Design Document 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

January 19, 2025

1 Versions, Roles, Contributions

1.1 Revision History

Date	Version	Notes
January 19, 2025	1.0	Initial Version

2 Purpose Statement

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 the overarching design of Nodify and its various components.

3 Component Diagram

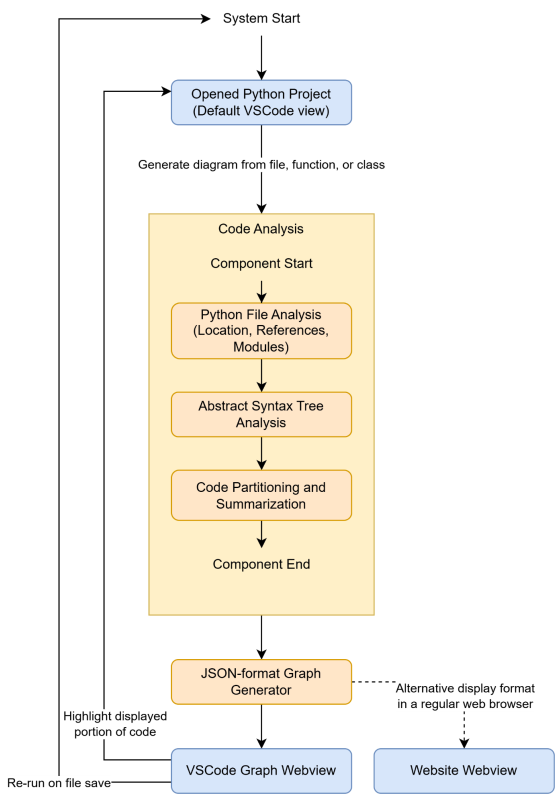

System Start
Opened Python Project (Default VSCode view)
Generate diagram from file, function, or class
Code Analysis
Component Start
Python File Analysis (Location, References, Modules)
Abstract Syntax Tree Analysis
Code Partitioning and Summarization
Component End
JSON-format Graph Generator
Alternative display format in a regular web browser
Highlight displayed portion of code
VSCode Graph Webview
Website Webview
Re-run on file save

Figure 1: A high level overview of Nodify components. Blue components represent interactive user-facing services and orange components provide insight on the backend structure. Further details on the Code Partitioning component are shown below.

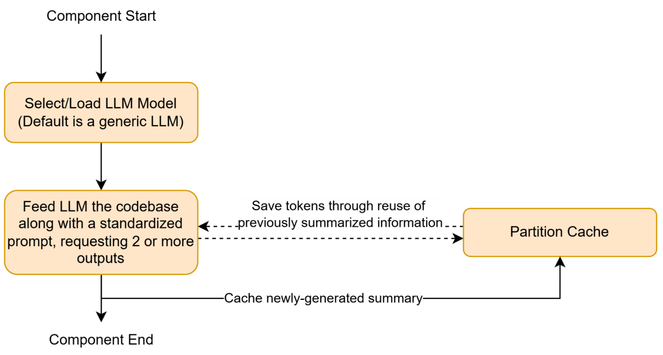

Component Start
Select/Load LLM Model (Default is a generic LLM)
Feed LLM the codebase along with a standardized prompt, requesting 2 or more outputs
Save tokens through reuse of previously summarized information
Partition Cache
Cache newly-generated summary
Component End

Figure 2: Code Partitioning in a production environment will make use of a cache to reduce the amount of tokens and processing required by the LLM. A similar setup may be used for the training environment if required.

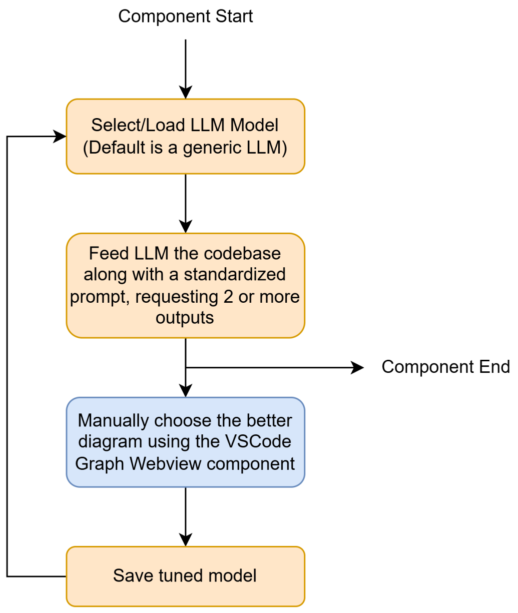

Component Start
Select/Load LLM Model (Default is a generic LLM)
Feed LLM the codebase along with a standardized prompt, requesting 2 or more outputs
Component End
Manually choose the better diagram using the VSCode Graph Webview component
Save tuned model

