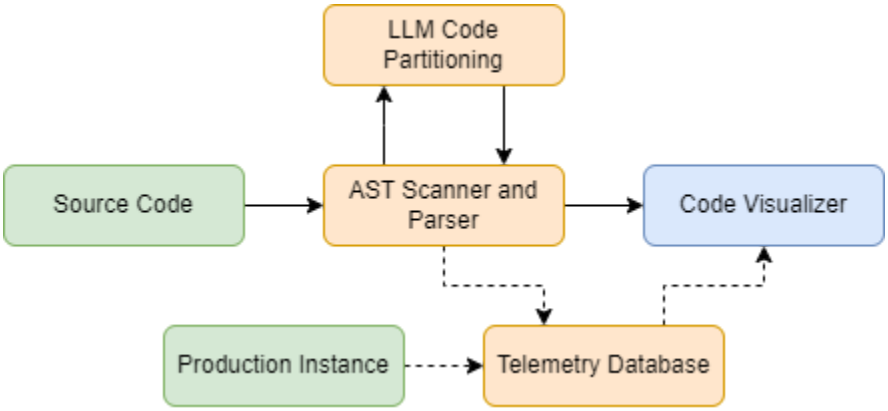

Figure 1: Flow of services available through our project. The telemetry component is optional and thus connected with dotted lines. Green nodes represent inputs or actions from the user, orange nodes will be the core segments of the project, and the blue node is the output given back to the user.

Figure 1 shows the major components of our project. To generate graphs as a form of documentation, we will need to statically analyze the source code and associated AST. As we organize this data, we will send requests to an LLM to partition and summarize large chunks of code for easier interpretation at the code visualization stage. Additionally, we plan to offer an opt-in API to fetch telemetry data in a real-world production instance of executing code. If we receive such data, we can present running time, function usage, and even execution flow statistics to users.

2.5.1 Abstraction Levels

One foreseeable issue with visualizing massive codebases is the potential for extremely messy and confusing graphs. Since the goal of Nodify is to provide a tool for better understanding and communication of code, having a too-messy graph would be extremely detrimental. Therefore, to prevent an overgrowth of useless nodes, Nodify will make use of 4 levels of abstraction. Each component of a program will have different levels of abstraction corresponding to different levels of detail, described in the table below.

Level	Description
Level 0 - Line by line Flow	At this level, users can visualize the flow of the program as it goes through each line of code. Nodes on the graph will still contain the original source code. Statements within a function/program will appear as content within a larger node. The larger nodes usually encompass a function or method. Sections of statements may be separated/partitioned into blocks. On hover, these sections will display a short summary of the code.
Level 1 - Block Flow	Users will still see a fine-grain execution flow, but instead of lines of code, blocks of code (within functions) are summarized by a specialized LLM model. Details of specific function calls and expressions are abstracted away. Function definitions become containers for their contents, and nodes within these containers are the names of partitions previously seen at Level 0. Function container nodes may be minimized and appear as a smaller code-block-level node.
Level 2 - Service Big Picture Flow	At this level, more implementation details are abstracted, giving users a high level view of the business logic within a particular service, the interfaces it provides, and any expected outputs. By default, each node will be created from a module, and modules can be grouped together to form higher-level flow hierarchies. Nested modules will be represented via nested nodes, similar to Level 1's representation of code blocks within a function.
Level 3 - Inter-Service Flow	At this level, each node is a single service with a high level description. This level of abstraction does not capture the execution flow of lower abstractions, rather edges represent inter-service dependencies, with informative but concise labeling.

It should be noted that different parts of a programming language will be presented differently at each level. For example, an if-statement will have its own structure and a while loop will have a different structure at each abstraction level. Thus, each piece of programming language syntax will likely require its own unique shape. This analysis will be discussed further in the Functional Requirements section.

