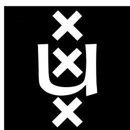
Bridging Incremental Programming and Complex Software Development in Modern Programming Environments

Max Boksem
max.boksem@student.uva.nl

March 5, 2025, 117 pages

Academic supervisor: Thomas van Binsbergen, l.t.vanbinsbergen@uva.nl
Daily supervisor: Thomas van Binsbergen, l.t.vanbinsbergen@uva.nl
Research group: Complex Cyber Infrastructure (CCI) group, https://cci-research.nl/



Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

In modern software development, programmers typically choose between two main types of coding environments: Incremental Programming Environments (IPEs), such as the Read-Eval-Print-Loop (REPL) interpreter IPython and the Jupyter Computational Notebook, and Integrated (text-based) Development Environments (IDEs), such as Visual Studio Code. Each type offers distinct advantages: IPEs excel in iterative development and immediate feedback, which is particularly useful in data science and computational sciences, while IDEs offer features to manage large-scale projects, which is particularly useful in software engineering. However, both types of environments also have their limitations. REPLs and notebooks struggle to manage complex dependencies between modules in larger software systems, and IDEs do not offer the same ease-of-use and incremental feedback as obtained by REPLs and notebooks.

This thesis addresses the challenge of bridging the gap between these two paradigms by proposing a new hybrid programming environment. The proposed environment proposes an alternative organization of code based on a graph structure, wherein nodes represent modular code blocks, functions, or components, and edges signify relationships such as dependencies, interactions, or execution paths. By shifting the focus from linear organization to a graph-based paradigm, we explore how such a system can better manage complexity, support incremental development, and improve code readability and modularity, particularly in large-scale software engineering.

The proposed environment was developed as a prototype, referred to as Incremental Graph Code (IGC), designed to fulfill key requirements derived from both IPEs and IDEs. These requirements include incremental programming features to support modular development, complexity management to handle large-scale projects, and high-level code abstractions to enhance readability. Extensibility and exploratory programming features were also considered to facilitate customization and improve the user experience. The prototype was created based on these requirements through a modern web-based application that could potentially be deployed both as a cloud-hosted solution or a standalone system application.

Four case studies were conducted to evaluate the effectiveness of the proposed environment. The first case study examined the environment's incremental programming capabilities by replicating a typical computational workflow found in Jupyter notebooks. The second case study explored the utility of code projections, demonstrating how the environment organizes and correlates scattered code and documentation. The third case study focused on architectural views, showcasing the ability to model complex software structures such as Model-View-Controller (MVC) architectures. The fourth case study investigated exploratory programming features, emphasizing how developers can compare execution paths and analyze program evolution over time.

The findings from these case studies demonstrate how the hybrid environment could potentially enhance code readability, modularity, and maintainability while preserving the flexibility and interactivity of incremental programming. By offering a graph-based perspective, IGC can allow developers to manage complexity better, visualize dependencies, and experiment with alternative program structures in ways not achievable in traditional environments.

This thesis presents an alternative to existing programming environments and lays the groundwork for future research into hybrid development workflows that merge incremental programming with large-scale software engineering practices. The proposed environment offers a new perspective on code organization and development, providing a stepping stone toward more efficient, readable, and scalable software systems.

Contents

1 Introduction **6**
1.1 Problem statement . . . 6
1.2 Motivation . . . 6
1.3 Research Questions . . . 7
1.4 Research method . . . 7
1.5 Contributions . . . 8
1.5.1 Incremental Programming for Software Engineers . . . 8
1.5.2 Exploratory Programming . . . 8
1.5.3 Complexity Management . . . 8
1.5.4 Enhanced Analysis Capabilities . . . 8
1.6 Scope . . . 8
1.6.1 Programming Environment . . . 8
1.6.2 Programming Languages . . . 8
1.6.3 Scale of Development . . . 9
1.7 Outline . . . 9

2 Background **10**
2.1 Prerequisite Concepts . . . 10
2.1.1 Incremental Programming . . . 10
2.1.2 Read-Eval-Print Loops (REPLs) . . . 11
2.1.3 Computational Notebooks . . . 12
2.1.4 Exploratory Programming . . . 12
2.1.5 Incremental Programming Environments (IPEs) . . . 13
2.1.6 Classical Software Development . . . 13
2.1.7 Text-based Programming Environments . . . 13
2.1.8 Integrated Development Environments (IDEs) . . . 13
2.1.9 Complexity (Management). . . . 14
2.2 Domain . . . 14
2.2.1 Data Structure / Code . . . 15
2.2.2 Interface / Programming Environment . . . 15
2.2.3 Engine / Compiler / Interpreter . . . 15

3 Design **16**
3.1 Requirements . . . 16
3.1.1 Formulation . . . 16
3.1.2 Overview . . . 19
3.1.3 Analysis . . . 20
3.2 Combining the Requirements . . . 22
3.2.1 Graph Structure. . . . 22
3.2.2 Sessions. . . . 26
3.2.3 File Management. . . . 27
3.2.4 Version Control. . . . 27
3.2.5 Extendability. . . . 27
3.2.6 Reproducibility . . . 28

4 Implementation **29**
4.1 General Technology / Tech Stack . . . 29

4.1.1 Mediums to present the environment . . . 29
4.1.2 Front End Technologies . . . 31
4.1.3 Back End Technologies . . . 33
4.1.4 Common Technologies . . . 34
4.2 Other Specifications . . . 34
4.2.1 Directory Structure . . . 34
4.2.2 IGC File . . . 35
4.2.3 Language Support . . . 36
4.2.4 Execution and State Management . . . 36
4.2.5 Dependency Tree Creation . . . 37
4.3 Front End Visualizations . . . 37
4.3.1 File Explorer . . . 37
4.3.2 Graph Editor . . . 39
4.3.3 File Editor . . . 40
4.3.4 Nodes . . . 41
4.3.5 Relationships: . . . 41
4.3.6 Views . . . 41
4.3.7 Custom IGC Components . . . 45
4.3.8 Hidden Menu Components . . . 47
4.3.9 Styling . . . 48
4.4 Back End / APIs . . . 50
4.4.1 Code Analysis / Execution . . . 51
4.4.2 File Operations . . . 51
4.4.3 Electron Specific Functionality . . . 52
4.5 Application Diagrams . . . 52
4.6 Software Development Practices . . . 54
4.6.1 General Practices . . . 54
4.6.2 Version Control and Management . . . 55
4.6.3 Feature Realization and Implementation . . . 55
4.6.4 Code Organization . . . 56
4.6.5 Testing . . . 59
4.6.6 Documentation . . . 59
4.7 Miscellaneous . . . 60
4.7.1 Code Templating . . . 60
4.7.2 Text Editor History . . . 60
4.7.3 Relationship Edge Path calculations . . . 60

5 Case Studies **67**
5.1 Case Study 1: Computational Notebook . . . 67
5.1.1 Introduction . . . 67
5.1.2 Showcase: Computing Basic Statistics . . . 68
5.1.3 Comparing IGC to Jupyter Notebook . . . 69
5.2 Case Study 2: Code/Documentation Projections . . . 70
5.2.1 Introduction . . . 70
5.2.2 Example Use Case: Managing Scattered Code and Documentation . . . 72
5.2.3 Comparing IGC to PescaJ . . . 72
5.3 Case Study 3: Architectural Diagrams and Composition . . . 74
5.3.1 Introduction . . . 74
5.3.2 Main Features of Graph and Composition Nodes . . . 74
5.3.3 Example Showcase: Modeling MVC Architecture . . . 74
5.3.4 Comparing IGC to Traditional Approaches . . . 76
5.4 Case Study 4: Exploratory Programming GUI . . . 77
5.4.1 Introduction . . . 77
5.4.2 Example Use Case: Managing Execution State and Branching . . . 78
5.4.3 Comparing IGC to the Exploratory Programming GUI . . . 80

6 Discussion **82**
6.1 Core Distinctive Features . . . 82

6.1.1 Graph-Based Model . . . 82
6.1.2 Sub Graphs . . . 82
6.1.3 Sessions . . . 82
6.1.4 View Representation . . . 83
6.2 System Evaluation: . . . 83
6.2.1 Performance . . . 83
6.2.2 Scalability . . . 83
6.3 Conflicts and Trade-offs . . . 84
6.3.1 Language Support . . . 84
6.3.2 Software Complexity and Structuredness . . . 86
6.3.3 Composition vs Inheritance / Flow vs Model-View-Controller . . . 86
6.4 Case Studies . . . 87
6.4.1 Incremental Programming . . . 87
6.4.2 Exploratory Programming . . . 88
6.4.3 Complexity Management . . . 88
6.5 Research Questions . . . 89
6.5.1 Research Question 1 . . . 89
6.5.2 Research Question 2 . . . 90
6.6 Limitations . . . 91
6.6.1 Threats to Validity . . . 91
6.6.2 Specific Feature Limitations . . . 92
6.6.3 Qualitative Nature of the Case Studies . . . 93
6.6.4 What we would do differently . . . 93

7 Related work **95**
7.1 Other Computational Notebooks . . . 95
7.2 Model-Driven Development . . . 95
7.3 Visual Programming Languages . . . 96
7.4 Projectional Views . . . 96
7.5 Alternative Code Structures . . . 97
7.6 Smalltalk . . . 98

8 Future Work **100**
8.1 Documentation / Visual Features . . . 100
8.1.1 Label Nodes . . . 100
8.1.2 Documentation Chaining . . . 100
8.1.3 More Granular Edge Control . . . 100
8.2 Language Support . . . 101
8.2.1 Implement different programming languages . . . 101
8.2.2 Language Manager / Control Component . . . 101
8.2.3 Language Execution / Interactions . . . 101
8.3 Integrated Version Control . . . 101
8.3.1 Direct Integration . . . 101
8.3.2 Graphical Diffing . . . 102
8.4 Usability Enhancements . . . 102
8.4.1 Make Projectional Views Editable . . . 102
8.4.2 Allow Input Terminals . . . 102
8.4.3 Execution Recommendations through Dependency Tree . . . 102
8.4.4 Threading Execution Relationships . . . 102
8.4.5 Reactive Programming . . . 102
8.5 Long-term Viability Features . . . 103
8.5.1 Graph Node Cycle Detection . . . 103
8.5.2 Graph Node Enhancing Functionality . . . 103
8.5.3 Purely Functional Programming Options . . . 103
8.5.4 Debugging Tools . . . 103
8.5.5 In-depth Scalability / Performance Testing and Optimization . . . 103
8.6 Transition / Compatibility Features . . . 104
8.6.1 Split Nodes . . . 104

8.6.2 Infer Node Types 104
8.6.3 Conversion Tools 104
8.7 Analysis Tools 104
8.7.1 User Study 104
8.7.2 User Data Metrics 104
8.8 Further in the Future 105
8.8.1 Component Marketplace 105
8.8.2 Live Collaboration 105
8.8.3 Porting IGC to Cloud-based Environment and System Application 105
8.8.4 AI-Powered Code Completion 105
8.8.5 Polyglot Programming 105

9 Conclusion **107**
9.1 Summary of Contributions 107
9.2 Addressing the Research Questions 107
9.3 Limitations and Future Directions 108
9.4 Final Reflections 109

Bibliography **111**

Chapter 1

Introduction

1.1 Problem statement

In modern software development, programmers typically choose between two main types of coding environments: Incremental Programming Environments (IPEs) such as the Read-Eval-Print-Loop (REPL) interpreter IPython [1] and the Jupyter Computational Notebook [2], and Integrated (text-based) Development Environments (IDEs) such as Visual Studio Code. Each offers distinct advantages but also suffers from significant limitations when applied outside their ideal contexts.

IPEs excel at providing immediate feedback, enabling a highly iterative and exploratory style of programming [3–5], particularly beneficial in the fields of data science and artificial intelligence where outcomes are unpredictable [6, 7]. Computational Notebooks promote an exploratory programming workflow by permitting code cells to be re-executed and re-ordered on the fly. This flexibility can help researchers and developers refine hypotheses iteratively when the exact goal is not fully known from the start. However, Notebooks are also found to be limited with regards to exploratory programming [8–10], which can become a drawback for large-scale software endeavors. Moreover, the linear and isolated interface of REPLs and Notebooks can become a liability for larger projects. Code fragments are typically provided in isolated cells, without ways of structuring or grouping fragments together (*e.g.*,, to form modules, classes, or packages), which can lead to difficulties in maintaining the complexity of traditional software projects. This property hinders the application of IPEs to complex software systems where managing relationships and dependencies between code fragments, *i.e.*,, the structure of the source code itself, is essential.

On the other hand, traditional IDEs are widely considered the standard for large-scale software development, as evidenced by the Stack Overflow Developer Survey of 2024[1]. IDEs typically offer powerful tools for comprehensive project management, debugging, and code maintenance. Their text-based paradigms allow developers to structure and scale projects efficiently, ensuring clarity of dependencies, version control, and modularity. However, IDEs often lack the interactive, incremental nature that is central to IPEs. For instance, instant code execution and rapid feedback loops are not as seamlessly integrated into most IDE workflows, which can limit efficiency during early experimentation or proof-of-concept development phases.

This duality creates a significant gap in the ecosystem of programming environments. While IPEs encourage rapid experimentation and iterative design, they struggle to provide the scaffolding needed for long-term software maintenance and collaboration. Conversely, IDEs excel at managing large projects but often forgo the immediacy and flexibility that can aid innovation and productivity in the early stages of software creation. Bridging this gap, or creating an environment that harmonizes the best of both worlds, remains an open challenge for data scientists and software developers alike.

1.2 Motivation

This work is important because it has the potential to transform how programmers approach both large-scale development and exploratory programming. Existing environments often force developers to choose between the immediacy of feedback found in interactive programming environments (IPEs) and the robust structural tools offered by integrated development environments (IDEs). In fact, prior studies

[1] https://survey.stackoverflow.co/2024/

have shown that high-quality code and effective tooling are critical for developers' productivity and long-term project success [11]. By introducing a hybrid approach based on a more flexible representation of source code, this work aims to combine rapid iteration with efficient complexity management.

A core contribution of this thesis is the introduction of a graph-based data structure for representing source code and its interdependencies. Recent research on computational notebooks has shown that the traditional 1D, linear arrangement of code cells can hinder non-linear explorations, such as branching analyses and comparative studies [12, 13]. By leveraging a graph-based model, our approach supports a more flexible, multi-dimensional view of code that facilitates incremental development and enables developers to navigate complex dependencies more intuitively. Moreover, the extensible nature of the graph permits the addition of new node and relation types, thereby accommodating a wide variety of programming paradigms and use cases.

In practical terms, this means that developers can rapidly explore alternative approaches to a problem by testing out different code fragments or modules while still maintaining a clear view of how these components interact. This design aligns with findings that advocate for the use of abstractions over displaying an entire codebase at once, which has been shown to reduce cognitive load and improve project maintainability [14, 15]. As projects scale, the graph-based view and its associated tools could further facilitate complexity management, enhance readability, and foster more effective collaboration. Ultimately, such an environment holds promise for elevating both developer productivity and code quality.

1.3 Research Questions

In order to bridge the gap between IPEs and IDEs, we aim to combine the benefits of incremental execution, exploratory programming, and structured project organization. To evaluate this proposition, we formulate two main research questions:

> **Research Question 1 (RQ1):**
> How can a programming environment be redesigned, specifically in the source code representation and interface, to take advantage of incremental programming techniques, exploratory programming methods, and the handling of complexity usually found in software projects?
>
> **Research Question 2 (RQ2):**
> What are the advantages/disadvantages of the proposed environment when comparing them to different environments that specialize in incremental programming, exploratory programming, or the handling of complexity usually found in software projects?

Answering these questions requires both a conceptual design and a tangible prototype that can be tested and analyzed. We hypothesize that a new organizational paradigm, built around a graph-based model of code, could address the limitations of existing tools and provide a cohesive development experience. The specifics of this approach, along with its comparative merits, will be investigated throughout the thesis.

1.4 Research method

To address the research questions, this thesis adopts a design-and-evaluate methodology:

1. **Conceptual Design:** We begin by identifying the essential features and requirements of both IPEs and IDEs. This involves analyzing common workflows in Jupyter-like environments and established IDEs, noting how each handles incremental development, exploratory programming, and complexity management.
2. **Prototype Development:** Based on these requirements, we design and implement an open-source proof-of-concept of a new programming environment. The system leverages a graph-based data structure to organize code, relationships, and execution paths, aiming to integrate incremental development with robust project organization.
3. **Comparative Case Studies:** We then compare our prototype with existing environments that have overlapping features. Specifically, four case studies will be introduced to explore similarities, differences, advantages, and disadvantages. For instance, one case study might recreate typical notebook functionality, while another might focus on complex software architectures, such as Model-View-Controller.

4. **Analysis and Synthesis:** Finally, we analyze the outcomes of these case studies to gauge how well the proposed environment addresses the identified limitations of current IPEs and IDEs. The findings will be used to answer the research questions, highlighting the system's strengths and any areas for improvement.

1.5 Contributions

Although the proof-of-concept environment targets multiple aspects of development, we can group its expected contributions into four categories:

1.5.1 Incremental Programming for Software Engineers

This project extends the concept of incremental programming outside a strictly REPL-driven context. By merging the iterative features of a REPL with the structural benefits of IDEs, the environment offers an agile and modular approach that can adapt to various software engineering needs.

1.5.2 Exploratory Programming

The environment supports exploratory programming by allowing developers to experiment with code fragments, modules, and relationships in a flexible and interactive manner. There will be an intuitive way to explore and visualize code, which can help developers refine hypotheses and test ideas more effectively. This capability is particularly useful for data scientists, researchers, and developers working on projects with uncertain or evolving requirements.

1.5.3 Complexity Management

Large software projects demand strong strategies to handle interdependencies and maintain readability. The proposed environment's graph-based structure and architectural views (similar to UML-like abstractions) empower developers to manage and visualize complex systems more effectively, improving scalability and maintainability.

1.5.4 Enhanced Analysis Capabilities

Building on the graph paradigm, the environment includes analysis tools that offer insights into how different code elements interact. For example, developers can visualize inheritance hierarchies, track execution paths, or quickly identify dependencies between modules. These capabilities streamline both debugging and onboarding, allowing for faster comprehension of complex architectures.

1.6 Scope

Here, we define the boundaries of the thesis by clarifying what is included and what is not.

1.6.1 Programming Environment

The core focus is creating a brand-new programming environment that merges the interactive strengths of REPLs and notebooks with the scalability and organizational tools of text-based IDEs. The aim is not to replicate every feature from existing tools but to introduce a cohesive hybrid model.

1.6.2 Programming Languages

The environment is intended to support both interpreted and compiled languages. An initial prototype may concentrate on a popular option, such as Python. Still, the intention is to allow the environment to be used with any programming language a user might want to use.

1.6.3 Scale of Development

Although this environment aspires to handle projects of various sizes, the initial evaluation will focus on simple software projects as a proof-of-concept to demonstrate viability as a programming environment. Future work will expand this focus to include larger software projects.

1.7 Outline

In Chapter 2, we describe the background of this thesis. Chapter 3 describes the requirements needed for the proof-of-concept as well as describing the overall design process. Chapter 4 goes over how the proof-of-concept was made and the challenges faced. Case studies are introduced and compared in Chapter 5 and discussed in Chapter 6. Chapter 7, contains the work related to this thesis. Finally, we present the planned future work of the research in Chapter 8 and our concluding remarks in Chapter 9.

Chapter 2

Background

This chapter will present the necessary background information for this thesis. First, we define some basic concepts that are essential for understanding the rest of the thesis. We then go over the domains that this thesis will interact with. Finally, we define some terms that will be used throughout the thesis.

2.1 Prerequisite Concepts

The following concepts are essential to understanding the rest of the thesis:

2.1.1 Incremental Programming

```
[1]: class Animal:
         def __init__(self, name, age):
             self.name = name
             self.age = age

         def speak(self):
             return "I'm an animal and my name is " + self.name

[2]: class Dog(Animal):
         def __init__(self, name, age, breed):
             super().__init__(name, age)
             self.breed = breed

         def speak(self):
             return super().speak() + ". Woof!"

[3]: class Cat(Animal):
         def __init__(self, name, age, color):
             super().__init__(name, age)
             self.color = color

         def speak(self):
             return super().speak() + ". Meow!"

A    dog = Dog("Buddy", 5, "Labrador")

B    cat = Cat("Whiskers", 3, "Black")

C    print(cat.speak())
     I'm an animal and my name is Whiskers. Meow!

D    print(dog.speak())
     I'm an animal and my name is Buddy. Woof!
```

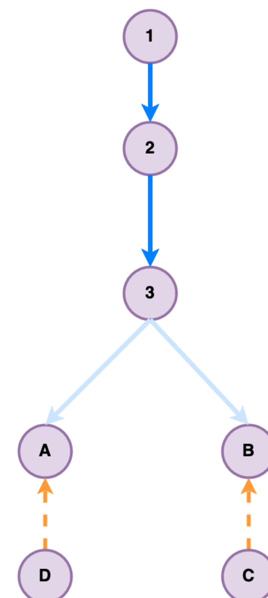


Figure 2.1: Incremental Programming with execution relationships (Showing possible paths)

Incremental programming is an approach to software development where the process is segmented into small, manageable programs that are built on top of one another. This differs from normal development, which focuses on writing a huge code feature and then testing everything afterward. Developing incrementally can help alleviate bugs faster therefore making development more efficient [16]. A common feature in incremental programming is incremental computing which aims to save time by selectively

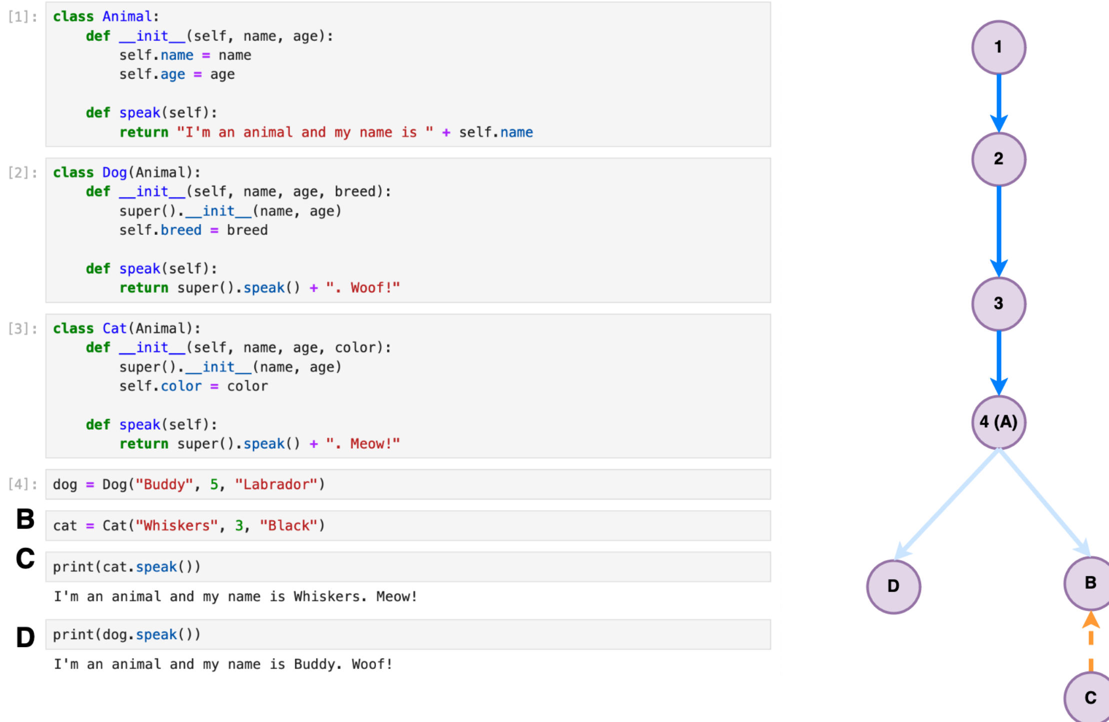


Figure 2.2: Incremental Programming with execution relationships (Node A was executed)

computing only the outputs affected by changes in data. It can lead to considerably faster results against completely recomputing new outputs.

In fig. 2.1 and fig. 2.2 the essence behind incremental programming is shown. In the figures, the dark blue arrows represent what has been executed and in what order. Two potential, valid executions continue the narrative after the third execution: cells A and B (represented by light-blue arrows). Technically, any cell provided can be executed, including re-executing previously executed cells. Still, for the given example, we are assuming we would like to execute every cell once without running into any execution errors. The orange, dashed arrows represent dependencies (*i.e.*, cell D is dependent on cell A and cell C is dependent on cell B). fig. 2.2 shows the result after cell A has been executed and the subsequent executions are shown. As we see, incremental programming is a method of progressing through a program step by step, building the execution through steps taken by the user.

2.1.2 Read-Eval-Print Loops (REPLs)

Incremental programming shines in environments such as Read-Eval-Print Loops (REPLs). REPLs are interactive programming environments that allow developers to write and execute code sequentially in small fragments. Common examples include command-line shells and similar environments for programming languages, such as IPython [1], and the technique is very characteristic of scripting languages [17]. However, REPLs are not limited to scripting languages; they can be found in languages like Java (JShell [18]), C++ (CLing [19]), and many more.

Characterized by its iterative development style, REPLs involve gradually building up a programming project piece by piece, allowing for continuous testing, adaptation, and refinement. REPL environments have a lot of advantages for code development, such as quick evaluation when manipulating code fragments, quick snippet testing of code, configuration or state independency (especially useful when a snippet has a lot of computation needed), and modularity of code fragments, to name a few [20]. For all the benefits REPLs have to offer, they cannot be utilized efficiently in the context of software development outside of small projects due to scalability concerns.

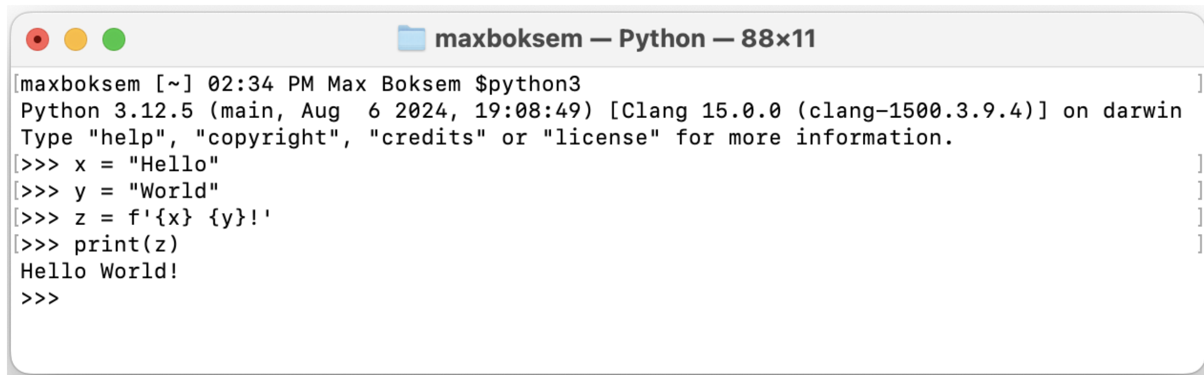


Figure 2.3: Python REPL environment (IPython).

2.1.3 Computational Notebooks

Building on the concept of REPLs, computational notebooks provide the developer with a friendly user interface and more functionality to interact with a REPL kernel. For example, users can create many code cells before any execution occurs and then execute the cells in whatever order they choose. Cells are often rerun to test different scenarios or to see how the code changes over time. Computational notebooks also allow the user to create documentation cells using markdown, which can explain the code or provide additional information [21].

Figure 2.4: Computational Notebook (Jupyter Notebook)

Computational notebooks like Jupyter Notebook [2] have become increasingly popular, especially in the fields of data science and artificial intelligence. This is shown by the rapid growth in the availability of Jupyter notebooks on GitHub, soaring from about 200,000 in 2015 to nearly 10 million by October 2020 [22, 23]. The Jupyter extension for Visual Studio Code is also the third most downloaded extension, with more than 74 million downloads at the time of writing this[24]. millions now use them for their work [21], including research communities [22].

2.1.4 Exploratory Programming

In 1983, Beau Shiel, a manager at Xerox's AI Systems, popularized the term "Exploratory Programming" to describe the challenges of applying rigid software development lifecycles to experimental AI code [25]. In open-ended tasks, where a program's behavior cannot be fully defined in advance, exploratory programming is a great practice to actively experiment with various code possibilities and discover workable solutions along the way [3]. The style of exploratory programming involves experimenting with different versions of the same code fragment and responding to the observed effects (*e.g.*, modifying and executing

a code cell in a notebook multiple times). Various approaches have been investigated to better support exploratory programming in Notebooks [7] or programming environments in general [5].

One approach to supporting exploratory programming is to allow developers to manipulate execution paths to explore and understand alternative branches of program evolution. This approach enables developers to test different scenarios by altering the sequence of code execution. Instead of modifying and executing one specific cell, a developer might also choose to execute an alternative cell to observe how the two compare based on their outputs. This flexibility facilitates deeper insights into how various parts of the code interact. Ideally, outputs from these different execution paths can be easily compared, providing clear and immediate feedback on how changes impact the code's behavior [4]. In [26], a generic approach for supporting exploratory programming in programming environments is suggested in which an execution graph keeps track of previously visited program states to which the programmer can return, similar to the record-keeping required for omniscient debugging [27].

2.1.5 Incremental Programming Environments (IPEs)

Incremental programming environments (IPEs) was a term coined initially by Medina-Mora and Feiler in "An Incremental Programming Environment" [28]. However, we will broaden the term to mean any programming environments that support incremental programming. IPEs encompass both REPLs and computational notebooks as they both allow developers to write and execute code incrementally.

IPEs are usually limited to data science and artificial intelligence as the emphasis on these projects are more about computational constraints utilizing powerful libraries versus project complexity[22, 29]. A complex project can be implemented entirely in IPEs, but this is equivalent to having all code exist in one file; almost impossible to read and analyze. It can be argued that this is one of the major bottlenecks of why IPEs are not used to develop large-scale software systems. For all the benefits IPEs have to offer, they cannot be utilized efficiently in the context of software development outside of smaller software projects.

2.1.6 Classical Software Development

The norm for creating large-scale software projects usually follows the following workflow: a developer opens an IDE or text editor, writes some code, and clicks run, hoping everything runs as expected. This method is adaptable and offers flexibility to almost every coding application. It does have its pitfalls, however. As project complexity increases, readability tends to decrease in software projects [30]. A large area of research is concerned with how to convert classical development methods into more readable forms such as Domain-Specific Languages (DSLs), UML generators, and Model Driven Development (MDD)[31–33]. Integrating incremental programming into this framework addresses these concerns by offering developers the advantage of immediate code feedback and simplified experimentation with code modifications.

2.1.7 Text-based Programming Environments

Text-based programming environments are a common type of programming environment. A text editor is a type of interface dedicated to the creation and modification of plain text data. Text-based programming environments are lightweight and are specifically designed to work with plain text. Since most programming languages utilize the use of file-based code storage, text-based programming environments are good for changing coding and general scripting, along with general text manipulation. Some examples include Sublime Text, Notepad++, and VIM. Text-based programming environments can sometimes offer features such as syntax highlighting, search and replace, and file operations/management. They do not include functions such as debugging or building, which are found in Integrated Development Environments (IDEs).

2.1.8 Integrated Development Environments (IDEs)

Integrated Development Environments (IDEs) are a type of programming environment that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of a source code / text-based editor, build automation tools, and a debugger. Most modern IDEs have intelligent code completion, syntax highlighting, and code refactoring. Some examples include Visual Studio, Eclipse, and IntelliJ IDEA. IDEs are generally used for larger projects that require more than just text editing. They are used for debugging, building, and running code. They are also used for

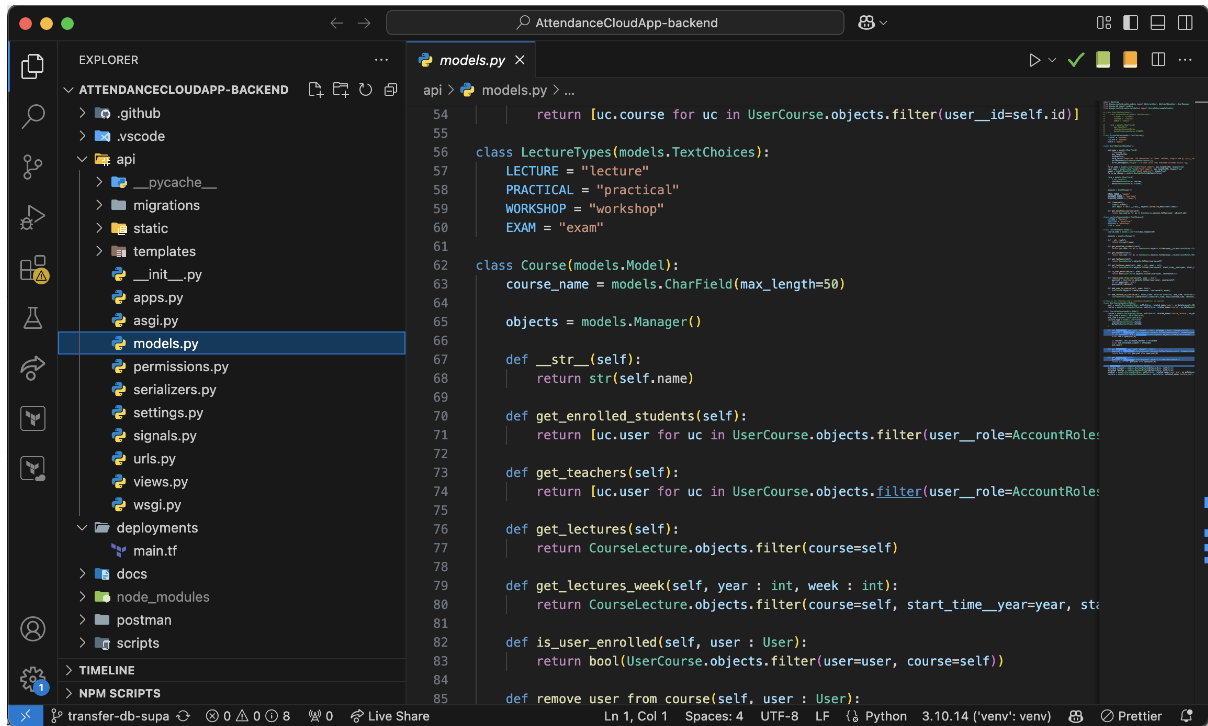


Figure 2.5: Visual Studio Code (VSCode)

version control, testing, and deployment. IDEs are generally more complex than text-based programming environments and are used for more complex projects.

A special case named, Visual Studio Code (VSCode) (Not to be confused with Visual Studio), tries to "toe the line" between text-based programming environments and IDEs. They offer a lightweight text-based editor along with extensions that mimic many of the features of an IDE, such as debugging, building, and running code. For these reasons, many developers have chosen to use VSCode as their primary development environment and has become the most used development environment, by far, according to the Stack Overflow Developer Survey of 2024[1].

2.1.9 Complexity (Management).

Complexity can be defined in many different ways in the scope of software engineering. Most software metrics define complexity specific to a project including number of statements, McCabe's cyclomatic number [34], Halstead's programming effort [35], and the knot measure [36, 37]. This project is focused on a programming environment instead of any specific project, so we will focus on how to manage complexity in a general sense. When we refer to complexity, we will be referring to the complexity of the code structure and how it is managed in the programming environment.

For structural complexity, we will be relating to the complexity model of object-oriented systems [38], specifically when it comes to encapsulation, and the reduction of cognitive load by simplifying the interface [39, 40]. Ejiogu refers to this type of structural complexity as 'structuredness' [14].

The management of the system through analyzation and visualization tools will be another part of complexity management. The goal of software visualization is to provide knowledge of a system, program artifacts, and understanding their relationships [41]. By limiting the data needed to analyze software, we can therefore reduce the complexity of the data shown [42].

2.2 Domain

We define and divide the typical pipeline of the software development process, from code to an actual application, into three main domains: the data structure for organizing and storing code, the programming environment (the interface through which developers interact with code), and lastly, the engine

[1] https://survey.stackoverflow.co/2024/

(how the code is physically run on the computer). Below explains how the proposal will interact with each of these domains:

2.2.1 Data Structure / Code

The data structure domain directly relates to how source code is stored and organized. For example, if a developer adds methods to a class, they change the code of the class. These classes, in turn, can be inside packages and those within files.

The data structure will be an important aspect of the project. It will need to be thoroughly considered to make the project a success. Due to the nature of how nodes can interact with each other, the structure will most likely be a graph-like structure. Each node will be a code fragment or abstraction connected to other nodes through a number of relationships. A careful investigation will need to be done to identify different types of relationships the data structure should have. The combination of programmer and tool will populate the relationships (add edges to the graph).

2.2.2 Interface / Programming Environment

The programming environment presents the source code data structure to the programmer and enables various kinds of interactions. Existing examples are computational notebooks (*e.g.*, Jupyter), IDEs (*e.g.*, VSCode), and REPLs (*e.g.*, IPython).

Out of all the domains, this will be the primary focus of the project. Incremental programming requires the representation of fine-grained code snippets/fragments in the source code data structure. Complex software (software engineering) requires that code fragments can be structured (classes, files, packages, etc.). The goal is to develop a programming environment in which a developer can build complex software while still benefiting from incremental programming. A solution to accommodate both goals is proposed by introducing an abstract interpretation of a node that can become more granular to the preference of the developer. The data nodes should be represented cleanly and legibly. Data nodes should be interactable, both to reorganize the nodes/relationships and to edit the nodes themselves.

Another feature that should be present in the interface is an exploratory way to pick and choose different nodes to quickly execute. This feature tries to capture the premise behind incremental programming. The developer should have a way to define an execution path (through nodes), then quickly swap nodes for different executions. They should also be able to investigate how attaching nodes can generate different execution paths.

The visualization should also allow the developer to do fast analysis. Potentially a developer wants to explore how a particular code fragment is executed. If so, they should be able to create a projection from a generated dependency graph to and from this node. This is one example, but this should be extendable to whichever relationship is desired from the developer.

2.2.3 Engine / Compiler / Interpreter

The engine domain involves how to convert the code and interface by executing or compiling to a machine-readable format. For example, how IPEs are executed or how software is compiled.

In the scope of incremental computing, the engine will need a quick way to keep track of the execution state. If a developer chooses to swap nodes or select a different execution path in the interface, the engine should not have to recompute everything. The engine should only have to recompute the outputs affected by changes in data.

The engine domain must also be considered in this proposal as it is responsible for making the application itself, however, the plan will be to utilize pre-made engines as much as possible. The development process that this proposal introduces is not catered to any specific language. Since different languages interact with the computer in different ways, this must be a consideration. As this proposal is geared toward creating large software projects, there must be some way to create a distributable, optimized, executable application. Compiled languages usually already do this when compiling and building the code. Interpreted languages, like Python, usually have to be executed "on the fly" which generally leads to slower execution time. To counteract this, the engine might have to convert to a compilable system.

Chapter 3

Design

3.1 Requirements

In this section, we will extract and outline the main requirements for the new programming environment proposed in this thesis. The process begins with the Requirements Formulation, where the initial set of requirements is derived from the analysis of existing incremental and classical text-based programming environments. Throughout the text, bolded keywords will correspond to the main requirements. Following this, we will summarize these main requirements into a table in the Requirements Overview section. This table will highlight the key requirements, their potential features, descriptions, and the importance of implementing them for the initial proof-of-concept. Finally, we will provide a detailed analysis of these requirements in the Requirements Analysis section, discussing how they can be effectively implemented.

3.1.1 Formulation

To answer Research Question 1 (RQ1), a proof-of-concept will be created to demonstrate the redesign of the programming environment. This new programming environment will need to consider both features and requirements specified by typical environments specialized in incremental programming and classical text-based environments. Once we classify the key ideals of each environment, we can then start to create a set of requirements needed. Once the requirements are defined, we can create a list of features for the new environment.

A popular programming environment that utilizes incremental programming techniques is a Read-Eval-Print-Loop (REPL). A common instance of a REPL is iPython (for Python), which is used to make the Jupyter Notebook. REPLs are used heavily in the fields of data science and artificial intelligence. These fields usually need to utilize incremental programming techniques as they often involve large computations, such as training models, and require a lot of experimentation to fine-tune. Incremental programming is excellent for both these uses as computation is selectively done to avoid repeated computation (known as incremental computation). Incremental programming allows users to break down individual cells, test them in isolation, and integrate them in stages, thus ensuring a more error-prone code.

One can ask, "Why are REPLs not commonly used for large-scale software development?" The short answer lies in their inherent design: while languages such as Clojure and Haskell integrate REPLs as a core part of an incremental programming workflow (typically in combination with an editor or IDE), this approach suits projects where rapid, iterative feedback is paramount rather than managing complex, interdependent modules. In contrast, languages designed for large-scale systems like C# and Java rely on environments that can handle the full breadth and intricacies of extensive codebases. Even Python, with its widely adopted IPython REPL, emphasizes a broader ecosystem beyond incremental development, spotlighting why REPLs are not the primary tool for large-scale software engineering. With so many different modules and systems a software project can contain, keeping track of everything in a linear set of cells is almost impossible. For similar reasons, 'readability' starts to degrade as a project expands. Poor readability can impede developers from understanding what the code is doing, cause sub-optimal changes, and introduce bugs into a code base [43]. For Jupyter Notebook, markup on cells helps with overall readability, but still is not substantial enough to support a typical software project.

Most software engineering projects are made using a classical text-based programming environment. Traditional text-based editors, such as Vi, Emacs, and even notepad [44], were the primary tools when

many programming languages were being created. These environments focus on raw text manipulation and often require developers to use command-line tools for compiling, debugging, and version control. Text-based programming environments are highly flexible and can be customized with plugins and extensions to support various programming languages and workflows. They provide a lightweight and distraction-free environment, allowing developers to focus purely on writing code without the overhead of additional features found in modern IDEs.

Text-based environments are inherently compatible with source control systems like Git, enabling seamless integration for version control. They allow developers to manage their code base effectively, track changes, and collaborate with team members. Many modern IDE features, such as syntax highlighting, code completion, and linting, are ported to text-based environments through plugins, making them versatile tools for development. Additionally, some programming environments, like Visual Studio Code, blur the lines between traditional text editors and full-fledged IDEs by offering extensive support for extensions that provide IDE-like features.

However, text-based programming environments have some limitations. Unlike REPLs, they do not inherently support incremental programming. Iterative development is the aim of many different software engineering agile methodologies [45]. However, with text-based programming, it is challenging to truly isolate changes to a specific code change. It is often the case that an obscure change in one part of code can change the output of another unrelated code snippet.

Regarding incremental computation, when testing minor changes, developers normally need to recompile and rerun large sections of code, which can be time-consuming. Incremental changes cannot be tested in isolation as easily as in a REPL environment. Readability can also become an issue as projects grow; large code bases can be challenging to navigate and understand, especially for new team members. Text-based environments can be customized with various tools to enhance readability, such as code folding and syntax highlighting. However, there are limits to how well this realistically can scale along with project size.