Figure 3: Detailed view of the Code Partitioning component in training mode. This structure enables powerful LLM refinements through crowdsourcing. Since the assessment of "accuracy" in the context of code summarization is highly subjective without concrete ground truths to compare against, the blue component contains an optional survey to gather a rough estimate on the satisfaction (accuracy) of the model's performance.[1]

4 Relationship Between Components and Requirements

System Component	**Requirements Covered**
Opened Python Project (Default VSCode Webview)	P1: Implementation as a VSCode extension Non-functional requirements: - Prevent overwhelming of new users - Provide a powerful tool for experienced developers and beginners
Code Analysis - Python File	P0: Parse any Python project, creating an accurate diagram

[1] Note: The manual selection feature is not available in the final version of Nodify. Instead, we have collected training data for a fine-tuned model manually.

Analysis	representation P1: Summarize code relationships
Code Analysis - Abstract Syntax Tree Analysis	P0: - Parse any Python project, creating an accurate diagram representation - Abstractions for all possible Python constructs
Code Analysis - Code Partitioning and Summarization	P0: Create and modify partitions P1: Summarize code relationships P2: Semantic search across the codebase for features, flows, and components
Code Analysis - Code Partitioning and Summarization (Training Mode)	P0: Generate partitions and labels with high satisfaction/approval rating (goal is 90%)
Code Analysis - Code Partitioning and Summarization (Production Mode) - Partition Cache	P1: Only analyze newly modified code Non-functional requirements: - Fast diagram generation to prevent developer workflow interruptions - Efficiency in graph modifications, for both speed and token costs
JSON-format Graph Generator	P0: - Enable loading and storing of the diagram - Create and modify partitions P1: Import/export diagrams in JSON format
VSCode Graph Webview	P0: - Canvas to visualize generated diagram - Intuitive interactions P1: Implementation as a VSCode extension P3: Export diagram as an SVG for permanent documentation Non-functional requirements: - Logical node placement - Prevent overwhelming of new users - Provide a powerful tool for experienced developers and beginners
Website Webview	In addition to the VSCode Graph Webview, the Website Webview satisfies: P2: Telemetry gathering features for production instances

5 Components

Opened Python Project (Default VSCode Webview)	
Normal Behaviour	Users will open their Python project in text format as normal. Nodify options will be held in a hidden menu/sidebar to reduce distractions from normal development as much as possible. The primary purpose of this component is to watch for hotkey/shortcut presses from the user to activate Nodify's graph generation capabilities. Once a graph is generated, this component will handle frontend text highlighting as described in Section 6.1.
API	Once the user decides to generate the graph, a single message will be passed to our extension (backend) through VSCode. If the user activates Nodify with a specific function/class/module, that function and associated location information will be attached to the message and passed to subsequent components.
Implementation	VSCode allows extensions to create listener callback functions for specific events[2]. A custom `nodify.openWebview` event will start the graph generation process and display the visualization using the Graph Webview component.
Potential Undesired Behaviour	As our extension focuses on enhancing productivity within a normal VSCode workflow, we must be careful to ensure minimal distractions. For minimal invasiveness, all interactions will be completed through the use of hotkeys, optional commands through VSCode's command palette[3] or side panel[4].

Python File Analysis	
Normal Behaviour	This component receives the name and location of the graph focus point from the Opened Python Project component. It then fetches all project Python files, references, and locations necessary to generate the graph.
API	Accepts a starting location and the root directory of the python project. Returns a list of required modules, files, and file locations.
Implementation	VSCode's Python Language Server extension provides an API to analyze all imports and uses of classes/functions. It also supports references to external libraries, even through a virtual environment. This API is sufficient enough to gather the information needed for detailed Abstract Syntax Tree Analysis.
Potential Undesired Behaviour	Nodify will not be able to handle dynamic imports through Python's `importlib`[5]; only static imports using `import * from _` syntax will be allowed. We also make use of careful error handling in case a file does not exist.