src/main.py | Level 0

```
delay = 0.1

# Score
score = 0
high_score = 0

# Create the Window
wn = turtle.Screen()
wn.title("Snake Game by @TokyoEdTech")
wn.bgcolor("green")
wn.setup(width=600, height=600)
wn.tracer(0)  # Turns off the screen updates

# Draw the snake's head at 0,0
head = turtle.Turtle()
head.speed(0)
head.shape("square")
head.color("black")
head.penup()
head.goto(0, 0)
head.direction = "stop"

# Draw the food at 0, 100
food = turtle.Turtle()
food.speed(0)
food.shape("circle")
food.color("red")
food.penup()
food.goto(0, 100)

segments = []

# Pen
pen = turtle.Turtle()
pen.speed(0)
pen.shape("square")
pen.color("white")
pen.penup()
pen.hideturtle()
pen.goto(0, 260)
pen.write(
    "Score: 0  High Score: 0",
    font=("Courier", 24, "normal"),
)

# Initialize keyboard bindings
wn.listen()
wn.onkeypress(go_up, "w")
wn.onkeypress(go_down, "s")
wn.onkeypress(go_left, "a")
wn.onkeypress(go_right, "d")

# Main game loop
while True:
    wn.update()

    # Check for a collision with the border
    if (
        head.xcor() > 290
        or head.xcor() < -290
        or head.ycor() > 290
        or head.ycor() < -290
    ):
        time.sleep(1)
        head.goto(0, 0)
        head.direction = "stop"

        # Hide the segments
        for segment in segments:
            segment.goto(1000, 1000)

        # Clear the segments list
        segments.clear()

        # Reset the score
        score = 0

        # Reset the delay
        delay = 0.1

        pen.clear()
```

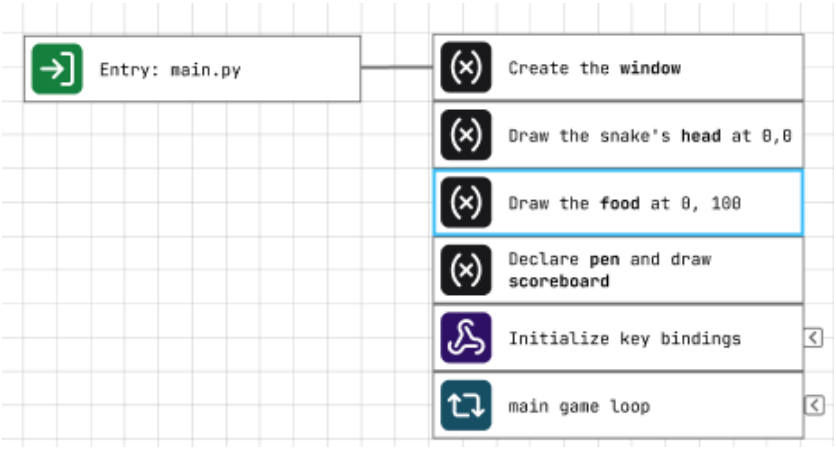

Figure 3: Level 1 abstraction graph for code provided in Figure 2. When the mouse hovers over each section, the corresponding code will be highlighted in the plain-text (level 0) version.

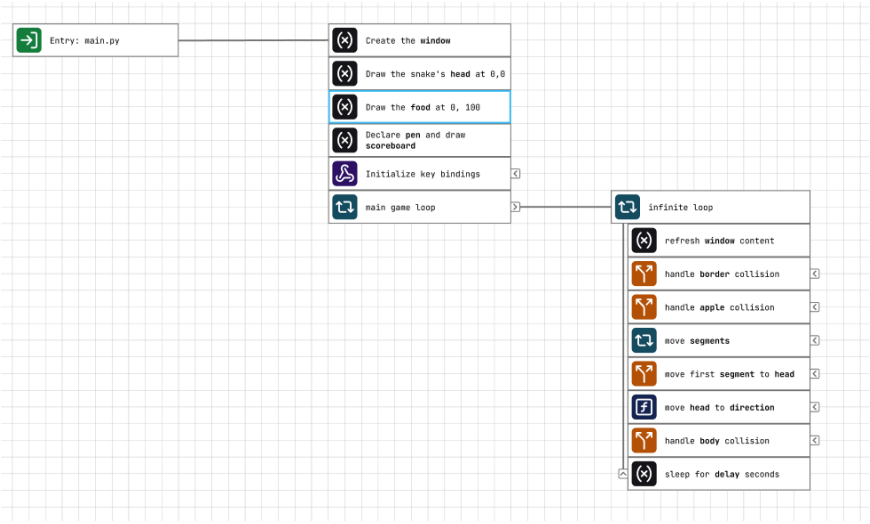

Figure 4: Level 1 abstraction with expanded main game loop code block. The large code block is separated from the rest of the function for clarity and easier visualization.