Despite these challenges, text-based programming environments' flexibility, customization options, and lightweight nature make them valuable tools for many developers. By understanding the strengths and weaknesses of incremental programming environments and classical text-based environments, developers can choose the best tools to suit their specific needs and workflows.

A reexamination of these advantages and disadvantages can be done to make a new environment that suits the needs of almost all developers. Several high-level requirements must be met to create a hybrid programming environment that leverages the benefits of both incremental programming and traditional text-based development. This hybrid environment should provide the **interactive and iterative features of REPLs**, allowing developers to test and refine their code in small, manageable increments. Simultaneously, it should offer the **structural support and comprehensive project management features found in classical text-based** environments, making it suitable for large-scale software projects.

To **handle the complexity of large software projects**, the environment should include features that help in managing dependencies, tracking changes, and organizing code effectively. The hybrid environment should utilize a more sophisticated structure rather than being limited to the linear sequence of cells typically found in REPLs. The one-dimensional constraint is too limited, so we need to expand it to two dimensions. The most generic 2-dimensional structure is a graph. By representing code and its relationships in a graph-like format, we can categorize and organize different sections of a project more effectively. This graph structure allows for cells to be grouped and abstracted, representing various systems and components within the software project. This method not only can improve organization if done properly but also enhance readability and manageability.

To further enhance the readability of the entire system, the graph representation should **support different architectural views**, allowing developers to visualize the structure and relationships of their code in various forms. For example, the environment could provide a diagram view similar to a UML diagram, which can illustrate classes, interfaces, modules, components, and/or systems. This capability would help developers better understand the project's overall architecture, identify dependencies, and spot potential design issues early. By offering multiple ways to visualize the code, the environment can cater to different levels of abstraction, from high-level architectural overviews to detailed code relationships, thereby improving the comprehensibility and maintainability of complex software systems.

Each code fragment should be capable of independent execution, **adhering to the principles of incremental programming and modular development**. This capability ensures that developers can run, edit, test, and analyze nodes in isolation, making identifying and addressing issues easier without affecting the entire code base. To support this modular approach, the environment should also **include**

tools for analyzing sub-graphs within the overall graph structure, allowing developers to focus on specific relationships and dependencies.

In addition to these core requirements, the environment must **include version control integration**, similar to traditional text-based environments, to facilitate collaborative work. This integration would enable seamless tracking of changes, branching, merging, and conflict resolution, ensuring that teams can work together efficiently. Shareable code is vital in software engineering teams. The environment should **ensure that code runs consistently across different machines** as done in various REPLs. This involves not only integrating with version control systems like Git but also ensuring that the state and environment in which the code runs are reproducible. This state consistency guarantees that the same code will produce the same results, regardless of the machine on which it is executed. To support collaboration and code sharing, the environment should also allow for the **easy sharing of projects and code snippets**. This can involve features for sharing code directly within the development environment (maybe even in real-time) or through external platforms, ensuring that shared code can be easily run and tested by others.

File support is necessary to integrate with version control systems, which rely on file structures to track changes and manage repositories. Therefore, the environment should **include a file manager**, similar to the file explorers found in modern IDEs, allowing developers to import, organize, and navigate files seamlessly. This integration ensures that developers can **maintain a logical structure** for their projects and leverage the full capabilities of version control systems.

To further enhance functionality, the hybrid environment should support **cross-compatibility between different development environments**. This means enabling the transformation of projects developed in the hybrid environment back **into traditional text-based formats or REPL environments**. This flexibility ensures that developers can switch between different tools as needed without losing any progress or functionality.

Another important requirement is the **ability to export projects into running software**. This involves packaging or compiling the code into executable applications, ensuring that the final product can be easily deployed and run in various environments. The environment should also allow for **experimentation with different versions of nodes**, allowing developers to select and run different versions of code fragments to compare their behavior and performance.

When considering adaptability, a **customizable interface** that allows developers to tailor their workspace to their specific preferences and needs. This flexibility is important for accommodating different workflows and enhancing productivity, drawing influence from the flexibility found in modern IDEs where users can rearrange panels, toolbars, and other elements to suit their individual styles.

Enhanced readability is another crucial requirement. The environment must incorporate features that **improve code clarity and organization**, such as **syntax highlighting**, **code folding**, and **integrated documentation (markup)**. These features, commonly found in traditional text-based environments, help developers quickly understand and navigate their code, reducing the cognitive load and potential for errors and making it easier to maintain and extend code bases.

To facilitate a smooth transition from traditional text-based IDEs to the new hybrid environment, it is essential to **support features commonly found in full-fledged IDEs**. These include advanced **search functionalities**, **code navigation tools**, and **comprehensive project management features**. By incorporating these elements, the hybrid environment can offer a familiar experience to developers, easing the learning curve and promoting adoption.

Debugging capabilities are essential for any development environment. The hybrid environment should include at least some basic debugging tools that allow developers to **set breakpoints** and **step through code execution**. These features help diagnose and fix issues efficiently, drawing from the comprehensive debugging support found in traditional text-based IDEs.

General tools to **track code metrics** are another important requirement. These tools should provide insights into various aspects of the code base, such as **complexity**, **code coverage**, and **performance**. By monitoring these metrics, developers can maintain high code quality and identify areas for improvement, ensuring that the software remains maintainable and efficient.

Expandability and modifiability are also critical requirements. The environment should be designed to **allow the addition of extensions and plugins**, enabling developers to introduce new features and functionalities as needed. This modularity ensures that the environment can evolve and adapt to new requirements and technologies over time, much like how modern IDEs support a wide range of extensions to enhance their capabilities.

By addressing these requirements, the hybrid programming environment can combine the strengths of REPLs and classical text-based development. The resulting environment would not only improve the ef-

ficiency and productivity of developers but also enhance the readability, maintainability, and shareability of complex software projects. This comprehensive approach ensures that the hybrid environment supports both the iterative, experimental nature of incremental programming and the structured, scalable needs of traditional software development.

To answer Research Question 2 (RQ2), a series of case studies will be conducted covering various key features of the proof-of-concept. These case studies will evaluate the effectiveness of the new programming environment in supporting incremental programming, exploratory programming, and managing project complexity. By analyzing the results of these case studies, we can gain insights into the strengths and weaknesses of the hybrid environment and identify areas for improvement.

3.1.2 Overview

The following is an overview of the requirements extracted above. We ranked the importance from Low, Medium, and High importance based on what we believed was appropriate. These ratings correspond to optional, would be nice to have, and mandatory requirements, respectively, when evaluating what the proof-of-concept should include.

Table 3.1: Requirements and Features (Part 1)

Requirement	Potential Feature	Description	Importance
Incremental Programming Features			High
	Incremental programming	Supports incremental programming methodology and modular development	High
	Interactive and iterative features of REPLs	Enables interactive and incremental code execution	High
Complexity Management			High
	Handle large software project complexity	Manages dependencies and complexity	High
	Comprehensive project management features	Tools for managing projects	Medium
	Track code metrics	Analyzes code complexity, coverage, and performance	Low
Code Architectural Visualization			High
	Support different architectural views	Visualize code structure in various formats	High
	Tools for analyzing graphs and sub-graphs	Analyze code relationships and dependencies	High
	Code abstraction	Abstract sub-graphs into groupings to symbolize systems or components	High

Table 3.2: Requirements and Features (Part 2)

Requirement	Potential Feature	Description	Importance
Version Control			Medium
	Version control integration	Seamless integration with tools like Git	Medium
Consistency			Medium
	Ensure consistent code execution	Reproducible results across different machines	Medium
File Management			High
	File-based data structure support	Environment stored in files with logical structure	High
	Easy sharing of projects and code snippets	Facilitates collaboration and sharing	Medium
Cross-Compatibility Export			Medium
	Cross-compatibility between environments	Convert projects between different programming environments	Medium
	Export projects into running software	Compile and package code for deployment	Medium
Extensibility			Medium
	Customizable interface	Let users customize view of the interface	Low
	Improve code clarity and organization	Features like syntax highlighting and code folding	Low
	Extensibility	Support for extensions and plugins	Medium
IDE-Like Features			Low
	IDE features	Search functionalities, code navigation tools	Low
	Debugging capabilities	Set breakpoints and step through code	Low

3.1.3 Analysis

The following is an analysis of the requirements and features extracted from the table above, along with the corresponding priority to implement in a proof-of-concept.

Incremental Programming Features [High Priority]

Incremental programming features are at the core of what IGC tries to offer developers. Coming from the ideals of REPLs, incremental programming allows the user to develop their system in a modular

fashion which is a core principle of many Agile software methodologies.

Complexity Management [High Priority]

This is everything we are trying to get from the text-based programming environments. We want to be able to handle large, complex projects and make sure they scale and are readable inside the environment.

Code Architectural Visualization [High Priority]

This refers to the graph structure of the environment to be able to handle much of the complexity needed for an average software project. Reviewing model-driven development can give me good insights into the advantages of implementing this requirement. It is an essential requirement to have as it relates directly to the readability of the project which we hypothesize solves both the complexity issue of REPL environments and the readability issue of text-based environments.

Version Control [Medium Priority]

Version control is a need for every software engineer as most software projects are done collaboratively. There are different ways to implement this because of the nature of the environment. More efficient "Diffing" algorithms exist in tree or graph-like data structures that can be implemented. However, using the standard Myers algorithm or some hybrid might be simpler due to time constraints. There are also different ways of thinking about version control, especially with incremental programming, such as the ideas expressed in the paper "Variolite" [7], however, this might be too ambitious.

Consistency [Medium Priority]

Consistency is important to be able to transfer the project to different systems and behave the same. If the behavior changes depending on, for example, the operating system, version of the software, or time of day, this destroys the confidence of the user and creates unneeded debugging time to figure out why.

File Management [High Priority]

This requirement refers mostly to the concept of being able to store the structure of the programming environment efficiently inside a file. This is vital as this relates to many other requirements such as version control and the following requirement.

Cross-Compatibility Export [Medium Priority]

These are both important to have as they allow the translation from different environments into this new proposed one, however, a proof-of-concept can exist without it. The research questions are more focused on the productivity and readability of the environment.

Extensibility [Medium Priority]

Extensibility includes customization, where the user can customize the interface to fit their own needs. This is to help with the transition from a text-based IDE to this environment. However, this is optional and low priority. What makes this requirement MEDIUM is the Extensibility. For this environment to adapt and become more versatile, it needs to be able to adapt and allow outside changes from different developers to contribute to the project.

IDE-Like Features [Low Priority]

Would be nice to have since we want to make the transition from text-based IDE to this environment seamless, however, this is not needed. Some of the features could be very quick to implement which might make it a valid feature to implement. For example, linting can easily be done with predefined "linting servers" where the only task would be to listen to feedback from the server to offer this feature. Linting features are incredibly valuable when programming, but again, not vital for a minimum viable product.

3.2 Combining the Requirements

In this section, we will review all of the key requirements and features that we have derived from the requirements above. We will discuss the importance of each requirement and feature and how they can be implemented in the proof-of-concept.

The core requirements of the new programming environment are Incremental Programming Features, Complexity Management, Code Architectural Visualization, Version Control, Consistency, and File Management. To realize these requirements, we need to implement features that create an interactive and iterative environment, handle large software project complexity, support different architectural views, integrate with version control systems, ensure consistent code execution, and manage project files effectively. We will first begin with the code organization and structure that will be used to implement these requirements. We will use a graph-based data structure to represent the code and its relationships for the proof-of-concept.

3.2.1 Graph Structure.

The graph structure will be the core of the new programming environment, representing code and its relationships in a visual format. It targets both the data structure and interface domains. We chose a graph structure as the increased dimension adds more flexibility and control over the interface. It attempts to fulfill the Complexity Management and Code Architectural Visualization requirements.

With a graph structure, we can categorize and organize different sections of a project more effectively by moving structural components in different ways. This could mean grouping nodes that relate to a specific system together or having the graph flow in such a way as to illustrate a potential workflow of the project. Whatever the design may be, the graph structure evolves from the typical linear flow of many programming environments. Think of writing code from top to bottom or creating cells in a typical REPL environment. Linear approaches can be limiting when trying to manage large software projects, as they can become difficult to navigate and understand as the project grows, causing data overload [15].

Although the graph structure in itself can be complex through overlapping elements [46] and even its own data overload issues, we believe the ability to organize through a graphical structure outweighs the issues. There is a reason that, in many professions, charts and graphs are used to represent data [47]. We try to combat data overload similarly to how many IDEs deal with it: with various features such as abstractions or compositions through data importation.

Nodes.

To build the graph structure, we need to have both nodes and edges. This section will describe how the nodes are defined. We will need to define the basic node payload that the GUI is required to handle and render the nodes. We also need to think about reproducibility. We believe that the graphs should look the exact same for every user who wants to manipulate the graph. Therefore, we will need to retain information such as node positions within the workspace and whether or not the node is selected. Finally, the nodes should be extendable. If the user wants to create a custom node, they should be able to do so by building upon another node.

Below is the definition, written in TypeScript, of the core elements needed for every node:

```typescript
type NodeData<T = any> = {
    id: string;                 // Identifier
    data: T;                    // Extendable payload
    type: string;               // Type classifier (Node Types)
    selected: boolean;          // If the node is selected
    isConnectable: boolean;     // If the node can be connected to edges
    xPos: number;               // X coordinate of the node location
    yPos: number;               // Y coordinate of the node location
};
```

Deviating from the initial node definition, we postulate that there are three essential building blocks (nodes) of any graph-structured programming environment: code, documentation, and an abstraction node representing sub-graphs. We will call the latter a “graph node” for simplicity.

The code node represents the actual code snippet or cell that the user writes. This node is where the user writes their code, and it can be executed independently. The code node is the core building block of the graph structure, representing the actual code fragments that make up the project. It is

important to keep track of the new definitions and the dependencies the code introduces, as it can potentially determine what nodes can be executed or need to be executed before others. The code node is language agnostic, meaning it can support any language that the environment is designed to work with. This flexibility allows developers to work with different languages within a similar-looking environment, making it suitable for a wide range of projects. The code node could potentially support multiple languages in a single environment, allowing developers to work on projects requiring different programming languages. However, for the proof-of-concept, we will focus on supporting a single language to simplify the implementation.

The definition of the payload of the code node is as follows:

```
interface Dependencies {    // All dependencies the code uses
    variables: string[];
    functions: string[];
    classes: string[];
    modules: string[];
}
interface Definitions {     // All new definitions the code creates
    variables: string[];
    functions: string[];
    classes: string[];
}

type CodeData = {
    code: string;           // The code payload
    scope?: string;         // Scope of the code (global by default)
    dependencies?: Dependencies;
    new_definitions?: Definitions;
};

type CodeNodeData<T = {}> = T & {
    codeData: CodeData;
};
```

To create a code node, we simply would have to pass the `CodeNodeData` type to the `data` payload for a node. Note that the `scope`, `dependencies`, and `new_definitions` are all optional parameters. This is because we often do not know this information until we actually analyze the code. For the `scope` parameter, this would be populated once an outside action puts the code node into a scope of something (*e.g.*, a "Method" node is attached to a "Class" node). For both `dependencies` and `new_definitions` parameters, this would be populated once analyzed (potentially before an execution or on `code` parameter change).

The documentation node represents the documentation or comments associated with the code. This node is used to provide additional context, explanations, or notes about the code snippet. The documentation node is essential for maintaining code readability and understanding, as it allows developers to document their code effectively. The documentation node can be linked to the code node, providing a clear connection between the code and its associated documentation. This connection ensures that developers can easily access and reference the documentation while working on the code, enhancing the overall readability and maintainability of the project helping realize the Complexity Management requirement.

The following is the payload definition of the documentation node:

```
type DocumentationNodeData<T = {}> = T & {
    documentation: string;
};
```

Documentation nodes are created by passing the `DocumentationNodeData` type to the `data` payload of a node. The documentation node is the simplest node type, but it is still extendable in case a user wants to build upon it.

The graph node represents an abstraction of a sub-graph within the overall graph structure. This node is used to group related code nodes together, representing systems, components, or modules within the project. The graph node allows developers to organize and manage different sections of the project

effectively, providing a high-level view of the code structure. By abstracting sub-graphs into graph nodes, developers can focus on specific systems or components, making navigating and understanding the project easier. The graph node is a key feature for managing project complexity and supporting different architectural views, as it enables developers to visualize the structure and relationships of their code in various forms.

The functional element to extract from a graph and import into another graph using a graph node is its execution. However, especially with exploratory programming where the user is experimenting with different executions, the user might want to execute a particular execution path that exists in different parts of a graph. To give the user full control, we should allow the user to specify which execution path they want to import. We will call this specific execution a "session". The session will be a directed path through the graph structure that represents an execution path. We will further discuss sessions in a later section (section 3.2.2).

The payload definition of the graph node is as follows:

```
type GraphNodeData<T = {}> = T & {
        filePath: string;           // Path to the graph to import
        selectedSession: string;    // Session id to use from this graph file
};
```

To create a graph node, we would pass the `filePath` and `selectedSession` parameters to the `data` payload of a Graph node. The `filePath` parameter is the path to the graph file to import, and the `selectedSession` parameter is the session ID to use from the imported graph file. The graph file that is being imported should have sessions that are already predefined. If not, there is no execution data and, therefore, no way to tell how to use the graph node.

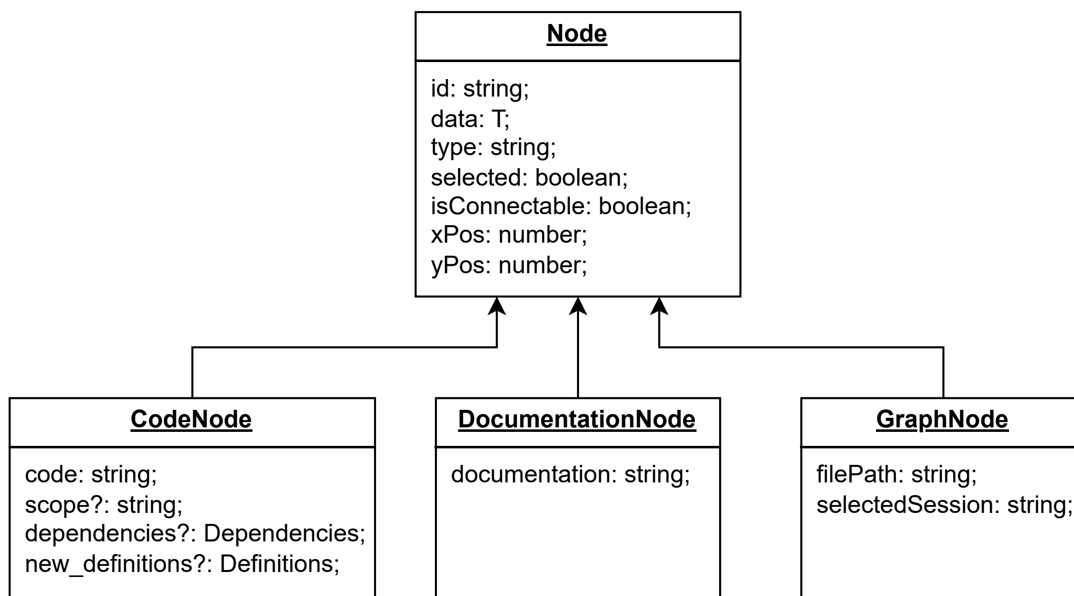


Figure 3.1: The node payloads hierarchy for the graphical programming environment. CodeNode, DocumentationNode, and GraphNode are all inherited from the base node type.

Figure 3.1 shows the relationships between the node types. All of the nodes are inherited from the base node type. This is to ensure that all nodes have the same basic structure and can be rendered in the same way. The base node type contains the core elements needed for every node. By defining a base node type, we can ensure consistency across all nodes and simplify the rendering process. The base Node type is extended by the CodeNode, DocumentationNode, and GraphNode types, which define the specific payloads for each node type. This hierarchy allows us to create different types of nodes with unique properties while maintaining a consistent structure across the graph.

Relationships (Edges).

To connect all of the building blocks together, we need to define relationships between the nodes. These relationships are represented as directed edges in the graph structure and define how different nodes are connected and interact with each other. There are several types of relationships that can be defined between nodes. Potentially there can be relationships self-loop into the same node.

The following is the definition of the core elements needed for every relationship to connect two nodes:

```
type RelationshipData<T = any> = {
    id: string;                  // Identifier
    data: T;                     // Extendable payload
    type: string;                // Type classifier (Relationship Types)
    source: string;              // Source node id
    target: string;              // Target node id
    selected: boolean;           // If the relationship is selected
};
```

One thing to note is that, unlike the node definition, the relationship definition does not have any positional data. This will be strictly inferred from the nodes themselves. Potentially, in the future, we could have the relationship definition include points that the relationship should go through, but for now, we will keep it simple.

Relationships can be used to represent various different types of meanings between nodes. Some relationships could be paradigm-specific (examples discussed later in the implementation), but some relationships are universal. For example, the execution relationship represents the ordering of code execution between nodes. This relationship defines the sequence in which code nodes are executed, ensuring that the code runs correctly and produces the expected results. A graphical representation of a codebase makes execution order unclear, as with a linear representation, the order is normally top-down.

Another universal relationship is the documentation relationship. This relationship links the documentation node to the code node, providing a connection between the code and its associated documentation. This relationship ensures that developers can easily access and reference the documentation while working on the code, enhancing the overall readability and maintainability of the project.

More relationships will be discussed in the implementation section, but the relationships are the core of the graph structure as they give more meaning to the codebase that is not normally seen in a linear representation. This is where the graph structure shines as it can represent relationships in a more meaningful way. The user is able to create custom relationships to fit the needs of their project, allowing for a more flexible and adaptable environment.

Views.

One thing to note with this new graphical organizational structure is that there is now a way to interpret more through relationships of the structure of the graph itself. This can translate to having different views than the typical text-based code editor. In an IDE, it is very difficult for the environment to interpret relationships between the code itself as relationships are not directly declared. There are instances where relationships can be inferred, such as in PescalJ[48] where they create a view directly correlating documentation a code snippets through selected definitions and references, however again, these relationships are not explicitly defined, so hard to infer. In a graphical environment, relationships can be directly declared through the graph structure.

The trivial view is the text-based code editor view. This is exactly the same as you would have in any other programming environment where the user can see the code in a text-based format. This view is essential for writing and editing code, as it provides a familiar interface for developers to work with. The text-based code editor view can support the IDE-Like Feature requirement providing features such as syntax highlighting, linting, and other features commonly found in traditional text-based programming environments. This view is the core interface for writing and editing code, providing a lightweight and distraction-free environment for developers to focus on their work.

The graphical environment differs in providing more views based on the relationships provided in the graph. For example, one main view we foresee is a view that shows the entire graph structure of the project. This view is crucial as it is the basis of the entire code organization, and it should always be shown on the user's screen. This view can be used to visualize the overall structure of the project, showing how different systems, components, and modules are connected.

From the relationships of the graphs, we can create different custom views to make meaning of the arrangements and combinations between the nodes and relations of the graph. The views can also be selectively shown depending on specific actions taken in the graph itself. For example, showing node specific views when a node is selected.

Views are not specific to a particular project but rather to the environment itself. This means that the views can be used across different projects and can be customized to fit the needs of the user. Unlike nodes and relationships, views do not need a payload to be saved to specific project data, thus only affecting the interface domain. Instead, views use the project data to render the graph structure in different ways.

When creating views, the user should be able to specify under what conditions to show the view. For example, the user might want to show a specific view when a node(s) or relationship(s) are selected. There would also need to be general views that are shown when nothing is selected. To capture this functionality, we will need to define a way to reference the nodes and relationships that the view is dependent on. The following is how we could define a view in the environment:

```
type Component = Node | Relationship;

interface ViewData {
    forComponents: Component[];
    weight: number;
}
```

The `forComponents` parameter is an array of components that the view is dependent on. If the array is empty, we will interpret this as a general view. The `weight` parameter is a number that determines the priority of the view. The higher the weight, the higher the priority of the view. The view with the highest weight will be shown first, followed by views with lower weights. This allows the user to customize the view hierarchy and ensure that the most relevant views are displayed at all times.

3.2.2 Sessions.

As the idea of execution is not as intuitive as a linear representation, we need to define a better way to handle code execution. We will define a session as a way to execute the code. A session is an ordered set of nodes that represent an execution path through the graph structure. Sessions will initialize at a null state containing no data or information. As the user selects nodes to execute, the data of each execution will incrementally build up in the session presenting a timeline of the execution to the user. This is a key feature that relates directly to the Incremental Programming Feature requirement as execution is done incrementally. The user is able to create multiple sessions to test and experiment between different execution paths.

Sessions are specific to a graph, meaning that different graphs have their own set of sessions. This is important as different graphs can represent different projects (if imported) or different parts of the same project. Sessions can be used to define different execution paths within the graph and, therefore, the project, allowing developers to test and analyze specific code fragments in isolation. Sessions can be created, saved, and loaded within the environment, allowing users to manage and organize their code execution effectively.

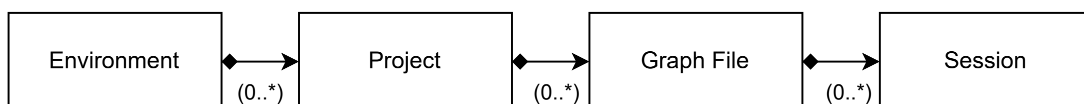


Figure 3.2: The relationship composition between the programming environment, projects, graph files, and sessions.

A session manager will be used to manage sessions within the environment. The session manager will allow developers to create and compare sessions throughout the development process, providing a way to experiment with different versions of code and analyze their behavior. This integration ensures that developers can leverage the full capabilities of the graph structure by letting the user execute in any direction of the graph they could want.

3.2.3 File Management.

The environment will support a file-based data structure to store the project files and data. A file-based environment was chosen as many developers are already familiar with file systems and version control systems that rely on file structures. This familiarity makes it easier for developers to manage and organize their projects effectively. The session data will also be expressed as files, allowing developers to save and load their work seamlessly. The file-based data structure will support the File Management requirement, enabling developers to import, organize, and navigate files effectively. The file structure will be designed to be logical and intuitive, allowing developers to maintain a structured project layout.

Similarly to many IDEs, the environment will include a file navigator to manage project files effectively. The file navigator will allow developers to import, organize, and navigate files seamlessly, ensuring that they can maintain a logical structure for their projects. The file navigator will support features such as creating new files and folders, renaming, copying, pasting, and deleting files, and moving files between folders. This integration ensures that developers can manage their project files efficiently and maintain a structured project layout just as a typical IDE would.

3.2.4 Version Control.

Version control is an important feature for any modern software project, as it enables developers to track changes, collaborate effectively, and manage project versions seamlessly.

Graph-based code structures open up the ability to utilize different version control systems. Specifically, the graph-based structure allows for more efficient "Diffing" algorithms to be implemented, such as tree-based [49] or graph-based diffing [50] algorithms. One inspiring example that was considered is the "Variolite" system [7]. This system allows for the user to track changes in an IPE that could be applied here.

However, for simplicity's sake, for the proof-of-concept, we will use a more traditional file-based version control system like Git. For simplicity, we do not provide a native way to interact with Git, but rather, the user can interact with Git through traditional methods such as through a terminal or a Git GUI referencing the project workspace. Although a better solution would be to integrate version control directly in the environment, having the capability to still use git supports the Version Control requirement. This is another reason why the file-based data structure is important, as it allows for easy integration with existing version control systems.

3.2.5 Extendability.

The environment needs to be extendable to allow the user to create anything the user might need. This is important as it allows the environment to adapt and evolve over time, supporting new requirements and technologies. The environment will support the Extensibility requirement by providing a way for developers to add custom nodes, relationships, and views.

The environment will have a custom library the user can import and use to create custom elements (nodes, relationships, views). The library has three base objects for nodes, relationships, and views that custom elements can inherit to import into the environment. This allows the user to create custom elements that fit the needs of any scenario the user might have. For example, the user might want to import the common elements of a specific language or paradigm to use in their project. The goal would be to have a global database full of these custom elements that the user can import and use in their project.

An Add-On manager will enable the user to manage and install custom elements within the environment. The Add-On manager will provide a way for developers to browse, install, and update custom elements, ensuring that they can extend the environment with new features and functionalities as needed. To streamline this, each of the base node, relationship, and view objects will inherit from a base object that will be used to import the custom elements into the environment. This will ensure that all custom elements have a consistent structure and can used in the Add-On manager in the same way. This base object will be called `RegistryComponent`. It will have the following structure:

```
interface RegistryComponent {
    key: string;                          // Identifier
    displayName: string;                  // Name of the component
    type: ``node'' | ``relationship'' | ``view''; // Type of component
}
```

The `key` parameter is a unique identifier for the component, the `displayName` parameter is the name of the component that will be displayed in the Add-On manager, and the `type` parameter is the type of component (node, relationship, or view) which will help with classifying every component that is imported.

Figure 3.3 shows the complete hierarchy of the registry system for the programming environment.

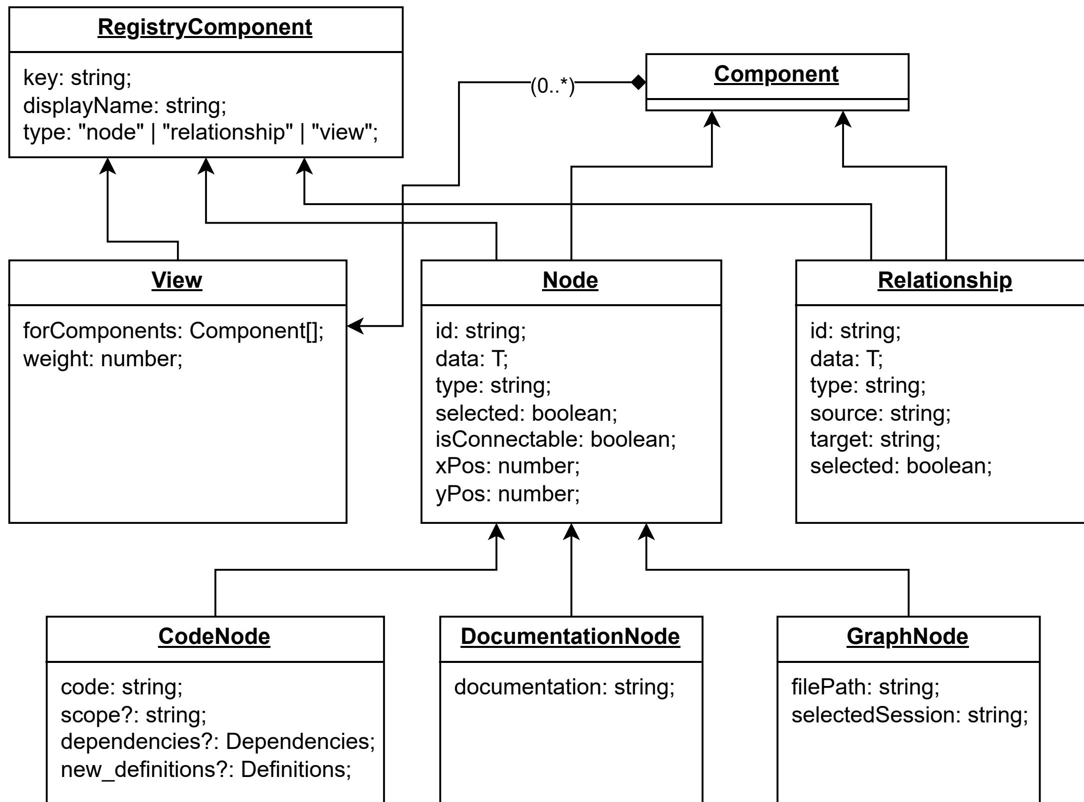


Figure 3.3: The hierarchy of the registry system for the programming environment.

3.2.6 Reproducibility

Reproducibility is a key feature of the environment, as it ensures that the code runs consistently across different machines. This is important for ensuring that the code behaves the same way regardless of the environment it is executed in. Disregarding operating system differences in how they interact with programming languages and packages, the environment will support reproducibility by providing a way to save and load project files, sessions, and custom elements that are independent of the hardware and operating system of the user. This means that the environment will look and function exactly the same for every user. This allows developers to share their work with others and ensure that the code runs consistently across different machines. This is inspired by computational notebooks and described in "Jupyter Notebooks – a publishing format for reproducible computational workflows" [2]. Reproducibility is important as it supports the Consistency requirement.

Chapter 4

Implementation

This chapter describes the implementation details of the Incremental Graph Code (IGC) environment. The chapter is divided into several sections, each focusing on a specific aspect of the implementation. The sections cover the front end and back end technologies used, the visual components of the environment, the APIs and back end functionality, software development practices, and other miscellaneous aspects of the implementation. The chapter provides an in-depth look at how the IGC environment was built, the technologies and tools used, and the design decisions made throughout the development process.

First, it is best to define some terms that will be used throughout the section and onward:

- **IGC/IncrGraph/Incremental Graph Code:** The proposed graphical programming environment.
- **Node:** The physical nodes of the graph.
- **Node Type:** The type of node that is being represented. This could be a class, method, or other element of the project.
- **Edge:** The physical edges of the graph.
- **Relationship:** The edge type that is being represented. This could be a dependency, inheritance, or other relationship between the nodes.
- **View:** A view that provides a representation of a set of nodes and edges.
- **IGC File:** A JSON file containing all the data of a graphical code representation formatted in a way that IGC can handle. These are the main files that are used by IGC.
- **Component:** A defined node type or relationship. Note that when we use the term "visual component," we specifically mean what front-end frameworks typically call a *display component* (e.g., a React component), which differs from the usage of the term "component" in our context.
- **IGC Component:** A defined node type, relationship, or view.
- **Registry:** A database of IGC Components that are available for the user.
- **Module:** A set of IGC Components inside the Registry.
- **Session:** A particular execution path for a given IGC file.
- **State:** A current snapshot of a step of the execution of a particular session.
- **Configuration:** All currently defined variables at a state.

4.1 General Technology / Tech Stack

4.1.1 Mediums to present the environment

There were several considerations on the platforms or mediums to present the programming environment. The following are several that were considered and the rationale behind every choice.

Visual Studio Code (VSCode) Extension. VSCode is a widely used, open-source code editor developed by Microsoft. It offers a community-based ecosystem of extensions and plugins, making it a popular choice among developers. Considering its popularity and familiarity among users (as evidenced by the StackOverflow developer survey of 2024 where over 73% of participants reported to using it[1]),

[1] https://survey.stackoverflow.co/2024/

creating a VSCode extension was initially considered as the primary option for presenting the new programming environment.

However, developing an extension for VSCode has certain limitations in customizing the user interface and controlling user interactions. The extension would be constrained by the APIs and capabilities provided by the VSCode platform, which may not allow the level of control required for this project. In addition, VSCode's extension framework is primarily geared towards enhancing code editing functionalities rather than building entirely new interactive experiences. This could restrict the implementation of specialized features essential for the project's goals.

While converting the environment into a VSCode extension is a potential future direction, we opted to develop an independent application for now. This choice allows us full control over the environment without the need to navigate the learning curve associated with VSCode extension development.

Software-based Application. Developing a standalone software application was considered a potential option for presenting the programming environment. This approach would provide complete control over the application's features and user interface, enabling the creation of a customized experience tailored to the project's specific requirements.

However, creating a software-based application has challenges regarding operating system independence, which is a key requirement for the project. Maintaining compatibility across different operating systems, such as Windows, macOS, and Linux, would increase the development complexity and resources needed. This diversity could lead to inconsistencies and complicate the support and maintenance processes.

Additionally, requiring users to install additional software could pose a barrier to entry. Users might be reluctant to install new applications due to concerns about security, system performance, or simply the inconvenience it entails. This could limit the accessibility and widespread adoption of the programming environment.

Given these considerations, we decided that a standalone software application may not be the optimal choice. Instead, we looked towards solutions that are platform-independent and do not require installation, leading us to consider web-based applications.

Web-based Application. Developing a web-based application emerged as the chosen option for presenting the programming environment. Web applications offer platform independence, allowing users to access the environment through a browser without the need for installation. This enhances accessibility and lowers the barrier to entry, as users can engage with the environment from any device with internet connectivity.

However, standard web applications face limitations in accessing the user's local file system due to browser security restrictions. This poses challenges for functionalities such as selecting directories, opening projects, or importing packages. To address this limitation, we adopted a hybrid approach that incorporates native capabilities for file system access (later discussed).

By integrating these functionalities, the environment can provide users with the necessary features while maintaining the benefits of a web-based platform. This approach allows the environment to be used both as a cloud-based application, similar to Jupyter Notebook, and as a native, operating system-independent application. This dual capability enhances flexibility and user experience, catering to different preferences and use cases.

Electron-based Application. Electron is an open-source framework that enables the development of cross-platform desktop applications using web technologies such as HTML, CSS, and JavaScript. By combining Chromium and Node.js into a single runtime, Electron allows developers to build applications that run seamlessly on Windows, macOS, and Linux without the need to modify code for each platform [51].

Using Electron for the programming environment offers several advantages. It retains the benefits of a web-based application—such as ease of development and a rich ecosystem of libraries—while providing access to native system functionalities. This includes features like directory selection, file system operations, and integration with local resources, which are essential for a full-fledged programming environment.

Electron applications can bypass the limitations imposed by standard web browsers regarding local file system access. This is crucial for functionalities that require reading from or writing to the user's file system, interacting with hardware, or performing other operations that generally need permissions beyond what a web application can obtain.

Electron allows for a smooth user experience with a native application feel. It supports extensively customizing the user interface, making it possible to design an environment tailored to the project's specific needs. Developers can leverage familiar web development tools and frameworks to build sophisticated interfaces quickly.

4.1.2 Front End Technologies

There are many different front end-based technologies to choose from. The following are a couple of the ones that were considered, as well as the reasoning behind every choice.

Vanilla JavaScript. Vanilla JavaScript refers to the use of plain JavaScript without any additional libraries or frameworks. While this approach offers complete control over the codebase and minimal overhead, it can be challenging to manage complex applications due to the lack of structure and organization provided by libraries or frameworks.

jQuery. jQuery is a fast, small, and feature-rich JavaScript library that simplifies HTML document traversal and manipulation, event handling, and animation [52]. It provides a concise syntax for common tasks, making it easier to write JavaScript code that works across different browsers. However, jQuery's popularity has waned in recent years due to the rise of modern JavaScript frameworks and libraries.

Similarly to Vanilla JavaScript, jQuery lacks the structure and organization provided by modern frameworks, making it less suitable for building complex applications. While it can be useful for simple interactions and animations, it may not be the best choice for developing a comprehensive programming environment.

Angular. Angular is a TypeScript-based open-source web application framework developed by Google. It provides a robust structure for building single-page applications and offers features such as two-way data binding, dependency injection, and modular design [53]. Angular's extensive ecosystem includes tools for routing, forms, HTTP clients, and more, making it a comprehensive solution for large-scale applications [54].

Angular's approach to development can be both a strength and a limitation. While it enforces best practices and consistency, it can also introduce a steep learning curve for developers new to the framework. Angular applications tend to be more complex due to the framework's architecture, which may not be necessary for smaller projects or applications with simpler requirements.

React. React is a JavaScript library maintained by Meta for building user interfaces. It focuses on component-based architecture, allowing developers to create reusable UI elements that manage their state independently. React's virtual DOM and efficient rendering process make it performant for complex applications, and its declarative approach simplifies UI development [55].

React's popularity has grown significantly in recent years due to its flexibility, performance, and strong developer community, making it the most popular web front end library according to the 2024 StackOverflow developer survey (footnote 1). React uses the Flux Architecture instead of the traditional MVC pattern. This architecture provides a unidirectional data flow, which helps with performance, specifically with component-based applications, and eliminates the complexity where many sources try to change the same data [55]. Flux does have its own set of challenges; as there is less access to the data store from anywhere in the application, it can be difficult to track changes and debug issues.

To address these challenges, React applications sometimes have data flow libraries like Redux or Zustand. These libraries help manage the state of the application and make it easier to track changes and debug issues.

Overall, due to React's flexibility, performance, and strong developer community, it was chosen as the primary front end technology for the programming environment.

Vite. Vite [2] is a modern build tool that focuses on speed and simplicity. It leverages native ES modules to provide fast, efficient builds without the need for bundling. Vite supports various front end frameworks, including react, and offers features like hot module replacement, optimized builds, and a built-in development server. Vite was chosen as the build tool for the programming environment due to its speed, simplicity, and previous experience we had.

[2] https://vite.dev/

Redux vs Zustand. Redux [3] and Zustand [4] are both state management libraries for React applications. Redux is a predictable state container, while Zustand is a small, fast, and scalable state management library based on hooks. Both libraries offer solutions for managing application state in React components, but they differ in their approach and complexity. For simplicity and performance, Zustand was chosen as the state management library for the programming environment. Zustand's minimalistic API and hook-based design make it easy to integrate and use, providing a lightweight solution for managing state in React components.

ReactFlow. ReactFlow [5] is a library for building node-based applications, such as flowcharts, diagrams, and visual programming interfaces. It provides a set of components and utilities for creating interactive graphs with nodes and edges, allowing users to visualize and manipulate data in a structured manner. ReactFlow's flexible API and customizable features make it well-suited for developing the visual components of the programming environment. We chose ReactFlow for its ease of use, extensibility, and compatibility with React applications.

ReactFlow provides an infinite, pannable workspace to create graphs. This allows the user to create any size project, as the workspace can always be extended. A limitation is that to capture the entirety of a large project, the user must zoom out, which will eventually make text hard to read and elements difficult to see or distinguish.

Matirial-UI. Material-UI [6] is a popular React UI framework that provides a set of components and styles based on Google's Material Design guidelines. It was used over other component libraries, such as Ant Design or Chakra UI, as we had prior experience with it and it offers a wide range of components and customization options. Material-UI offers a wide range of pre-built components, icons, and themes for creating modern, responsive user interfaces. Material-UI's modular design and customization options make it a great choice for designing the visual components of the programming environment.

Front End Libraries Used

The following is a list of every library used on the front end:

Table 4.1: All Dependencies (Production + Development), Excluding `@types` Libraries

Library	Version	Purpose / Need
`@emotion/react`	11.11.4	CSS-in-JS styling for React components.
`@emotion/styled`	11.11.5	Styled-components API built on Emotion.
`@monaco-editor/react`	4.6.0	React wrapper for the Monaco code editor.
`@mui/icons-material`	6.0.2	Material UI icons for React.
`@mui/material`	6.0.2	Material UI core components for React.
`@mui/x-tree-view`	7.15.0	Tree view component for MUI (hierarchical data).
`@reduxjs/toolkit`	2.2.7	Simplifies Redux store setup and usage.
`@xterm/addon-fit`	0.10.0	Xterm.js addon for dynamically resizing terminals.
`@xterm/xterm`	5.5.0	Terminal emulator for browser-based apps.
`axios`	1.7.7	Promise-based HTTP client for browsers/Node.js.
`js-cookie`	3.0.5	Utility for handling browser cookies.
`lodash`	4.17.21	Utility library for data manipulation, etc.
`path-browserify`	1.0.1	'path'-like utilities for use in the browser.
`react`	18.3.1	Core React library for building UIs.