[2] https://code.visualstudio.com/api/references/activation-events
[3] https://code.visualstudio.com/api/ux-guidelines/command-palette
[4] https://code.visualstudio.com/api/ux-guidelines/activity-bar
[5] https://docs.python.org/3/library/importlib.html#importlib.import_module

Abstract Syntax Tree Analysis	
Normal Behaviour	Parses plaintext Python files into an Abstract Syntax Tree (AST) suitable for graphing, keeping track of references, redefinitions, and docstrings.
API	This component analyzes all files provided by the File Analysis and returns a modified version of the AST suitable for code partitioning and graphical display. See Section 7.1 for Nodify's version of the AST schema, focusing on modules, functions, classes, control flow statements, and expressions. Each part of the parsed AST is then transformed to a generic node data structure in preparation for partitioning and graph generation (see Section 7.2).
Implementation	We whittle down the extensive information given by AST-Grep, a JavaScript library aimed at parsing several unique programming languages into general AST constructs. It further provides many tools to search through and modify the provided AST as needed.
Potential Undesired Behaviour	Unexpected expressions in the AST or versions of Python may cause this component to fail. In general, we attempt to anticipate every possible combination of valid Python syntax based on heuristics and the available grammar[6] (see Section 7.3 for our current list of cases).

Code Partitioning and Summarization	
Normal Behaviour	The reduced AST from previous components is passed to an LLM for partitioning and summarization. A cache of previous summarizations is used to reduce token usage and increase efficiency, so only newly changed code is analyzed. This component fills in all leftover information needed to generate a cohesive flow diagram.
API	Given our Nodify AST without partition information and summaries, this component returns the same AST with each node assigned a particular partition/label and summary.
Implement ation	OpenAI APIs are used to communicate with cloud-based LLMs. Our group is currently exploring the use of local LLMs for computers with capable GPUs. Section 7.4 contains our existing prompt preamble.
Potential Undesired Behaviour	Hallucinations and inaccurate partitioning could massively decrease the usefulness of Nodify. To counteract known LLM weaknesses, we utilize crowdsourcing techniques to assess and increase (subjective) labelling accuracy. Performance of the LLM should be addressed by use of the Partition Cache

[6] https://docs.python.org/3/reference/grammar.html

	subcomponent, as shown in Figure 2.

JSON-format Graph Generator	
Normal Behaviour	Prepares the AST for input to the graph webview. Removes all unnecessary information and reorganizes partitions into a more suitable flow data structure. This component also serves as the data host for the Graph Webview component and allows for JSON exporting of the prepared AST.
API	Transforms, stores, and prepares the AST for the Graph Webview.
Implementation	Only requires simple mapping and JSON manipulation functions. The graph AST is stored as a JSON object for easy comparison and manipulation. The storage may be converted to a database if lower latency is required.
Potential Undesired Behaviour	Mangled data input or bugs in JSON manipulation would cause this component to fail.

VSCode Graph Webview / Website Webview	
Normal Behaviour	As the main page of Nodify, this component will display the codebase diagram on an infinite canvas, as described in Section 6.2. This component handles all graph interaction, correspondence with VSCode's text editor, and any other GUI related tasks.
API	Accepts JSON-formatted nodes as input and displays the corresponding interactive diagram. Outputs in the form of SVG, JSON, or PNG will contain snapshots of the current graph view.
Implementation	React and VSCode libraries will be used to create buttons, handle layout interactions, and display the generated graph. A modified version of this webview will be used to enable crowdsourced partition model training.
Potential Undesired Behaviour	Large projects may contain an overwhelming number of graph nodes, leading to incomprehensible layouts. Simplifying the navigation and enabling saving/loading layouts should remedy most issues for small-to-medium-large projects.