Figure 2: Plaintext code corresponding to the graph in Figures 3 and 4.

2.6 High-Level Usage

Users will generally interact with Nodify in two ways: as a tool for documentation and static local analysis, or as a production mode runtime analyzer with telemetry data.

2.6.1 Local Editor

A majority of the time, developers will use Nodify to track and document developments and modifications in their code. As such, a VSCode extension allows for seamless integration with normal development workflows, described below.

1. Install our VSCode extension.
2. Open a Python project and run the graph initialization command.
3. Local config and state files are created on the system to store the state of and any manual changes added to the graph.
    a. For large existing projects, this step could take a long time, as chunks of code have to be processed by an LLM.
4. Run the view graph VSCode command to bring up the big picture graph that shows you the whole codebase.
5. Double click on a node to focus on it, expanding it to reveal more details, and hiding other irrelevant nodes.
6. Above every function and class in the codebase, is small gray clickable text that allows you to inspect the part of the graph relevant to that function, on a level 1 abstraction by default.
7. On save, changes are detected, and the graph is modified based on the change diff. A small model is used to update the summaries for fast and cheap updates.

2.6.2 Production Mode

We will provide an API for developers to send telemetry data to Nodify from an executing production instance of their codebase. The generated graph will then appear in a web browser with all local editing features and additional usage statistics.

1. Login to our website.
2. Link a GitHub repository.
    a. Config and state files are required to view a project on the web view, these are generated locally as described in part 2.6.1. Since there is no “local” instance in production settings, we will create these files on our own server.
3. Users will be provided with an API endpoint to enable sending of telemetry data to our backend. Must be compatible with OpenTelemetry.

2.7 Limitations & Exclusions

Nodify is geared towards generating visual aids for Python projects with a clear singular flow of execution. While Nodify may work to some extent with library functions or examples within a standalone library, its primary focus is on projects with a clear start and end. From the team’s perspective, choosing to target scripts with a clear execution flow will allow for much more valuable documentation and clearer visualization of a codebase.

This project will focus only on the Python programming language because of its extensive tooling support and the availability of clear AST documentation and libraries. By limiting our scope to Python, we aim to keep the project feasible and provide a practical solution. Additionally, using Python allows for simple opt-in integration of telemetry data collection via “monkey-patching” existing functions, a feature only readily available in Python and some other interpreted languages.

3 The Client & Stakeholders

- Client/Primary Stakeholder: Software Developers
    - Contributors of medium to large scale Python projects, especially those mentoring or onboarding other developers
- Other Stakeholders:
    - Educators and Learners: Assist those teaching and learning programming through visualizing code structure and performance
    - Devops: Used to monitor real-time performance of a project in production with telemetry data
    - Software Architects: To ensure efficient function interactions and spot bottlenecks or issues across codebase
    - Quality Engineers: Performance and function flows for debugging and optimizing code and performing comprehensive unit testing

4 Project Constraints

4.1 Relevant Facts and Assumptions

Assumptions:

- Users should have a basic level of programming experience and familiarity with Python
- Users should be able to navigate around IDEs like VSCode and git repositories

Facts:

- Documentation, especially for large codebases, easily becomes outdated as code changes over time, which can slow down further development and onboarding processes.
- Onboarding new developers can be resource-intensive especially for big companies with complex projects. This process often requires additional resources like use of licensed softwares and the time of other team members to provide knowledge transfer and support.
- Local environments may not be able to execute a Python project, so any local visualizations must be statically generated from the source code and associated AST.

4.2 Constraints

- As this project is used for credit in a year-long course with deadlines throughout the year, we will have a limited amount of time for prototyping and completing the final product.
- Efficiency: Nodify must have minimal impact on the performance of the code editing experience. As in, the IDE or text editor should not freeze and stutter with code changes.
- Our visualizer will only function on Python-based repositories since Python is a flexible, simple, and popular language. There is a significant amount of available tooling to aid in the scanning and parsing of Python ASTs.
- While Python is a relatively simple language, its flexibility allows for functionality that cannot be detected through static analysis. For example, methods on an object may be called just by the string name through getattr. Thus, Nodify's static analysis will only work on basic-to-somewhat-advanced Python syntax.