Continued on next page

[3] https://redux.js.org/
[4] https://zustand-demo.pmnd.rs/
[5] https://reactflow.dev/
[6] https://mui.com/

Continued from previous page

Library	Version	Purpose / Need
`react-dom`	18.3.1	DOM-specific methods for React app rendering.
`react-draggable`	4.4.6	Adds draggable functionality to React elements.
`react-json-view`	1.21.3	JSON viewer/editor React component.
`react-markdown`	8.0.6	Renders Markdown as React components.
`react-resizable`	3.0.5	Resizable container component for React.
`react-router-dom`	6.23.1	Declarative routing in React (for browsers).
`reactflow`	11.11.3	For building flow diagrams / node-based editors.
`shared`	file:../shared	Local shared code for cross-project usage.
`vis-network`	9.1.9	Graph/network visualization in the browser.
`zustand`	4.5.4	Small, fast state-management for React.
`@typescript-eslint/eslint-plugin`	7.8.0	ESLint rules for TypeScript-specific linting.
`@typescript-eslint/parser`	7.8.0	ESLint parser for analyzing TypeScript code.
`@vitejs/plugin-react`	4.2.1	Vite plugin enabling React fast refresh, etc.
`@vitejs/plugin-react-swc`	3.6.0	SWC-based React plugin for faster Vite builds.
`eslint`	8.56.0	Pluggable JS/TS linter and code-quality tool.
`eslint-plugin-react-hooks`	4.6.2	ESLint rules enforcing correct usage of React Hooks.
`eslint-plugin-react-refresh`	0.4.7	ESLint integration for React Refresh in dev.
`typescript`	5.4.5	Superset of JavaScript that adds static typing.
`vite`	5.2.11	Fast dev server and build tool for modern web apps.

4.1.3 Back End Technologies

There are many different back end-based technologies to choose from. The following are a couple of the ones that were considered and the reasoning behind every choice.

Flask/Django. Flask and Django are popular Python web frameworks used for building web applications. Flask is a lightweight, micro web framework that provides the essentials for web development, while Django is a full-featured framework with built-in components for authentication, database management, and more. We have experience in both Flask and Django, but these frameworks add another programming language to the project, which could complicate the process for new IGC developers.

Node.js. Node.js is an open-source, cross-platform runtime environment for executing JavaScript code outside a web browser. It allows developers to build server-side applications using JavaScript, providing a unified language for both client-side and server-side development. Node.js is known for its event-driven, non-blocking I/O model, which makes it lightweight and efficient for handling concurrent requests. Given our experience with Node.js, the wide community support, and the unified language approach, we chose Node.js as the back end technology for the programming environment.

Express. Express is a minimal and flexible Node.js web application framework that provides a robust set of features for building web and mobile applications. It offers a simple, unopinionated design that allows developers to create APIs and web servers quickly and easily. Express is widely used in the Node.js community due to its simplicity, performance, and extensibility. We chose Express as the back end framework for the programming environment due to its lightweight nature, ease of use, and compatibility with Node.js.

Back End Libraries Used

The following is a list of every library used on the back end:

Table 4.2: All Dependencies for the Back End (Production + Dev)

Library	Version	Purpose / Need
`async-mutex`	0.5.0	Provides mutex/locking primitives for async usage.
`body-parser`	1.20.2	Parses incoming request bodies in Express.
`cors`	2.8.5	Enables Cross-Origin Resource Sharing in Express.
`dotenv`	16.4.5	Loads environment variables from a `.env` file.
`dotenv-expand`	11.0.6	Expands variable references in `.env` files.
`express`	4.19.2	Minimalist web framework for Node.js.
`fs-extra`	11.2.0	Extra filesystem methods beyond Node's built-in `fs`.
`shared`	file:../shared	Local shared code for cross-project usage.
`uuid`	10.0.0	Generate UUIDs (v1, v4) per RFC4122.
`nodemon`	3.1.4	Auto-restarts Node app on file changes.
`ts-morph`	23.0.0	TypeScript compiler API wrapper for AST manipulation.
`ts-node`	10.9.2	Run TypeScript directly without pre-compiling.
`typescript`	5.4.5	TypeScript language and compiler.

4.1.4 Common Technologies

The following are technologies that are used in both the front end and back end:

TypeScript. TypeScript is a superset of JavaScript that adds static typing and other features to the language. It helps developers catch errors early in the development process and improve code quality by providing type-checking and code analysis tools. TypeScript transpiles to plain JavaScript, making it compatible with existing JavaScript code and libraries. It is widely used in modern web development projects due to its ability to catch run-time bugs through string typing.

ESLint. ESLint is a pluggable JavaScript linter that helps identify and fix problems in code. It enforces coding standards, catches common errors, and improves code quality by analyzing JavaScript and TypeScript code. ESLint is widely used in modern web development projects to maintain consistency, readability, and best practices across codebases.

Prettier. Prettier is a code formatter that automatically formats code to ensure consistency and readability. It supports various programming languages, including JavaScript, TypeScript, HTML, CSS, and more. Prettier integrates with popular code editors and build tools, making it easy to enforce code style conventions and improve collaboration in development teams.

4.2 Other Specifications

4.2.1 Directory Structure

The directory structure of the project is listed at fig. 4.1.

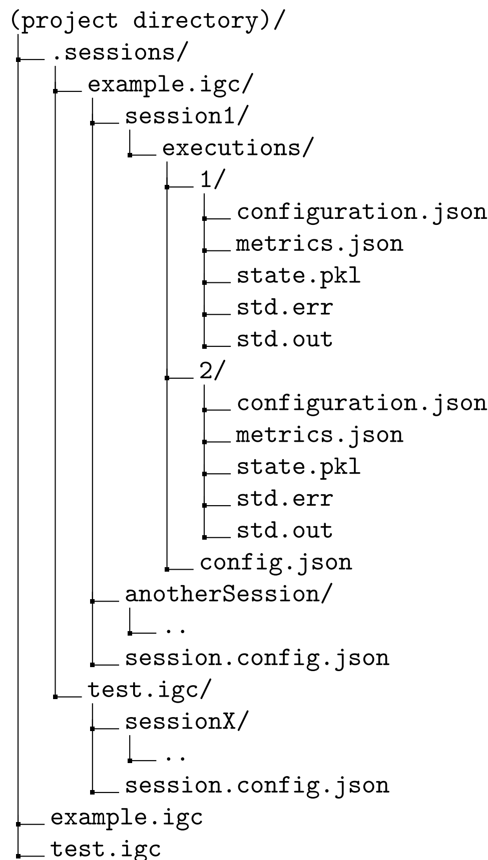


Figure 4.1: IGC Directory Structure

4.2.2 IGC File

The IGC file is a JSON file that contains all the data of the graphical code representation for a particular graph. There can be many IGC files in a project, each representing a different part of the project. The basic IGC file is structured as follows:

```
{
    "nodes": [...],
    "edges": [...],
}
```

The `nodes` array contains all the nodes in the graph, and the `edges` array contains all the edges in the graph. Since IGC uses ReactFlow, for simplicity's sake, the node and edge objects are exactly the same as the definitions from ReactFlow. This means that IGC would not have to recreate all the basic functionality that ReactFlow provides and can instead use the objects directly.

An example of a node object is shown below:

```
{
    "id": "0",                      // Unique id of node
    "type": "codeFragmentNode",     // Type of node
    "data": {                       // IGC specific data
        "label": "Import Libraries",
        "code": "import pandas as pd\nfrom datetime import datetime",
        "metaNodeData": {
            "dependencies": {
                "variables": [],
                "functions": [],
                "classes": [],
                "modules": [
                    "pandas",
```

```
                    "datetime"
                ]
            },
            "new_definitions": {
                "variables": [],
                "functions": [],
                "classes": []
            }
        }
    },
    "position": {                       // Position of node on graph
        "x": -60.76447523242331,
        "y": -117.91026780703368
    },
    "selected": false,                  // Whether the node is selected
    "width": 154,                       // Width of node
    "height": 84,                       // Height of node
    "dragging": false                   // Whether the node is being dragged
}
```

The node above is a *Code Fragment* node. Since ReactFlow does not allow for custom attributes directly in the object, all IGC-specific node attributes are stored in the generic `data` attribute. We can see that the `label` attribute is the name of the node, the `code` attribute is the code that the node represents, and the `metaNodeData` attribute is the IGC-specific analyzed data that is used for dependencies.

An example of an edge object is shown below:

```
{
    "id": "execution-0-start>2",        // Unique id of edge
    "source": "start",                  // Source node id
    "target": "2",                      // Target node id
    "type": "ExecutionRelationship",    // Type of edge
    "data": {                           // IGC specific data
        "label": "1"
    }
}
```

The edge above is an *Execution* relationship. Similar to the node object, all IGC-specific edge attributes are stored in the generic `data` attribute. We can see that the `label` attribute is the edge label which, in this case, represents the execution number of a session.

4.2.3 Language Support

IGC attempts to be language agnostic meaning that it can be used with any programming language. The implementation currently only supports Python, but the system is designed to be able to handle any future language. Every REST API related to code execution and analysis requires a language parameter. This parameter is used to determine which language the code is written in and how to handle it.

On the back end, the language parameter is checked with available language handlers. Since the environment is a prototype and is still in its infancy stage, there is just a switch statement (only containing Python), which then will point to a Python binary file on the system. Every user that wishes to use IGC needs to customize IGC to point to the correct binary file for the language they wish to use. Currently, the python path points to `IncrGraph/languages/python/venv/bin/python`. To import libraries, the user has to manually go to the virtual environment at this path and install the necessary libraries. This is a limitation of the current implementation, but in the future, the system will have a language management system that will allow users to add, remove, and interact with languages as they wish. This topic is important and is further discussed in section 6.3.1.

4.2.4 Execution and State Management

In traditional incremental programming environments, the state of the program is stored in memory as an ongoing REPL instance. This allows for quick access to the state and fast execution of code.

However, this approach has limitations when dealing with large projects, shareability, and exploratory programming. As the state grows, the memory usage increases, leading to performance issues and potential crashes. Additionally, the state is lost when the program is closed, requiring the user to re-execute the code to restore the state. This can be time-consuming and frustrating, especially when working on complex projects or debugging issues. With exploratory programming, having the execution state saved in memory can lead to exponential growth as the user could potentially be branching into different execution paths. This can quickly overwhelm the system and make it difficult to manage the state effectively.

To address these challenges, IGC takes a different approach to execution and state management. Instead of storing the state in memory, IGC saves the state to disk as files. This allows the state to persist across sessions, enabling users to resume their work without losing progress. By saving the state to disk, IGC can handle larger projects and exploratory programming as disk space is typically more plentiful than memory. This also allows the user to export and share the state with others, which is not possible with a pure memory-based system. The state files are stored in a structured format that captures the configuration, metrics, and output of each execution step. All execution and state data is located in the same directory as the IGC file under a directory named `.sessions/<IGC File Name>` (the structure is shown above in fig. 4.1).

To be realistic, any approach to manage state with exploratory programming will lead to exponential growth with every branch from the execution path. In the future, it would be beneficial to have a system that can manage this growth effectively. This is further discussed in section 6.3.1.

Another issue is that every programming language has its own implementation of saving state and execution management to the disk. In Python, this was done using the `dill` module (an improvement of the `pickle` module), which is a module that serializes and deserializes Python objects. This is something that would favor using a live memory REPL system, as the user would not have to worry about the implementation of saving state to disk. This is further discussed in section 6.3.1.

4.2.5 Dependency Tree Creation

One feature of IGC is the ability to analyze code and extract a dependency tree from the code snippets. IGC is a great environment to use when displaying these dependencies as it already is a graphical structure that can display trees within the graph itself. The dependency tree is created by analyzing the code of each node and determining the new definitions that the code creates and the dependencies that the code uses. From this information, dependency relationships are created matching a new definition to a dependency.

To extract the new definitions and dependencies, the code is analyzed using the Python AST (Abstract Syntax Tree) module. The AST module parses the code and creates a tree structure that represents the code's syntax. The AST is then traversed to extract the necessary information for creating the dependency tree. The new definitions are extracted by identifying variable, function, and class declarations in the code. The dependencies are extracted by identifying variable, function, class, and module references in the code.

The intention behind the dependency tree is to give the user more insight over what they are executing. For example, if the user attempts to run a node that needs specific dependencies that have not been executed, IGC could give the user a warning that the dependencies are missing. This is a feature that is not yet implemented, but is a potential future feature that could be added to IGC.

One downfall of the current implementation is that the analysis code is specific to Python. Although most languages have AST modules, the analysis code would need to be rewritten for each language. This alone is already a problem since the analysis code is not trivial and would require a lot of work to implement and ensure that it is correct. This is further discussed in section 6.3.1.

4.3 Front End Visualizations

4.3.1 File Explorer

The File Explorer is the far-most left column of the environment. It contains both the File Navigator and the Session Manager. Below is a description of each component:

File Navigator:

The File Navigator is responsible for file management and navigation. It allows the user to create new files, open existing files, and navigate through the file system. The File Navigator is shown in fig. 4.2.

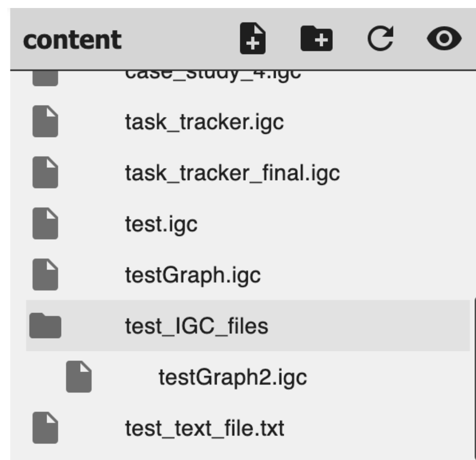
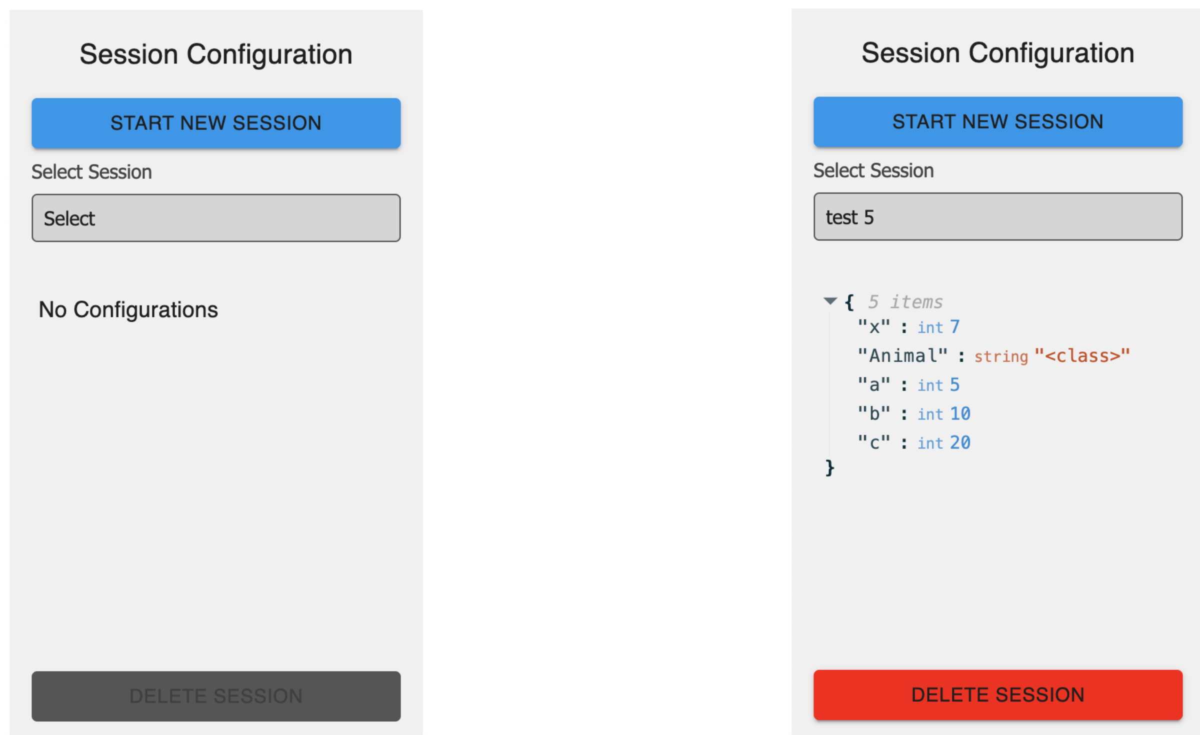


Figure 4.2: File Navigator.

At the top portion of the file navigator (from left to right) are buttons to create a new file, create a new directory, refresh the file navigator, and hide the entire file explorer. If a file or directory is right-clicked, a context menu will appear with options to open, rename, copy, paste, delete, copy the path, or copy the relative path.

Session Manager:

The Session Manager is responsible for all the sessions currently available in the current IGC file. It allows the user to create new sessions, delete sessions, and set the primary session. The primary session is the session that is currently being viewed by the user. The session manager also allows the user to view the global configuration of everything that has been executed. The session manager is shown in fig. 4.3.



(a) Session manager on an IGC file that has no execution data.

(b) Session manager on an IGC file that has execution data.

Figure 4.3: Session Manager.

Sessions are essential to the environment as they allow the user to view the state of the project at different points in time. This is especially useful when debugging or trying to understand the flow of the project. For a particular session, the user can go through each Node that has been executed and view the configuration at that point in time.

4.3.2 Graph Editor

The graph editor is the main component of the environment and is located in the middle of the screen. It is where the user can view and manipulate the graph. The graph editor is divided into several components:

Graph Toolbar:

At the top of the graph editor is the graph toolbar. From left to right, there are buttons to add a new node, filter the relationships shown, re-execute the current session, debug (currently used to generate all dependencies), and automatically navigate back to the origin of the graph.

Graph Container:

The graph container is where the graph is displayed. The graph is interactive and allows the user to zoom in and out, pan around, and select nodes and edges. The graph is rendered using the ReactFlow library, which provides a flexible and customizable API for building node-based applications. The graph editor is shown in fig. 4.4.

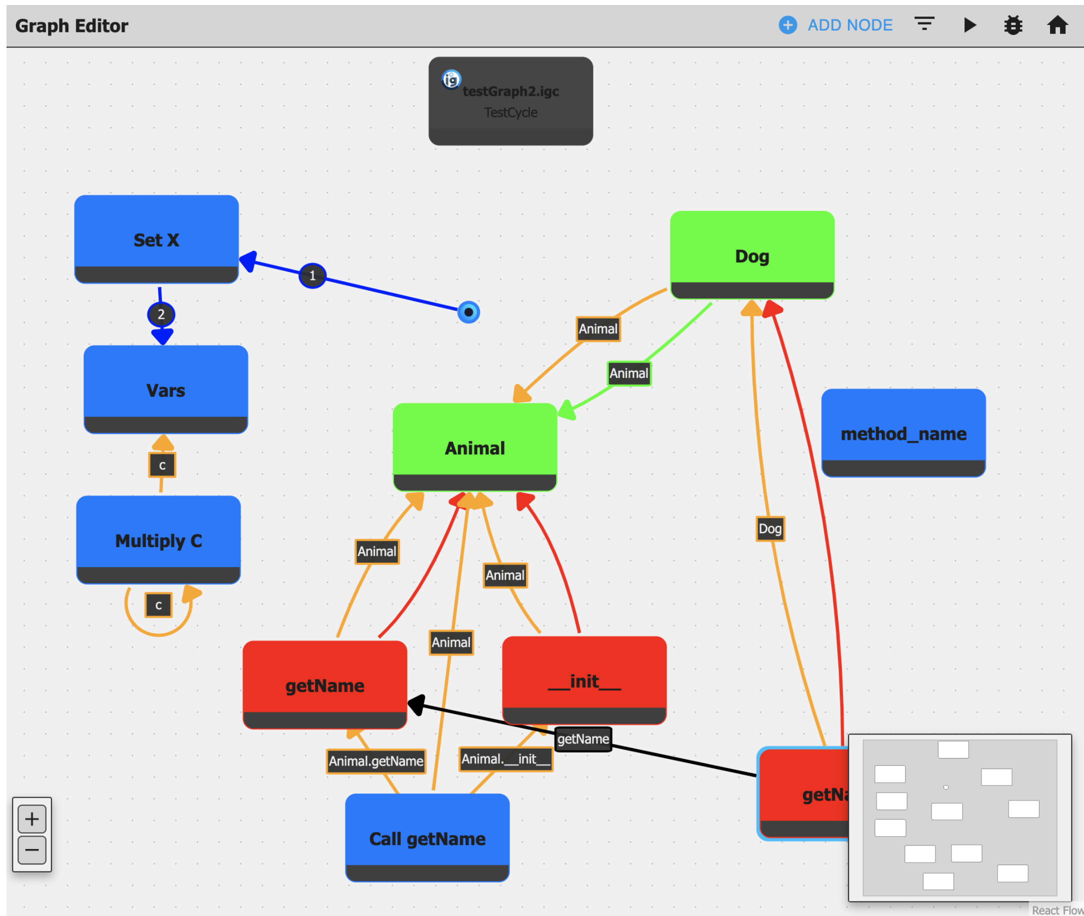


Figure 4.4: The Graph Editor.

On the bottom left of the graph container is a zoom slider that allows the user to have granular control to zoom in and out of the graph. On the bottom right of the graph container is a minimap of the current view of the graph. This allows the user to see the entire graph at a glance and navigate to different parts quickly.

4.3.3 File Editor

The file editor is on the right side of the environment. It is the container that displays all the IGC views. By default, the General view is shown which contains the raw data of the file which is open. Views are shown based on the usage of the environment. The view is dependent on the file selected in the file navigator and any selections made in the graph container. The top toolbar serves as a way for the user to interact with the view, such as executing code or checking the save status. Below the toolbar is the view tab selector which displays all relevant views. As the visuals of the file editor are always changing, section 4.3.6 will describe all the views that are currently available in IGC.

File Editor Toolbar Attachments:

The file editor toolbar contains components that can be attached to the view. The components are attached based on the view that is currently being shown. Currently, there are two attachments:

Save Indicator:
The save indicator is a small circular icon. It indicates whether the file has been saved or not. If the save indicator is green, the file has been saved. If the save indicator is orange, the file has not been saved. Finally, if the save indicator is red, there was an error saving the file.

Execution Button:
The execution button is a play button icon. It is used to execute the code that is currently being shown in the view. The execution button is only enabled when there is an active session in the

IGC file and there is valid executable code (nonempty). If either of these conditions is not met, the execution button will be disabled.

File Editor Tabs:

All views are ordered by a given weight defined by the views.

The following describes all of the components that are customizable and vary based on the application state.

4.3.4 Nodes

Base Node
The default node includes the basic node rendering process that most nodes utilize (the Start Node is an exception).

Code, Documentation, Graph Nodes
The foundational building blocks from which all other nodes are derived.

- Code Node is *abstract*—it cannot be instantiated directly, only inherited from (see Code Fragment node).
- Documentation Node represents textual or descriptive content associated with another node.
- Graph Node is a fundamental type for representing graphs or graph-like structures.

Code Fragment Node
A concrete node derived from the abstract Code Node. It specifically represents a piece of code.

Start Node
A special, symbolic node representing the initial execution state. It can be considered to have a "null" state. Any execution relationships begin at this node.

Abstract Class, Class, Interface, Library, Method Nodes
These belong to an object-oriented (OO) module and represent typical OO artifacts:

- *Abstract Class* is a class that cannot be instantiated directly.
- *Class* is a standard OO class type.
- *Interface* defines a set of methods without implementations.
- *Library* is a set of related classes, methods, or utilities that are typically imported.
- *Method* is a callable function within a class.

4.3.5 Relationships:

Base Relationship
The default type of relationship (similar to the Base Node).

Documentation Relationship
Connects a Documentation Node to another node, signifying reference or annotation.

Execution Relationship
Establishes an execution path between nodes. Often created automatically when a node is executed.

Dependency Relationship
Symbolizes that one node depends on another for its execution. Often generated automatically if, for instance, a node uses a variable defined elsewhere.

Inheritance, Overrides, and Method Relationships
These belong to the object-oriented module:

- *Inheritance* is used when a class extends another class.
- *Overrides* occurs when a subclass method overrides a method from its parent class.
- *Method Relationship* attaches a method node to a class node.

4.3.6 Views

Views are the primary way that the user interacts with and understands the environment. They are the visual representations of the data that is stored in the IGC file. Views are dynamic and change based on the state of the environment. Before describing the views, it is best to go over some commonly used visual components that are used in the views:

Documentation Display
The documentation display is a visual component that is used to display documentation. The documentation itself comes from a documentation node/relationship combination directed at the current node.

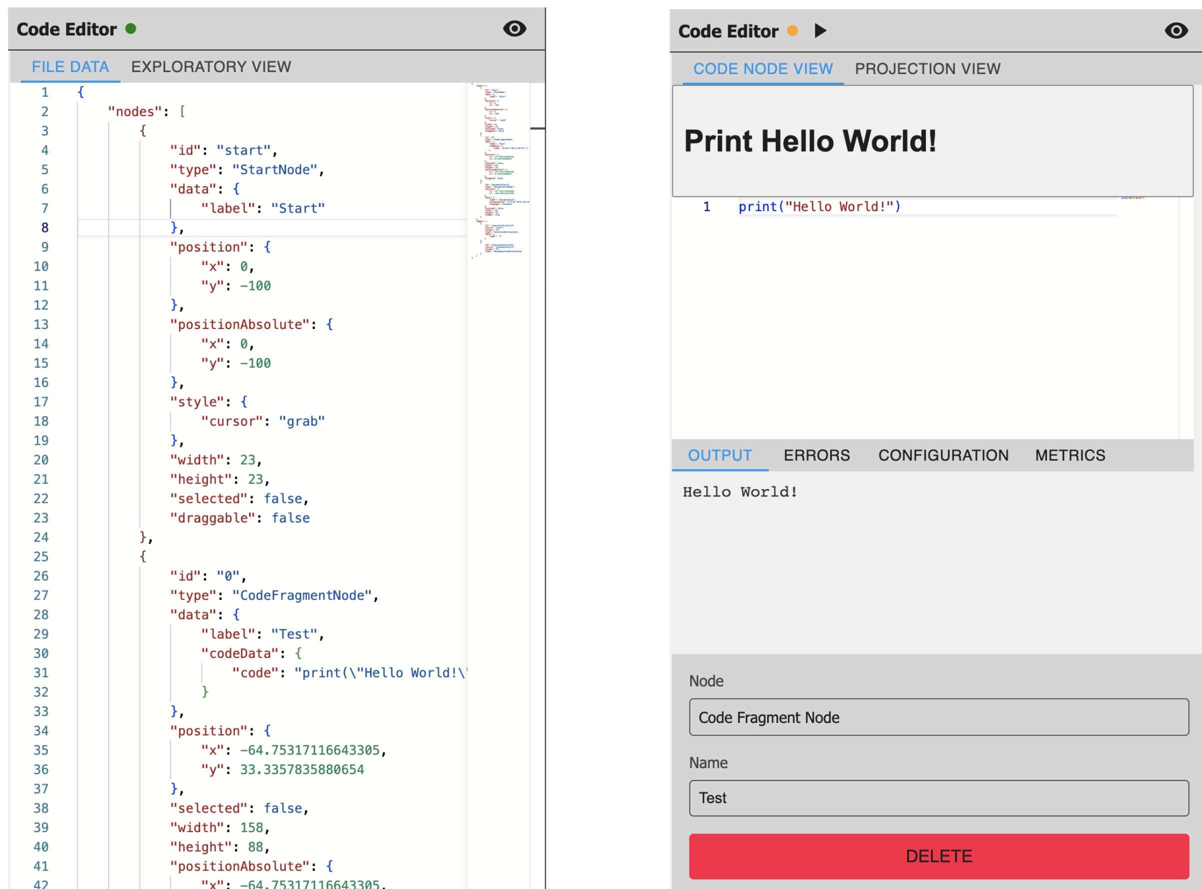
If there is no documentation, the documentation display will be empty with a "plus" icon. If this icon is double-clicked, a new documentation node/relationship will be created and attached to the current node, after which the corresponding documentation node will be opened in the documentation view to allow the user to add documentation.

If there is documentation, the documentation display will show the relevant documentation in a markdown format. If the documentation is double-clicked, the documentation view will be opened with the corresponding documentation node.

Monaco Text Editor
The Monaco Text Editor is a visual component that is used to display text and code. It is a code editor that is used in many popular code editors such as Visual Studio Code. The Monaco Text Editor has many features such as syntax highlighting, code completion, basic linting, search functionality, and more. The editor keeps track of history and allows the user to undo and redo changes. When a file is opened in the file navigator, the editor automatically detects the file type to provide the correct syntax highlighting and formatting.

Component Selector
The component selector is a visual component that is used to select a component to be displayed in the view. It normally is available if a node and/or relationship is selected. If multiple nodes and/or relationships are selected, the component selector will show the corresponding list of components to choose from to mark as the active component the views will use. The component selector also has a text field to set the name of the component. Next, there is a toggle to view and select the type of component (node or relationship). Finally, there is a button to delete the selected component.

Figure 4.5 shows all the views that are currently available in IGC. The following are the descriptions of each view:



(a) The General View. (b) The Code View.

Base

The base view is an abstract view that all other views inherit from. It contains the basic structure of a view as well as capturing which components to trigger the view to display.

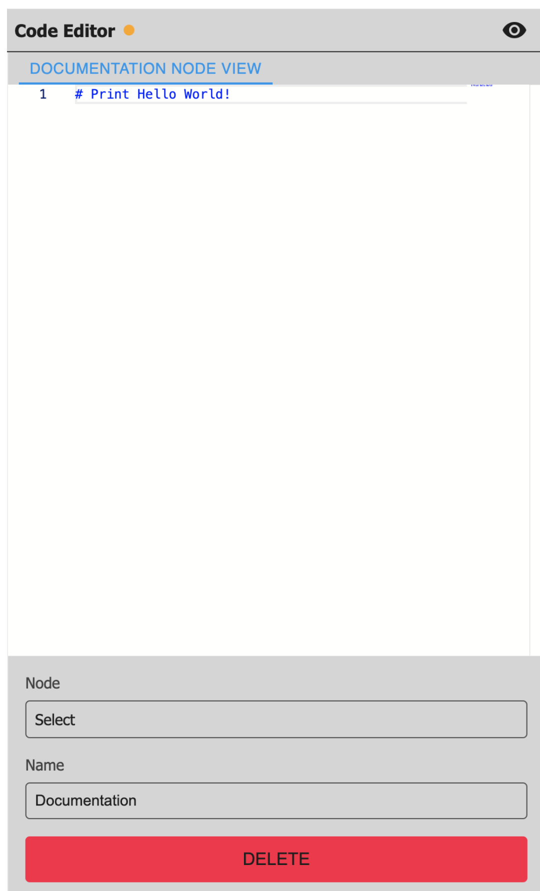
General

The general view is the default view that is shown when a file is opened. It contains all the raw text of the file. Most of the space in the view is taken up by the Monaco Text Editor to display the text or code. The general view is shown in fig. 4.5a. This view is useful for also viewing non-IGC files such as native code files. The general view attaches the *Save Indicator* visual component to the view.

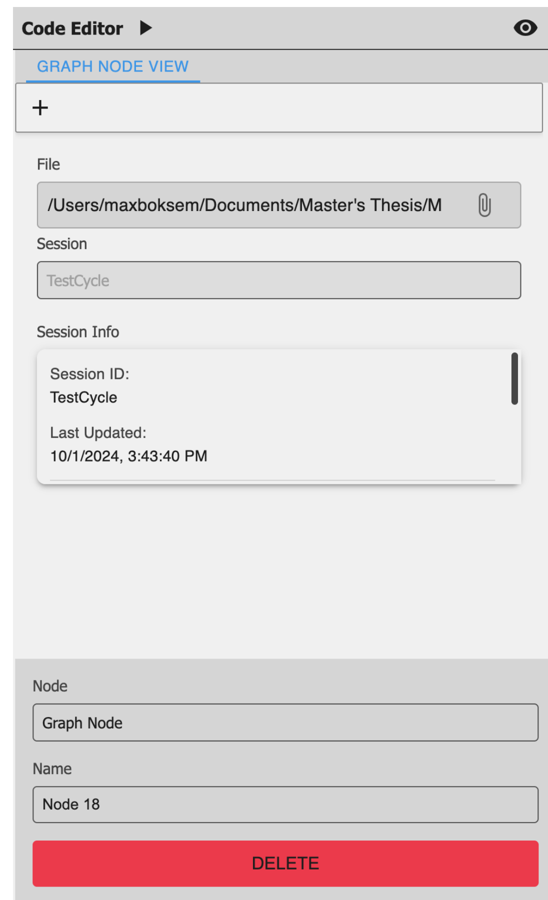
Code Node

The code view is a view that is shown when a code node is selected and is shown in fig. 4.5b. The code node view consists of the documentation display, the Monaco Text Editor, and the component selector. The code node view attaches the *Save Indicator* and the *Execution Button* visual component to the view.

If a code node is executed, it will have a tabbed code output visual component. The tabbed code output has four tabs: *Output*, *Errors*, *Con guration*, and *Metrics*. The *Output* tab shows the standard output of the code that was executed. The *Errors* tab shows the standard error of the execution. The *Con guration* tab shows the configuration of the code that was executed as well as any configuration that was present before. The *Metrics* tab shows the metrics of the code that was executed, such as execution duration and the session ID of the session the execution was attached to.



(c) The Documentation View.



(d) The Graph View.

Documentation Node:

The documentation view is a view that is shown when a documentation node is selected and is shown in fig. 4.5c. The documentation view consists of a Monaco Text Editor and the component selector. The documentation view attaches the *Save Indicator* visual component to the view.

Graph Node:

The graph view is a view that is shown when a graph node is selected and is shown in fig. 4.5d. The graph node view has a documentation display, a graph selector visual component, and a component selector. The graph view attaches the *Execution Button* visual component to the view.

The graph selector visual component consists of three main sections: the file selector, the session dropdown, and the session info display. The file selector allows the user to input a path to an IGC file to import the graph. This can either be a relative or an absolute path. After the IGC file is specified, the session dropdown allows the user to select a session to import. These sessions are automatically detected from the userdata of the IGC file. The session info display shows the data of the session that is currently selected. This includes the session ID, when the session was last updated, and the execution path (given by node IDs) in the session.

Projection:

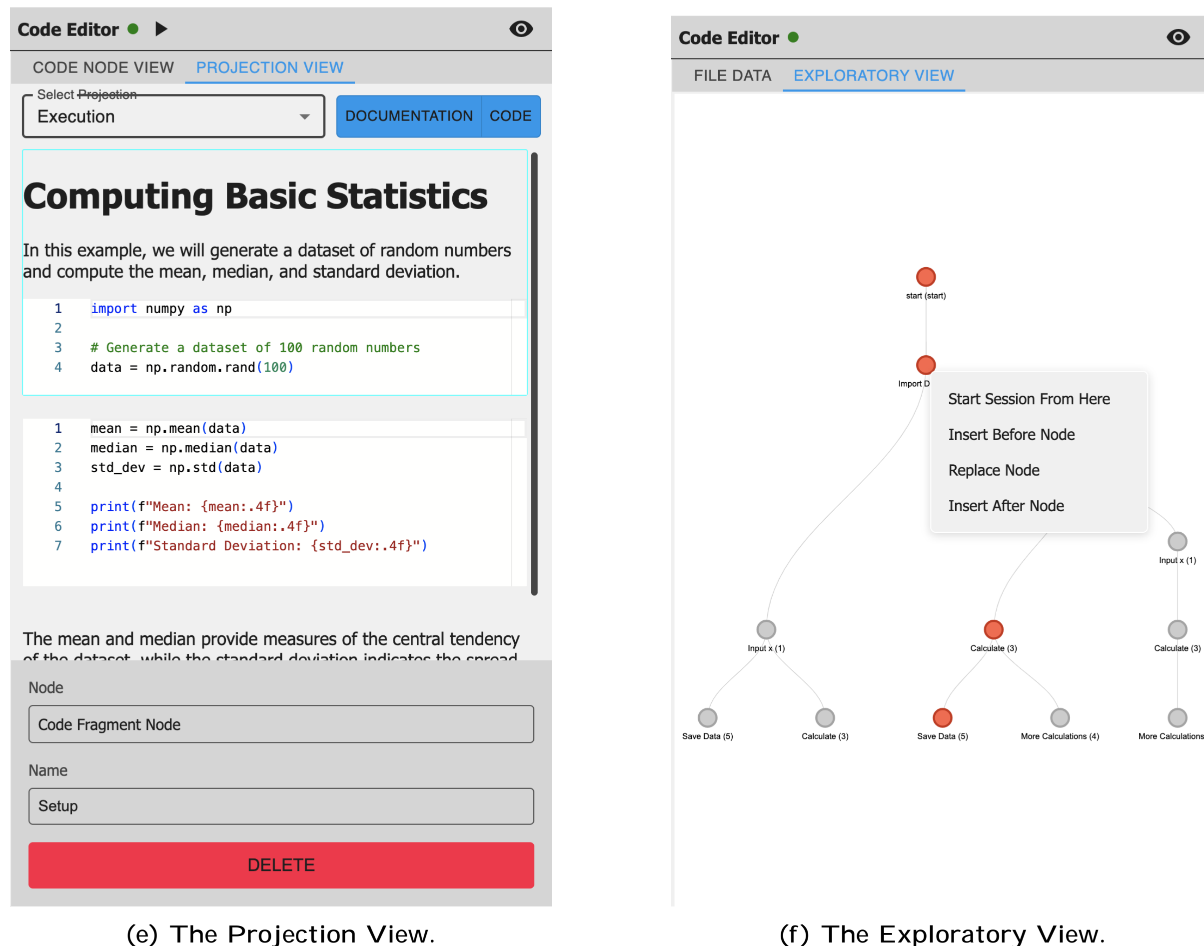
The projection view is a view that is shown when a code node is selected and is shown in fig. 4.5e. The main content of the projection view is a projection type dropdown, documentation and code toggle buttons, and the projection itself. There are three types of projections: *Execution*, *Dependency*, and *Class*. If the dependency projection is selected, there will be another dropdown to select the specific dependency the user wants to see. Much of this view is discussed in case study 2 (section 5.2) about code/documentation projections.

Exploratory:

The exploratory view is a view that is shown as a default view (when no components are selected) and is shown in fig. 4.5f. The exploratory view is a view that is used to sessions. The only content is a Viz-Network graph (tree) that shows the current session indicated by red nodes. The tree shows the path that each session takes and what nodes are executed in each session. The tree is also useful to see where sessions overlap and where they diverge.

When right-clicking on a node, a context menu will appear with options to create new sessions. If a node of the current session is right-clicked, the context menu will have four options[7]: "Start Session From Here", "Insert Before Node", "Replace Node", and "Insert After Node". If the node is not of the current session, the context menu will only show the option to "Start Session From Here". The "Start Session From Here" option will create a new session that starts from the selected node. The "Insert Before Node" option will create a new session identical to the current session but it will insert a new node before the selected node. Once the button is pressed, all nodes will have a green highlight indicating to the user to select the node they want to insert before. The "Replace Node" option will create a new session and replace the selected node with a new node (similar node selection process as "Insert Before Node"). The "Insert After Node" option will create a new session and insert a new node after the selected node (similar node selection process as "Insert Before Node"). Much of this view is discussed in case study 4 (section 5.4) about exploratory programming GUI.

[7] Some options will not show depending if it is the rst or last node.



(e) The Projection View.

(f) The Exploratory View.

Figure 4.5: All the current views of IGC.

4.3.7 Custom IGC Components

Note: The next section contains some overlapping terminology. Normally, the term "component" is used in two different contexts: IGC specific components and visual react components, however, this section uses the term which can be used in both contexts. This is because this section discusses the creation of custom IGC components, which are visual react components. A distinction will be made when necessary.

Extensibility is a key factor considered throughout the development process of IGC. The environment needed to allow users to create custom IGC components that were not part of the default set. This is especially important as the environment is designed to be used in a variety of programming environments and languages. Custom IGC components are handled in a registry system. The registry system keeps track of all directories that potentially contain custom IGC components defined by the user. In the context of IGC, these directories are also known as "Modules".The registry system is handled by the "Add-On Manager" visual component (mentioned in section 4.3.8). In short, the Add-On Manager allows the user to add, remove, and disable modules from the registry.

Initially, the plan was to use inheritance to create custom IGC components. This would be simple as the user would only have to inherit the component or building block they wanted to extend and then override the necessary methods. However, this approach was not feasible because of the way ReactFlow and React prioritize composition instead of inheritance (this issue is discussed in section 6.3.3). Instead of using direct inheritance, the user can use a mixture of "sudo-inheritance" and composition to create custom IGC components.

To create a custom IGC component, the user firsts creates a directory to use a workspace. This directory will be the path of the module IGC will import to the registry. The user can make a npm package allowing them to import custom react components or libraries to use in their component. The user can then create a new javascript or typescript file in the workspace directory that will contain the

custom IGC component. Next, the user will import the custom IGC library, which contains basic types and functions to link the component back to IGC. To create a component (node or relationship), the user simply needs to use the `createComponent` function from the IGC library. The following is an example of how you would create a custom node component:

```
import BaseNode, { IGCNodeProps } from "@/IGCItems/nodes/BaseNode";
import { createComponent } from "@/utils/componentCache";

const TestNodeComponent: IGCNodeProps = ( props ) => (
    <BaseNode {...props} data={{
        ...props.data,
        backgroundColor: TestNode.color
    }}/>
);

const TestNode = createComponent(
    TestNodeComponent,
    "TestNode",
    "Test Node",
    {
        color: "cyan",
        parentComponent: BaseNode,
        settable: true,
        abstract: false,
        type: "node",
    },
);

export default TestNode;
```

Let us break down the code above:

1. Imports the necessary types and the `createComponent` function. Since we are creating a node, we import the `IGCNodeProps` type. The `BaseNode` component is also imported as it is a typical node rendered view that most nodes use, however, it is not necessary to use it.
2. Creates the outline of a new component that will be used to render the custom IGC component. It is important to note that the component is of type `IGCNodeProps`. The equivalent of this for a relationship or view is `IGCRelationshipProps` and `IGCViewProps`, respectively. The component returns the `BaseNode` component to render.
3. Uses the createComponent function as a wrapper to make the actual custom IGC component. The function takes in the rendering component, the component's unique key for the registry, the human-readable name of the component, and optional parameters of the component. The optional parameters of the component are important as they allow the user to set the color of the component, the parent component (in this case BaseNode is the most basic node), if the component is settable (if the type can be changed by the user), if the component is abstract (displayable or only used for inheritance), and what type the component is.
   There are two things to note. Firstly, the type of the component will be inferred by the parent component if specified. If neither the parent component nor the type optional parameter is specified, IGC will throw an error. Secondly, `createComponent` cannot be used to create views. This is handled by the function `createView` which takes in the view rendering component, the view unique key for the registry, the human-readable name of the view, the list of components (nodes and/or relationships) that the view will appear for (if empty, the view will be a general view), the weight of the view (lower weight means the view will appear first), and some optional parameters. The optional parameters are the parent view and whether the view is abstract or not.
4. Finally, the component is exported to be used in the module. This step is important as the component will not be available to use in the environment if it is not exported.