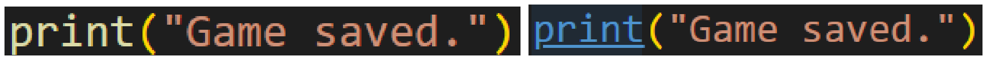
6 User Interface

Interactions between the default VSCode text editor and Nodify's main graphical webview allow for a seamless workflow between writing code and immediately visualizing the resulting structural changes. In particular, we will make extensive use of customizable hotkeys, commands, and shortcuts to assist in navigation of the codebase for beginners and power users alike. Nodify itself only makes use of one “page”, where the project diagram is placed on an infinite canvas with buttons to change the contents and level of detail. The canvas is then updated as the developer modifies the source code files.

6.1 VSCode Text Editor

Creating a VSCode extension instead of a standalone application allows our tool to seamlessly integrate with the setup of all VSCode users. Therefore, rather than designing a proprietary tool for code development, we will make use of VSCode’s comprehensive extension libraries to provide additional interactions and functionality to the existing text editor.

In VSCode, the Python language extension offers several essential productivity-focused features. For example, to navigate to the source code of a Python function, users can simply CTRL+click on the function name, similar to opening a link in a Word document. Hovering over a function name while holding CTRL temporarily underlines the text to signify a hyperlink, as seen in Figure 4.



```
print("Game saved.")
```

```
print("Game saved.")
```

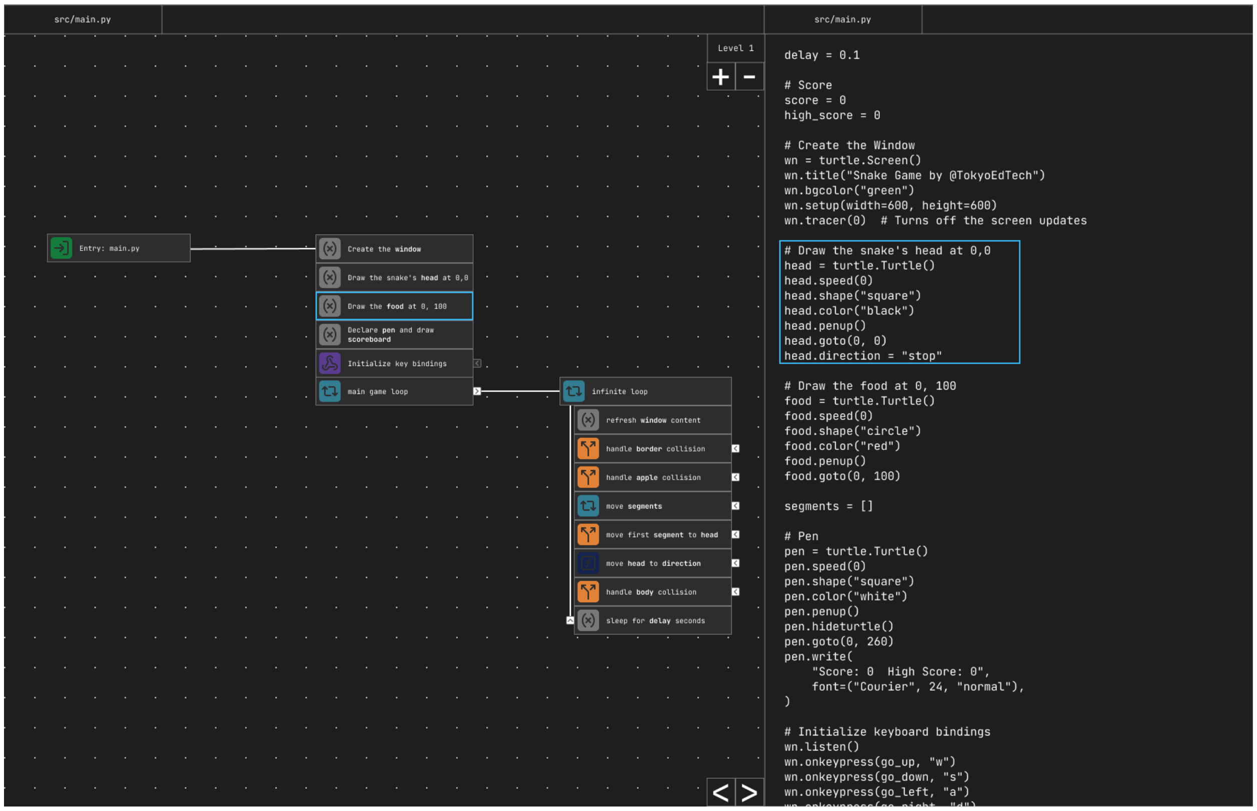
Figure 4: The left image shows the function as it appears normally. CTRL-hover highlighting is shown on the right.