- Nodify will be available both in VSCode for local development and through a web interface when collecting production telemetry data.
- Nodify will primarily be written in Typescript since both websites and VSCode use Javascript interfaces.
- Nodify must be able to statically analyze and document code without executing it (in case code only executes fully in a “production” setting and not in a local IDE).
- For practicality and usefulness in understanding a codebase, Nodify should be able to run in an IDE, updating visualizations of the codebase in real-time
- Nodify should work on all Python codebases out-of-the-box, without any extraneous modifications to the source code. This means we must take a non-invasive approach to generating the visualization.

5 Functional Requirements

P0 (Minimal Viable Product)

- Nodify should be able to parse any Python project and create an accurate and representative graph data structure. This will be done through static analysis of source code files.
- Nodify should provide a canvas which visualizes the generated graph data structure with reasonable default placements.
- Abstractions should depend on well-defined rules for how to visualize all possible Python constructs (conditions, loops, function calls, try..except, etc). These concepts will be defined for all levels of abstractions (see 2.5.1 for the 4 levels of abstraction)
- Visual changes to the graph, such as moving nodes around, should be saved to a plain text file, which would enable storing and loading the graph through git commits.
- Ability to modify partitions and create custom partitions, triggering a new summary for different segments of the code.
- Intuitive interactions with the graph to grant the user more/less detail when needed, primarily, the ability to collapse and expand nodes.
- The fine-tuned LLM should be able to create code partitions with a human approval rating of over 0.90. The partitions will be used to provide concise summaries of code blocks within the graph visualization.
- The fine-tuned LLM should be able to provide reasonable labels to aforementioned partitions with a human approval rating of over 0.90.

P1 (High priority features)

- A fine-tuned LLM should be able to summarize code partitions, labels, and their relationships into reasonable business logic services. This will serve as the foundation for Level 3 abstractions (Inter-Service Flow).
- A smaller diff model should be trained as follows: it will accept previously generated partitions, summaries, and some code changes, and output updated summaries and partitions. Without this smaller model, we would need to rerun the original model on all code related to a diff. This requirement is to maintain responsiveness and functionality of the graph.
- Ability to import/export custom expanded flows (views of the graph) to a JSON file.

- Implementation of the webview canvas as a VSCode extension, for better local documentation generation.

P2 (Medium priority features)

- Addition of telemetry-gathering features for production instances. This involves creating an API for users to send usage data through and then displaying received data as usage statistics.
- Create text embeddings for all partitions, and partition summaries based on the meaning of the text. This will be used to support the following two requirements.
- The ability to use semantic search across the codebase to find code and flows that might be relevant to a specific question the user has.

P3 (Nice to have features)

- Automatically group services based on the semantic similarity of the code.
- Ability to export graph as an SVG or other format to work with existing documentation applications/formats (eg. Confluence, markdown).
- Implementation of Nodify as part of a debugger for use within the local IDE. This would combine the benefits of a tool like PythonTutor with high level abstractions.

6 Data and Metrics

- Nodify will primarily require synthetic LLM training data for automatic segmentation and summarization of code blocks. Summaries can be nested for subsequent levels of abstraction. The following is an example of how we will organize and use the data:
    a. First, we will manually segment and label 3 examples.
    b. In addition to a detailed prompt about how to segment and label chunks, these examples will be given to an LLM in context. We will then run inference 5 times on a training set of 200 code chunks, resulting in 1000 labeled examples.
    c. We will manually go through these results and select 1 or 2 good outputs for each of the code chunks. This will give us a training set of about 300 examples.
    d. We will use the OpenAI fine tuning API to fine tune the LLM on these examples.
- In addition to synthetic training data, we will use real-world open-source Python projects to test and train on. We need a large selection of open source Python projects to create both training and test sets for the visualizer as a whole, code segmentation, and labeling.
- The same Python projects will be used to test our graph generation capabilities and to ensure we cover all constructs of Python. We will also use these repositories to determine the most useful interactions for the visualizer.

7 Non-Functional Requirements

7.1 Performance and Speed Requirements

The graph generation performance and speed requirements are as follows:

- Graphs should be completely accurate to the source code, otherwise it will not be useful as a form of documentation.