That is everything that is needed to create a custom IGC component. The user can create as many custom components as they wish inside of a module. IGC will automatically and dynamically look at all the source files and detect which components are usable based on the types of the exported components.

The user can then import the module to the registry and the component(s) will be available to use in the environment.

If the user wishes to create a user-friendly name for the module, they can create a `igc.module.json` file in the module directory. The `igc.module.json` file will contain the following:

```
{
    "name": "<Example Module Name>"
}
```

4.3.8 Hidden Menu Components

There are a few key visual components that are not actively visible in the environment but are used to provide functionality to the user. These are the *Add-On Manager* and the *Project Info* visual components. The Add-On Manager is used to manage the modules that are in the registry. The Project Info component is used to display the information of the current project. Below is a description of each component:

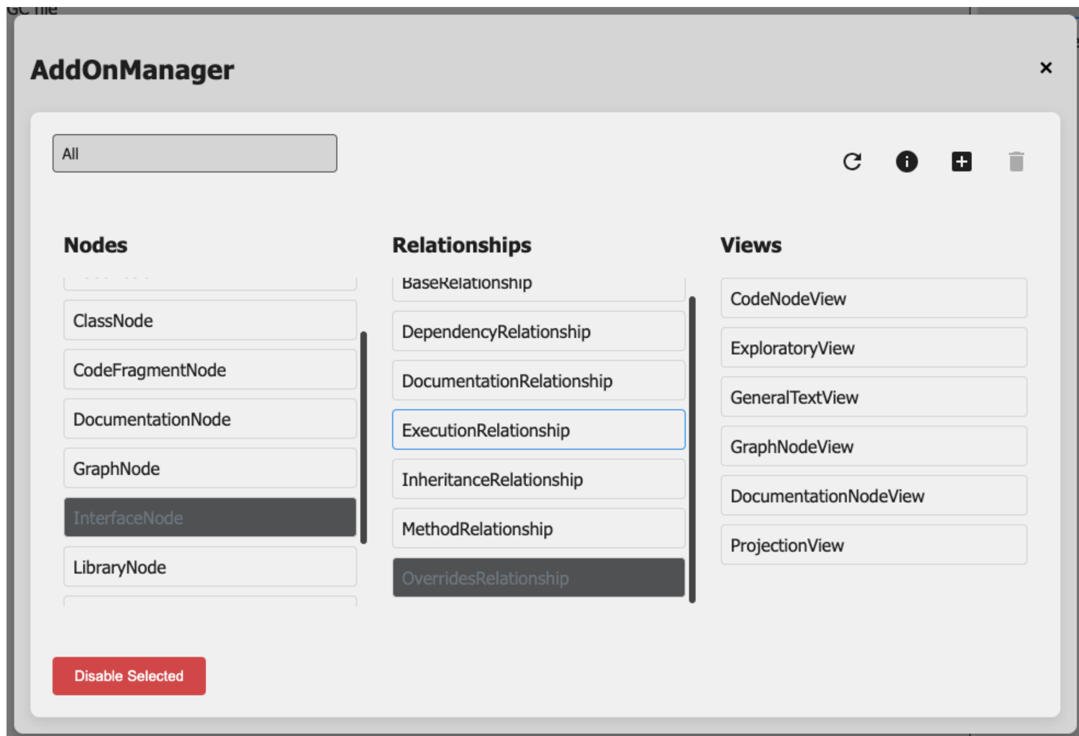


Figure 4.6: Add-On Manager.

Add-On Manager

The Add-On Manager is a visual component that is used to manage the modules in the registry. It is accessed by clicking the “Manage Add-ons” button under the file dropdown menu on the navigation bar. The Add-On Manager is shown in fig. 4.6. The main content displays all the IGC components from all modules that are currently in the registry. The components are separated into three columns: nodes, relationships, and views. The user can select any component and choose to enable or disable it. This is saved in the cache so the user does not have to re-enable or disable the component every time they open the environment. Above the main content from left to right is the module dropdown selector, the refresh modules button, the module info button, the add module button, and the delete module button. The module dropdown selector allows the user to select a module to view the associated components. By default, an aggregation of all modules is shown in the “All” module (the user is not able to delete this aggregation module). If the module has a `igc.module.json` file, the name from this file will be used in the dropdown selector. The refresh modules button will refresh the modules and their status in the registry. The module-info button will show the information of the module that is currently selected. The add module button will allow the user to add a new module to the registry. This will open a directory selector dialog for the user to select. The delete module button will allow the user to delete the module

that is currently selected. The Add-On Manager is a key visual component as it allows the user to manage the components that are available to use in the environment.

The Add-On Manager is an essential visual component for extensibility. It is the interface that allows the user to add and manage any custom modules and IGC components the user might want to create.

Project Info

The Project Info component is a visual component that is used to display the information of the current IGC file. The Project Info visual component is accessed by clicking the "Project Info" button on the navigation bar.

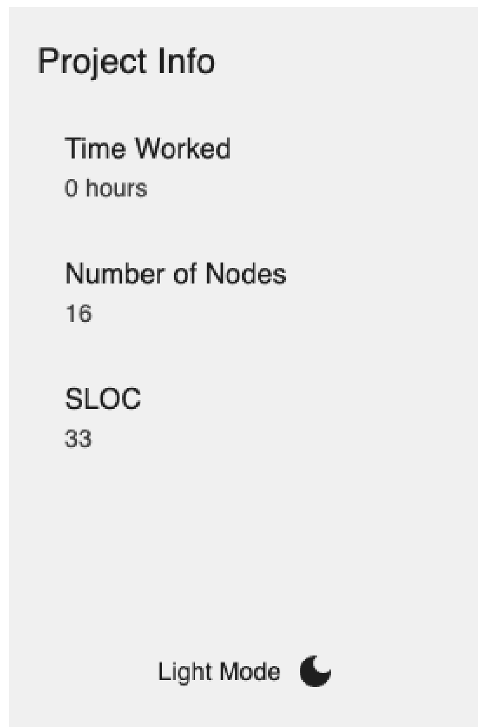


Figure 4.7: Project Info.

Currently, there are three metrics that are displayed: Time Worked, Number of Nodes, and SLOC. The Time Worked metric is the total time the user has spent working on the IGC file, however, this metric is still running into issues. The Number of Nodes metric is the total number of nodes that are in the current IGC file. The SLOC metric is the total number of source lines of code that are in the current IGC file. At the very bottom of the Project Info visual component is a button to change the color scheme between "light mode" and "dark mode". By default, the color scheme is matched to the user's operating system color scheme (more info in section 4.3.9). The Project Info component is shown in fig. 4.7.

4.3.9 Styling

The styling of IGC was handled either from native Material-UI components or custom CSS. The custom CSS styling was mostly all done using CSS modules. CSS modules are a way to locally scope CSS by automatically creating unique class names. This is done by creating a CSS file with the same name as the component file and then importing the CSS file into the component file. The CSS file will then be automatically scoped to the component file. This is useful as it prevents CSS class name collisions and makes it easier to maintain the CSS.

Specifics of the styling choices are discussed below:

Color Scheme

IGC has two color schemes: light and dark mode. Originally, IGC was only developed using a dark theme as we wanted to natively follow the trend where more websites use this color scheme. The claim is that dark themes lower the strain on the user's eyes [56]. However, as the project grew, it was clear that the dark background looked too harsh when sharing images with a default light theme.

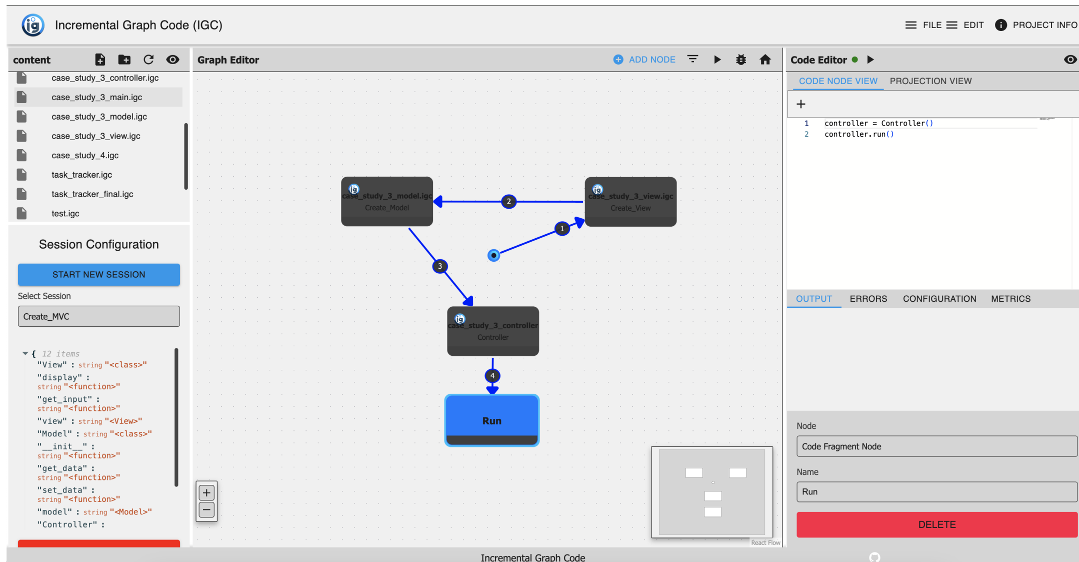


Figure 4.8: IGC in light theme.

To compromise, both light and dark themes are available to the user. By default, the user's color scheme is set by the operating system color scheme (if this fails, the light color scheme is selected). This selection can always be changed as there is a switch at the bottom of the Project Info visual component.

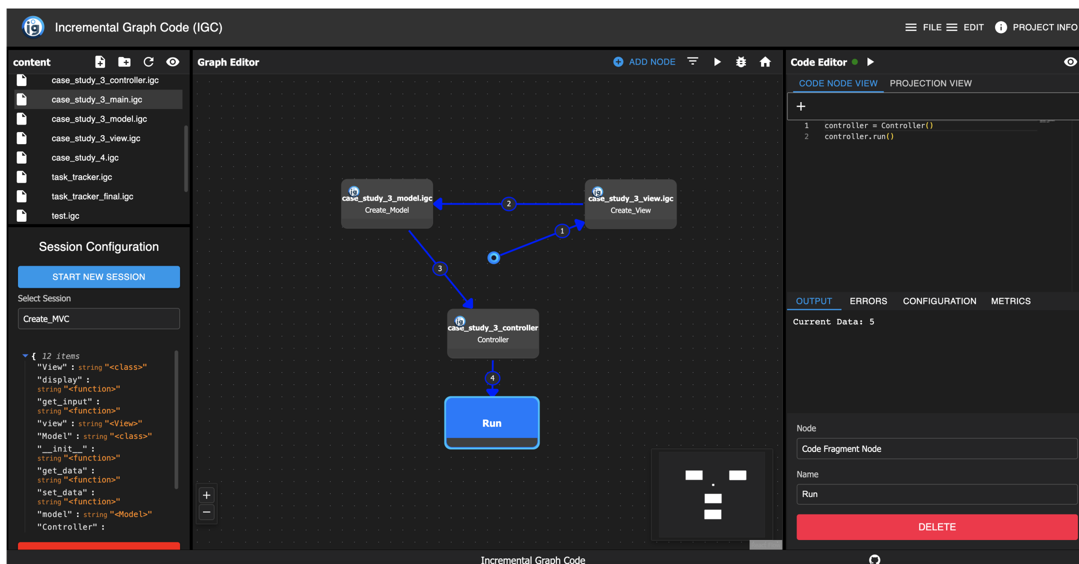


Figure 4.9: IGC in dark theme.

During the development of the webpage, the Web Content Accessibility Guidelines (WCAG) [57] were considered to ensure an inclusive user experience. While the design does not strictly adhere to every guideline, special attention was given to try to have distinct color contrasts and appropriate text sizes throughout the application. The goal was to strike a balance between aesthetic appeal and accessibility, ensuring that key elements remain easily distinguishable for users with varying visual capabilities.

The color schemes are created through two central mechanisms. The first uses CSS injection. This evolves creating general CSS variables along with theme-equivalent versions. For example, the following:

```
{
    ...
    mainBackgroundColorLight: "#f0f0f0",
    mainBackgroundColorDark: "#1e1e1e",
    mainBackgroundColor: "var(--mainBackgroundColorLight)",
    ...
}
```

Then, to change the color scheme, we set the general CSS values to their corresponding theme on a color scheme change event. This CSS inject works fine and it consolidates all color scheme colors inside variables located in one CSS file. On the negative side, it is a bit of a 'hack' that relies on dynamically changing CSS variables which can be a bit slow and can cause some flickering. All static colors are defined in `IncrGraph/frontend/src/styles/constants.ts`.

The second mechanism is through the use of the Material-UI library. IGC uses many Material-UI pre-built components that follow the Material Design guidelines. Material-UI has a built-in theming system that allows the user to create custom themes and apply them to the components. Two themes were created: one for light mode and one for dark mode. Then, from the constants color file, the colors are applied to the themes. The themes are then loaded into the Material-UI theme provider which is wrapped around the entire application. The theme that is passed is determined by a variable inside the Zustand data store as it is accessible throughout the entire application. This method is normally much faster and more reliable than the CSS injection method.

Layout

The layout of IGC is designed to be flexible and responsive. The environment is divided into three main columns between the navigation bar and the footer. Each column is meant to have its own independent purpose and have its own sub-navigation bar. Each column can be resized by dragging the column lines left and right. The user can also hide a column by clicking the hide (eye) icon. The navigation bar is at the top of the environment and contains all the main environment actions that the user can take. The footer is at the bottom of the environment.

Material-UI is used to create much of the layout and components of IGC. These components were used as many are responsive, modern-looking with a lot of functionality, and highly customizable. The layout is responsive to the screen size making sure that the interface looks almost the same for every user. Many of the sizes are set by variables inside the constants file. This makes it easy to quickly change and experiment to get the view just right.

Icons

Besides the IGC specific icons that were created and designed for the environment, like the logo, Material-UI icons were used throughout IGC. Material-UI has a large selection of icons that are free to use and are designed to be used in modern web applications. The Material-UI icons are used in buttons, tabs, and other components that require an icon.

4.4 Back End / APIs

The back end of IGC is very simple compared to most full-stack applications. The entire application does not use a database system; instead, it opts to use file-based data. This is the case for many programming environments such as Visual Studio Code or Eclipse, which allows the user to send the entire state of a project through the transfer of files. IGC currently has two general routes that each serve their own distinct purpose: File Operations (everything to do with manipulating files) and Code Analysis / Execution (everything to do with code).

The following are all the APIs that are used throughout IGC:

Route	Endpoint	Method	Description
/code-handler	/execute	POST	Executes a code fragment and attaches it to a given session.
/code-handler	/execute-many	POST	Executes multiple code fragments and attaches them to a given session.
/code-handler	/analyze	POST	Analyzes a code fragment.
/file-explorer	/file-tree	GET	Gets the file directory tree from a specific path.
/file-explorer	/file-exists	GET	Checks if a file exists at a specific path.
/file-explorer	/file-content	GET	Gets the content of a file at a specific path.
/file-explorer	/rename	PUT	Renames a file at a specific path.
/file-explorer	/copy	POST	Copies a file to a new location.
/file-explorer	/delete	DELETE	Deletes a file at a specific path.
/file-explorer	/new-file	POST	Creates a new file at a specific path.
/file-explorer	/new-igc-file	POST	Creates a new IGC file at a specific path.
/file-explorer	/new-directory	POST	Creates a new directory at a specific path.
/file-explorer	/module	POST	Adds a module to the IGC registry.
/file-explorer	/module	DELETE	Removes a module from the IGC registry.
/file-explorer	/find-components	GET	Finds all components in every module in the registry.
/file-explorer	/session-data	GET	Gets the data of a session.
/file-explorer	/session-data-node	DELETE	Removes a node's execution "contribution" in all sessions.
/file-explorer	/session-data-execution	DELETE	Removes an execution's "contribution" in all sessions.
/file-explorer	/primary-session	POST	Sets the primary session.
/file-explorer	/session	POST	Creates a new session.
/file-explorer	/session	DELETE	Deletes a session.

Table 4.3: Example API Routes

There are two main routes that are used in the back end of IGC: /code-handler and /file-explorer.

4.4.1 Code Analysis / Execution

The /code-handler route is used for all code-related operations. This includes executing code, analyzing code, and managing the code execution sessions. The route has three main endpoints: /execute, /execute-many, and /analyze. All of these endpoints takes in a language string that corresponds to a language binary (described in section 4.2.3). The /execute endpoint is used to execute a single code fragment and attach it to a session. Similarly, the /execute-many endpoint is used to execute multiple code fragments and attach them to a session. The /analyze endpoint is used to analyze a code fragment and return the results offering information such as dependencies and the new definitions that are defined by the code.

4.4.2 File Operations

The /file-explorer route is responsible for handling all file-related operations within the IGC environment. This includes tasks such as navigating the file system, creating, renaming, copying, and deleting files and directories, as well as managing custom modules and session data.

To interact with the file system, the route provides several endpoints that allow users to perform various operations. For example, the /file-tree endpoint retrieves the directory tree structure from a specified path. There are several other endpoints that handle file content retrieval, renaming, copying, deletion, and creation. All these endpoints are key for the functionality of the file explorer component.

To interact with custom IGC components or modules, the route provides endpoints for adding and removing modules from the IGC registry. This registry stores information about all the components

available in the environment, allowing users to access and utilize them in their projects. The /find-components endpoint retrieves all components from the paths listed in the registry. Normally this will only happen on startup of IGC or when refreshing/updating the registry.

The route also manages session data, which includes information about the code execution sessions in the environment. The /session-data endpoint retrieves the data of a specific session, while the /session-data-node and /session-data-execution endpoints handle when a node or relationship ever gets removed from the project. IGC is smart enough to automatically detect which sessions would be affected by the deletion, and recreate them accordingly. The /primary-session endpoint sets the primary session, and the /session endpoint creates or deletes a session as needed.

4.4.3 Electron Specific Functionality

The Electron-specific functionality is used only very sparsely. In particular, there is only one function that is used in the entire application. This function is used to open a file dialog box that allows the user to select a file or directory. This function is used to allow a user to open a project or to select a directory to save a project. This function is also used in the Add-On Manager to allow the user to select a package to add to the registry. This was needed as there is no way to access the file system in a web application due to security reasons. This was the only way to get around this limitation. In Electron lingo, a context bridge is used to allow the renderer process to communicate with the main process. This is done by using the ipcRenderer and ipcMain modules in Electron. The context bridge is used to expose the dialog function to the renderer process, allowing the user to interact with the file system.

4.5 Application Diagrams

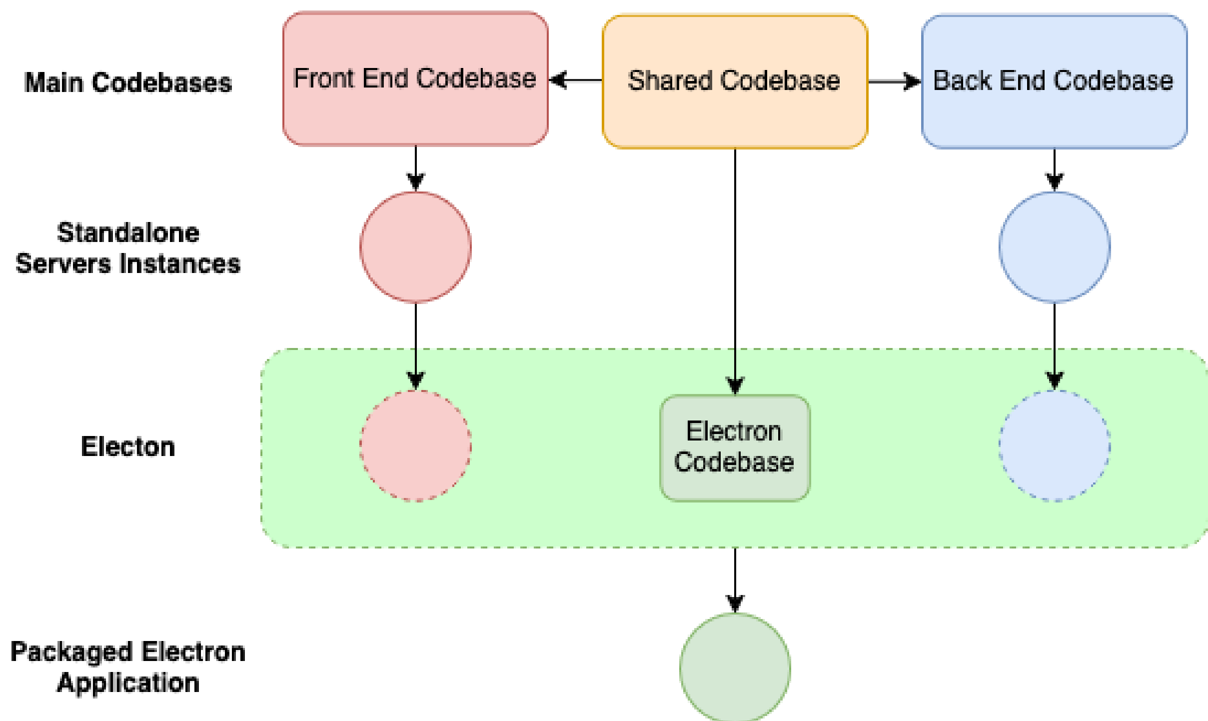


Figure 4.10: The figure shows the process that each project takes to actually run. A server/application is represented by a circle. For development, most of the time, we just need to run the front end (Vite) and back end (Node) servers on a local environment. Electron is more complicated. Internally, Electron recreates these local servers and attaches some procedures to allow the front end to access the user's system. After which, all the processes get packaged and distributed into an application native to the OS specified.

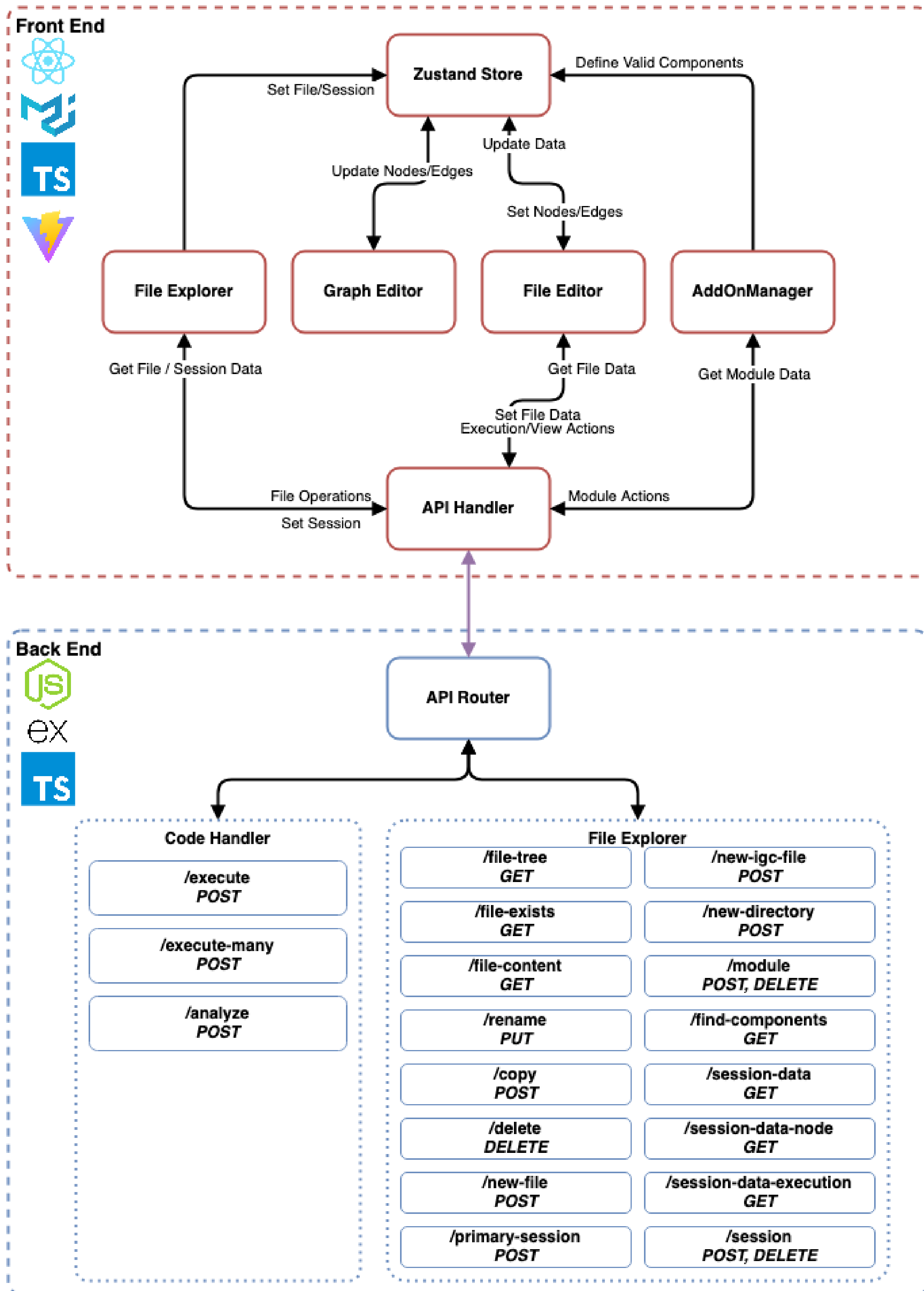


Figure 4.11: Application diagram showing the data flow between different key components.

4.6 Software Development Practices

To maintain a high level of code quality and ensure the project's success, several software development practices were followed throughout the development of IGC. The following sections detail the practices followed in each of these areas. fig. 4.12.

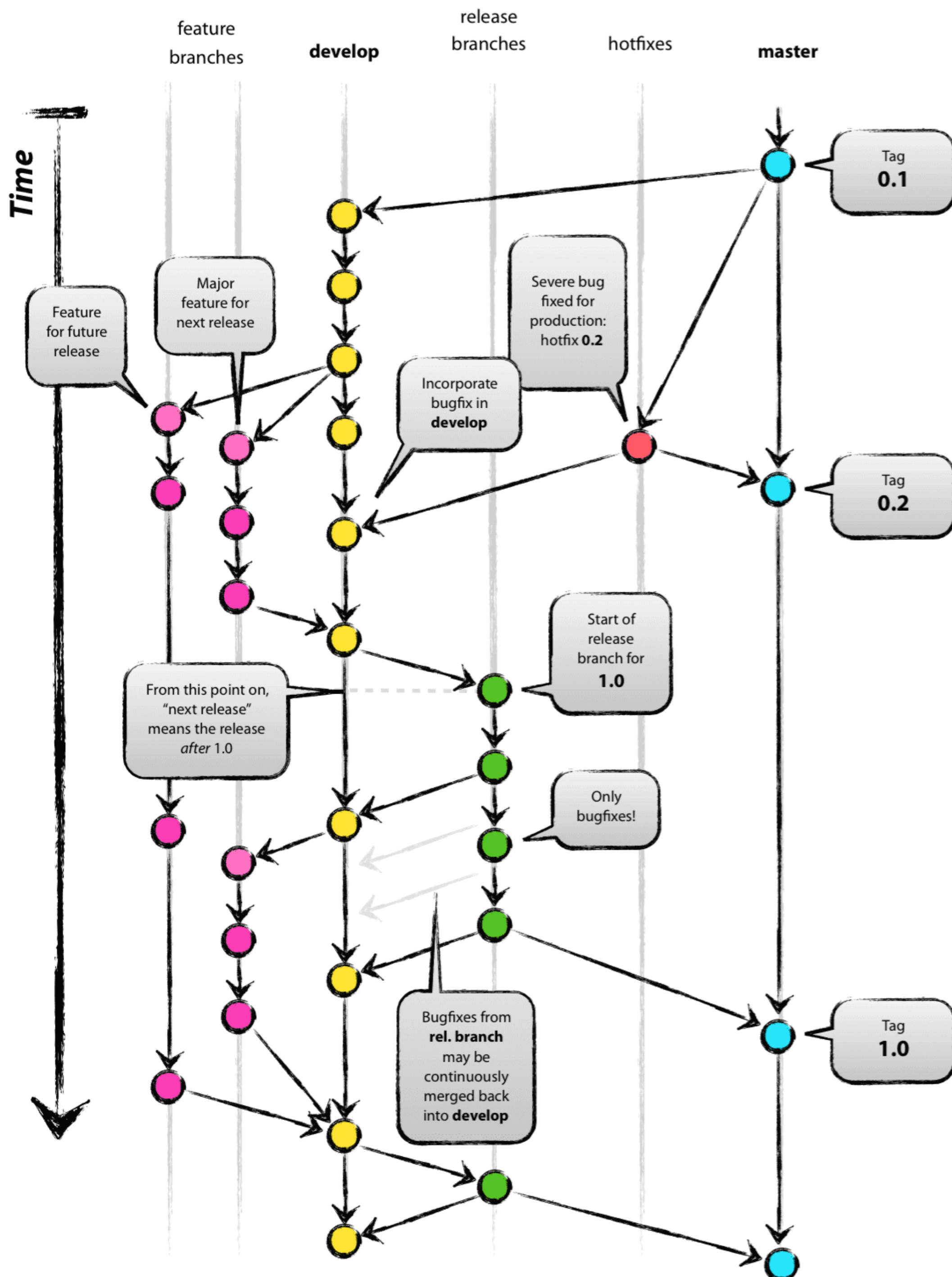


Figure 4.12: Git-Flow branching system. Diagram created by Vincent Driessen [58].

4.6.1 General Practices

To ensure code is standardized and consistent, ESLint and Prettier were used. ESLint is a static code analysis tool that checks for errors and enforces coding standards. Prettier is a code formatter that

automatically formats code to a consistent style. Both tools were configured to run automatically on save in the code editor to ensure that all code adheres to the defined standards. To see the structure rules that were used, see the `.eslintrc.base.json` (for base structure), `.eslintrc.json` (for module-specific structure), and `.prettierrc.json` files in the root of the project roots.

To ensure potential sensitive data is not leaked, the `.gitignore` file was used to exclude certain files and directories from being committed to the repository. This includes the `.env` file, which contains sensitive information such as back end and front end server information. A template `.env_example` file is provided to guide users on what information is required. Many cloud services have ways to emulate this `.env` file. This means if IGC ever gets deployed to a cloud service, the user can easily set up the environment variables.

Modularity was a key focus throughout the development process. The project was divided into separate modules, each with its own distinct purpose. The modules are front end (for all visuals), back end (for all APIs), electron (for all electron-specific functionality), and shared (for all shared code between the front end and back end). This modular approach allowed for better organization of the codebase and made it easier to manage and maintain the project. The use of modules also made it easier to add new features and functionality to the project without affecting other parts of the codebase.

The shared module contains all shared code and types between the other modules. Since all modules are written in TypeScript, the shared module contains all the types that are used throughout the entire project. Making sure all code is not reused is an essential software engineering practice. It ensures that changes only need to be made in one place and that the code is consistent across the entire project.

4.6.2 Version Control and Management

Throughout the development of IGC, git was used as the version control system through Github. GitHub provides a platform for hosting code repositories and managing the Git version control system. The development strategy was inspired by the Git-Flow branching system [58]. It is used to manage the different branches and ensure that changes are properly tested before being merged into the main branch. The Git-Flow branching system is a branching model that defines a strict branching structure designed to facilitate parallel development and release management. The model consists of two main branches: the main branch and the develop branch. The main branch contains the official release history, while the develop branch contains the latest development changes. Feature branches are created off the develop branch for new features, and release branches are created off the develop branch for preparing a new release. Hotfix branches are created off the main branch to fix critical bugs in the production release. Although not strictly enforced as there was only one developer, the Git-Flow branching system helped ensure that changes were properly tested and reviewed before being merged into the main branch, reducing the risk of introducing bugs or breaking changes. The Git-Flow branching system is shown in

4.6.3 Feature Realization and Implementation

Since the project is fairly grand in scale, an ongoing question was posed: "where to start?" To answer this question, the project was broken down into the smaller requirements that were defined in the design phase. A set of issues were created for each requirement, trying to capture all the necessary work that needed to be done to complete the requirement. Each issue was labeled with the corresponding requirement label. This allowed for easy tracking of the progress of each requirement and ensured that all requirements were completed before moving on to the next phase. As the project grew, new issues were created to address any issues or introduce new features. This process was repeated throughout the development process. The labels used on the sprint board are shown in fig. 4.13.

We manage our development tasks using a sprint board with columns for Backlog, Ready, In Progress, In Review, and Done. Rather than working in time-boxed sprints, we let tasks flow through these stages on a continuous basis. When a new feature request or bug fix comes up, we add it directly to the Backlog. We order all tasks in terms of priority. Once we are confident about scope, priority, and believe that it can be accomplished in the present state of IGC, we move it to Ready. At this point, we will pick it up for development and mark it as In Progress. As soon as we finish coding and basic validation, we move it to In Review for a final check, even if it is just us confirming that the change meets the requirement. Finally, we mark the task Done once we have verified and merged it into the development branch. The sprint board is shown in fig. 4.14.

Because there is no strict end date to any "sprint," this approach resembles a Kanban system more than a traditional Scrum cycle. We inject new work as needed without waiting for a formal sprint

23 labels

Label	Description	Open issues
Basic Essentials	Needed to make everything work	
bug	Something isn't working	1
*Code Architectural Visualization	Related to "Code Architectural Visualization" requirement	
*Complexity Management	Related to "Complexity Management" requirement	
*Consistency	Related to "Consistency" requirement	1
*Cross-Compatibility Export	Related to "Cross-Compatibility Export" requirement	2
*Data Collection	Related to "Data Collection" requirement	1
dependencies	Pull requests that update a dependency file	4
documentation	Improvements or additions to documentation	
duplicate	This issue or pull request already exists	
enhancement	New feature or request	2
*Extensibility	Related to "Extensibility" requirement	
*File Management	Related to "File Management" requirement	
*IDE-Like Features	Related to "IDE-Like Features" requirement	
*Incremental Programming Features	Related to "Incremental Programming Features" requirement	2
Large	Requires a high amount of effort	
Medium	Requires a decent amount of effort	
old	Previous issues before relating to requirements	3
Small	Not much effort needed	
story	Story point outlining a feature	3
UI/UX	Improvements to UI/UX	2
*Version Control	Related to "Version Control" requirement	1
wontfix	This will not be worked on	

Figure 4.13: All labels used on the sprint board. Labels that reference a design requirement are prefixed with "*".

boundary, which helps us stay flexible when priorities change. However, we do not conduct sprint retrospectives or time-bound planning sessions the way you might in Scrum. The development process is simple, adaptive, and focuses on a steady flow of tasks from backlog to completion.

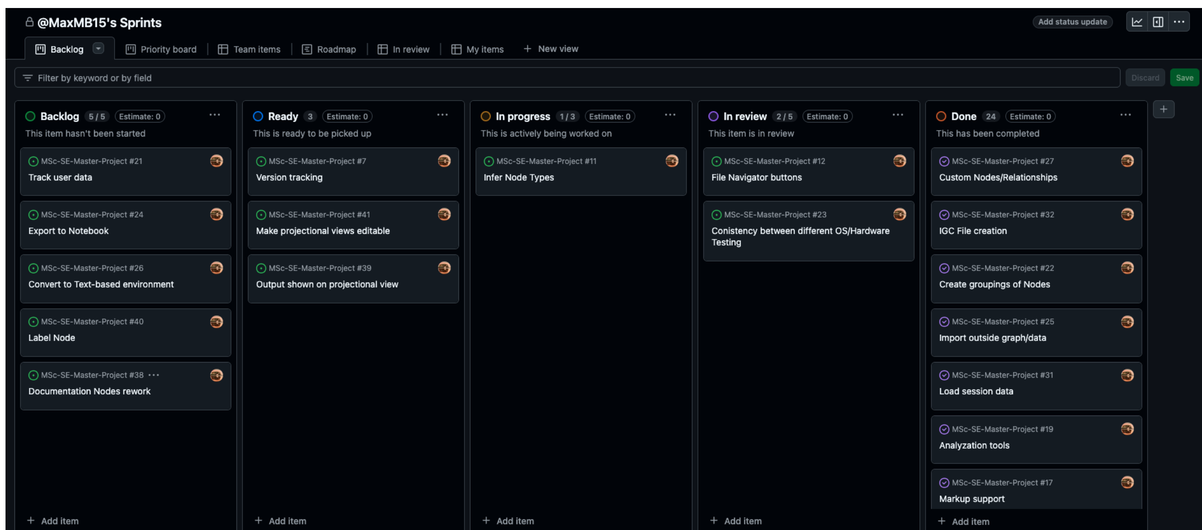


Figure 4.14: Sprint board with tasks/issues.

4.6.4 Code Organization

The codebase is organized into four main modules: frontend, backend, electron, and shared. Each module was carefully structured to ensure that the code was organized and easy to maintain. Below is a brief

overview of each module:

Front End

Below is the frontend directory structure (fig. 4.15). The frontend module is the largest module in the project and contains the majority of the codebase. The frontend module is divided into several subdirectories, each containing code specific to its purpose. One thing to note is the structure of visual components. Each visual component has three files: the component code, the CSS module styling, and an `index.tsx` symlink pointing to the component code file. The symlink is created to remove redundant import statements. For example, without `index.tsx` symlink for the example listed in the directory structure, the import for "AddOnManager" would be `@components/AddOnManager/AddOnManager`. As we see, the visual component name is duplicated. We can avoid this by putting the code inside an `index.tsx` file, but then, throughout development, all files open will have the same `index.tsx` name. The symlink is a good compromise as it allows for a clean import statement and a clean file structure. Another strategy to reduce import names was the use of aliases for commonly used paths. For example, `@components/` is an alias for the `frontend/src/components` directory. This allows for cleaner and more concise import statements.

```
IncrGraph/
└─ frontend/
   ├─ node_modules/ (Dependency modules installed by npm)
   ├─ public/ (Public assets served directly)
   ├─ src/
   │  ├─ assets/ (Images, fonts, etc.)
   │  ├─ components/ (Visual components)
   │  │  ├─ AddOnManager/
   │  │  │  ├─ AddOnManager.module.css (CSS Module for Add-On Manager)
   │  │  │  ├─ AddOnManager.tsx (Add-On Manager component)
   │  │  │  └─ index.tsx (Symlink to ./AddOnManager.tsx)
   │  │  └─ ...
   │  ├─ hooks/ (Custom React hooks for shared logic)
   │  ├─ IGCItems/ (IGC Components)
   │  │  ├─ nodes/
   │  │  ├─ relationships/
   │  │  └─ views/
   │  ├─ requests/ (For API requests)
   │  ├─ store/ (Zustand store)
   │  ├─ styles/ (Main Styles)
   │  ├─ types/ (Front end Specific types)
   │  ├─ utils/ (Utility functions)
   │  ├─ index.css
   │  └─ main.tsx
   ├─ .eslintrc.json
   ├─ .gitignore
   ├─ index.html
   ├─ package-lock.json
   ├─ package.json
   ├─ README.md
   ├─ tsconfig.json
   ├─ vite-env.d.ts
   └─ vite.config.ts
```

Figure 4.15: Front End Directory Structure

Back End

The back end directory structure is shown in fig. 4.16. The back end module is fairly simple. The key files are the `index.ts` file, which is the main entry point for the back end, and the `routes` directory, which

contains all the API routes. Language-specific scripts are stored in the scripts/<language> directory (python is currently available).

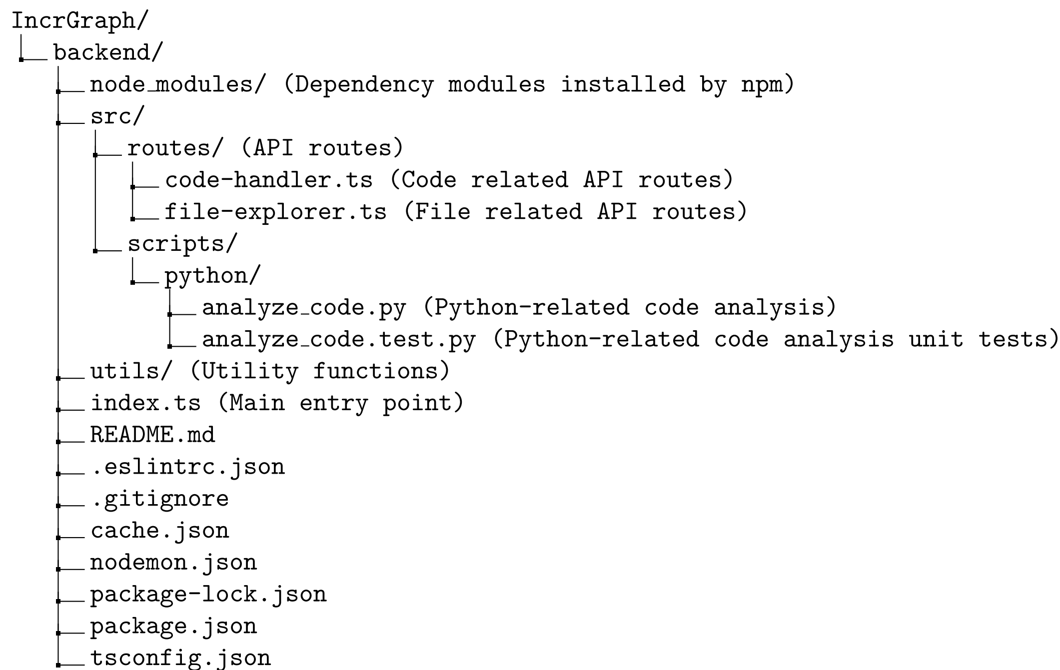


Figure 4.16: Back End Directory Structure

Electron

The electron directory structure is shown in fig. 4.17. The key files are the main.ts file, which is the main entry point for Electron, and the preload.ts file, which is the preload script for Electron that adds functionality such as opening up the user's directory selector dialog. The dist directory contains the compiled Electron application images for different platforms.

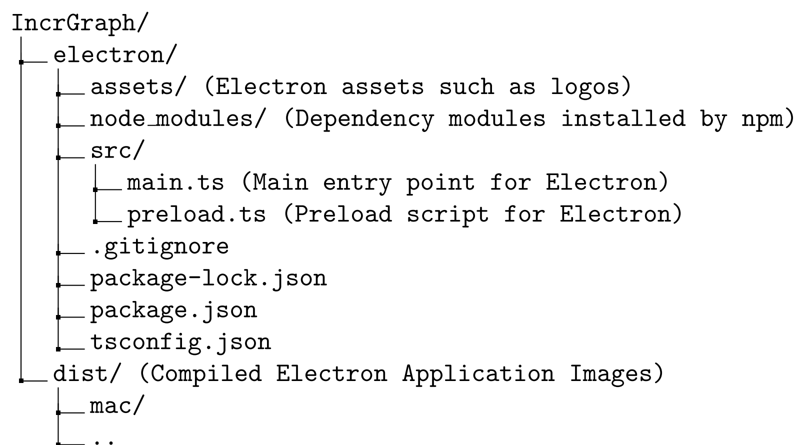


Figure 4.17: Electron Directory Structure

Shared

The shared directory structure is shown in fig. 4.18. The shared module contains all shared code between the front end, back end and electron. This includes shared types, utility functions, and the main entry point that links all the files together. It is important to know what other modules only access the dist/ directory in the shared module. This means if any changes occur in the shared module, the shared module must be recompiled and the other modules must be restarted.