A similar interaction would be highly beneficial for users of Nodify, where a <hotkey>+click interaction will generate and navigate to that location on the graph.

Conversely, we plan to support a reverse operation where users can click on a graph node to immediately navigate to the source code. The text editor will then highlight the source code as the user moves through nodes on the graph. Therefore, users will be able to navigate through both the text-first codebase and Nodify’s visual representation of the codebase.

6.2 Graphical Webview

The diagram webview of Nodify is built-in to VSCode as a standalone tab, as seen in Figure 5. Interactions with the webview mirror those of free-drawing canvas tools like Figma or draw.io, ie. drag to move the canvas view, scrolling to zoom in/out, and click and drag to rearrange components of the generated graph. Additional buttons will be available to progress through the program flow step-by-step, similar to a debugger. For aesthetics and logical consistency, we will make use of VSCode’s built-in themes to ensure uniformity between Nodify’s UI and the rest of the application.



```
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")
```

Figure 5: VSCode view of Nodify (left tab) and the source code (right tab). The blue outline shows a variation of in-graph and in-code highlighting, as described in section 6.1.

7 Appendix

7.1 AST Schema

```
export const flowKinds = [
  "expression_statement",
  "if_statement",
  "for_statement",
  "while_statement",
] as const;
export type FlowKind = (typeof flowKinds)[number];

export const importKinds = [
  // different types of imports
  "import_statement",
  "import_from_statement",
  "future_import_statement",
] as const;
export type ImportKind = (typeof importKinds)[number];

export const definitionKinds = [
   "function_definition",
   "decorated_definition",
   "class_definition",
] as const;

export type DefinitionKind = (typeof definitionKinds)[number];

export const kinds = [
   "comment",
   ...importKinds,
   ...definitionKinds,
   ...flowKinds,
] as const;
export type Kind = (typeof kinds)[number];
```

7.2 Recursive Node Structure for Nodify Diagrams

```
export type Reference = {
  symbol: string;
  location: vscode.Location;
  file: vscode.Uri;
};

export type CodeBlock = {
  id: number;
  text: string;
  location: vscode.Range;
  file: vscode.Uri;
  references?: Reference[];
  children?: CodeBlock[];
};
```

7.3 Handled AST Structures

```
switch (node.kind()) {
    case "attribute":
    case "call":
    case "lambda":
    case "augmented_assignment":
    case "tuple":
    case "set":
    case "list":
    case "dictionary":
    case "pair":
    case "parenthesized_expression":
    case "conditional_expression":
    case "binary_operator":
    case "boolean_operator":
    case "not_operator":
    case "comparison_operator":
    case "string":
    case "interpolation":
    case "subscript":
    case "slice":
    case "generator_expression":
    case "set_comprehension":
    case "tuple_comprehension":
    case "list_comprehension":
    case "dictionary_comprehension":
    case "for_in_clause":
    case "unary_operator":
    case "assignment":
    case "yield":
};
```

7.4 LLM Partitioning Prompt

```
`you are a python code partitioning and labeling expert. Your job is to convert the above json into a more useful, abstract format.

your input will be a json of the following format:

\`\`\`ts
type inputItem = {
 id: number;
 text: string;
 children?: inputItem[];
};
```

````
type inputList = {
 input: inputItem[];
}
\`\`\`

your output type must match the following:

\`\`\`ts
type outputItem = {
 // an incrementing id for each group of code
 groupID: number;

 // a short description 2-8 word description of the code
 label: string;

 // the range of ids that this code represents, from the input list
 idRange: [number, number];

 // a one word category of the code chunk, for example "event_handler_setup"
 // try to be as broad as possible for this field
 type: string;

 // any nested code blocks. Only include children if the corresponding input item has children.
 children?: outputItem[];
}

// your actual output will be an array of these items
type outputList = {
 output: outputItem[];
}
\`\`\`

Note:
- don't include whitespace in your json output, keep it compact like the input.
- don't overlap your id ranges
- you can do this, follow the schema, and you will be successful.
`
````