- Graph generation should be efficient enough to run on a consumer-grade CPU, without interrupting the workflow of developers. That is, graph generation should not cause lag to any other operation within the IDE.
- Adding changes to the graph should be efficient. In other words, we should not regenerate the graph from scratch every time a small change is made. Instead, small changes in the codebase should reflect small modifications in an existing graph. Small graph modifications should take less than 5 seconds.

The LLM performance and speed requirements are as follows:

- LLM segmentation and partitioning summaries will be accurate to the semantic meaning of the source code with a 90% approval rating.
- LLM summary generation will be asynchronous with graph generation to prevent slowdowns caused by idly waiting for a response. In general, the LLM is expected to respond within 10 seconds.

7.2 Operational and Environmental Requirements

In order to reduce costs when sending requests to an LLM, small changes in a previously-generated graph should only require a small amount of new request tokens.

Users of this project will require at minimum a consumer-grade computer and a Python codebase to visualize. We further assume that devices using the LLM and telemetry features will be connected to the internet. In order to use IDE extension features, users will need to work inside of VSCode.

7.3 Security and Privacy Requirements

Any (confidential) code parsed by our AST scanner will require a secure channel, especially when using the LLM and telemetry features. Any collected telemetry data will be stored on our own servers and freely available for deletion by users.

7.4 Look and Feel requirements

- The nodes should be placed logically to create clear visuals. Also, they should avoid overlap for simple and concise navigation.
- The tool should have a responsive design so it adapts to different screen sizes.
- The interface should be intuitive to navigate so that the users can explore the software without a high learning curve.
- The design must accommodate users with colorblindness. This can be achieved through a high-contrast theme or other symbols to represent code partitions.

7.5 Usability and Humanity Requirements

The graph visualization should not overwhelm first-time users or new developers but should be powerful enough for experienced users or programming experts to navigate effectively and efficiently. The graph should be concise yet informative enough for users to quickly grasp the flow of a program at a glance. Since the tool is intended to serve as a form of interactive documentation, new contributors should especially gain value from the graph as a form of guidance to the program's execution. Experienced users should be able to use the graph as a form of visual reference when modifying a component and examining its impact.

7.6 Legal Requirements

Our software will use the Apache 2.0 License. When using open-source projects as training data, we must be careful of the licenses used by those projects. Local static analysis of codebases will not result in the collection of data, but the telemetry service will require some knowledge of the information being executed.

8 Risks & Predicted Issues

8.1 Risks

- Since the code will be partitioned and summarized by an LLM, there is a risk of inaccurate summary generation, potentially leading to hallucinations, misleading visuals, or otherwise unintended representations of the code.
- Generating these graphs could require a lot of computing power, especially for larger projects, which might slow down the development process and impact productivity.
- Developers may not find the graphs intuitive or helpful if the output is too complex or too simple. Interacting with the graph-based code visualizations must be useful and simple for developers, otherwise they would not use the tool.
- The abstraction algorithm must accommodate all aspects of the programming language. Our algorithm must be flexible and robust enough to work through different components of a language's AST. Although the general structure of Python's AST has not changed much in recent years, major version updates could modify unknown components

8.2 Predicted Issues

- Complex graphs may be overwhelming to users, reducing the effectiveness of communicating the flow of execution. We address this concern through abstraction levels (see section 2.5)
- It is difficult to guarantee good LLM partitioning and summarization for complex codebases. If the selected segments are not appropriate or concise enough for the project, the resulting graph may become overly complicated and hard to understand, which would defeat the purpose of simplifying the code. Similarly, our abstraction algorithm must accommodate both complex and simple codebases for effective functionality.
- The LLM might misinterpret or incorrectly divide parts of the code, particularly in highly modular or dynamically typed projects, resulting in graphs that don't accurately reflect the system's architecture. This may be a concern especially for uncommented/disorganized code. If the LLM does not perform well enough for the task, we may opt to use a comment tagging system similar to Swagger.
- Another potential problem is the tool's lack of flexibility in customizing the abstraction. We will address this concern through well-tested interactions with our stakeholders.
- Our program may not be able to link wildcard imports properly (of the form `from module import *`) since we would not be able to match functions to their originating module. Since this practice is discouraged according to PEP-8 style, we will treat functions used this way as plain statements without any of the specialized features discussed above.