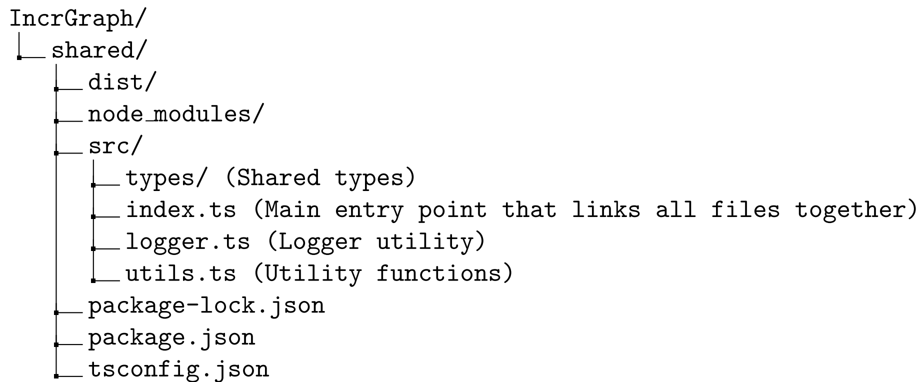


Figure 4.18: Shared Directory Structure within the IncrGraph Project

4.6.5 Testing

As this project was never regularly published in a central, online space, most of the testing was done manually in a local environment. This was done by running the application and testing the functionality to ensure that it worked as expected. The front end was tested by running the application and interacting with the visual components to ensure that they worked as expected. The back end was tested by sending requests to the API routes and checking that the responses were correct. The Electron module was tested by running the compiled application and checking that the Electron functionality worked as expected. The shared module was tested by running the frontend and backend modules and checking that the shared code worked as expected.

Overall, the testing process was fairly informal and, in retrospect, insufficient for the long term. In the future, more formal, automated testing processes should be implemented, such as using selenium for the front end and Newman / Postman unit testing for the back end. Both these methods would work great with GitHub actions. The testing could run before a feature pull request gets merged into the development or main branches. This would ensure that the code is always working as expected and that new features do not break existing functionality.

The following describes the few testing practices that were used throughout the development of IGC:

Unit Testing

Unit testing was used to ensure the Python analysis scripts were working as expected. The unit tests were written using the built-in Python unittest module. The tests were run before any changes were made to the code to ensure that the code was working as expected. The tests were also run after any changes were made to the code to ensure that the changes did not break the code. The unit tests ensured that the analysis scripts were working as expected.

Logging

The Winston module was used for any logging in both the back end and electron. The logging module is defined in the shared module so every system is able to use it. The logging module was used to log any errors or information that needed to be logged. Logging was an important part of the development process as it allowed for easy debugging and monitoring of the application if there were any issues. On the front end, the console was used for logging as it was the easiest way to log information.

4.6.6 Documentation

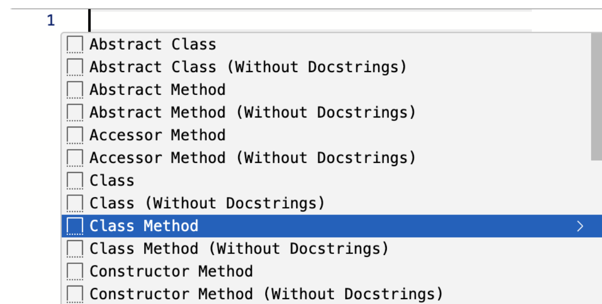
There is not much to mention about the documentation. Where applicable, documentation would be written in the code itself. This would be done sometimes using JSDoc comments for TypeScript code and docstrings for Python code. Other times, simple comments were used to make sure that future developers understood the purpose of the code. The README file was used to provide an overview of the project and how to run the project.

4.7 Miscellaneous

The following are some miscellaneous topics that are important to mention but did not fit into any of the other sections.

4.7.1 Code Templating

To help guild the user of IGC, code templating was used when creating nodes. When a new node is created, a template option is automatically shown and, if selected, generated based on the node type. This template is then displayed in the code editor for the user to edit. The template is generated using a code template file that contains the templates for each node type. This template file is expandable and can be easily modified to add new templates or modify existing ones.



(a) Template options when creating a new method node.

```
1  @classmethod
2  def method_name(cls, args):
3      """
4      Method docstring
5      """
6      pass
```

(b) Template shown when clicking on \Class Method" option.

4.7.2 Text Editor History

The Monaco text editor history is a feature that allows the user to undo and redo changes made to the code editor. It is fairly common that the code editor will have to change between the code of different nodes. Normally, when the editor dismounts (not rendered), the code is lost. We can always load back the code, but then the user cannot undo or redo changes that they made in the past. To solve this, we initially stored the editor state in a cache as well as a "last edited" datetime object to the file so that we could tell if the file was ever edited externally. If it was edited externally, we would ask the user what version they would like to keep. The caching system came with complications as potentially the cache would be overriden by opening the same node in a different editor (potentially on another instance of IGC). To solve this, we recreated the history tracking system. We instead use a system where every file (and node) has a unique identifier that is stored in a Monaco internal storage. This unique identifier is then used to pull data if the editor is ever remounted. This system is much more reliable and does not have the same issues as the cache system.

4.7.3 Relationship Edge Path calculations

When drawing multiple edges between nodes, it is important to prevent edges from overlapping. Overlapping edges can obscure the structure of the graph, making it difficult for readers to follow the connections. We introduce two custom pathing algorithms to prevent intersections of edges at least for relationships generated from the same two nodes. We are unable to prevent intersections between relationships between different nodes as it is impossible for some graphs to prevent all intersections between edges. Realizing this goal in general is an NP-hard problem [59], and, for some graphs, is even undecidable [60].

Offset Calculations

A general formula for the offsets for two distinct nodes, with spacing d (50 pixels currently), is:

$$\text{offset}_i = \left(i - \frac{n-1}{2}\right) d, \quad i = 0, 1, \ldots, n-1.$$

Figure 4.20 below illustrates the concept of offset values used to visually separate multiple edges that connect the same pair of nodes. Instead of drawing all edges directly along the line connecting the two

nodes, we assign each edge an *offset* value. This offset is applied along a perpendicular line passing through the midpoint of the two nodes, ensuring that each edge is drawn at a unique position, equally spaced from the other edges, no matter the orientation. As a result, even when multiple edges connect the same nodes, each edge can be clearly distinguishable, helping with the readability and reducing visual clutter.

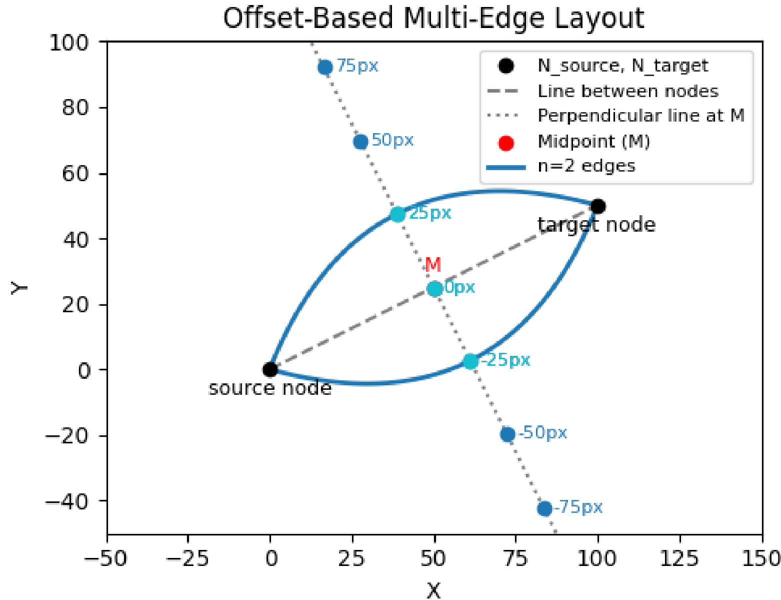


Figure 4.20: Demonstration of using offsets to avoid overlapping edges. Two nodes N_{source} and N_{target} are connected by two edges. By placing offset points along the perpendicular line through the midpoint M, each edge is routed differently. For $n = 2$, the offsets are symmetrically placed at $\pm$25px.

These offsets are chosen symmetrically around the midpoint so that their average remains at zero. By using a consistent spacing and distributing the offsets evenly no matter the orientation, we maintain a balanced, aesthetically pleasing layout. The method scales naturally to any number of edges, spacing them out in a visually uniform manner and minimizing overlap without resorting to complex and computationally relatively expensive algorithms.

For self-loop edges, the offset calculation is simplified as the edges are circular in nature and do not have to be centered around a "zero point" of a line. Instead, the offset is applied to the radius, while maintaining the top-most point of the circle passes through the center point of the node. As the number of edges increases, the offset values of the radius increase, ensuring that the edges do not overlap and are visually distinct. The starting value of the radius is 30 pixels, and the offset is 20 pixels for each additional edge.

The general formula for the offsets for self-loop edges, with a starting radius r (30 pixels currently) and spacing d (20 pixels currently), is:

$$\text{offset}_i = r + i \cdot d, \quad i = 0, 1, \ldots$$

Distinct Source and Target Edge Paths

Definitions. We define the following:

- **Point**: A point in two-dimensional space, represented as (x, y).
- **Rectangle**: An object defined by its center position (x, y), width w, and height h.

Variables.

- N_s: The *source node*, a rectangle with center at point $P_s = (x_s, y_s)$.
- N_t: The *target node*, a rectangle with center at point $P_t = (x_t, y_t)$.
- offset $\in \mathbb{R}$: A scalar representing the offset distance.

Calculations.

1. **Vector from Source to Target**:

$$\mathbf{v} = P_t - P_s = \begin{pmatrix} x_t - x_s \\ y_t - y_s \end{pmatrix}.$$

2. **Unit Vector in the Direction of v**:

$$\mathbf{u} = \frac{\mathbf{v}}{\|\mathbf{v}\|} = \frac{1}{\sqrt{(x_t - x_s)^2 + (y_t - y_s)^2}} \begin{pmatrix} x_t - x_s \\ y_t - y_s \end{pmatrix}.$$

3. **Midpoint between P_s and P_t**:

$$M = \frac{P_s + P_t}{2} = \begin{pmatrix} \frac{x_s + x_t}{2} \\ \frac{y_s + y_t}{2} \end{pmatrix}.$$

4. **Perpendicular Vector to u**:

$$\mathbf{p} = \begin{pmatrix} -u_y \\ u_x \end{pmatrix}, \quad \text{where} \quad \mathbf{u} = \begin{pmatrix} u_x \\ u_y \end{pmatrix}.$$

5. **Offset Point O**:

$$O = M + \text{offset} \cdot \mathbf{p} = \begin{pmatrix} M_x - \text{offset} \cdot u_y \\ M_y + \text{offset} \cdot u_x \end{pmatrix}.$$

6. **Initial Control Point C**
The initial control point C is calculated using the offset point O and the centers of the source and target nodes P_s and P_t:

$$C = 2O - \frac{1}{2}(P_s + P_t) = \begin{pmatrix} 2O_x - \frac{x_s + x_t}{2} \\ 2O_y - \frac{y_s + y_t}{2} \end{pmatrix}.$$

7. **Intersection of Bézier Curve with Node Boundaries**
To determine where the quadratic Bézier curve intersects the boundaries of a node (represented as a rectangle), we define a function:

$$\text{getBezierNodeIntersection}(N, P_0, P_1, P_2) \rightarrow I,$$

where:

- N: The rectangle represents the node.
- P_0, P_1, P_2: The control points of the quadratic Bézier curve.
- I: The intersection point between the Bézier curve and the rectangle N.

Algorithm 1 outlines the algorithm to calculate the intersection points of a Bézier curve with the boundaries of a node.
Using the function, we compute the intersections with the source and target nodes:

$$I_s = \text{getBezierNodeIntersection}(N_s, P_s, C, P_t),$$
$$I_t = \text{getBezierNodeIntersection}(N_t, P_s, C, P_t).$$

8. **Adjusted Control Point C'**
Finally, we adjust the control point C using the intersection points I_s and I_t:

$$C' = 2O - \frac{1}{2}(I_s + I_t) = \begin{pmatrix} 2O_x - \frac{I_{s_x} + I_{t_x}}{2} \\ 2O_y - \frac{I_{s_y} + I_{t_y}}{2} \end{pmatrix}.$$

9. **Constructing the SVG Path String**
   Using the intersection points I_s, I_t, and the adjusted control point C', we construct the SVG path string for the quadratic Bézier curve:

   $$\text{SVG Path} = \texttt{M}\ I_{s_x}, I_{s_y}\ \texttt{Q}\ C'_x, C'_y\ I_{t_x}, I_{t_y}$$

   This path string represents:

   - `M`: Move to the starting point I_s.
   - I_{s_x}, I_{s_y}: The coordinates of the source intersection point.
   - `Q`: Draw a quadratic Bézier curve.
   - C'_x, C'_y: The coordinates of the adjusted control point C'.
   - I_{t_x}, I_{t_y}: The coordinates of the target intersection point.

   In SVG path notation, this is written as:

   ```
   M I_s.x,I_s.y Q C'.x,C'.y I_t.x,I_t.y
   ```

   This string can be used directly in an SVG path HTML element to render the curve connecting the two nodes with the calculated control point, ensuring the curve passes through the offset point and intersects the node boundaries appropriately. The offset point can be used to place an associated label or annotation.

Self-Loop Edge Paths

Definitions. We define the following:

- **Point**: A point in two-dimensional space, represented as (x, y).
- **Rectangle**: An object defined by its center position (x, y), width w, and height h.

Variables.

- N: The node is represented as a rectangle.
- $r \in \mathbb{R}$: The radius used for creating the self-loop.

From N, we have its center:

$$P_n = (x_n, y_n).$$

Calculations.

1. **Circle Center for the Self-Loop**: We place the circle center of the self-loop an offset r units above the node's center:

   $$\text{circleCenter} = (x_n,\ y_n + r).$$

   This would mean that the circle's topmost point passes through the node's center point.
2. **Label Point**: To place a label for the self-loop, we choose a point further above by an additional r units:

   $$\text{labelPoint} = (x_n,\ y_n + 2r).$$

3. **Intersections of the Node with the Circle**: To determine where the circle curve intersects the boundaries of a node (represented as a rectangle), we define a function:

   $$\text{getNodeIntersectionWithCircle}(N, \text{circleCenter}, r) \to \{I_0, I_1\}.$$

   where:

   - N: The rectangle represents the node.
   - circleCenter: The center point of the circle.
   - r: The radius of the circle.

   Let the returned intersection points be:

   $$I_0 = (I_{0_x}, I_{0_y}), \quad I_1 = (I_{1_x}, I_{1_y})$$

Algorithm 1 Find Intersection of Bézier Curve with Node Boundaries

1: **procedure** GETBEZIERNODEINTERSECTION(N, P_0, P_1, P_2)
2: $intersections \leftarrow \emptyset$ ▷ Initialize empty list of intersections
3: Define quadratic Bézier curve:

$$B(t) \leftarrow (1-t)^2 P_0 + 2(1-t)tP_1 + t^2 P_2 \quad \text{for } t \in [0,1]$$

4: **for all** vertical sides $x_{\text{side}} \in \{N_{\text{left}}, N_{\text{right}}\}$ **do**
5: Compute coefficients:

$$a \leftarrow P_{0x} - 2P_{1x} + P_{2x}, \quad b \leftarrow -2P_{0x} + 2P_{1x}, \quad c \leftarrow P_{0x} - x_{\text{side}}$$

6: Solve $at^2 + bt + c = 0$ for t
7: **for all** t where $0 \leq t \leq 1$ **do**
8: $B_y(t) \leftarrow y$-coordinate of $B(t)$
9: **if** $N_{\text{top}} \leq B_y(t) \leq N_{\text{bottom}}$ **then**
10: Add $(x_{\text{side}}, B_y(t))$ to $intersections$
11: **end if**
12: **end for**
13: **end for**
14: **for all** horizontal sides $y_{\text{side}} \in \{N_{\text{top}}, N_{\text{bottom}}\}$ **do**
15: Compute coefficients:

$$a \leftarrow P_{0y} - 2P_{1y} + P_{2y}, \quad b \leftarrow -2P_{0y} + 2P_{1y}, \quad c \leftarrow P_{0y} - y_{\text{side}}$$

16: Solve $at^2 + bt + c = 0$ for t
17: **for all** t where $0 \leq t \leq 1$ **do**
18: $B_x(t) \leftarrow x$-coordinate of $B(t)$
19: **if** $N_{\text{left}} \leq B_x(t) \leq N_{\text{right}}$ **then**
20: Add $(B_x(t), y_{\text{side}})$ to $intersections$
21: **end if**
22: **end for**
23: **end for**
24: **if** $intersections \neq \emptyset$ **then**
25: **return** $intersections[0]$
26: **else**
27: **return** $null$
28: **end if**
29: **end procedure**

Algorithm 2 outlines the algorithm to calculate the intersection points of a circle with the boundaries of a node.

Using the function, we compute the intersections of the circle and the node. If fewer than two intersection points are found, the loop cannot be drawn:

$$\text{if } |\{I\}| < 2 : \quad \text{return an empty path.}$$

4. **Constructing the SVG Arc Path**: With two intersection points on the circle, we can draw an arc that represents the self-loop. Using SVG path syntax:

$$\text{path} = \texttt{M}\ I_{0_x}, I_{0_y}\ \texttt{A}\ r, r\ 0\ 1, 0\ I_{1_x}, I_{1_y}.$$

Here:

- `M` I_{0_x}, I_{0_y}: Move to the first intersection point.
- `A` r, r 0 1, 0 I_{1_x}, I_{1_y}: Draw an arc with radius r from I_0 to I_1. The parameters 0 1, 0 define the large arc and sweep flags to ensure the arc is drawn in the correct direction (above the node).

This path string can be used in an SVG `<path>` element to render the self-loop. The label point can be used to place an associated label or annotation.

Algorithm 2 Find Intersection of Circle with Node Boundaries

1: **procedure** GETNODEINTERSECTIONWITHCIRCLE(N, C, r)
2: $intersections \leftarrow \emptyset$ ▷ Initialize an empty list of intersections
3: Define $sides$ of rectangle N:

$$sides \leftarrow \begin{cases} \text{Top: } (N_{\text{left}}, N_{\text{top}}, N_{\text{right}}, N_{\text{top}}), \\ \text{Bottom: } (N_{\text{left}}, N_{\text{bottom}}, N_{\text{right}}, N_{\text{bottom}}), \\ \text{Left: } (N_{\text{left}}, N_{\text{top}}, N_{\text{left}}, N_{\text{bottom}}), \\ \text{Right: } (N_{\text{right}}, N_{\text{top}}, N_{\text{right}}, N_{\text{bottom}}) \end{cases}$$

4: **for all** $side \in sides$ **do** ▷ Iterate over all rectangle sides
5: $(x_1, y_1) \leftarrow (side.x_1, side.y_1)$
6: $(x_2, y_2) \leftarrow (side.x_2, side.y_2)$
7: $dx \leftarrow x_2 - x_1, \quad dy \leftarrow y_2 - y_1$
8: $A \leftarrow dx^2 + dy^2$
9: $B \leftarrow 2 \cdot (dx \cdot (x_1 - C_x) + dy \cdot (y_1 - C_y))$
10: $C \leftarrow (x_1 - C_x)^2 + (y_1 - C_y)^2 - r^2$
11: $\det \leftarrow B^2 - 4 \cdot A \cdot C$ ▷ Calculate the determinant
12: **if** $\det \geq 0$ **then** ▷ Circle intersects the side
13: $t_1 \leftarrow \frac{-B+\sqrt{\det}}{2 \cdot A}$
14: $t_2 \leftarrow \frac{-B-\sqrt{\det}}{2 \cdot A}$
15: **for all** $t \in \{t_1, t_2\}$ **do** ▷ Check both roots
16: **if** $0 \leq t \leq 1$ **then** ▷ Intersection lies within the segment
17: $i_x \leftarrow x_1 + t \cdot dx, \quad i_y \leftarrow y_1 + t \cdot dy$
18: Add (i_x, i_y) to $intersections$
19: **end if**
20: **end for**
21: **end if**
22: **end for**
23: Sort $intersections$ by (x, y)
24: **for** $i = 1$ **to** $|intersections| - 1$ **do** ▷ Remove duplicate points
25: $dx \leftarrow intersections[i].x - intersections[i-1].x$
26: $dy \leftarrow intersections[i].y - intersections[i-1].y$
27: $dist \leftarrow \sqrt{(dx)^2 + (dy)^2}$
28: **if** $dist < 10^{-5}$ **then** ▷ If distance is negligible
29: Remove $intersections[i]$
30: **end if**
31: **end for**
32: **return** first two points in $intersections$
33: **end procedure**

Chapter 5

Case Studies

To demonstrate the unique features of the programming environment, we present four case studies that showcase its performance across a variety of scenarios drawn from previous literature. These case studies were carefully selected to represent the core concepts essential to addressing our research questions. Each case study focuses on a particular set of overlapping features of IGC, highlighting a key attribute essential to the design of that specific tool, even though none covers the entire feature set.

Our selection process involved a systematic review of existing studies and practical applications, ensuring that each case study was relevant and representative of the critical functionalities under investigation. For each study, we executed targeted experiments to directly compare the performance of the programming environment against established benchmarks. In these experiments, we analyzed both the strengths and limitations of IGC, providing a nuanced perspective on its design. Finally, we reflected on the outcomes to discuss what the tool and IGC do well, as well as areas where they fall short. This methodological approach not only grounds our evaluation in concrete evidence but also offers insights into potential avenues for future improvement.

5.1 Case Study 1: Computational Notebook

5.1.1 Introduction

The first case study focuses on the Jupyter Notebook [2], a computational notebook environment that has gained widespread popularity [22] due to its interactive and incremental programming capabilities. Evolving from the IPython project [1], it allows users to combine executable code, narrative text, and visual outputs into a sequence of cells, thus forming a coherent computational narrative that embraces the literate programming philosophy. This structure not only enhances experimentation and iterative development but also supports reproducibility thanks to its export and import functionality, where the entire notebook is saved as a JSON file that captures both the code and its accompanying documentation, enabling easy sharing and replication of research [10].

Main Features of Jupyter Notebook. Jupyter Notebook offers several key features that make it a powerful tool for interactive computing. It enables interactive code execution, where users can write and run code in individual cells, receiving immediate feedback and results. This interactivity supports exploratory programming and data analysis, allowing for rapid iteration and experimentation.

The notebook supports multiple programming languages through its kernel architecture, enabling users to work with Python, R, Julia, and over 100 other languages [22]. This flexibility makes Jupyter a versatile tool suitable for a wide range of applications.

Jupyter also supports various graphical outputs. Users can display images, videos, interactive plots, and mathematical equations directly within the notebook, enhancing the visualization of data and results. The integrated documentation feature allows users to write explanatory text using Markdown cells, facilitating the creation of comprehensive computational narratives.

Additionally, Jupyter's extensibility allows for customization and the addition of extensions, enabling users to tailor their environment to specific workflows or domains [22]. The platform supports collaboration and sharing through tools like GitHub and JupyterHub, and services like Binder allow others to interact with notebooks without requiring local installations [1, 22].

5.1.2 Showcase: Computing Basic Statistics

Jupyter Notebook Showcase. To illustrate the capabilities of Jupyter Notebook, consider a data analyst tasked with computing basic statistics on a dataset. The dataset consists of numerical values, and the goal is to calculate the mean, median, and standard deviation.

The user begins by opening a new Jupyter Notebook, which provides an interactive environment with cells containing code or Markdown text. The incremental and interactive nature of Jupyter allows the user to build the analysis step by step, receiving immediate feedback at each stage.

Computing Basic Statistics

In this example, we will generate a dataset of random numbers and compute the mean, median, and standard deviation.

```python
import numpy as np

# Generate a dataset of 100 random numbers
data = np.random.rand(100)
```

[1] ✓ 0.0s Python

```python
mean = np.mean(data)
median = np.median(data)
std_dev = np.std(data)

print(f"Mean: {mean:.4f}")
print(f"Median: {median:.4f}")
print(f"Standard Deviation: {std_dev:.4f}")
```

[2] ✓ 0.0s Python

```
Mean: 0.4926
Median: 0.4627
Standard Deviation: 0.2842
```

The mean and median provide measures of the central tendency of the dataset, while the standard deviation indicates the spread of the data around the mean.

Figure 5.1: Jupyter Notebook Showcase: Computing Basic Statistics

The showcase of the Jupyter Notebook is shown in fig. 5.1.

By executing these cells sequentially, the user interacts with the data, performs computations, and documents the process. The immediate feedback from code execution and the integration of explanations enhance understanding and facilitate communication of the results.

Applying the Showcase in IGC. To apply the same showcase as above in IGC, the user creates a graph structure with nodes representing code fragments and documentation. The user first creates the corresponding code fragment nodes from the Jupyter Notebook cells. Documentation nodes are attached to code fragments, providing Markdown explanations similar to the Markdown cells in Jupyter. Now, to execute the code, the user needs to initiate a new session and execute the nodes in sequence, following the defined execution path. The session records the execution history, which can be revisited or shared with collaborators. Textual outputs from code execution are displayed in the code view pane. The entire session, including execution data and documentation, can be exported, ensuring that the computational narrative and execution path are preserved.

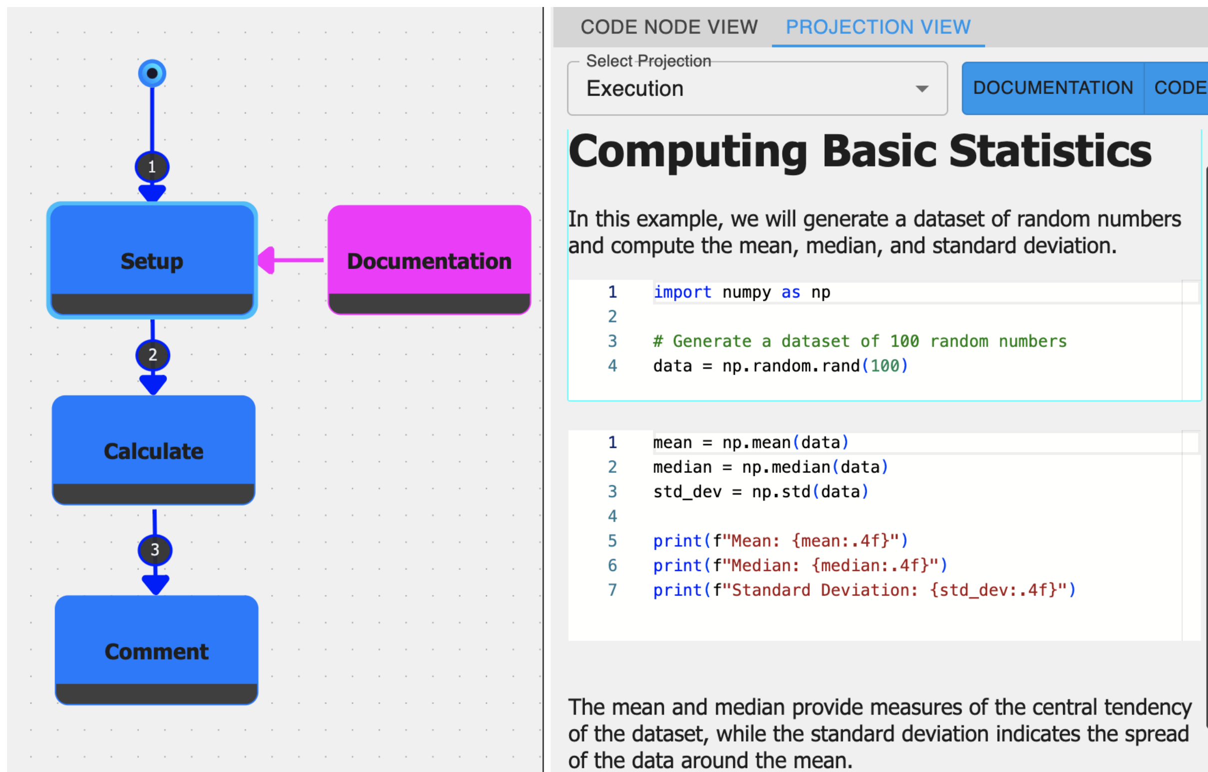


Figure 5.2: IGC Showcase: Computing Basic Statistics

The showcase of the IGC version of the previous showcase is shown in figure fig. 5.2 using the projectional view for the execution.

5.1.3 Comparing IGC to Jupyter Notebook

Comparison of Differences in the Showcase. When comparing the experience of performing the basic statistics showcase in Jupyter Notebook versus IGC, several distinctions become apparent. In Jupyter, while cells are typically executed in order, the user can execute them in any sequence, potentially leading to inconsistencies such as incorrect cell execution. In contrast, IGC's visual representation of the execution path through its graph structure enforces a defined execution sequence, reducing the risk of executing code fragments in an unintended order. In Jupyter, code cells display numbers in square brackets (e.g., [1]) that indicate the order of execution, but this numeric notation is less visual and cannot show if a cell has been executed multiple times.

Jupyter allows for quick prototyping and immediate visualization within the notebook. However, as projects grow in complexity, the linear cell structure can make it difficult to manage and understand dependencies between different parts of the code [61]. IGC's graph-based organization allows users to manage complex codebases more effectively by visually mapping out code fragments and their interdependencies.

However, IGC's current limitation to textual outputs means that users cannot visualize data directly within the environment. In the basic statistics showcase, while numerical results are displayed, any graphical representations would need to be generated externally.

Another notable difference in the projectional view of IGC is that the direct outputs are not displayed under the code cells, as seen in Jupyter. In Jupyter, the immediate feedback from code execution is visible below the code cell, providing a quick overview of the results. In IGC, users need to refer to the textual outputs in the code view pane, which may require additional scrolling and navigation.

The last key difference is the documentation or textual explanations. The current implementation of IGC is that one code node can have one documentation node that is displayed above the code. This differs from Jupyter because you can make as many documentation markdown cells as the user wants in any order, whereas in IGC, you would have to condense them all into one documentation cell. In the showcase, the user had to create an empty code node to attach the final documentation node, a workaround to display the documentation at the end of the code. This limitation highlights an area for

improvement in IGC's current implementation.

General Comparison of IGC and Jupyter Notebook. While Jupyter Notebook excels in interactivity and ease of use, it has certain limitations that can impede productivity, especially in complex projects. A significant challenge is maintaining the execution order of cells. Since cells can be executed in any order, the notebook's state can become inconsistent, leading to reproducibility issues and confusion [10].

IGC addresses this challenge by introducing a graph-based programming environment where code fragments, represented as nodes, and their relationships, represented as edges, are explicitly defined. Execution relationships in IGC are visualized as edges in a directed graph, providing a clear and reproducible execution path within sessions. This explicit representation mitigates the risk of inconsistencies arising from out-of-order execution, a common issue in traditional notebooks [10].

IGC's two-dimensional graph structure allows users to organize code fragments spatially and define relationships such as dependencies and inheritance, enhancing code manageability. This approach can potentially better handle complexity compared to the linear arrangement of cells in Jupyter, which can become unmanageable in large notebooks.

IGC does have some limitations compared to Jupyter Notebook. Jupyter supports various media outputs like images and interactive plots embedded directly within the notebook, while IGC currently supports only textual outputs. This limitation may hinder users who rely heavily on visual data exploration. IGC also lacks the output data shown in the projectional view, which is a feature that Jupyter has. Both platforms integrate documentation with code: Jupyter uses Markdown cells for inline documentation, whereas IGC allows documentation nodes to be attached to code fragments, displaying markdown above the code view, similar in effect to Jupyter's approach, but not as flexible.

Conclusion. Jupyter Notebook has established itself as a powerful tool for interactive computing, enabling users to integrate code, data, and narrative seamlessly. Its strengths in facilitating exploratory analysis and providing immediate feedback make it indispensable in many fields. Nevertheless, challenges with execution order and code organization can hinder productivity in complex projects [61].

IGC offers an alternative approach by leveraging a graph-based interface that emphasizes explicit relationships and execution paths. This structure enhances the management of complex codebases and ensures reproducibility by maintaining clear execution sequences. Although IGC currently lacks support for complex media outputs, its features address some of the pain points identified in traditional notebooks, such as execution order confusion and difficulty in handling complexity [10].

By exploring and comparing different approaches to computational notebooks, we can work towards creating environments that better support the needs of researchers, data scientists and software engineers.

5.2 Case Study 2: Code/Documentation Projections

5.2.1 Introduction

Introduction. The second case study focuses on PescaJ [48], a projectional editor designed for Java that addresses the challenge of scattered code and documentation through aggregated views. Projectional editors provide the ability to look at the program from multiple views, which is often requested as a means to simplify program comprehension. Different projections can be useful in different contexts and generally have positive effects on program comprehension by providing a higher-level view of the system [62].

PescaJ departs from conventional text-based editing by allowing developers to create customizable, overlapping views that consolidate both code and documentation fragments, which are often dispersed across different files and classes in traditional Integrated Development Environments (IDEs). This approach is particularly beneficial for managing the complexity of large software systems, where the relationships between code segments and their associated documentation can be difficult to navigate. This capability is valuable in maintaining synchronization between code and its documentation. Users debugging their project can alleviate the cognitive load typically caused by navigating dispersed data by using an interface that consolidates related information due to the Spatial Split Attention Effect [63]. This can especially be valuable to programmers unfamiliar with a codebase trying to explore the context and the project call flow.

Main Features of PescaJ. PescaJ has several features that improve code comprehension and documentation management. Unlike traditional text editors, PescaJ employs projectional editing, where the editing representation is distinct from the storage representation. This allows for the creation of views that can display and edit code fragments from different classes or files in a single workspace. Users can form views that aggregate methods and documentation scattered across a codebase, facilitating simultaneous editing and viewing of related elements.

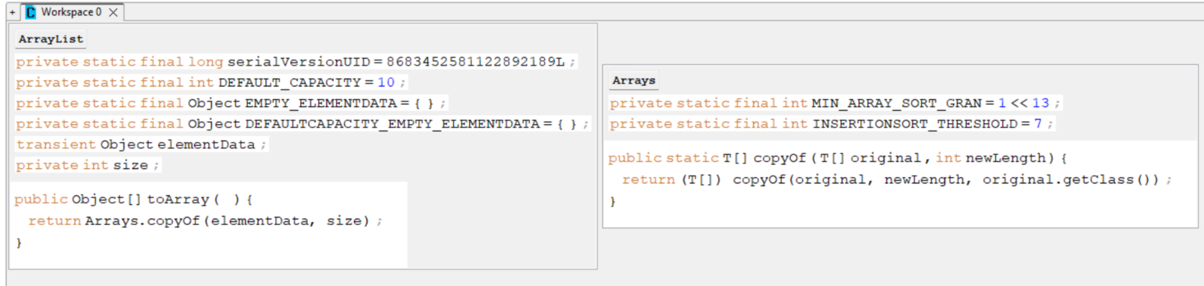


Figure 5.3: (Image from Lopes et al [48]) An inter-class view illustrated with a fragment of the classes ArrayList and Arrays (Java libraries), where the method toArray depends on method copyOf, which are juxtaposed.

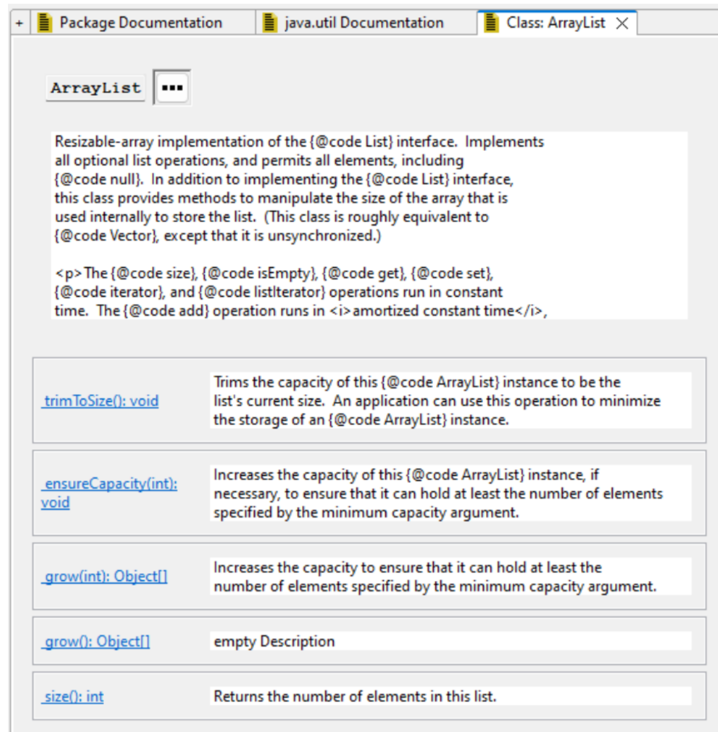


Figure 5.4: (Image from Lopes et al [48]) View that aggregates the documentation of the methods in a class, showing a documentation header for each method.

PescaJ provides both code and documentation views, which can be intra-class (within a single class) or inter-class (across multiple classes). Code views allow developers to juxtapose methods based on dependencies, facilitating easier navigation and understanding of code relationships. Documentation views aggregate scattered Javadoc comments, presenting them in a cohesive manner that helps maintain consistency and coherence.

Workspaces are described as canvases where users can place different views. This organization allows developers to show related information and switch between different contexts without the need to navigate through multiple files or scroll extensively within a single file. The views within a workspace are synchronized, ensuring that changes made to code or documentation are reflected across all relevant views.

5.2.2 Example Use Case: Managing Scattered Code and Documentation

PescaJ Use Case. Consider a developer working on a Java project with multiple classes that have interdependent methods and associated documentation. In a traditional IDE, the developer would need to navigate between different files and scroll through large codebases (perhaps aided by call hierarchies and "jump to definition" navigation options) to understand the relationships and update documentation.

Using PescaJ, the developer can create a workspace that contains:

- **Code View**: An inter-class code view that displays methods from different classes based on their call dependencies. For instance, methods `A`, `B`, and `C` from classes `Class1`, `Class2`, and `Class3`, respectively, can be displayed side by side.
- **Documentation View**: A documentation view that aggregates Javadoc comments for these methods, presenting them in a single pane. This view allows the developer to update and maintain documentation consistency across methods and classes.

By clicking on a method call within the code view, PescaJ expands the workspace to include the called method's code and documentation, placing it adjacent to the calling method. This juxtaposition reduces cognitive load and enhances the developer's ability to comprehend and modify code and documentation efficiently.

Applying IGC for the Use Case. In IGC, the graph-based code structure is different from that of PescaJ, but there, the views can be created in a similar manner. IGC represents code fragments and documentation as nodes within a graph. When a documentation node is attached to a code node, it appears directly above the code in the code node view (section 4.3.6). This is a case of using projections to correlate source code to its documentation. More of the functional overlap between PescaJ and IGC can be shown in the Projectional View(section 4.3.6).

IGC's projectional view allows users to select different projection settings:

- **Execution Projection**: Displays all nodes that are part of the current execution path are displayed. This view allows developers to visualize the sequence of code execution and navigate through the code fragments in the order they are executed.
- **Class Projection**: Displays all nodes belonging to the class of the selected node. This view enables developers to focus on the code and documentation within a specific class.
- **Dependency Projection**: Displays all nodes that are dependent on a variable chosen from the selected node. The user will have a dropdown of variables used in the code fragment of the selected node. Once selected, the projection will show all nodes that are dependent on the selected variable. This view helps developers understand the relationships between code fragments and their dependencies.

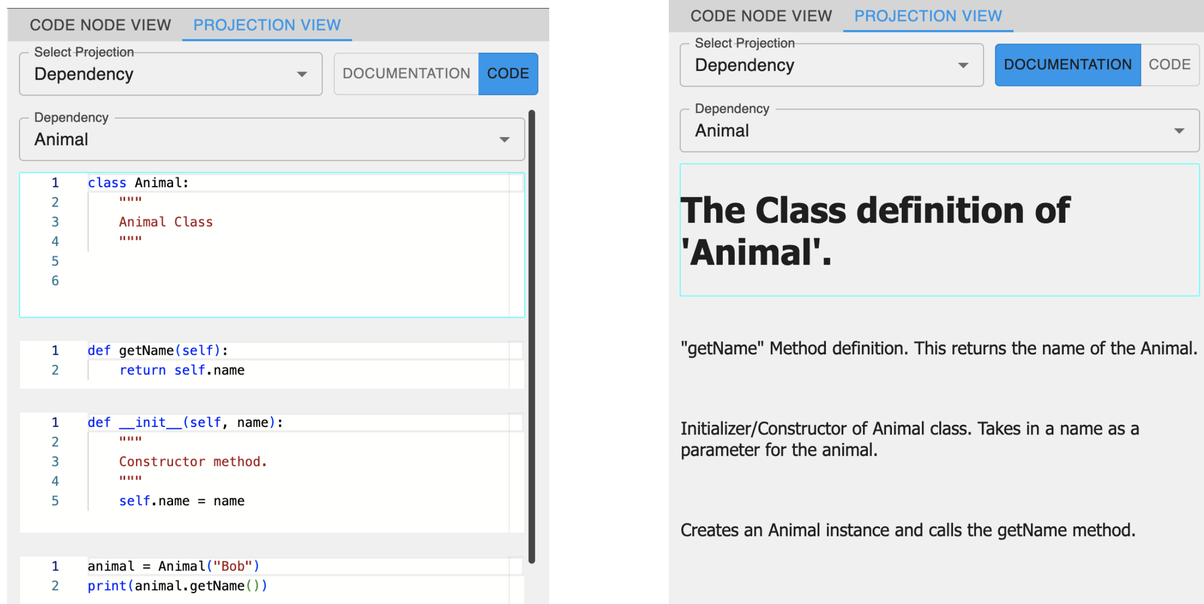
Users can toggle the visibility of code and documentation within these projections. By customizing the projection settings, developers can create views that aggregate related code and documentation, similar to PescaJ's workspace views.

5.2.3 Comparing IGC to PescaJ

When comparing PescaJ and IGC in handling the example use case, several similarities and differences emerge. Both PescaJ and IGC enable the aggregation of code and documentation that are scattered across different files or classes. PescaJ achieves this through customizable views within a workspace, while IGC utilizes a projectional view to display related nodes through different projection settings. PescaJ visually represents method dependencies by placing called methods adjacent to their callers, forming a chain of code fragments. IGC, on the other hand, represents dependencies relationships explicitly as edges in a graph, allowing users to navigate dependencies through the graph structure or through the projectional view.

In PescaJ, the Model-View-Controller (MVC) architecture ensures that changes in code or documentation are synchronized across all views. IGC maintains synchronization through its node-based (flow) model, where updates to a node are reflected wherever the node is displayed.

PescaJ provides a specialized interface focused on reducing cognitive load by minimizing navigation and leveraging spatial contiguity. IGC offers flexibility in organizing code and documentation through its graph interface, but also allows for the automatic, specialized interface of the aggregation of code fragments and documentation within specific views, such as the code node view and projectional view.



(a) Dependency chain projection of the code fragments (Only code enabled).

(b) Dependency chain projection of the documentation (Only documentation enabled).

Figure 5.5: Both images show the dependency chain projection of the same code; both show the corresponding dependencies for both the implementation and usage of the "Animal" class. The top-right toggle determines what is shown.

While both platforms aim to improve code comprehension and management, there are differences. IGC's graph-based interface allows for a more explicit representation of relationships through edges, potentially allowing the user to create projections of relationships outside the scope of PescaJ.

IGC's projectional view offers customizable ways to display code and documentation based on execution paths, class membership, or dependencies. PescaJ's views are more focused on call hierarchies but in general have a more customizable way of creating different views within many workspaces.

Another difference is that PescaJ allows users to edit code and documentation directly within the aggregated views, whereas IGC's projectional view is currently read-only. This difference may impact the user experience, as PescaJ users can make changes directly in the aggregated view, while IGC users currently need to navigate to the specific node to make edits.

Another distinction is that PescaJ automatically extracts Javadoc documentation [64], which can be easily created when writing source code, whereas IGC requires users to attach documentation nodes to code nodes. This difference may affect the user experience, as PescaJ users can rely on existing Javadoc comments, while IGC users need to create separate documentation nodes to associate with code fragments. However, there is nothing stopping us in the future from creating a feature to extract Javadoc comments just as PescaJ does.

Conclusion. PescaJ introduces an innovative approach to code and documentation management by leveraging projectional editing to aggregate scattered elements, reducing the cognitive load associated with navigating large codebases. Its ability to create overlapping views that synchronize code and documentation can enhance developer productivity and code comprehension.

IGC shares similar goals in improving code organization, but implements them through a graph-based interface with projectional views that offer flexibility in displaying related code and documentation. While both platforms address the challenges of scattered code and documentation, they differ in their representation of relationships, synchronization mechanisms, and editing capabilities.

By comparing PescaJ and IGC, we can identify the strengths and limitations of each approach and explore how different interfaces can support developers in managing complex software systems.

5.3 Case Study 3: Architectural Diagrams and Composition

5.3.1 Introduction

Introduction. As a software system grows in size, generally, the code complexity of said system tends to increase. A solution to mitigate complexity is to create sub-systems or components to reduce the complexity into smaller manageable parts [65]. One of the primary challenges in managing and maintaining large-scale software systems is the ability to clearly represent and understand the relationships between software components [66]. Traditional tools like the Unified Modeling Language (UML) diagrams provide high-level abstractions to represent system components and their interactions, aiding developers in understanding and maintaining the overall architecture [67]. However, creating and maintaining these diagrams can be time-consuming and may not always accurately reflect the underlying codebase, leading to inconsistencies and misunderstandings [66].

The Model-View-Controller (MVC) design pattern is a widely adopted architectural style that separates an application into three interconnected components: Model (data and business logic), View (user interface), and Controller (input processing)[68]. Visualizing the relationships and interactions between components is informative for developers to comprehend and maintain the system effectively[69].

IGC introduces *Graph Nodes* as a means to encapsulate and compose complex graphs, enabling developers to manage large-scale systems by abstracting and consolidating components. By allowing one graph to contain another, along with its execution context (session), IGC facilitates a modular approach to software architecture, similar to importing libraries or designing "sudo-UML" diagrams for high-level design.

5.3.2 Main Features of Graph and Composition Nodes

Graph Nodes in IGC. Graph Nodes in IGC are specialized nodes that reference an external IGC graph file and a specific session within that graph. Essentially, a Graph Node acts as a composite node, encapsulating an entire IGC graph and its execution state. This feature allows developers to modularize their code by encapsulating functionality within separate graphs and then composing them within a larger graph.

Composition and Abstraction. By using Graph Nodes, developers can build abstractions and higher-level components, assembling complex systems from simpler, reusable parts. This approach is similar to the concept of modules or libraries in traditional programming, where functionality is encapsulated and made available for reuse. In the context of IGC, this modularity extends to both the code and its execution context, enabling developers to manage and interact with components at different levels of abstraction.

Modeling Software Architectures. Graph Nodes enable the visualization and modeling of software architectures directly within the development environment. Developers can represent architectural patterns, such as MVC, by creating separate graphs for each component and composing them using Graph Nodes. This approach allows for a dynamic and interactive representation of the system's architecture, which can be more accurate and up-to-date compared to static UML diagrams.

5.3.3 Example Showcase: Modeling MVC Architecture

Graph Nodes for MVC Components. Consider an application that follows the MVC architectural pattern. The application consists of a Model Graph, which contains nodes representing data structures and business logic; a View Graph, which contains nodes representing the user interface and presentation logic; and a Controller Graph, which contains nodes managing user input and orchestrating interactions between the Model and View.

Each component is developed and maintained as a separate IGC graph file with its own execution sessions. Developers can work on these components independently, promoting modularity and separation of concerns [68].

Composing the MVC Architecture. Using Graph Nodes, the developer creates a Main Graph that composes the Model, View, and Controller graphs. In the Main Graph, the developer includes a Model Node, which is a Graph Node referencing the Model Graph and a specific session, a View Node, which is

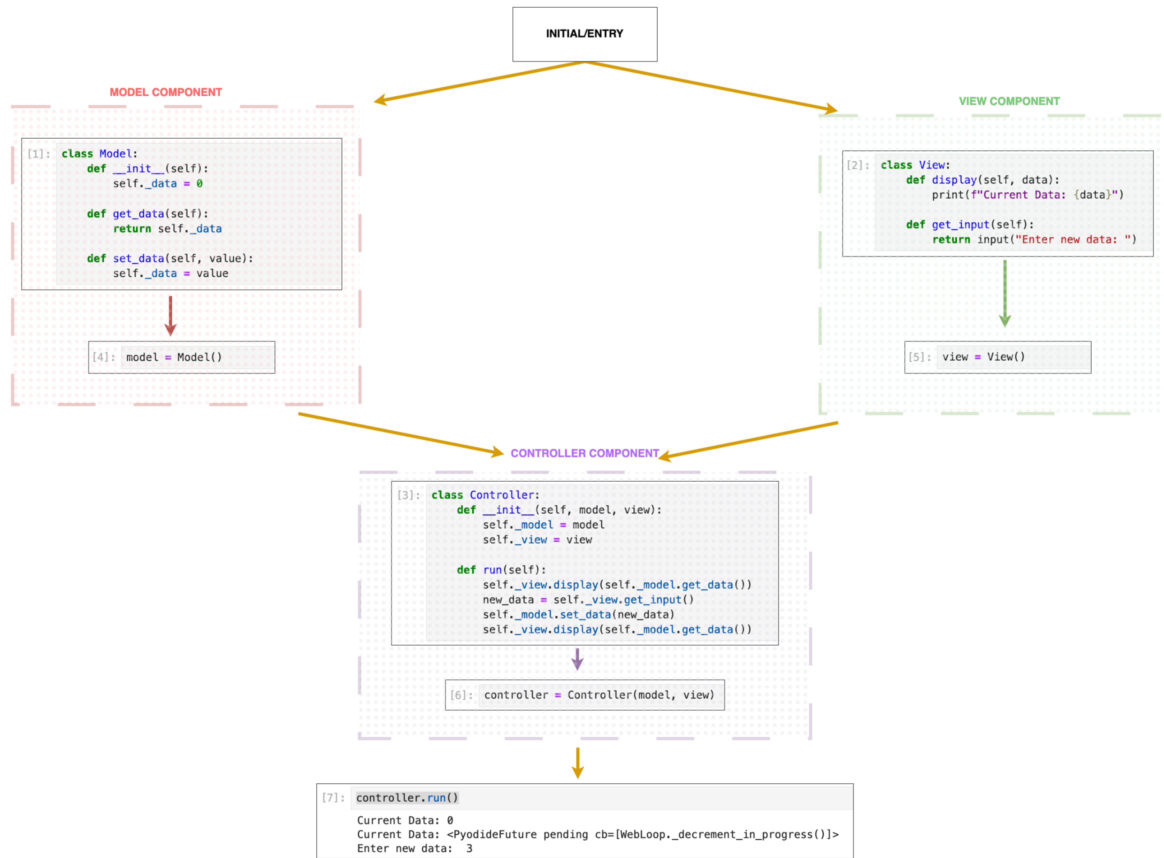


Figure 5.6: Example of how the test MVC architecture is modeled

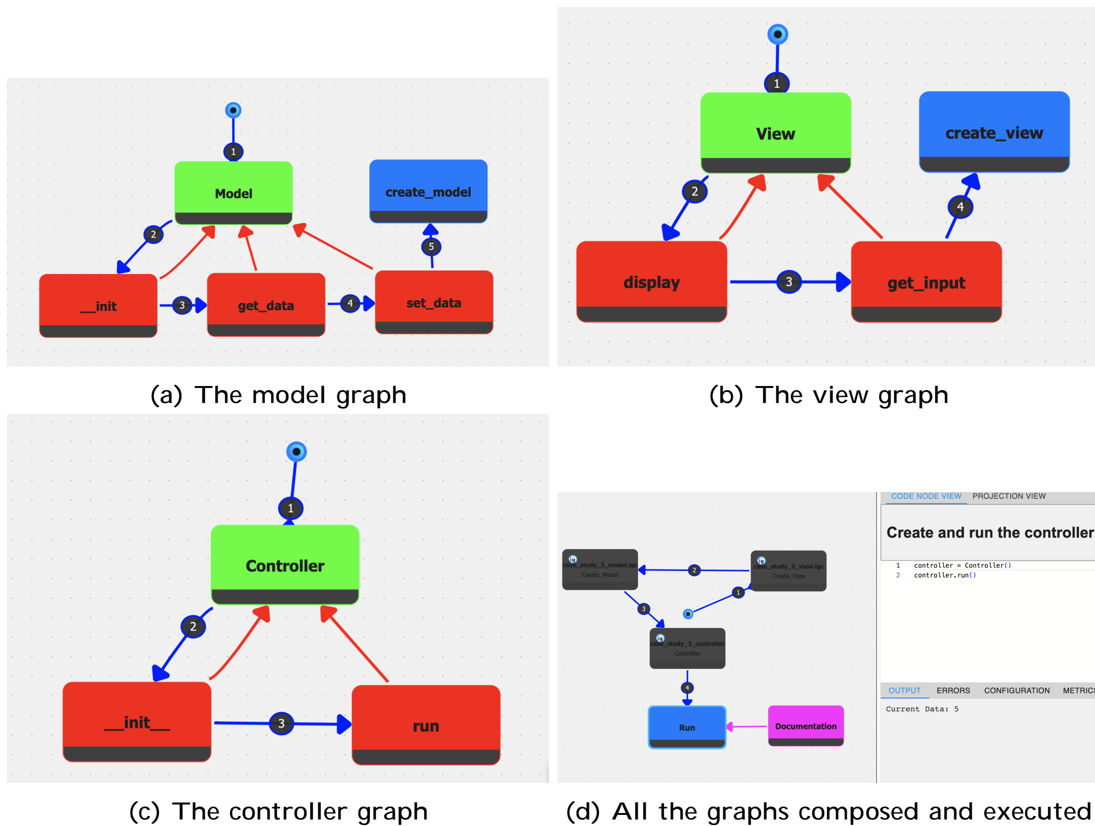
a Graph Node referencing the View Graph, and a Controller Node, which is a Graph Node referencing the Controller Graph.

Edges are established between these Graph Nodes to represent the interactions and dependencies among the MVC components. For example, the Controller Node has edges connecting it to both the Model Node and the View Node, indicating that it mediates communication between them.

Visualization and Interaction. This composition allows developers to visualize the overall architecture within IGC, seeing how components are interconnected. Since Graph Nodes encapsulate both the code and its execution context, developers can execute the Main Graph and observe the behavior of the entire application, stepping into individual components as needed.

Applying the Example in IGC: Creating Component Graphs. Developers begin by creating separate IGC graph files for the Model, View, and Controller. The Model graph, named `Model.igc`, contains nodes defining data models and business logic functions. The View graph, named `View.igc`, includes nodes responsible for rendering the user interface. The Controller graph, named `Controller.igc`, comprises nodes that handle user input and coordinate interactions between the Model and View. Each graph can have its own sessions representing different states or versions of the component, facilitating independent development and testing.

Composing with Graph Nodes. To compose the MVC architecture in the Main Graph, the developer first creates a new file named *MVC.igc* containing a Graph Node for each component. The IGC File Path is set to point to the respective component graph file, and the desired session for each component is selected. Next, edges are established between the Graph Nodes to represent the flow of data and control. The Controller Node is connected to both the Model Node and the View Node, defining the relationships and dependencies explicitly. Finally, any additional nodes or logic required to integrate the components are added, such as running the controller.



(a) The model graph

(b) The view graph

(c) The controller graph

(d) All the graphs composed and executed

Figure 5.7: All the graph nodes for the MVC architecture

Executing and Interacting. With the Main Graph set up, developers can execute the application as a whole, with the Graph Nodes handling the execution of their respective components. They can also inspect individual Graph Nodes to examine or modify the underlying graphs and sessions. Additionally, developers can switch sessions associated with Graph Nodes to test different versions or configurations of components.

Visualization of Architecture. The Main Graph provides a visual representation of the application's architecture, similar to a UML diagram but directly tied to the actual code and execution context. This dynamic visualization aids in understanding and maintaining the system [66].

5.3.4 Comparing IGC to Traditional Approaches

Comparison with UML Diagrams. Traditional UML diagrams offer static representations of software architectures, useful for planning and communication but often detached from the actual codebase [67]. In contrast, IGC's Graph Nodes provide a dynamic and executable representation. The architecture visualization in IGC reflects the current state of the code, reducing discrepancies. Developers can interact with the Graph Nodes to inspect or modify components, facilitating deeper understanding. Execution sessions within Graph Nodes allow for testing and debugging within the architectural view.

Comparison with Model-Driven Development. Model-driven development (MDD) emphasizes creating models that can be transformed into executable code [70]. IGC's approach shares similarities but differs in key aspects. In IGC, the model (graph structure) and the code are intertwined, allowing for simultaneous development and visualization. Unlike MDD, where models are transformed into code, IGC's graphs are the code, eliminating potential inconsistencies. Additionally, Graph Nodes in IGC enable modular composition without the overhead of maintaining separate models and codebases, providing flexibility and modularity.

Advantages of IGC's Approach. IGC's use of Graph Nodes offers several benefits. Firstly, it encourages modularity by allowing components to be developed independently, promoting separation of

concerns. Secondly, it enhances reusability, as components encapsulated in Graph Nodes can be reused across different projects. Thirdly, it simplifies scalability by breaking down large systems into manageable pieces, making it easier to manage complex architectures. Lastly, it provides clarity by offering a clear and accurate visualization of the system's architecture, aiding comprehension and communication.

Potential Limitations. While IGC's approach has advantages, it may also present challenges. Overusing nesting graphs might lead to complex hierarchies that are difficult to navigate, posing complexity management challenges. Also, integration with other development tools and workflows may require additional effort, which could impact tooling support.

Conclusion. IGC's introduction of Graph and Composition Nodes offers a powerful mechanism for managing and visualizing complex software architectures. By encapsulating graphs within Graph Nodes, developers can create modular, reusable components that reflect architectural patterns like MVC directly within the development environment. This approach bridges the gap between high-level architectural visualization and low-level code implementation, providing an integrated platform for development, visualization, and execution.

Compared to traditional UML diagrams and Model-Driven Development, IGC's method offers a dynamic and accurate system representation, promoting better understanding and maintenance. While there may be challenges in adopting this paradigm, the benefits of modularity, scalability, and clarity present a compelling case for incorporating graph nodes into software development practices.

5.4 Case Study 4: Exploratory Programming GUI

5.4.1 Introduction

Introduction. The final case study we will present is an implementation of exploratory programming techniques that allow the user to alter execution paths or sessions seamlessly. Exploratory programming is a programming style characterized by prototyping, experimentation, and iterative refinement without a predefined end goal [5]. It is a key practice in which programmers actively experiment with different possibilities using code [3]. This approach is valuable in contexts such as data science, education, and rapid prototyping, where the ability to test ideas and explore solutions quickly is essential. Traditional programming environments often fall short in supporting this style due to rigid edit-compile-run cycles and limited feedback mechanisms [5].

The paper, "A Language-Parametric Approach to Exploratory Programming Environments" [5], discusses an experimental computational Notebook front-end in Section 6. For simplicity, we will refer to this as the Exploratory Programming GUI. The GUI in this Notebook allows the user to create and manipulate execution paths called traces. Each trace can be compared to internal runtime state values with other execution traces. Having a GUI to compare execution paths to promote exploratory programming similar to the example above is a great feature to have as experimentation and exploration have been found to be beneficial to data science and computational science [6, 7]. However, these tools have their own constraints, particularly in managing execution state, versioning code fragments, and exploring alternative execution paths [21].

Main Features of the Exploratory Programming GUI The Exploratory Programming GUI introduces several key features to support exploratory programming. One of the central components is the use of execution graphs, data structures that record the execution history as a tree of configurations, allowing users to navigate and manipulate different execution states easily. Additionally, the Exploratory Programming GUI supports branching and merging, enabling users to create branches from any point in the execution graph. This feature allows for the exploration of alternative code paths without losing previous work, a key aspect of exploratory programming.

Incremental code execution is another key feature of the Exploratory Programming GUI. Users can execute code fragments incrementally, observing the effects on the program state at each step. This incremental execution supports an iterative development process, where users can refine their code gradually based on intermediate results.

The Exploratory Programming GUI also provides interactive state management, offering immediate feedback by displaying the effects of code execution, including outputs and state changes. This interactivity enhances the user's understanding of how code modifications affect the program.

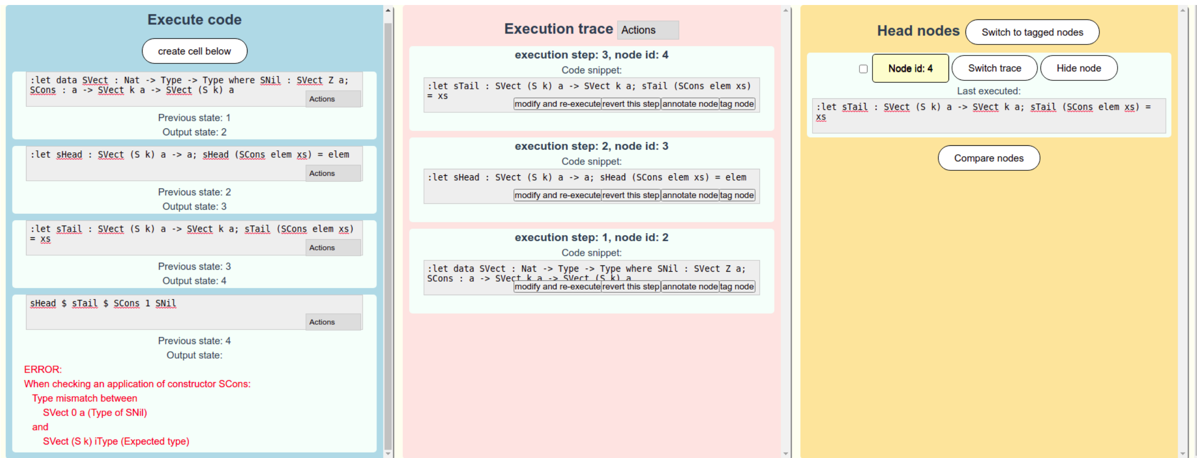


Figure 5.8: (Image from van Binsbergen et al [5]) Overview of the exploratory programming environment prototype GUI.

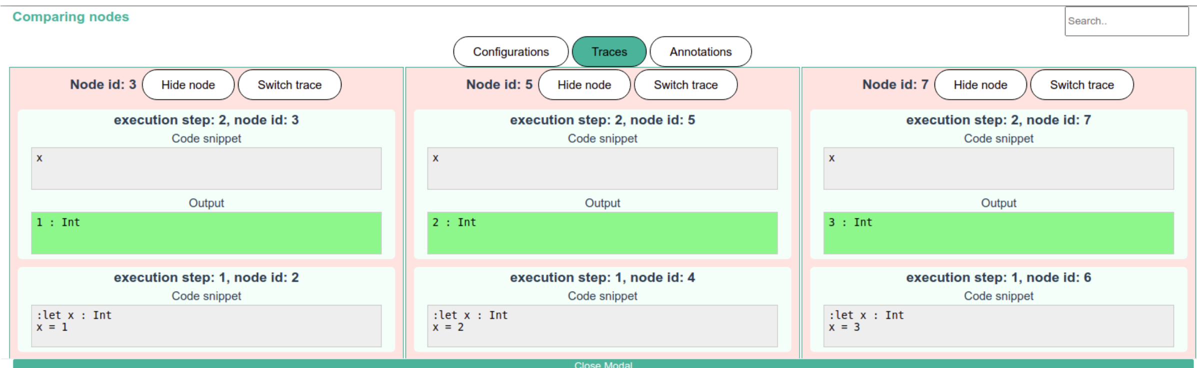


Figure 5.9: (Image from van Binsbergen et al [5]) Demonstration of how traces are compared in the exploratory programming environment prototype.

Finally, features like undo and redo actions allow users to revert to previous states and redo actions, adding to the iterative experimentation and refinement features. Together, these features enhance the exploratory programming experience by providing tools to manage execution state, experiment with code variations, and receive immediate feedback.

5.4.2 Example Use Case: Managing Execution State and Branching

Exploratory Programming GUI Use Case. Consider a developer experimenting with code in the Exploratory Programming GUI, which features a main screen divided into three sections: the notebook interface on the left, the execution trace in the center, and the head and tagged nodes on the right.

The developer begins by writing code fragments in the notebook's code cells. After executing a code cell, the output and the resulting program state are displayed. Each execution adds a new node to the execution trace in the center panel, representing the path from the initial state to the current state.

To explore different implementations, the developer modifies a code cell and re-executes it. The GUI allows them to revert to previous states or create new branches in the execution tree. By using the "Modify and Re-execute" feature, the developer can execute the modified code from a prior state, generating a new branch and preserving the original execution path.

The right panel displays head nodes, corresponding to the leaves of the execution tree, and tagged nodes, which are specific states the developer has marked as significant. Selecting different head or tagged nodes lets the developer switch between execution traces, facilitating comparison of various code paths and outcomes.

For in-depth analysis, the developer can select multiple nodes and use the "Compare Nodes" function to view configurations side by side. This comparison includes code differences, runtime states, and any annotations, aiding in understanding the impact of changes.

The GUI also supports sharing and collaboration by allowing the export and import of execution trees or individual traces. This functionality enables developers to share their exploration sessions with others or incorporate version control, enhancing reproducibility and reuse of code experiments.

Exploritory Programming in IGC. In IGC, the exploratory programming capabilities are facilitated through the use of *sessions* and the *Session Manager*. A session represents a distinct execution path defined by a sequence of executed code nodes. Users can create, manipulate, and compare different sessions to explore various execution states and code modifications.

The **Session Manager** in IGC provides a comprehensive view of all sessions in a tree structure (see fig. 5.10). The root of the tree is the *start node*, representing the initial empty state from which all execution paths begin. Each branch corresponds to a different session, with nodes representing executed code fragments. Overlaps in the tree indicate shared prefixes in execution paths among sessions.

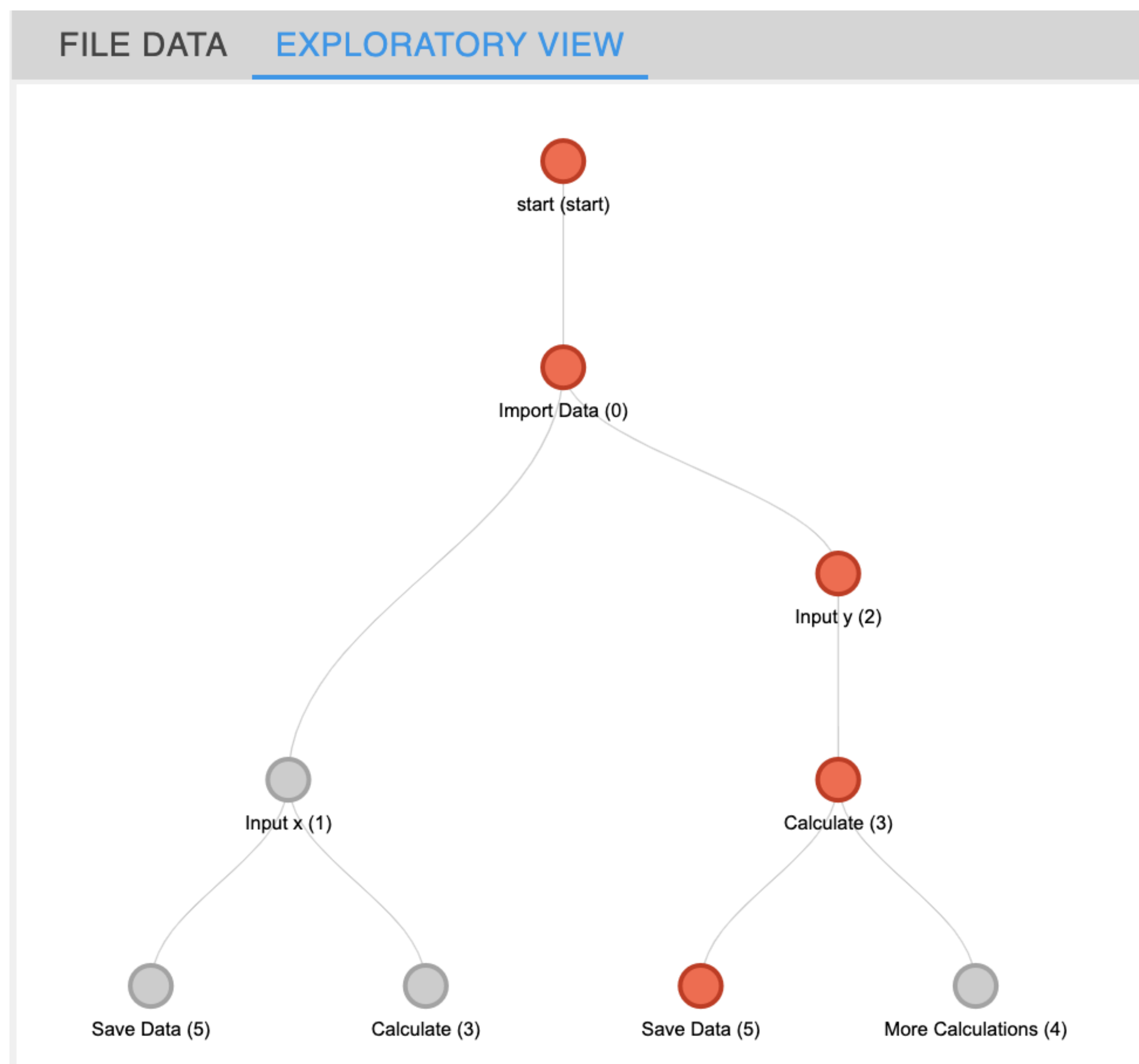


Figure 5.10: IGC Exploratory view showing the execution tree with multiple sessions. The selected session is highlighted in red.

Users can interact with the Session Manager to manipulate sessions in several ways by right-clicking on a node in the current (selected) session:

- **Start Session from Here**: Creates a new session beginning at the selected node, allowing exploration from a specific point in the execution path.
- **Insert Before Node**: Generates a new session identical to the current one but with an additional node executed *before* the selected node. After right-clicking, the user selects the node to insert.

- **Insert After Node**: Creates a new session identical to the current one but with an additional node executed *after* the selected node. The node to insert is selected after right-clicking.
- **Replace Node**: Produces a new session where the selected node in the current session is replaced with another node chosen by the user.

These manipulation options enable rapid experimentation and iteration, key aspects of exploratory programming. By quickly making changes and creating new sessions, users can compare different execution results efficiently.

Switching between sessions is straightforward within the Session Manager. The selected session is highlighted in red, while inactive sessions are displayed in dark grey. This visual distinction assists users in keeping track of multiple exploration paths.

Example Workflow in IGC. Suppose a user is developing a function and wants to explore different implementations. They can proceed as follows:

1. Execute the initial code nodes to create the first session.
2. In the Session Manager, right-click on a node where they want to modify the execution path.
3. Choose "Replace Node" and select an alternative code node representing a different implementation.
4. A new session is created with the modification, and the execution continues from that point.
5. Compare the outputs or execution states between the original and new sessions by switching between them in the Session Manager.

This approach lets the user explore various code modifications without losing previous work. By creating new sessions from specific points in the execution path, users can compare different outcomes and make informed decisions about code changes.

5.4.3 Comparing IGC to the Exploratory Programming GUI

Differences in Approach. While both environments support exploratory programming, their approaches differ in several key aspects. IGC uses a graph-based interface that provides a visual representation of code dependencies and execution paths, enhancing the understanding of complex relationships. In contrast, the exploratory programming environment employs a cell-based interface, which may be more straightforward for linear code exploration.

In terms of session manipulation granularity, IGC offers specific actions, such as inserting nodes before or after a selected node, allowing for precise control over the execution path. The exploratory environment relies on branching and re-execution of code cells, which may offer less granularity.

Regarding visualization of execution history, IGC displays the execution history as a tree of sessions, highlighting the active session. The exploratory environment presents the execution graph and allows users to navigate between different execution traces.

In editing capabilities, IGC allows code nodes to be edited within the graph with saved execution order due to execution relationships being defined through sessions. The exploratory environment supports code cell modification and provides undo and redo actions.

General Comparison of IGC and the Exploratory Programming GUI. Both IGC and the Exploratory Programming GUI aim to support exploratory programming, but they differ in their underlying models, interfaces, and capabilities.

IGC is built around a graph-based representation of code, where code fragments are nodes connected by edges that represent relationships such as execution order and dependencies. This structure allows for explicit visualization of code relationships and execution paths. The use of sessions within the session manager and the exploratory view in IGC enables users to create, manipulate, and compare different execution paths with fine-grained control. Actions like inserting nodes before or after a certain point, replacing nodes, and starting new sessions from any node provide precise management of the execution state.

In contrast, the Exploratory Programming GUI uses a notebook interface with code cells that can be executed and modified. The execution history is represented as an execution graph, where nodes correspond to execution states or configurations. Users can create branches from any point in the execution graph, allowing for the exploration of alternative code paths. However, the manipulations are generally at the level of code cells and execution traces, which may offer less granularity compared to IGC's node-level manipulations.

Unlike the Exploratory Programming GUI, which represents execution traces as individual paths, IGC displays the entire execution tree, providing a comprehensive overview of all execution paths. This visualization makes it easier for users to see the relationships between different sessions and understand the overall structure of the execution history.

Regarding collaboration and reproducibility, both environments support sharing execution histories. IGC's sessions can be exported and shared, providing a complete representation of the execution path and code state. The Exploratory Programming GUI allows users to share execution graphs and traces, facilitating collaborative exploration and comparison of different code paths.

In terms of user interaction, IGC may have a steeper learning curve due to its graph-based interface and the concepts of nodes and sessions. Users need to understand how to create and manipulate graphs, sessions, and relationships between nodes. The Exploratory Programming GUI, with its notebook-like interface, may be more accessible to users who are accustomed to traditional computational notebooks.

Conclusion. Both IGC and the Exploratory Programming GUI facilitate exploratory programming by offering tools for managing execution state, experimenting with code variations, and receiving immediate feedback. Still, they differ significantly in their approaches and interfaces. IGC uses a graph-based interface with a Session Manager that allows precise manipulation of execution paths with fine-grained control.

In contrast, the Exploratory Programming GUI emphasizes a cell-based interface similar to traditional computational notebooks, making it more accessible to users familiar with such environments. It employs execution graphs to represent the execution history, enabling users to create branches from any point and explore alternative code paths.

By comparing these approaches, we see that while both support exploration and iterative development, they differ in implementation and user interaction. IGC focuses on detailed visual representation and fine-grained control, which are suitable for managing complex code relationships. The Exploratory Programming GUI offers a more familiar interface with broader language support, benefiting users seeking accessibility across different programming languages. Ultimately, the choice between them depends on user preferences, desired level of control, and familiarity with interfaces.

Chapter 6

Discussion

In this chapter, we discuss the core distintive features that make IGC different than other programming environments. Next, we go over some key factors of performance and scalability. Then we mention conflicts and trade-offs that we encountered during the development of IGC. We then we discuss the main points discovered throughout the four case studies. Next, we review the research questions and try to answer them. Finally we discuss the limitations of the tool and what we would have done differently.

6.1 Core Distinctive Features

The following are the core features that are unique to IGC compared to typical programming environments:

6.1.1 Graph-Based Model

The graph-based model of IGC is a distinctive feature that sets it apart from traditional computational notebooks. By representing code fragments as nodes and their dependencies as edges, IGC provides a visual representation of the code structure that is not possible in a linear, text-based environment. This graph-based model allows users to visualize and focus on the relationships between different parts of their code, allowing them to better understand the dependencies and interactions within their projects. This feature is particularly useful for managing complex codebases, where traditional text-based representations can become difficult to navigate.

6.1.2 Sub Graphs

Due to the graph-based structure, sub graphs are a unique feature of IGC that allows users to encapsulate parts of their project into modular components. This feature enables users to break down complex codebases into smaller, manageable parts that can be developed, maintained, and visualized independently. By composing these modular components into higher-level architectures, IGC creates a dynamic and interactive visualization of the overall system. This method of abstraction and composition supports the management of complexity in large-scale software projects, allowing users to focus on specific parts of their code while maintaining an overview of the entire project.

6.1.3 Sessions

The session system in IGC is a unique feature that allows users to manage multiple execution paths and states within a single project. This feature is a direct result of supporting exploratory programming. In normal programming environments, one distinguished execution path is normally followed. In many IPEs, the execution path is usually more blurred as the user can execute and interact with cells in any order they choose. However, the experimentation that is enabled by incremental programming is specific to a particular execution process in a REPL kernel and is lost when the kernel is restarted. The session system in IGC allows the user to manage multiple execution paths and states within a single project. The user can even go back to a previous state and continue from there. This is a powerful feature that allows the user to fully explore the code and experiment with different execution paths without losing any work.

6.1.4 View Representation

The view representation in IGC is a unique feature that provides users with a flexible and intuitive interface for interacting with their code. Besides the graph-based model, the views located in the file editor control the entire project. There is not just one view. Views are the way the user writes code, provides documentation, analyzes relationships, and manages tools that interact with the project as a whole. This feature allows users to customize their workspace to suit their needs, enabling them to focus on specific aspects of their code and documentation while maintaining an overview of the entire project. Most programming environments have a single view based on a way to input code fragments. IGC breaks from this norm by focusing on the project structure as a whole and allows the user to interact with the pieces of the project in a more modular way.

6.2 System Evaluation:

The main system evaluation is broken down into the following sections:

6.2.1 Performance

In terms of performance, we can use one main metric to evaluate the system: execution time. The execution time is the time it takes for the code to be executed and the output to be displayed. This is a critical metric as the user will be executing code many times and will want the output to be displayed as quickly as possible. The execution time is dependent on many factors, such as the size of the code, the complexity of the code, and the system resources. This is a metric that is difficult to evaluate as it is dependent on many factors. However, IGC does not change anything with the program-specific process or "engine" (section 2.2.3). The only added execution time is the wrapped process of starting the new execution process, loading the previous state, and saving the new state to disk. This is a process that is not computationally expensive and should not add much to the execution time. The execution time should be the same as a normal execution time in a REPL kernel, and these wrapped processes should be negligible for the more intensive computations.

The only other process that might take some time is the initial loading of custom IGC components. This normally happens at startup and should not be a problem for the user. The user should only have to wait for this process once, and then the components should be loaded into memory. This process should not take more than a couple of seconds and should not be a problem for the user as this is a one-time initial process.

Every other visual process in IGC is not computationally expensive and should not add any waiting time for the user. The user should be able to interact with the tool as if it were a normal programming environment.

6.2.2 Scalability

A goal for IGC is to be able to handle large software projects, so scalability is a key consideration for its development. React was chosen as its virtual DOM and re-renders only updated components, ensuring smooth performance as the graph scales. ReactFlow can take advantage of React to optimize rendering.

ReactFlow provides an infinite, pannable workspace to create the graphs. This allows the user to create any size project, as the workspace can always be extended. A limitation is that to capture the entirety of a large project, the user must zoom out, which will eventually make text hard to read and elements difficult to see or distinguish. A possible solution currently in development is to consolidate subgraphs into a node by introducing a node type that can represent a (sub-)graph. This feature would allow users to simplify graphs by effectively introducing layers of abstraction. The sub-graph can be defined in a separate file, corresponding to the common development practice in which files often represent an individual component of a larger software project.

Zustand is used to record program state as it is lightweight, minimizes overhead and maintains consistency. Node.js is used for the back-end and is seen as highly scalable [71] owing to its non-blocking, event-driven architecture.

6.3 Conflicts and Trade-offs

The following are some of the conflicts and trade-offs that we encountered during the development of the tool that are important to discuss:

6.3.1 Language Support

IGC was designed to allow the use of many different programming languages. This was an important design decision, as it allows the tool to be used by a wider audience. It also matches the availability of many IPEs that support multiple languages such as Jupyter Notebook. Jupyter Notebook uses a kernel system that allows the user to simply use a REPL of the programming language they want to use [72]. They then are able to use the familiar interface they are used to. This is a powerful feature that allows the user to use the notebook with whichever language they choose.

This was the plan for IGC as well; to import some kernel or system that allows the user to use the tool with any language they choose. However, we prioritized developing a proof-of-concept for the interface itself over implementing full multi-language functionality. This meant that wherever possible, we designed features and functionality to be language agnostic. However, for certain aspects (particularly code execution and state management), we had to implement Python-specific solutions. Throughout development, we remained mindful of our goal to support additional programming languages eventually. As a result, whenever we introduced functionality tailored to Python, we were careful to limit the scope of these language-specific choices, ensuring that our overall architecture could accommodate other languages with minimal adjustments. Specific reasons for the Python-specific choices will be discussed in more detail in the following sections.

The interface itself is not dependent on any specific language. Nodes and relationships have no specific language attached to them and are meant to be generic to many different programming languages and paradigms. For example, the object-oriented programming paradigm is not specific to Python and can be used by any language that supports it. If a user were to use a different language that does not support object-oriented programming, they would be able to disable the object-oriented programming module.

We can also see the language-agnostic nature of the tool in the way that the APIs are designed. The APIs are designed to be generic and normally require a language to be passed in as a parameter. This allows the user to use the tool with any language they choose as long as they have the language support on the back end.

One feature that is difficult to make language agnostic is code analysis. In the current implementation, call the `analyze` endpoint along with the raw code and specific language parameters. The goal of this API is to return all the new definitions the code creates and all of the dependencies the code is dependent on. From this information, we can create a dependency graph of the code which can hopefully inform the user on which nodes can be executed next without running into any errors. In Python, we use the `ast` module in Python to go through the code and identify the new definitions and dependencies. This implementation is specific to Python and would need to be re-created for each language, as every language processes its own AST due to differences in syntax and code structure. This feature is not critical to the tool and could be removed, but is an example of a feature that is difficult to make language agnostic.

Potentially, in the future, we could use a language server to execute and analyze the code. Language servers are standardized for every language [73], so it would hopefully be easy to just import the language server for the language the user wants to use. The process behind doing this would need to be explored further to see if it is possible.

Execution Environment. Continuing from the language support section, we had to make some choices about how to handle code execution. We decided to support Python as the initial implementation of IGC. This was because Python is a popular language for almost all types of programming, which is evident by the Stack Overflow Developer Survey[1]. This was also because Python is a language that is simple and readable [74]and can be used in data science, web development, and many other fields. It also supports many different programming paradigms such as object-oriented programming, functional programming, and procedural programming. This made it a good choice for the initial implementation of IGC.

With many computational notebooks, the user normally can import a REPL kernel for the language they want to use. The REPL kernel lives in memory as a process that can be interacted with. There were

[1] https://survey.stackoverflow.co/2024/

many reasons why we decided against using a REPL kernel for IGC. These reasons can be divided into two categories: exploratory programming and the need to share execution outcomes without recompilation.

The main issues with having a memory-based system for exploratory programming is that it is hard to manage the branching execution of the code. In theory, the user would almost have to create a fork [75] of the kernel for each node they execute. The reason why we need to fork at each node instead of each branch is that we need to allow the user to go back in the execution history. The extent of this might not be known at the current execution state. Creating a fork at each node could easily lead to an exponential growth of the number of processes that are running most likely contributing to high memory usage and slow execution times. This is not a problem that is easily solved and would require a lot of work to implement.

The second reason is that we wanted to allow the user to share the execution outcomes with others. This is not possible with a memory-based system unless exporting all state to disk. If we need to export all state to disk anyway, why not use a disk-based system in the first place? This is why we decided to use a disk-based system for IGC. By default all the state is set to disk and can be shared with others. By always saving state to disk, we can also avoid issues where the user forgets to save their work and loses it when the kernel is restarted or corrupted in some way.

Both these reasons are non-critical and could actually still allow for a memory-based system to allow for plug-and-play language kernels. However, we decided to use a disk-based system for the reasons mentioned above. Potentially, in the future, we could implement an option to choose a memory-based system instead of a disk-based system.

The disk-based system is not perfect and has its own issues. The two main issues are that it is slow and requires a language-specific implementation to save to disk. The first issue is not a critical issue as the user can still use the tool. The execution time increase is just that of reading and saving to disk. The actual processing of the code should be the same, whether it is memory-based or disk-based. The second issue is more critical. The user would have to implement a language-specific solution to serialize state to disk. This is not ideal as the user would have to implement a solution for each language they want to use. The following section (section 6.3.1) will discuss the state management system in more detail.

Even with a disk-based system, the issue of exponential growth from exploratory programming remains as a user's project expands. However, disk storage is generally cheaper and more abundant than available memory.

State Management Every execution has a wrapper that loads the previous state (if it exists) and saves the new state to disk. This allows the user to go back to a previous state and continue from there. This is a powerful feature that allows the user to explore different paths of execution and go back to a previous state if they make a mistake without any re-execution. This feature is not available in many computational notebooks and is unique to IGC. For the Python implementation, we used the `dill` library (an extension to the `pickle` library) to serialize and deserialize the state to and from disk. The process to be able to serialize and deserialize state is called pickling [76]. The `dill` library was a good choice for Python as it is a powerful library that can serialize almost any Python object. However, this implementation is specific to Python.

Serializing state to disk is not a trivial task and would require a lot of work to implement for each language. Many languages have pickling libraries, but not all of them can serialize all objects. For example, "Pickling state in the javaTM system" [77] goes over the process of implementing pickling in Java. Since the pickling system is a requirement for execution, this might hinder the implementation of other languages to be included in IGC. This is potentially a critical bottleneck that would need to be resolved before specific languages could be included in IGC.

Implementing an additional language. The following is a list of steps that would need to be taken to implement an additional language in IGC:

1. **Language binary.** The language binary would be the binary that is used to execute the code. This would be the same as the Python binary that is used in the current implementation. Currently, this is located in `IncrGraph/languages/`. It would also be beneficial to implement a check to see if the binary is installed on the user's system (already done in Python through the `checkPythonInstallation` function).
2. **New `executeCode` function.** The `executeCode` function is used to execute the code in a specific language. This function would need to be implemented for every new language. The following are what is needed inside this function:

   (a) **Language specific state pickling system.** The state pickling system would be a system that serializes and deserializes the state to and from disk. This would be a required step as the state pickling system is required for execution.
   (b) **Execution and output logging system.** The execution step should take in the code to execute. Whether to put the code in a file before execution or to execute the raw code is the user's choice. Either way, this execution should then be wrapped between the unpickling (deserializing) of the previous state and the pickling of the new output state. After all of this, the stdout, stderr, and any appropriate output data should be directed into files of the "execution directory" (the directory that stores all data specific to a particular execution).

   Once this is implemented, there would then need to be a check under the `execute` endpoint to see if the given language parameter is supported and, if so, to call the appropriate `executeCode` function.
3. **(OPTIONAL) Analyzation function** The analyzation function is used to analyze the code and return the new definitions and dependencies. This is used to create the dependency tree and to provide some other useful metadata, however, this is not a critical step and could be ignored if the user does not care for these features. If the user does want these features, they would need to implement the `analyzeCode` function for the new language.

6.3.2 Software Complexity and Structuredness

Ejiogu's work in "A simple measure of software complexity" [14] presents a model for measuring software complexity using tree abstractions based on three key variables: Level (L), Explosion Number (E), and Monadicity (M). The degree of complexity is calculated as

$$C = \frac{1}{L \quad E \quad M}$$

This model captures how increased depth, breadth, or detail in a software system can reduce overall comprehensibility. This model departs from traditional metrics such as Lines of Code or Cyclomatic Complexity by offering a language-agnostic, empirical, and verifiable measure that reflects the inherent complexity of a system's structure.

At the same time, Ejiogu's model provides a framework for distinguishing between software complexity and software structuredness. While complexity is an absolute property of the system, affecting maintainability and understandability, structuredness arises from deliberate design choices aimed at organizing code through modularization and adherence to design principles. In our discussions of IGC and related case studies, we observe that breaking down a system into sub-graphs or components can enhance local modularity and clarity much like importing libraries to encapsulate functionality. However, this approach also introduces a trade-off. Increasing the number of sub-graphs may improve the structuredness of individual components, yet it can also raise the overall complexity of the project due to additional interconnections and dependencies.

The key challenge is to balance the advantages of modularity with the risk of increasing overall complexity. A well-structured system should isolate intricate functionality into manageable sub-graphs without burdening the overall architecture with too many interdependencies. Thoughtful design decisions, such as the strategic use of libraries and deliberate modularization, can help reduce the negative impact of a high component count while still enhancing structuredness. In a programming environment like IGC, users can leverage sub-graphs instead of libraries to encapsulate parts of their projects, thereby improving organization and clarity.

6.3.3 Composition vs Inheritance / Flow vs Model-View-Controller

In IGC, we adopted React largely because of its popularity and its modern approach to UI development, which emphasizes composition over inheritance [2]. React's component model enables us to build interfaces by composing simple, reusable units that can be easily customized through props and hooks. This compositional approach aligns with the ideas presented in [78], where the benefits of adaptable, modular design components are highlighted. By favoring composition, our architecture remains flexible and explicit in its data flow, allowing for straightforward reuse and extension of functionality.

At the same time, there is merit to the traditional inheritance-based MVC structure, as exemplified by Angular. Angular's framework offers a clear separation between controllers, views, and models, enforcing

[2] https://legacy.reactjs.org/docs/composition-vs-inheritance.html

a well-defined architectural boundary. This structure can be particularly beneficial in managing complex state and interactions within large applications. While we chose React for IGC due to its widespread adoption and the agility it provides, one could argue that Angular's approach has its own advantages, especially in scenarios where a more rigid hierarchy might simplify certain aspects of traceability and consistency.

Our implementation of IGC further illustrates a hybrid mindset. For example, our pseudo-hierarchy for node types, relationships, and views. For instance, a node uses a BaseNode type that is extended into specialized types such as CodeNode, DocumentationNode, and GraphNode. This strategy leverages the clarity of a unified structure while retaining the flexibility of composition, ensuring that each node maintains the core attributes required for consistent rendering and behavior. The design of relationships and views follows a similar pattern, with a base class providing a foundation for specialized implementations. This approach allows us to balance the benefits of inheritance with the adaptability of composition, creating a foundation for future development.

In conclusion, while React's compositional model offers significant benefits in terms of adaptability and ease of maintenance, we recognize that Angular's inheritance-based MVC framework has its own strengths. Current trends and practical considerations drove our decision to use React, but we remain open to future adjustments. As the project evolves, further examination of these architectural paradigms may guide us toward an even more thorough and flexible solution for IGC.

6.4 Case Studies

The following are the main points discovered throughout the four case studies and how they relate to the research questions:

6.4.1 Incremental Programming

The examination of Jupyter Notebook alongside IGC, in this case, study provides insights into the strengths and limitations of environments designed for incremental programming. Both systems offer unique benefits that speak directly to the research questions. In addressing **RQ1**, we find that the redesign of the programming environment through a graph-based model in IGC significantly alters how code execution and documentation are managed. While Jupyter Notebook has long excelled at providing immediate feedback through its cell-based execution model, IGC's visual representation of code fragments and their execution order offers a clear structure that mitigates the risk of inconsistent state, a challenge often encountered in traditional notebook environments.

A key discovery from this comparison is that the explicit execution sequence in IGC, visualized as edges in a directed graph, provides a robust framework for managing complex projects. This directly aligns with our goal of leveraging incremental execution techniques to reduce the pitfalls of arbitrary execution orders observed in Jupyter. The graph structure in IGC not only reinforces reproducibility but also enhances clarity in the computational narrative—a critical requirement for complex software projects.

In the context of **RQ2**, our comparative analysis reveals both advantages and disadvantages of the proposed environment. On the one hand, IGC addresses limitations such as the ambiguity introduced by the flexible execution order of Jupyter Notebook. Its structured graph-based interface facilitates better management of code dependencies and can prove beneficial in scaling projects where maintaining a coherent execution flow is paramount. On the other hand, the current implementation of IGC shows limitations in handling media-rich outputs and the flexibility of documentation. For instance, while Jupyter allows for multiple, interleaved Markdown cells, IGC's one-to-one association between code and documentation nodes necessitates workarounds that may impede the seamless integration of narrative and code.

The incremental nature of both environments supports exploratory programming and rapid prototyping, yet the experiences diverge when complexity increases. Jupyter Notebook's immediate feedback through embedded outputs is highly effective for quick iterations, whereas IGC's approach of segregating textual outputs into a dedicated code view pane introduces an extra layer of navigation that may affect user efficiency. These observations underscore the trade-offs inherent in each design choice.

Overall, the case study reinforces the hypothesis that a graph-based model can offer significant benefits in organizing and executing code incrementally. While Jupyter Notebook remains a powerful tool for exploratory analysis and rapid prototyping, IGC's design highlights potential improvements in handling the execution order and dependencies in more complex codebases. The insights gathered here lay the

groundwork for further refinements in IGC, emphasizing the need to balance the structured execution flow with flexible, user-friendly features traditionally offered by established computational notebooks.

6.4.2 Exploratory Programming

The third case study, focusing on exploratory programming, offers significant insights into how environments can be tailored to support a prototypical, iterative programming style. In addressing **RQ1**, the case study demonstrates that enabling manipulation of execution paths and states is key to enabling exploratory programming techniques. The Exploratory Programming GUI's features, such as branching and the ability to compare execution traces, embody the essence of prototyping and iterative refinement. These capabilities align with the hypothesis that a flexible, interactive interface can better accommodate the fluid nature of exploratory programming compared to traditional, rigid edit-compile-run cycles.

An important observation is that the use of execution graphs as a central feature provides an intuitive representation of the program's evolving state. This visual approach not only allows navigation through different sessions but also allows for the detailed tracking of changes as a project is executed. Such a mechanism directly addresses the common challenges in managing execution state and versioning code fragments, which are crucial when experimenting with alternative code paths. By permitting users to branch and manipulate execution histories at a granular level through actions such as inserting or replacing nodes, IGC reinforces reproducibility and comparative analysis, thereby enhancing the overall exploratory process.

In considering **RQ2**, the comparative analysis between IGC and the Exploratory Programming GUI reveals both strengths and constraints in each approach. IGC's graph-based model provides fine-grained control over code execution and state management, which is particularly effective in environments where explicit visualization of code relationships is paramount. In contrast, the Exploratory Programming GUI leverages a more familiar, cell-based interface that may offer greater accessibility to users already accustomed to traditional computational notebooks. This difference underscores a key trade-off: while IGC's detailed session manipulation and comprehensive execution tree offer high precision, the Exploratory Programming GUI's approach can lower the learning curve and facilitate broader language support.

Both environments enhance the exploratory programming experience by providing immediate feedback and supporting iterative development, yet they diverge in user interaction and interface complexity. The Exploratory Programming GUI excels in presenting a unified view of the execution state through its trace comparison and state management features, thereby simplifying the exploration of alternative execution paths. However, its reliance on a cell-based paradigm may limit the level of fine-grained control afforded by a graph-based approach. Conversely, IGC's session manager and node-level editing enable precise control over execution sequences, albeit at the potential cost of a steeper learning curve.

Overall, this case study confirms that enabling dynamic manipulation of execution states is critical for supporting exploratory programming. The features demonstrated by the Exploratory Programming GUI—such as incremental execution, branch management, and execution trace comparison—highlight the benefits of an environment designed specifically for experimentation. At the same time, the analysis underscores the importance of balancing detailed control with accessibility, suggesting that future work might explore hybrid models that integrate the best of both approaches. These insights contribute to a deeper understanding of how rethinking source code representation and user interaction can overcome the limitations of conventional environments, ultimately guiding the development of more effective tools for exploratory programming.

6.4.3 Complexity Management

Since complexity management is fairly large and the focus on two of the case studies, we will separate it into two sections:

Simplifying Complexity

Both case study 2 (Code/Documentation Projections) and case study 3 (Architectural Diagrams and Composition) provide valuable insights into addressing the challenges of complexity management, thereby contributing to **RQ2** regarding the advantages and disadvantages of the proposed environment when compared to traditional approaches. In the case of PescaJ, complexity is simplified by aggregating scattered code fragments and documentation into unified, overlapping views. This design leverages spatial contiguity to alleviate the cognitive load associated with navigating dispersed information, allowing developers to readily access related code and documentation in a single, coherent workspace. Such

an approach not only improves code comprehension but also helps ensure that documentation remains synchronized with the underlying code, directly addressing the need for handling complexity in large-scale software systems.

Similarly, the introduction of Graph Nodes in IGC, as demonstrated in case study 3, simplifies complexity by encapsulating entire subsystems into modular components. This modular approach enables developers to break down a complex codebase into smaller, manageable parts that can be developed, maintained, and visualized independently. By composing these modular components into higher-level architectures, IGC creates a dynamic and interactive visualization of the overall system. This method of abstraction and composition supports **RQ1** by showcasing how a redesigned programming environment, specifically through a graph-based representation, can manage complexity in ways that traditional tools may not. In both cases, the techniques for simplifying complexity—whether by aggregating scattered elements or by modularizing the system—demonstrate how rethinking the interface and underlying code structure can offer significant advantages over conventional approaches.

Code Navigation

Navigating complex codebases is a critical concern that directly relates to both **RQ1** and **RQ2**, and both case study 2 and case study 3 offer distinct approaches to this challenge. In the PescaJ example, code navigation is enhanced through projectional views that consolidate interrelated code and documentation into easily accessible workspaces. This approach minimizes the need for extensive file switching and scrolling, thereby reducing the cognitive effort required to understand intricate method call hierarchies and dependency relationships. By spatially organizing code fragments and their associated documentation, PescaJ provides a clear pathway through which developers can trace dependencies and contextual relationships. This method of navigation directly supports **RQ2** by illustrating an advantage of the proposed environment: improved manageability of complex codebases through enhanced visual navigation.

In contrast, IGC's use of Graph Nodes, as illustrated in case study 3, offers a graph-based approach to code navigation that is particularly well-suited to representing complex interdependencies. The explicit visualization of relationships through graph edges, combined with the modular encapsulation of code components, allows developers to traverse from high-level architectural overviews to specific code fragments seamlessly. This multi-layered navigation strategy not only facilitates rapid identification of dependencies and interactions but also enables detailed inspection of individual modules. Such an approach embodies the principles outlined in **RQ1** by demonstrating how a redesigned interface—rooted in a graph-based model—can handle complexity more effectively than traditional linear code representations. At the same time, it contributes to **RQ2** by highlighting both the potential benefits, such as precise control over code navigation, and some of the limitations, like the possible challenges in managing deeply nested graph structures.

Overall, the strategies discussed in both case studies underline that reimagining code and documentation representation through aggregated views and modular graph compositions not only simplifies complexity but also enhances code navigation. These findings provide strong evidence in support of our research questions, illustrating that innovative interface designs can address the limitations of existing tools while offering clear advantages in managing and navigating complex software systems.

6.5 Research Questions

From the design, implementation, and insights gathered throughout the case studies, we can directly address the research questions as follows:

6.5.1 Research Question 1

RQ1 asks: *How can a programming environment be redesigned, speci cally in the source code representation and interface, to take advantage of incremental programming techniques, exploratory programming methods, and the handling of complexity usually found in software projects?*

The case studies clearly demonstrate that a graph-based model provides a rich and expressive way to represent and manage code. For incremental programming, as shown in the comparison between Jupyter Notebook and IGC, the graph-based interface enables an explicit depiction of code fragments and their execution order. This design not only preserves the benefits of immediate feedback and rapid prototyping but also reduces the risk of inconsistencies by visualizing the execution sequence through

edges in a directed graph. The clear, modular presentation of code through nodes directly addresses the challenges found in traditional, linear notebooks.

In the realm of exploratory programming, the session system and projectional views in IGC offer a significant advancement over conventional cell-based interfaces. By allowing users to manipulate execution paths and branch out from any point in the execution graph, IGC supports a fluid, iterative exploration process. The ability to compare different execution sessions and visualize the complete execution history addresses the shortcomings of rigid edit-compile-run cycles, thereby fulfilling the needs identified in **RQ1**.

Regarding complexity management, the graph-based representation excels by providing multiple, customizable views. As demonstrated in the Code/Documentation Projections and Architectural Diagrams and Composition case studies, developers can aggregate dispersed code and documentation into coherent workspaces and encapsulate entire subsystems into modular Graph Nodes. Such modularity not only simplifies the visualization of complex software architectures but also allows for the consolidation of large graphs into subgraphs, akin to constructing dynamic, executable UML diagrams. This flexibility in presentation, achieved through various projection settings, directly supports the goal of rethinking source code representation to better handle the complexity inherent in modern software projects.

Overall, the redesigned environment, centered around a graph-based model, effectively leverages incremental and exploratory programming techniques while also offering powerful mechanisms for complexity management. This comprehensive approach directly responds to **RQ1** by demonstrating that a graph-centric interface can capture and present a wealth of information, thereby enhancing both usability and the overall development experience.

6.5.2 Research Question 2

RQ2 asks: *What are the advantages/disadvantages of the proposed environment when comparing them to di erent environments that specialize in incremental programming, exploratory programming, or the handling of complexity usually found in software projects?*

Across the case studies, several key points emerge regarding the comparative merits of the proposed graph-based environment:

Incremental Programming: By contrasting IGC with Jupyter Notebook, we see that the graph-based model in IGC introduces transparency in execution order and dependency management. While Jupyter Notebook excels in providing immediate, embedded feedback through its cell-based execution model, it suffers from potential inconsistencies due to arbitrary cell execution. IGC mitigates this risk by enforcing a clear, visual execution sequence through its graph representation. However, IGC's current limitations, such as the handling of rich media outputs and the one-to-one mapping of code to documentation, highlight areas for further refinement. These advantages and disadvantages provide a balanced perspective on how a graph-based interface can both improve and challenge established practices.

Exploratory Programming: The exploratory programming case study illustrates that IGC's session system, with its capacity for fine-grained control over execution paths, offers enhanced reproducibility and precision compared to traditional cell-based approaches. The ability to insert, replace, or branch nodes provides developers with a robust toolkit for iterative experimentation. Nonetheless, while IGC delivers detailed control, it comes at the cost of a steeper learning curve compared to more familiar, linear interfaces like the Exploratory Programming GUI. This trade-off underscores the importance of balancing detailed control with usability—a core consideration when evaluating the benefits of the proposed environment.

Complexity Management: Both the aggregation of code and documentation in PescaJ and the modular composition via Graph Nodes in IGC reveal significant benefits in managing software complexity. IGC's graph-based interface not only supports the visualization of intricate execution paths and interdependencies but also allows for the creation of modular subgraphs that serve as high-level abstractions. This capability facilitates navigation through complex codebases and enables developers to create dynamic, executable representations of system architecture. Yet, the richness of the graph can potentially overwhelm users if not properly managed through effective projection and consolidation tools. Thus, while the graph-based approach provides a more comprehensive view compared to linear representations, it also introduces challenges that must be carefully addressed to avoid cognitive overload.

In summary, the comparative analysis across the case studies highlights that the graph-based model in IGC offers distinct advantages in terms of explicit execution ordering for incremental programming, enhanced exploratory capabilities, and different complexity management tools. At the same time, it

reveals certain limitations, such as handling media outputs, documentation flexibility, and potential interface complexity, that require further investigation. These findings, therefore, provide an answer to **RQ2**, affirming that while the proposed environment has considerable strengths in addressing the shortcomings of traditional tools. However, areas still remain for improvement to realize its potential fully in various development contexts.

6.6 Limitations

While the case studies presented provide valuable insights into the potential advantages of a graph-based programming environment, it is important to acknowledge several limitations and threats to validity that may affect the generalizability and interpretation of these findings.

6.6.1 Threats to Validity

The following are potential threats to the validity of the research findings focusing on construct, internal, and external validity:

Construct Validity

Construct validity is challenged by both the measures used to evaluate the environment's benefits and the potential for vague or inconsistent definitions of key criteria. In the case studies, we relied on qualitative observations to assess aspects such as the clarity of execution paths, code modularity, and overall system visualization. Although these observations provide a useful narrative, they may not capture the full spectrum of performance metrics, such as productivity improvements or error rates, that would be ideal for evaluating incremental, exploratory, and complexity management techniques. Additionally, without precise definitions for improvements in usability, performance, or functionality, there is a risk that our evaluations might inadvertently measure aspects different from those we intended to capture. For example, while our studies highlight features like explicit execution pathing, session manipulation, and modular code aggregation, it remains unclear whether these improvements translate into faster project development, better user comprehension, or more effective code maintainability.

We acknowledge that this ambiguity can lead to subjective interpretations, where personal judgment may fill the gaps left by the lack of clear, objective criteria. As a result, the benefits observed in IGC might be overemphasized or underreported based on the evaluator's understanding of what constitutes an improvement. Given the current constraints and in the absence of a dedicated user study with more concrete quantitative metrics, we believe that our qualitative assessments represent the best analysis possible at this stage of research. In future work, we plan to incorporate a user study with quantitative measures and to establish more rigorous and explicit definitions for these evaluation metrics. This approach will help ensure that our measures consistently reflect the intended aspects of incremental programming, exploratory programming, and complexity management within IGC.

Internal Validity

Internal validity is influenced by the subjective nature of our evaluations. Since the analysis is primarily based on qualitative observations of IGC's features. There is a possibility that personal preferences and expectations have shaped our assessments. For example, a strong belief in the superiority of the graph-based model may have led to rating its benefits, such as the clarity of execution paths or the ease of managing complex architectures, more favorably than they might otherwise be perceived.

This subjectivity presents a clear threat to internal validity. Our judgments, made during the examination of IGC alongside tools like Jupyter Notebook, PescaJ, and the Exploratory Programming GUI, may reflect inherent researcher bias. The controlled conditions of the case studies and the specific design choices of IGC could have contributed to these biases. An evaluator predisposed to favor a modular and visual representation of code is more likely to highlight the strengths of IGC, while alternative perspectives may have emphasized different aspects or limitations.

To mitigate these issues in future work, it would be valuable to incorporate controlled experiments or blinded assessments. These approaches would help ensure that evaluations of IGC's features are based on more objective criteria and reduce the influence of individual bias. In doing so, a more balanced comparison of the benefits and limitations of the proposed environment could be achieved. This would provide

a better basis for drawing conclusions about its effectiveness in supporting incremental programming, exploratory programming, and complexity management.

External Validity

External validity is limited by the qualitative nature of these case studies and by the particular contexts in which they were conducted. Although we have identified which features are present in the proposed environment, such as explicit execution sequencing, session manipulation, and modular code aggregation, it is difficult to state whether one version is better than another in terms of qualitative performance. We can describe the existence and functionality of these features, but measuring their impact beyond qualitative observations and simple quantitative metrics like execution time remains challenging.

The scenarios used in these case studies may not fully represent the diversity of real-world software development environments. While the detailed examples and use cases illustrate the potential of a graph-based model for handling incremental and exploratory programming as well as complexity management, these results may not generalize to all programming environments or development contexts. The case studies were conducted in specific settings with targeted examples, and the extent to which these findings can be applied to broader or different software systems remains an open question. A user study incorporating quantitative measures such as task completion time, error rates, and user satisfaction would be necessary to further validate these observations and to assess the practical impact of the proposed environment in a broader range of development contexts.

6.6.2 Specific Feature Limitations

The following are specific limitations of the current implementation of IGC:

No Interactable Terminal. The current implementation of IGC lacks an interactable terminal, which limits the user's ability to execute code snippets directly within the environment. While the graph-based model offers a structured approach to code execution, the absence of a terminal interface may hinder the user's ability to interact with code in real time. Future iterations of IGC could benefit from incorporating a terminal feature that allows users to execute code fragments and view outputs within the environment, thereby enhancing the interactive programming experience.

Language Support. The current implementation of IGC is tailored to Python, which may restrict its usability for developers working with other programming languages. While the graph-based model is language-agnostic in theory, the lack of language-specific support for other languages could limit the tool's adoption in diverse development environments. Future work should focus on expanding language support by implementing language-specific execution environments and analyzers, thereby broadening the tool's applicability across different programming paradigms.

Projection Views are Read-Only. The projection views in IGC are read-only, meaning that users cannot directly interact with the code or documentation displayed in these views. This is not the case for computational notebooks, where this view is the main view and is interactable. This limitation may hinder the user's ability to edit or modify code fragments within the projection view, potentially reducing the tool's flexibility for exploratory programming and rapid prototyping. Future iterations of IGC could enhance the projection views by enabling direct editing of code and documentation, thereby providing a more seamless and interactive programming experience.

Not Functionally Pure. The current implementation of IGC is not functionally pure, meaning that it does not enforce a purely functional programming paradigm. IGC graph nodes have no direct input or output. Instead, they just take the entire execution state at the time of execution. Functionally pure programming would allow for the user to explicitly define the variables and what they expect as an output. This would especially be useful for GraphNodes where the contents of this node are not always known. Having explicit typing would allow the user to know exactly which nodes can be switched out and which nodes are dependent on other nodes. It might be beneficial in the future to have an option to make a project functionally pure if they so choose.

Graph Node Cycles. Another limitation of the current implementation is the potential for graph node cycles. In the current system, there is no check to see if a graph node references an IGC file that contains another graph node referencing the original graph node. This could cause an infinite loop in the system. In the future, it would be beneficial to add a check to see if a graph node references another graph node that references the original graph node. However, this might be the intended functionality for the user. This limitation relates to issues relating to the Halting Problem [79], which could potentially be mitigated by estimation techniques such as the work done in [80].

6.6.3 Qualitative Nature of the Case Studies

As reiterated in the threats to validity, it is also important to note that the current evaluation is largely qualitative. While this approach has allowed us to explore and illustrate the benefits and limitations of the graph-based model in depth, a quantitative user study would offer additional insights. Such a study could measure aspects such as task completion time, error rates, and user satisfaction, thereby complementing the qualitative findings and providing a more robust evaluation of the environment's effectiveness.

6.6.4 What we would do differently

Systematic Execution Stategy. The current implementation of IGC lacks a systematic execution strategy inside the IGC source code that would allow users to define the order of execution explicitly. More specifically, execution is entangled with many different processes and dependencies within the IGC codebase. This is not only a problem for the maintainability of IGC, but more importantly, when it comes to extensibility for the user if they wanted to make custom IGC functionality. Execution is one of the most important elements of a programming environment. Currently, there are some APIs that the user would need to call in their components that handle the execution, however, these APIs are not explicit and the side effects of calling these APIs are not well documented. If this execution process was better outlined, but would be more maintainable and extensible for the user.

Language Manager Point of View. As the IGC was just a prototype, the effort was devoted towards the interface and functionality. The programming language was just a Python implementation. The programming language decisions should have been more of a standpoint of using an unknown language instead of knowing about the Python implementation. The implementation did try to make the implementation as language-agnostic as possible; however, there are no great ways to deal with managing a generic language since we knew the features of Python beforehand. For example, state management and interacting with a language binary is not well-defined for a generic system in the current implementation.

In the future, we will create a language manager that will allow the user to add, remove, and/or interact with a language instance. Then the specific IGC functionality will be defined by the user at the language manager level. This will allow for a more generic system that can be used for any language. It would also be nice in the future to allow for a REPL kernel system in case the user wants to use a language that is not supported by IGC.

Experiementing with Different Front End Technologies. There were several different front end technologies that could have been experimented with:

Since we were more familiar with React and React has a larger community, we decided against using Angular. In retrospect, Angular could have been a better choice due to its MVC architecture. Specifically, when it comes to the views and the component inheritance, Angular might have been a better choice as these are explicit features normally found in MVC architectures.

Another technology that could have been experimented with is different graph libraries. The current implementation uses ReactFlow. ReactFlow is a great library for creating graphs and was chosen as it met all of the requirements in the design section, however, it is not the most performant. There are other libraries, such as D3.js, that are more performant and have more features. In the future, it would be beneficial to experiment with different graph libraries to see if there is a better alternative.

The current implementation uses Material-UI and custom CSS modules for styling. There have been issues with conflicting styles coming from the two systems. An interesting system for styling could have been to use Tailwind CSS. Tailwind is a CSS framework that is gaining popularity where the user uses

predefined CSS classes directly in the front end code. It would have been beneficial to experiment with Tailwind CSS to see if it would have been a better choice for styling the IGC interface.

Chapter 7

Related work

The following section will discuss related work in the field of code visualization, interactive programming environments, and model-driven development. We will also discuss the relevance of visual programming languages and alternative code structures in the context of IGC.

7.1 Other Computational Notebooks

The following are some notable computational notebooks that have some unique features that could be relevant to IGC in terms of immediate code feedback and shareability:

Observable. Observable[1] is a reactive computational notebook that updates as users interact with it. Observable is a web-based platform that allows users to create and share interactive visualizations, data analyses, and other computational narratives. Observable's reactive programming model enables users to create dynamic, interactive visualizations that update in real time as users interact with the code. Observable's notebook interface is similar to IGC's graph-based editor, but Observable focuses on data visualization and reactive programming, while IGC is designed for general-purpose programming and code visualization. The reactive programming model in Observable might be an interesting feature to explore in IGC to enable real-time updates in the graph-based editor. However, a major reason for not adopting this model is that it might run into conflicts potentially with graph nodes cycles (discussed in section 6.6.2)

Deepnote. Deepnote[2] is a cloud-based computational notebook designed for real-time collaboration, enabling teams to work on data science projects together seamlessly, much like Google Docs for Jupyter notebooks. It extends the traditional Jupyter environment with features such as live co-editing, inline comments, and version control, making it ideal for collaborative data analysis and educational settings. Deepnote supports Python, SQL, and other languages, with built-in integrations for popular data sources like Snowflake, BigQuery, and PostgreSQL, allowing users to query, analyze, and visualize data directly within the notebook. Its automatic environment management ensures that dependencies are consistently configured across projects, eliminating setup issues. Implementing real-time collaboration features in IGC could enhance its usability for pair programming and team-based software development. This way, users would not necessarily need to share notebooks through files, but rather live representations. This is an ambitious feature that would require a high level of server synchronization through web sockets or other technologies.

7.2 Model-Driven Development

The more abstraction that can be done to resemble design patterns, the more this approach seems to imitate the goals of Model-Driven Development (MDD). MDD is a software development approach that focuses on creating and utilizing abstract models. MDD prioritizes the design aspect of the software, using models as simplified representations to guide code generation and other development processes [33]. However, the consequences of a model-first approach in MDD include challenges like redundancy in

[1] https://observablehq.com/
[2] https://deepnote.com/

managing multiple model representations, difficulties in formalizing a standard modeling language, and increased complexity in managing model relationships [81]. IGC differentiates itself from MDD by offering a classical coding-first approach through REPLs that can later be abstracted to design patterns instead of the other way around.

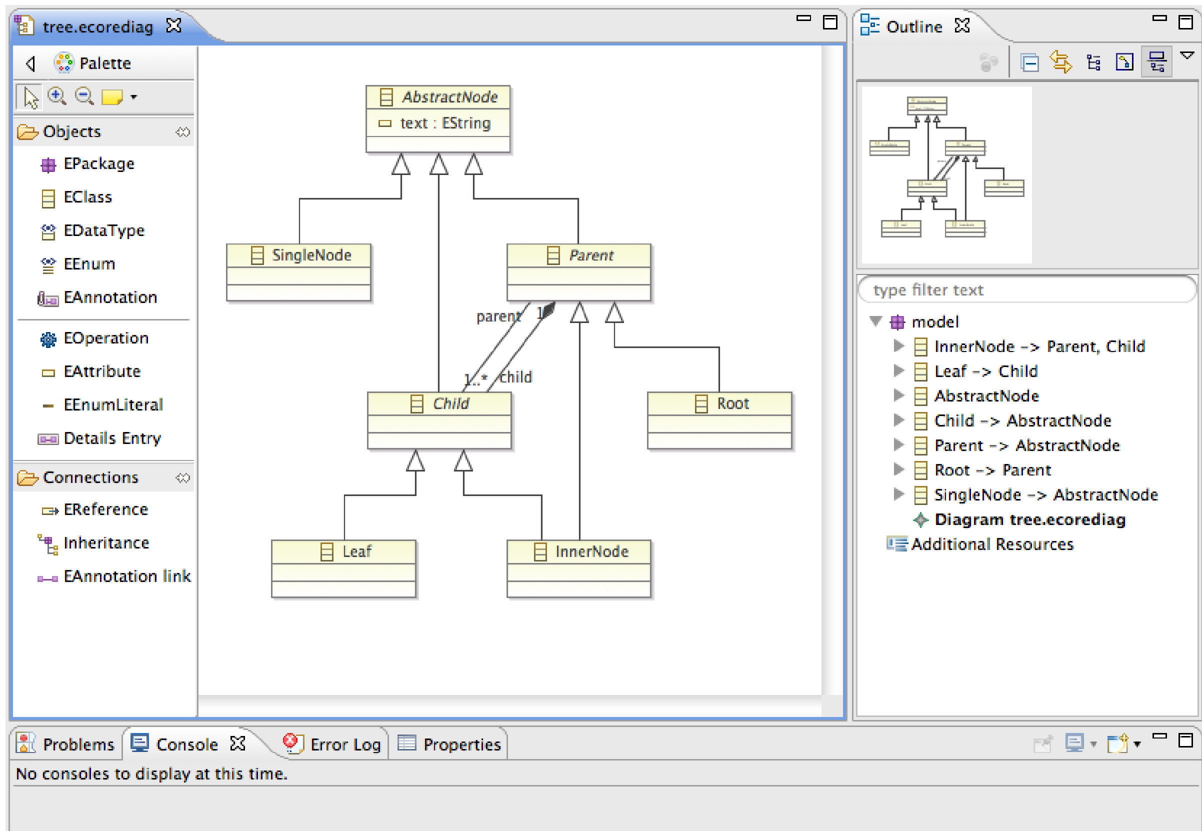


Figure 7.1: Example of a Model-Driven Development (MDD) environment in Eclipse.

7.3 Visual Programming Languages

Visual Programming Languages (VPLs), like MDD, have many overlaps with the aspirations of IGC. VPLs are about transforming visual representations into program logic. One example of a VPL is Scratch. Scratch is a block-based programming language that allows users to create interactive stories, games, and animations by snapping together visual blocks that represent code constructs. Scratch is designed to be intuitive and accessible, making it ideal for beginners and children to learn programming concepts without needing to write code. VPLs like Scratch offer a visual, interactive environment that simplifies the programming process and encourages creativity and experimentation.

One of the most relevant subsets of VPLs is called Node Graph Architecture (NGA), which is about representing code statements into nodes of a graph. This architecture is at the extreme end of the visualization and abstraction of what IGC is trying to accomplish. IGC differs as it is meant to be a hybrid environment between classical text development. However, one can not understate the relevance of VPLs.

7.4 Projectional Views

Besides PescaJ [48] that was mentioned in the case study, there are other works that have explored projectional views. The following are some notable examples:

Hover Tooltips / IntelliSense. Hover tooltips and IntelliSense are common features in modern code editors that provide contextual information about code elements as users interact with them. These

features offer a form of projectional view by displaying additional details like type signatures, documentation, and method signatures when users hover over or select code elements. IntelliSense, in particular, offers code completion suggestions based on the current context, helping users write code more efficiently and accurately. These features are widely adopted in popular code editors like Visual Studio Code, IntelliJ IDEA, and Eclipse, enhancing the coding experience by providing real-time feedback and guidance.

Lorgnette. Lorgnette [82] is a framework that allows developers to create, customize, and integrate code projections within their code editors. It is designed to be language-agnostic and adaptable, supporting projections based on textual patterns, syntax trees, and runtime data. Unlike many projection systems that are rigid and hardcoded for specific languages or environments, Lorgnette allows users to define their own projections without altering the underlying editor. This enables dynamic, context-aware interactions such as inline data visualizations, variable tracing, and form-based code manipulation directly within the coding environment.

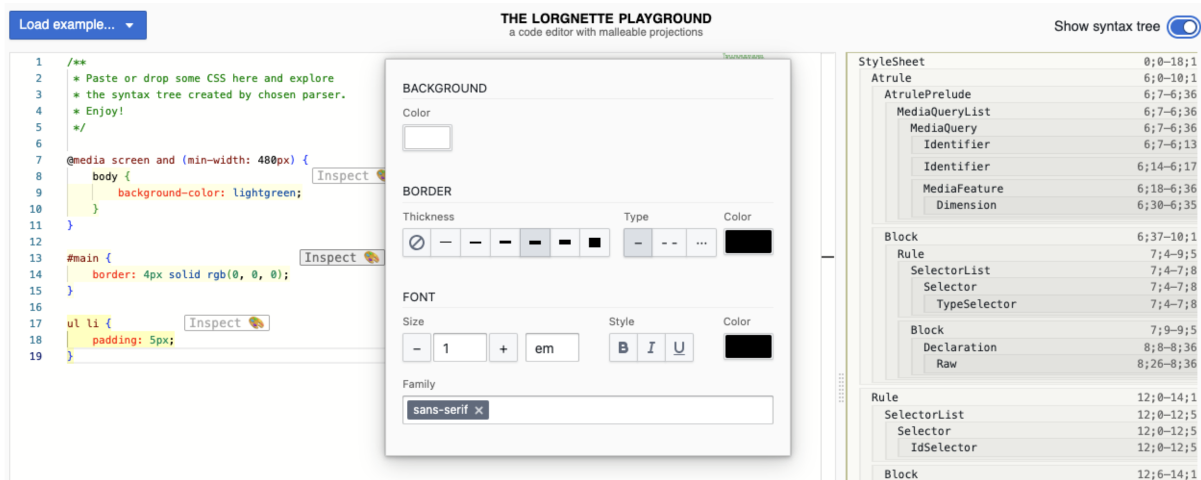


Figure 7.2: The Lorgnette interface.

At its core, Lorgnette operates on a bidirectional mapping system, ensuring seamless synchronization between the code and its visual projections. Users can define projections through specifications that outline conditions for activation, data extraction patterns, and UI behaviors. The framework's modular design promotes reusability, allowing the same projection to be applied across multiple languages and contexts with minimal adjustments. Lorgnette integrates smoothly with popular code editors like VSCode, supports runtime data interaction via the Debug Adapter Protocol, and offers a rich ecosystem for creating interactive, domain-specific projections that enhance code readability, debugging, and exploratory programming workflows.

7.5 Alternative Code Structures

Other works have investigated alternative ways of structuring and visualizing code beyond the conventional file-system organization found in IDEs or the linear sequence of code cells in computational notebooks. The following sections describe several notable approaches.

Code Bubbles. Code Bubbles affords the visual organization of code fragments by representing them as individual "bubbles" that can be arranged and grouped on the working pane. This approach enables developers to visually manage code fragments, facilitating reading, editing, and navigation. Evaluations of Code Bubbles indicate that users spend significantly less time navigating when performing code comprehension tasks and benefit from improved support for multitasking and breakpoint debugging [83, 84].

The Link Environment. The Link environment, developed by MakinaRocks[3], enhances Jupyter Notebooks by adding a visual display that represents code cells as nodes within a graph. In this system, code cells are depicted as pipeline components that are interconnected by dependency edges. Users can click on a node to trigger the execution of that code cell along with the cells it depends on, following the prescribed dependency order. This visual pipeline assists users in tracking and managing the evolution of their code, although it does not allow for the simultaneous exploration of alternative execution paths.

2D Cell Layout in Notebooks. Harden *et al.* have explored an alternative Notebook design featuring a two-dimensional cell layout [12]. This 2D layout enables non-linear code narratives and introduces a form of branching that conventional Notebooks do not support. Their user studies reveal that this structure promotes exploratory programming by making it easier to branch analyses and compare results.

7.6 Smalltalk

Smalltalk [85] is the first object-oriented programming language and offered the first graphical user interface [86]. Both of these transformed the way developers interact with software. Introduced in the early 1970s, Smalltalk built upon earlier ideas from LISP [87] and improved on just-in-time (JIT) compilation techniques. In Smalltalk, translation to machine code is performed on demand, with the compiled code cached for later use. When memory becomes limited, the system discards some of this code and regenerates it as needed [88]. Later, Sun's Self language refined these approaches, achieving performance close to half that of optimized C while retaining a fully object-oriented framework [89].

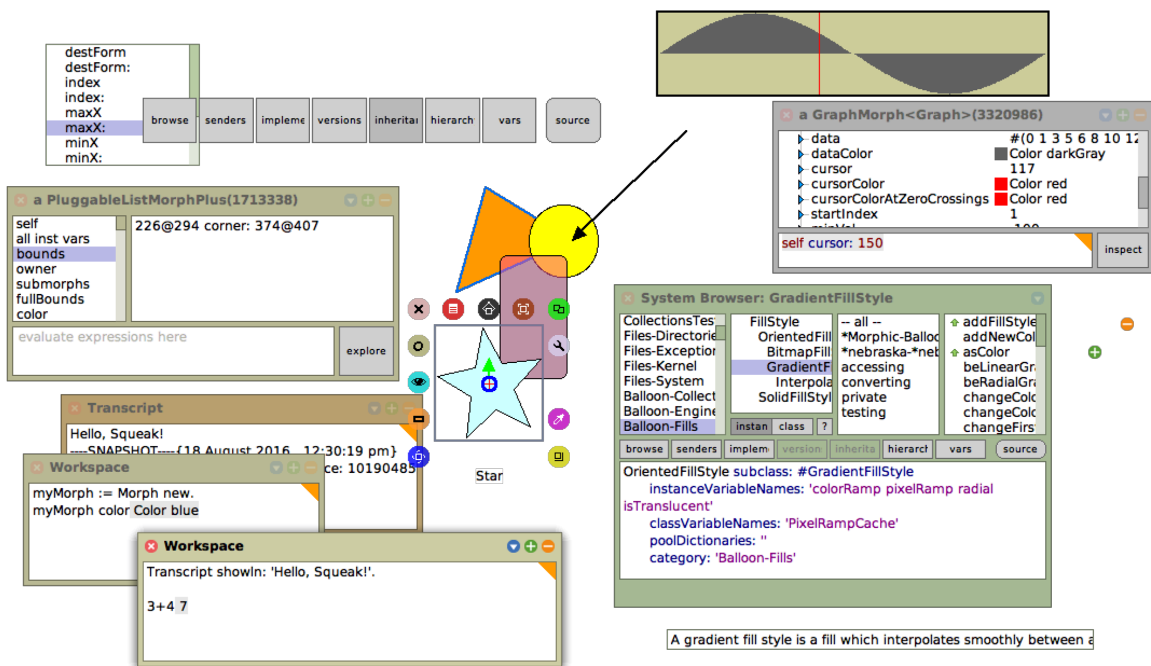


Figure 7.3: A Visual Smalltalk Development Environment (Squeak).

A key similarity between Smalltalk and IGC is their execution approaches. Smalltalk's JIT mechanisms support dynamic, on-the-fly code execution instead of the classic compile everything approach. There are many overlaps with IGC's focus on incremental execution. Both systems aim to provide a more interactive and dynamic programming experience by allowing developers to modify code and see the results immediately.

Another important overlap lies in Smalltalk's adoption of an object-oriented programming paradigm. Smalltalk introduced the concept of treating every element as an object, a design choice that not only fosters modularity but also lays the groundwork for visual environments where developers can interact with objects directly [90]. IGC similarly abstracts code where we represent the code base flow as a graph representing all components of a software system as nodes. In both cases, the systems encapsulate

[3] https://link.makinarocks.ai/

functionality and manage complexity through different abstractions, though IGC leverages this idea in a visual and structured manner through sub-graphs.

A major difference, however, is that Smalltalk is fundamentally a programming language with an integrated, live development environment, whereas IGC is primarily focused in being a programming environment. Smalltalk is centered on its language and the manipulation of a live system image, enabling comprehensive object manipulation. In contrast, IGC focuses on providing tools for managing large-scale software complexity by visualizing project flow.

In summary, Smalltalk and IGC share many overlapping principles. Both systems promote dynamic programming and code abstraction. However, while Smalltalk is a complete programming language with an integrated development environment, IGC is designed as a programming environment to manage the complexity of modern software projects through a visual, graph-based approach.

Chapter 8

Future Work

IGC is an ongoing project with many features and improvements to be made. This chapter outlines some of the proposed ideas and features to be implemented in the future.

8.1 Documentation / Visual Features

The following features are focused on improving the documentation and strictly visual aspects of IGC.

8.1.1 Label Nodes

Currently, only documentation nodes are used to provide information about the project. This is useful for explaining code or other specific context relating directly to a node; documentation nodes must be attached to node. However, there are times when a user may want to provide a label for a node that is not directly related to a node but the system as a whole. For example, if there was a project containing many IGC graphs, the user might want to create a label to provide documentation concerning what the purpose of this particular graph is. We propose adding a “Label Node” that the user can place anywhere on the graph to provide this information.

Another aspect of the label node could be to outline a specific area of the graph. This could be useful for explaining a specific section of the graph or for providing a title for a section of the graph. We see many use cases where this would particularly be useful in the context of trying to make more descriptive sudo-uml diagrams. The label node would be an excellent tool for narrative purposes that do not relate to specific nodes.

8.1.2 Documentation Chaining

Documentation nodes are currently fairly limited. One documentation node can be attached to one node. When this documentation node is attached, it will always show above the context it is referring to (for example, above the code in a code node). We propose adding the ability to chain documentation nodes together. This would allow the user to attach multiple documentation nodes to a single node. The user could then also control whether the documentation node should show before or after the node it is attached to. This would allow for more complex documentation to be created and displayed. This would solve a limitation currently present compared to typical computational notebooks where the user has full access over how many markdown cells they can add and where they can place them.

8.1.3 More Granular Edge Control

Currently, the user has no control over the actual edge that is drawn between two nodes. We propose adding the ability to control the edge that is drawn by potentially adding a control point when dragging an edge on the graph. This would allow the user to create more complex graphs and potentially make the graph more readable. It is very difficult to create an edge pathing algorithm that does not overlap with any other edges in the graph. The feature would allow the user to manually control the edge pathing to avoid overlapping edges or if they have some specific design in mind. With the creation of control points, the implementation would also have to take into account on how to remove these control points.

8.2 Language Support

Language support was a large topic of discussion throughout this thesis. The following features are focused on improving the language support in IGC.

8.2.1 Implement different programming languages

Currently, IGC only supports Python. We propose adding support for other programming languages, especially those with different paradigms. This would allow IGC to be used in a wider variety of projects and would make it more accessible to a wider audience. It would also serve as a good test of the flexibility of the system. Some languages in mind are purely functional languages such as Haskell and strong typed, compiled, and heavily used industry languages such as C# or Java. It might be interesting as well to experiment with some obscure cases such as HTML/CSS or SQL. The easiest language we have in mind to implement is JavaScript, as it is very similar to Python as they are both interpreted and offer many of the same paradigms (OOP, functional, etc). Adding a new language would be a great test of the system and would allow us to see how well the system can be adapted to new languages.

8.2.2 Language Manager / Control Component

With additional languages being added to the system, it would be useful to have a language manager or control component. This component would allow the user to provide a language server or a language grammar to the system. It would also allow the user to interact with the language. For example, the user could install pip libraries or npm packages. This would allow the user to have more control over the language they are using and would make it easier to use IGC with a wider variety of languages.

8.2.3 Language Execution / Interactions

One potential method of integrating new languages into IGC would be to use language servers. Language servers are programs that provide language-specific features to an editor or IDE. They provide features such as code completion, go to definition, and find references. By using language servers, we could potentially add support for many languages with minimal effort. This would need to be further investigated to see what features it can offer and what features are missing.

Another method of integrating new languages into IGC would be to use memory-based kernels. Kernels are how many computational notebooks execute code, however, IGC uses a disk-based execution model (described in section 6.3.1) which offers many features that are not present in the kernel system. If the user would like to use a language that is not supported by IGC, they could potentially use a memory-based kernel to execute code in that language in the meantime. Since the kernels are already developed as part of the computational notebook ecosystem, it would offer almost immediate support for many languages. Maybe this option between disk-based and memory-based kernels could be a user choice through the language manager component.

8.3 Integrated Version Control

Version control is an important feature for many large projects. The following features are focused on improving the version control in IGC.

8.3.1 Direct Integration

Currently, IGC does not have a direct implementation of version control. We satisfied the version control condition by making everything file-based and allowing the user to use their own version control system. However, it would be beneficial to have a direct implementation of version control in IGC, as is the case with many IDEs.

Two examples of an implementation are from Kery *et al.*: Verdant [9] and Variolite [7]. Verdant is a cell versioning tool, and Variolite promotes exploratory programming features through code cells that can have 'variants'. This would allow the user to see the history of a node, see what changes have been made, and potentially revert to an older version of a node. This would be a very useful feature for many users.

8.3.2 Graphical Diffing

Given the graph structure, IGC has the potential to utilize more efficient diffing algorithms. Traditional text-based diffing methods like the Myers Algorithm can serve as a fallback, but exploring hybrid approaches could optimize performance. This would allow the user to see what changes have been made to a graph and would make it easier to understand the changes that have been made. This would be a great optimization and really utilize the environment to the fullest.

8.4 Usability Enhancements

The following features are focused on improving the usability of IGC.

8.4.1 Make Projectional Views Editable

Currently, projectional views are read-only. We propose making projectional views editable. This would allow the user to edit the code in the projectional view and have the changes reflected in the graph. The projectional view allows users to have a computational notebook-like experience, but it is currently read-only. Making it editable would allow users who commonly use computational notebooks to have a more familiar experience. It would also allow users to edit the code as they see it instead of going back and forth to the code view.

8.4.2 Allow Input Terminals

IGC does not accept any input from programmable sources; a user can not enter any input that the program asks for. Allowing the user to input data through a terminal would allow the user to do the same functionality as they would in any other environment. This would be useful for many scripts that require user input and allow the user to interact with the system in a more dynamic way.

Different types of inputs could also prove useful. For example, the user could use a slider to manipulate a variable or a color picker to select a color. This would allow the user to interact with the system in a more visual way and would allow the user to see the changes in real time.

8.4.3 Execution Recommendations through Dependency Tree

The dependency tree creation is a powerful tool that can give clarity to the user on how the nodes are connected without any execution. We propose using the dependency tree to give execution recommendations to the user. Helpful recommendations would be useful for the user to know the prerequisites before executing a particular node and, conversely, to prevent the user from executing nodes that are not necessary.

8.4.4 Threading Execution Relationships

Currently, IGC executes nodes in a linear fashion. We propose adding the ability to execute nodes in parallel. For example, if the user wants to execute thread-blocking tasks (such as a server or a long-running task), they could execute these tasks in parallel with other tasks. The idea is to allow the user to define diverging execution branches. The diverging branches would signify that the nodes are to be executed in parallel. Potentially, the user could define a case where execution paths converge, signifying the two threads should be joined back together. This would allow the user to have more control over the execution of their code and would allow them to execute tasks in parallel.

One consideration to take into account is that this would work well in a disk-based language implementation, as both processes could start from a shared state file. The method could still work with a memory-based kernel, but the implementation might get complex as there would have to be a fork in the process and it is unclear if kernels support this.

8.4.5 Reactive Programming

Reactive programming is a programming paradigm that is based on the propagation of change. It is a way of programming with asynchronous data streams. While there are use cases where reactive programming is not necessary, there are many cases where it is very useful. Reactive programming works well with exploratory programming as it allows the user to see many different changes in real time. Reactive

programming could also prove useful in educational or demonstrative environments. We propose adding an optional reactive programming implementation to IGC if the user wants to use it.

8.5 Long-term Viability Features

The following features are focused on improving the long-term viability of IGC. These include features that would make IGC more scalable, more maintainable, and general 'quality-of-life' improvements.

8.5.1 Graph Node Cycle Detection

Currently, IGC does not have any cycle detection. We propose adding methods to estimate cycle detection to IGC. The equivalent of this in common programming terms is an infinite loop. The user still can have these infinite loops in their code, it is just that the added implementation of graph nodes allow for another class of infinite loops. By warning the user of potential infinite loops or creating cycles, we can ensure that the graph is always in a good state.

8.5.2 Graph Node Enhancing Functionality

Currently, graph nodes are very limited. They can only be used to group nodes together. If a graph node is included in a projectional view or a dependency calculation, it is simply ignored. In the future, more support for graph nodes in several different applications should be implemented. Graph nodes are an essential building block of IGC so it is important to support them in as many ways as possible.

8.5.3 Purely Functional Programming Options

IGC currently has no pure functional options. This makes it hard to know what IGC components can be substituted for other components. We propose adding the option to specify strict types for graph nodes. There are many ways we could implement this. A simple way is to have the user specify directly specify the input and the output types for a graph. Another, in our opinion, more interesting way is to have the user define sources and sinks. Sources would behave as expected input variables (similarly to the start node that takes in *null*) to the graph, and sinks would behave as expected output variables. From these sources and sinks, the user can explicitly define expected types for both input and output. The graph nodes would then have associated types that allow for type checking and type inference.

8.5.4 Debugging Tools

Debugging is a valuable tool for any programmer. Even simple debugging tools that allow the user to stop and start execution line by line would be very useful. This would allow the user to see what is happening in the graph and would allow them to see what changes are being made. This would be a very useful feature for many users.

8.5.5 In-depth Scalability / Performance Testing and Optimization

For any programming environment, scalability and performance are crucial. We propose doing in-depth scalability and performance testing and optimization. This would involve testing the system with many different graphs and many different nodes. It would also involve testing the system with many different users and many different use cases. This would allow us to see how well the system performs and how well it scales. We would also be able to see where the system is lacking and where it needs to be improved.

The data structure of the IGC files and sessions could potentially be enhanced. Currently, nodes and edges exist as lists in the JSON file. This could be changed to a more efficient data structure like a hash table or tree. This would allow for faster access to nodes and edges and would allow for faster graph creation and manipulation.

Different technologies could also be used to enhance the performance of IGC. For example, a different graphing library focused on scalability and the exact needs for IGC could be tested to compare with the current ReactFlow system. This could potentially make the graph render faster and make the graph more responsive. Different UI enhancements could also be used to make the system more responsive and more user-friendly.

More testing and profiling could also be done to see where the system is lacking and where it needs to be improved. It would also be useful to create more comprehensive unit testing throughout the system to ensure that all features and the system in general is working as expected throughout development (especially on commit pushes).

8.6 Transition / Compatibility Features

The following are features that would make it easier for users to transition to IGC and would make it easier for users to use IGC in their current workflow.

8.6.1 Split Nodes

The ability to split nodes would be a useful feature for many users. This would allow the user to split a node that contains many different statements of logic into multiple nodes, each with its own purpose. We can see cases where potential users define a class along with methods in a single node out of convenience. The user could then split this node into a class node and the corresponding method nodes. Splitting nodes would be a convenient feature that would help users quickly develop their graphs and make them more readable.

8.6.2 Infer Node Types

Another quality-of-life feature would be to infer node types. This would allow the user to write code in a base node and have the system infer the type of the node automatically based on the syntax. The analysis functionality would most likely need to get involved to do this code inference. This would be useful for many users as it would allow them to quickly develop their graphs and would make it easier to understand the types of nodes.

8.6.3 Conversion Tools

This thesis attempts to bridge the gap between incremental programming and complex software development environments. A great way to help the transition from these environments to IGC would be to have conversion tools. For example, a user could convert a computational notebook or a typical text-based environment to an IGC project. This would allow the user to take their existing work and convert it to IGC. The same goes in the other direction. If the user wants to have a final solution in a computational notebook or a text-based environment, they could convert their IGC project to that format. This feature would be valuable for many users who want to experiment and explore the IGC workflow.

8.7 Analysis Tools

The following features are focused on improving IGC through user feedback.

8.7.1 User Study

A user study would be very insightful for IGC. A user study would allow us to see how users are using IGC and what features they are using. It would also allow us to see what features are missing and what features need to be improved. Originally, we wanted to do a user study for this thesis, but due to time constraints and the difficulty finding participants willing to do long coding tasks in multiple environments, we were unable to do so. A user study would be very beneficial for IGC and would allow us to get quantifiable data on how well the system is working.

8.7.2 User Data Metrics

A potential feature that would be very insightful for IGC would be to collect anonymized user data. This data would give similar data as a user study, but would be collected over a longer period of time. This data would allow us to see how users are using IGC and what features they are using. Currently, we have no way of knowing how users are using IGC which is a significant bottleneck for improving the environment.

8.8 Further in the Future

The following features we find interesting and ambitious for IGC, but not vital to the main functionality. These features would take a fair amount of time to implement.

8.8.1 Component Marketplace

A component marketplace where users could upload and download custom components would be a very useful feature. This would allow users to share their work with others and would allow users to use components that they would not have been able to create themselves. It is a key feature that supports extensibility and promotes the open-source community. Various programming paradigms could be implemented in these components, and special unthought-of views could be added to the system. Creating a database of components is almost an independent project in itself, but it would be very useful for the future of IGC.

8.8.2 Live Collaboration

Collaboration is very important within software development. Version control supports a lot of the needed collaboration efforts, however, an extreme approach would be to have a live session that multiple users on different computers could interact with. We can see this being used in team stand-ups or educational settings for explaining code. Deepnote (section 7.1) is an example of a computational notebook that supports live collaboration. Similarly, there are VSCode extensions that allow for live collaboration in text-based environments as well. This would be a very ambitious feature for IGC, but it would be very useful for many users.

8.8.3 Porting IGC to Cloud-based Environment and System Application

Out of the other potential future features in this section, this is not as ambitious as the others. IGC is a web-based application with electron integration. For a cloud-based environment, development would have to be done to become compatible with a cloud-based file system, but it can mostly be ported over seamlessly. This would be a very similar process as many computational notebooks have already been ported to the cloud. The system application can actually be made with the current implementation of IGC. We would just need to use electron to package the application and distribute it. More testing would have to be done to ensure that the system application works as expected, but it should be fairly straightforward. Porting IGC to both of these environments would make it more accessible to a wider audience.

8.8.4 AI-Powered Code Completion

The use of AI-powered code completion is becoming more common in software development [91]. It can help developer productivity, reduce cognitive load and improve code quality. By leveraging machine learning and natural language processing, this feature provides intelligent, context-aware suggestions that help developers write code faster and with fewer errors. It streamlines repetitive tasks and promotes best coding practices through smart recommendations. A potential use case for this feature is to estimate logic between two nodes given a description or a set of example inputs and outputs. Incorporating AI-driven code completion not only can make IGC more competitive but also create a more user-friendly environment that can adapt to the evolving needs of modern developers.

8.8.5 Polyglot Programming

Polyglot programming is the practice of writing code in multiple languages. This would be useful as different languages are better suited for different tasks. IGC could potentially support polyglot programming by allowing the user to write code in code nodes, each using a different programming language. This would be a very ambitious feature for IGC, but it could be very useful. This would allow the user to use the best language for the task at hand and would allow the user to use IGC in a wider variety of projects. Polynote[1] is an example of a computational notebook that supports polyglot programming.

[1] https://polynote.org/latest/

A way to implement this feature would be to wrap each node with a language-specific microservice translation layer. For example, a code node containing Python could be requested to execute the code with parameters containing generic data from the previous state. The translation layer would be in charge of taking the request parameters and converting them to a format that the Python interpreter can understand. The translation layer would then execute the code and translate results back to the original input format as a new state. This would be a very complex and ambitious feature to implement, but it would be useful for many users.

Chapter 9

Conclusion

This thesis sets out to address a challenge in software development: the need to reconcile the rapid, interactive feedback and exploratory nature of IPEs with the structured, scalable, and maintainable characteristics of traditional text-based IDEs. By reimagining source code representation as a graph, containing a network of interconnected nodes and relationships, this work introduced IGC, a new programming environment that captures multiple executions along with the explicit orderings, code dependencies, and their corresponding relationships. In doing so, it addresses both the limitations of linear, cell-based systems and the challenges of managing complexity in large-scale software projects.

9.1 Summary of Contributions

The central contributions of this work can be summarized as follows:

- **Graph-Based Code Representation:** IGC redefines source code organization by representing code, documentation, and higher-level abstractions as nodes within a graph-based model. This model captures relationships and execution sequences visually allowing for a clear understanding of the underlying structure and facilitating an extendable way to manage code.
- **Incremental Programming for Software Engineers:** Extending beyond traditional IPEs, IGC incorporates incremental programming techniques into the realm of full-scale software engineering. The environment supports modular, step-by-step code execution that enables rapid prototyping and systematic refinement without sacrificing maintainability.
- **Exploratory Programming Interface:** IGC offers an interface designed with exploratory programming in mind. By employing a session system that records execution paths and views that support the branching, insertion, and replacement of code nodes, the environment empowers developers to experiment iteratively and compare alternative execution strategies.
- **Complexity Management:** Recognizing that complexity in software is not solely a function of code volume, but of its structural organization, IGC emphasizes techniques to manage and reduce structural complexity. Through modularization (via subgraphs) and visualization tools that focus on encapsulation and the reduction of cognitive load, the environment enhances "structuredness" while offering visualization tools necessary for codebase analysis.
- **Multi-View Analysis Capabilities:** IGC supports multiple customizable views of the codebase, such as projectional, exploratory programming, and graphical views. These diverse perspectives enable developers to navigate, analyze, and understand different aspects of their project, whether focusing on execution flow, dependency structures, or high-level architectural composition.

9.2 Addressing the Research Questions

RQ1: Redesigning the Programming Environment

This work demonstrates that a graph-based representation of source code can effectively merge the immediacy and flexibility of IPEs with the rigor and scalability of traditional IDEs. By:

- Allowing code fragments and their execution order to be visualized explicitly,
- Enabling exploratory programming through a session manager that tracks and manipulates execu-

tion paths,
- Supporting modularization via subgraphs that encapsulate complexity and offering analyzation tools through custom views,

IGC answers the question of how the programming environment can be redesigned to capitalize on incremental programming techniques, exploratory methods, and complexity management. The graph model retains the agility of rapid prototyping while also establishing a disciplined structure that scales as projects grow.

RQ2: Comparative Advantages and Disadvantages

The case studies and evaluations reveal several key insights regarding the benefits and limitations of the proposed environment:

- **Incremental Programming:** In comparison to traditional linear, cell-based IPEs, IGC's explicit execution sequencing through execution paths, represented by execution relationships, offers improved reproducibility and clarity. However, the current prototype lacks the full range of rich media outputs and flexible documentation found in some other IPEs.
- **Exploratory Programming:** The session system in IGC, which allows fine-grained control over execution branching and code replacement, offers a significant advantage over traditional cell-based approaches. Nonetheless, the new system could introduce a learning curve compared to the more familiar interfaces found in cell-based IPEs.
- **Complexity Management:** Techniques drawn from both PescaJ's aggregated views and the modular composition of Graph Nodes illustrate that the graph-based model can manage interdependencies and code navigation more effectively than linear representations. Yet, the complex structure of the graph poses its own challenges in terms of potential cognitive overload, necessitating further work in ways to filter and abstract the graph model.

9.3 Limitations and Future Directions

Despite the promising contributions, several limitations warrant consideration:

- **Language Support:** Although designed to be language agnostic in principle, the current implementation of IGC is centered on Python features for execution, state management, and analysis. Future iterations must extend language support and try to generalize disk-based state management and analysis potentially through the use of the language server protocol. The decision to employ a disk-based execution model was guided by the need for reproducibility and shareability. However, this choice introduces additional latency and language-specific challenges. Investigating hybrid models that leverage memory-based kernels alongside persistent state management remains a critical avenue for future research.
- **Subjective Nature of Case Studies:** The case studies presented in this thesis provide valuable insights into the potential benefits and disadvantages of IGC. However, their evaluations are primarily qualitative and subject to individual interpretation. Without systematic, quantitative user studies or controlled experiments, it remains challenging to generalize the observed advantages across a broader developer population. Future research should incorporate user studies with well-defined metrics, such as task completion time, error rates, and user satisfaction, to objectively assess the environment's efficacy and potentially validate these preliminary findings.
- **Interface Complexity and Usability:** While IGC's graph-based interface offers a rich, expressive means of representing code structure and execution flow, its inherent complexity may present a steep learning curve for new users. Enhancing usability remains a critical area for future improvement. This could involve the development of intuitive translation tools that facilitate seamless conversion between traditional IPEs and IDEs, as well as the incorporation of more fine-tuned documentation features. Streamlining these aspects would not only reduce cognitive load but also foster smoother adoption by programmers used to traditional development environments.
- **Scalability and Performance:** Although the current prototype demonstrates promising performance on small to moderately sized projects, scalability challenges are likely to emerge as project complexity increases. As the graph grows in size, issues such as rendering latency, memory/disk consumption, and overall responsiveness may become significant. Future work should explore optimization strategies, including more efficient graph rendering libraries and hybrid execution models

that blend in-memory processing with disk-based state management. Such enhancements will be essential to ensure that IGC can maintain high performance and remain responsive even as project demands scale upward.

9.4 Final Reflections

In conclusion, this thesis has demonstrated that re-envisioning source code as a graph can bridge the gap between incremental, exploratory programming and the structured demands of large-scale software development. IGC's design emphasizes explicit code representation, modular subgraphs, and multi-view analytical capabilities, which together enhance reproducibility, clarity, and flexibility. The approach not only preserves the benefits of rapid prototyping but also instills a structured framework that scales with project size.

Despite the promising results, the prototype reveals certain limitations that offer a clear direction for future features and research. The current implementation, centered on Python, must evolve to support additional languages while refining state management and execution strategies. Moreover, the expressiveness of the graph-based interface, though powerful, poses challenges in usability and may require advanced abstraction and filtering techniques to mitigate cognitive load. Scalability and performance under increasingly complex scenarios also remain important areas for further investigation.

In reflecting on the broader implications, it is evident that rethinking how we represent and interact with code can significantly improve both developer productivity and the maintainability of complex systems. By bridging incremental programming and complex software development in modern programming environments, this work lays a solid foundation for future advances in programming environment design.

Acknowledgements

I would like to extend my gratitude to the Software Engineering Master's program at the University of Amsterdam for their invaluable support. A special thanks to Dr. L. Thomas van Binsbergen for the weekly meetings that always pointed me in the right direction and guided the development of this work.

I am also deeply grateful for the support from my friends and family, whose encouragement and understanding have been invaluable throughout this journey. Thank you for believing in me and providing the motivation to strive for more.

Bibliography

[1] F. Pérez and B. E. Granger, "IPython: A system for interactive scientific computing," *Computing in science & engineering*, vol. 9, no. 3, pp. 21–29, 2007, Publisher: IEEE.

[2] T. Kluyver *et al.*, "Jupyter Notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, IOS Press, 2016, pp. 87–90. DOI: 10.3233/978-1-61499-649-1-87. [Online]. Available: https://ebooks.iospress.nl/doi/10.3233/978-1-61499-649-1-87.

[3] M. Beth Kery and B. A. Myers, "Exploring exploratory programming," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, ISSN: 1943-6106, Oct. 2017, pp. 25–29. DOI: 10.1109/VLHCC.2017.8103446. [Online]. Available: https://ieeexplore.ieee.org/document/8103446.

[4] P. Rein, S. Ramson, J. Lincke, R. Hirschfeld, and T. Pape, "Exploratory and Live, Programming and Coding," en, *The Art, Science, and Engineering of Programming*, vol. 3, no. 1, 1:1–1:33, Jul. 2018, Publisher: AOSA, Inc., ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2019/3/1. [Online]. Available: https://programming-journal.org/2019/3/1/.

[5] L. T. van Binsbergen *et al.*, "A Language-Parametric Approach to Exploratory Programming Environments," in *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2022, New York, NY, USA: Association for Computing Machinery, Dec. 2022, pp. 175–188, ISBN: 978-1-4503-9919-7. DOI: 10.1145/3567512.3567527. [Online]. Available: https://dl.acm.org/doi/10.1145/3567512.3567527.

[6] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers, "The Story in the Notebook: Exploratory Data Science using a Literate Programming Tool," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI '18, New York, NY, USA: Association for Computing Machinery, Apr. 2018, pp. 1–11, ISBN: 978-1-4503-5620-6. DOI: 10.1145/3173574.3173748. [Online]. Available: https://dl.acm.org/doi/10.1145/3173574.3173748.

[7] M. B. Kery, A. Horvath, and B. Myers, "Variolite: Supporting Exploratory Programming by Data Scientists," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, ser. CHI '17, New York, NY, USA: Association for Computing Machinery, May 2017, pp. 1265–1276, ISBN: 978-1-4503-4655-9. DOI: 10.1145/3025453.3025626. [Online]. Available: https://doi.org/10.1145/3025453.3025626.

[8] A. Head, F. Hohman, T. Barik, S. M. Drucker, and R. DeLine, "Managing Messes in Computational Notebooks," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, ser. CHI '19, New York, NY, USA: Association for Computing Machinery, May 2019, pp. 1–12, ISBN: 978-1-4503-5970-2. DOI: 10.1145/3290605.3300500. [Online]. Available: https://dl.acm.org/doi/10.1145/3290605.3300500.

[9] M. B. Kery and B. A. Myers, "Interactions for Untangling Messy History in a Computational Notebook," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, ISSN: 1943-6106, Oct. 2018, pp. 147–155. DOI: 10.1109/VLHCC.2018.8506576. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8506576.

[10] S. Chattopadhyay, I. Prasad, A. Z. Henley, A. Sarma, and T. Barik, "What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, ser. CHI '20, New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 1–12, ISBN: 978-1-4503-6708-0. DOI: 10.1145/3313831.3376729. [Online]. Available: https://dl.acm.org/doi/10.1145/3313831.3376729.

[11] J. Börstler *et al.*, “”I know it when I see it” Perceptions of Code Quality: ITiCSE ’17 Working Group Report,” in *Proceedings of the 2017 ITiCSE Conference on Working Group Reports*, ser. ITiCSE-WGR ’17, New York, NY, USA: Association for Computing Machinery, Jan. 2018, pp. 70–85, ISBN: 978-1-4503-5627-5. DOI: 10.1145/3174781.3174785. [Online]. Available: https://dl.acm.org/doi/10.1145/3174781.3174785.

[12] J. Harden, E. Christman, N. Kirshenbaum, J. Wenskovitch, J. Leigh, and C. North, “Exploring Organization of Computational Notebook Cells in 2D Space,” in *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, ISSN: 1943-6106, Sep. 2022, pp. 1–6. DOI: 10.1109/VL/HCC53370.2022.9833128. [Online]. Available: https://ieeexplore.ieee.org/document/9833128.

[13] J. Harden, “Exploring and Evaluating the Potential of 2D Computational Notebooks,” in *Companion Proceedings of the 2023 Conference on Interactive Surfaces and Spaces*, ser. ISS Companion ’23, New York, NY, USA: Association for Computing Machinery, Nov. 2023, pp. 97–99. DOI: 10.1145/3626485.3626554. [Online]. Available: https://dl.acm.org/doi/10.1145/3626485.3626554.

[14] L. O. Ejiogu, “A simple measure of software complexity,” *ACM SIGPLAN Notices*, vol. 20, no. 3, pp. 16–31, 1985, Publisher: ACM New York, NY, USA.

[15] D. D. Woods, E. S. Patterson, and E. M. Roth, “Can We Ever Escape from Data Overload? A Cognitive Systems Diagnosis,” en, *Cognition, Technology & Work*, vol. 4, no. 1, pp. 22–36, Apr. 2002, ISSN: 1435-5558. DOI: 10.1007/s101110200002. [Online]. Available: https://doi.org/10.1007/s101110200002.

[16] K. Petersen and C. Wohlin, “The effect of moving from a plan-driven to an incremental software development approach with agile practices,” en, *Empirical Software Engineering*, vol. 15, no. 6, pp. 654–693, Dec. 2010, ISSN: 1573-7616. DOI: 10.1007/s10664-010-9136-6. [Online]. Available: https://doi.org/10.1007/s10664-010-9136-6.

[17] A. J. Hey and G. Pápay, *The computing universe: a journey through a revolution*. Cambridge University Press, 2015, ISBN: 978-0-521-15018-7, ISBN: 978-0-521-15018-7.

[18] F. Cheng and F. Cheng, “Jshell,” *Exploring Java 9: Build Modularized Applications in Java*, pp. 57–65, 2018, Publisher: Springer.

[19] V. Vasilev, P. Canal, A. Naumann, and P. Russo, “Cling–the new interactive interpreter for ROOT 6,” in *Journal of Physics: Conference Series*, Issue: 5, vol. 396, IOP Publishing, 2012, p. 052 071.

[20] L. T. van Binsbergen, M. Verano Merino, P. Jeanjean, T. van der Storm, B. Combemale, and O. Barais, “A principled approach to REPL interpreters,” in *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Re ections on Programming and Software*, ser. Onward! 2020, New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 84–100, ISBN: 978-1-4503-8178-9. DOI: 10.1145/3426428.3426917. [Online]. Available: https://dl.acm.org/doi/10.1145/3426428.3426917.

[21] A. Rule, A. Tabard, and J. D. Hollan, “Exploration and Explanation in Computational Notebooks,” in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI ’18, New York, NY, USA: Association for Computing Machinery, Apr. 2018, pp. 1–12, ISBN: 978-1-4503-5620-6. DOI: 10.1145/3173574.3173606. [Online]. Available: https://dl.acm.org/doi/10.1145/3173574.3173606.

[22] J. M. Perkel, “Why Jupyter is data scientists’ computational notebook of choice,” en, *Nature*, vol. 563, no. 7729, pp. 145–146, Oct. 2018. DOI: 10.1038/d41586-018-07196-1. [Online]. Available: https://www.nature.com/articles/d41586-018-07196-1.

[23] A. Guzharina, *We Downloaded 10,000,000 Jupyter Notebooks From Github { This Is What We Learned | The Datalore Blog*, en, Dec. 2020. [Online]. Available: https://blog.jetbrains.com/datalore/2020/12/17/we-downloaded-10-000-000-jupyter-notebooks-from-github-this-is-what-we-learned/.

[24] *Visual Studio Extension Marketplace*, en-US. [Online]. Available: https://marketplace.visualstudio.com/vscode.

[25] B. Sheil, “Datamation®: Power Tools for Programmers,” in *Readings in arti cial intelligence and software engineering*, Elsevier, 1986, pp. 573–580.

[26] D. Frolich and L. T. van Binsbergen, "A Generic Back-End for Exploratory Programming," en, in *Trends in Functional Programming*, V. Zsók and J. Hughes, Eds., Cham: Springer International Publishing, 2021, pp. 24–43, ISBN: 978-3-030-83978-9. DOI: 10.1007/978-3-030-83978-9_2.

[27] E. Bousse, D. Leroy, B. Combemale, M. Wimmer, and B. Baudry, "Omniscient debugging for executable DSLs," *Journal of Systems and Software*, vol. 137, pp. 261–288, Mar. 2018, ISSN: 0164-1212. DOI: 10.1016/j.jss.2017.11.025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121217302765.

[28] R. Medina-Mora and P. Feiler, "An Incremental Programming Environment," *IEEE Transactions on Software Engineering*, vol. SE-7, no. 5, pp. 472–482, Sep. 1981, Conference Name: IEEE Transactions on Software Engineering, ISSN: 1939-3520. DOI: 10.1109/TSE.1981.231109. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1702873.

[29] K. J. O'Hara, D. Blank, and J. Marshall, "Computational notebooks for AI education," 2015. DOI: 10.13140/2.1.2434.5928. [Online]. Available: http://doi.org/10.13140/2.1.2434.5928.

[30] Y. Tashtoush, N. Abu-El-Rub, O. Darwish, S. Al-Eidi, D. Darweesh, and O. Karajeh, "A Notional Understanding of the Relationship between Code Readability and Software Complexity," en, *Information*, vol. 14, no. 2, p. 81, Feb. 2023, Number: 2 Publisher: Multidisciplinary Digital Publishing Institute, ISSN: 2078-2489. DOI: 10.3390/info14020081. [Online]. Available: https://www.mdpi.com/2078-2489/14/2/81.

[31] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, Dec. 2005, ISSN: 0360-0300. DOI: 10.1145/1118890.1118892. [Online]. Available: https://dl.acm.org/doi/10.1145/1118890.1118892.

[32] W. Ben Abdessalem Karaa, Z. Ben Azzouz, A. Singh, N. Dey, A. S. Ashour, and H. Ben Ghazala, "Automatic builder of class diagram (ABCD): An application of UML generation from functional requirements," en, *Software: Practice and Experience*, vol. 46, no. 11, pp. 1443–1458, 2016, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2384, ISSN: 1097-024X. DOI: 10.1002/spe.2384. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2384.

[33] O. Pastor, S. España, J. I. Panach, and N. Aquino, "Model-Driven Development," en, *Informatik-Spektrum*, vol. 31, no. 5, pp. 394–407, Oct. 2008, ISSN: 1432-122X. DOI: 10.1007/s00287-008-0275-8. [Online]. Available: https://doi.org/10.1007/s00287-008-0275-8.

[34] T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976, Conference Name: IEEE Transactions on Software Engineering, ISSN: 1939-3520. DOI: 10.1109/TSE.1976.233837. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1702388.

[35] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[36] M. R. Woodward, M. A. Hennell, and D. Hedley, "A measure of control flow complexity in program text," *IEEE Transactions on Software Engineering*, no. 1, pp. 45–50, 1979, Publisher: IEEE.

[37] E. Weyuker, "Evaluating software complexity measures," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1357–1365, Sep. 1988, Conference Name: IEEE Transactions on Software Engineering, ISSN: 1939-3520. DOI: 10.1109/32.6178. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6178.

[38] D. P. Tegarden, S. D. Sheetz, and D. E. Monarchi, "A software complexity model of object-oriented systems," *Decision Support Systems*, Information technologies and systems, vol. 13, no. 3, pp. 241–262, Mar. 1995, ISSN: 0167-9236. DOI: 10.1016/0167-9236(93)E0045-F. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167923693E0045F.

[39] H. M. Reis *et al.*, "Towards Reducing Cognitive Load and Enhancing Usability through a Reduced Graphical User Interface for a Dynamic Geometry System: An Experimental Study," in *2012 IEEE International Symposium on Multimedia*, Dec. 2012, pp. 445–450. DOI: 10.1109/ISM.2012.91. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6424704.

[40] A. M. Figueroa and R. Juárez-Ramírez, "The Problem of Cognitive Load in GUI's: Towards Establishing the Relationship between Cognitive Load and Our Executive Functions," in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, ISSN: 0730-3157, vol. 2, Jul. 2017, pp. 561–565. DOI: 10.1109/COMPSAC.2017.238. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8029990.

[41] J. Rilling, A. Se ah, and C. Bouthlier, \The CONCEPT project-applying source code analysis to reduce information complexity of static and dynamic visualization techniques," in Proceedings First International Workshop on Visualizing Software for Understanding and Analysis, IEEE, 2002, pp. 90{99.

[42] M. Sari et al., \SBGNViz: A Tool for Visualization and Complexity Management of SBGN Process Description Maps," en, PLOS ONE, vol. 10, no. 6, e0128985, Jun. 2015, Publisher: Public Library of Science, issn: 1932-6203. doi: 10.1371/journal.pone.0128985 . [Online]. Available: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0128985 .

[43] U. A. Mannan, I. Ahmed, and A. Sarma, \Towards understanding code readability and its impact on design quality," in Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering, ser. NL4SE 2018, New York, NY, USA: Association for Computing Machinery, Nov. 2018, pp. 18{21, isbn: 978-1-4503-6055-5. doi: 10.1145/3283812.3283820. [Online]. Available: https://dl.acm.org/doi/10.1145/3283812.3283820 .

[44] olawanletjoel, What is an IDE? IDE Meaning in Coding, en, Sep. 2022. [Online]. Available: https://www.freecodecamp.org/news/what-is-an-ide-for-beginners/ .

[45] A. Srivastava, S. Bhardwaj, and S. Saraswat, \SCRUM model for agile methodology," in 2017 International Conference on Computing, Communication and Automation (ICCCA), May 2017, pp. 864{869. doi: 10.1109/CCAA.2017.8229928. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8229928 .

[46] H. Haleem, Y. Wang, A. Puri, S. Wadhwa, and H. Qu, \Evaluating the Readability of Force Directed Graph Layouts: A Deep Learning Approach," IEEE Computer Graphics and Applications, vol. 39, no. 4, pp. 40{53, Jul. 2019, Conference Name: IEEE Computer Graphics and Applications, issn: 1558-1756. doi: 10.1109/MCG.2018.2881501. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8739137 .

[47] H. Kennedy and M. Engebretsen, \Introduction: The relationships between graphs, charts, maps and meanings, feelings, engagements," Data visualization in society, pp. 19{32, 2020, Publisher: Amsterdam University Press Amsterdam.

[48] J. Lopes and A. Santos, \PescaJ: A Projectional Editor for Java Featuring Scattered Code Aggregation," in Proceedings of the 2nd ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments, ser. PAINT 2023, New York, NY, USA: Association for Computing Machinery, Oct. 2023, pp. 44{50. doi: 10.1145/3623504.3623571. [Online]. Available: https://dl.acm.org/doi/10.1145/3623504.3623571 .

[49] B. Moosherr, \Constructing variation di s using tree di ng algorithms," en, Aug. 2023, Publisher: Universität Ulm. [Online]. Available: https://oparu.uni-ulm.de/items/3e366edf-9db5-4822-a117-39b8ba28a063

[50] E. Rinaldi, D. Sforza, and F. Pellacini, \NodeGit: Di ng and Merging Node Graphs," ACM Trans. Graph., vol. 42, no. 6, 265:1{265:12, Dec. 2023, issn: 0730-0301. doi: 10.1145/3618343. [Online]. Available: https://dl.acm.org/doi/10.1145/3618343 .

[51] D. Alymkulov, \Desktop Application Development Using Electron Framework: Native vs. Cross-Platform," 2019.

[52] R. York, Web development with jQuery. John Wiley & Sons, 2015.

[53] B. Green and S. Seshadri, AngularJS. " O'Reilly Media, Inc.", 2013.

[54] G. Geetha, M. Mittal, K. M. Prasad, and J. G. Ponsam, \Interpretation and Analysis of Angular Framework," in 2022 International Conference on Power, Energy, Control and Transmission Systems (ICPECTS), Dec. 2022, pp. 1{6. doi: 10.1109/ICPECTS56089.2022.10047474. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10047474 .

[55] C. Gackenheimer, Introduction to React. Apress, 2015.

[56] F. J. Pathari, Y. Nielsen, L. I. Andersen, and G. Marentakis, \Dark vs. Light Mode on Smartphones: E ects on Eye Fatigue,"

[57] B. Caldwell et al., \Web content accessibility guidelines (WCAG) 2.0," WWW Consortium (W3C), vol. 290, no. 1-34, pp. 5{12, 2008.

[58] V. Driessen, A successful Git branching model, en, Jan. 2010. [Online]. Available: http://nvie.com/posts/a-successful-git-branching-model/ .

[59] S. Cabello and B. Mohar, \Adding One Edge to Planar Graphs Makes Crossing Number and 1-Planarity Hard," SIAM Journal on Computing, vol. 42, no. 5, pp. 1803{1829, 2013, doi: 10.1137/120872310. [Online]. Available: https://doi.org/10.1137/120872310.

[60] O. Levin, Discrete Mathematics: An Open Introduction. Amazon Digital Services LLC - Kdp, 2018, ISBN: 978-1-79290-169-0, isbn: 978-1-79290-169-0. [Online]. Available: https://books.google.nl/books?id=YTAWwQEACAAJ

[61] K. Grotov, S. Titov, V. Sotnikov, Y. Golubev, and T. Bryksin, \A large-scale comparison of Python code in Jupyter notebooks and scripts," in Proceedings of the 19th International Conference on Mining Software Repositories, ser. MSR '22, New York, NY, USA: Association for Computing Machinery, Oct. 2022, pp. 353{364, isbn: 978-1-4503-9303-4. doi: 10.1145/3524842.3528447. [Online]. Available: https://dl.acm.org/doi/10.1145/3524842.3528447.

[62] J. Juhar and L. Vokorokos, \A review of source code projections in integrated development environments," in 2015 Federated Conference on Computer Science and Information Systems (FedCSIS), Sep. 2015, pp. 923{927. doi: 10.15439/2015F289. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7321542.

[63] N. L. Schroeder and A. T. Cenkci, \Spatial Contiguity and Spatial Split-Attention E ects in Multimedia Learning Environments: A Meta-Analysis," en, Educational Psychology Review, vol. 30, no. 3, pp. 679{701, Sep. 2018, issn: 1573-336X. doi: 10.1007/s10648-018-9435-9. [Online]. Available: https://doi.org/10.1007/s10648-018-9435-9.

[64] L. Friendly, \The design of distributed hyperlinked programming documentation," in Hypermedia Design: Proceedings of the International Workshop on Hypermedia Design (IWHD'95), Montpellier, France, 1{2 June 1995, Springer, 1996, pp. 151{173.

[65] M. AlSharif, W. P. Bond, and T. Al-Otaiby, \Assessing the complexity of software architecture," in Proceedings of the 42nd annual ACM Southeast Conference, ser. ACMSE '04, New York, NY, USA: Association for Computing Machinery, Apr. 2004, pp. 98{103, isbn: 978-1-58113-870-2. doi: 10.1145/986537.986562. [Online]. Available: https://dl.acm.org/doi/10.1145/986537.986562.

[66] B. Dobing and J. Parsons, \How UML is used," Communications of the ACM, vol. 49, no. 5, pp. 109{113, 2006, Publisher: ACM New York, NY, USA.

[67] G. Booch, I. Jacobson, J. Rumbaugh, et al., \The uni ed modeling language," Unix Review, vol. 14, no. 13, p. 5, 1996.

[68] I. H. Sarker and K. Apu, \MVC Architecture Driven Design and Implementation of Java Framework for Developing Desktop Application," en, International Journal of Hybrid Information Technology, vol. 7, no. 5, pp. 317{322, Sep. 2014, issn: 17389968. doi: 10.14257/ijhit.2014.7.5.29. [Online]. Available: http://gvpress.com/journals/IJHIT/vol7_no5/29.pdf.

[69] A. Ramrez-Noriega, Y. Martnez-Ramrez, J. Chavez Lizarraga, K. Vazquez Niebla, and J. Soto, \A software tool to generate a Model-View-Controller architecture based on the Entity-Relationship Model," in 2020 8th International Conference in Software Engineering Research and Innovation (CONISOFT), Nov. 2020, pp. 57{63. doi: 10.1109/CONISOFT50191.2020.00018. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9307798.

[70] M. Brambilla, J. Cabot, and M. Wimmer, Model-driven software engineering in practice. Morgan & Claypool Publishers, 2017.

[71] P. Teixeira, Professional Node. js: Building Javascript based scalable software. John Wiley & Sons, 2012, ISBN: 978-1118185469, isbn: 978-1-118-18546-9.

[72] J. Anderson and K. Keahey, \A Case for Integrating Experimental Containers with Notebooks," in 2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), ISSN: 2330-2186, Dec. 2019, pp. 151{158. doi: 10.1109/CloudCom.2019.00032. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8968868.

[73] R. Rodriguez-Echeverria, J. L. C. Izquierdo, M. Wimmer, and J. Cabot, \Towards a Language Server Protocol Infrastructure for Graphical Modeling," in Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, ser. MODELS '18, New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 370{380, isbn: 978-1-4503-4949-9. doi: 10.1145/3239372.3239383. [Online]. Available: https://dl.acm.org/doi/10.1145/3239372.3239383.

[74] M. Lutz, *Programming python.* " O'Reilly Media, Inc.", 2011, ISBN: 978-1-4493-0285-6, ISBN: 978-1-4493-0285-6.

[75] A. Baumann, J. Appavoo, O. Krieger, and T. Roscoe, "A fork() in the road," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '19, New York, NY, USA: Association for Computing Machinery, May 2019, pp. 14–22, ISBN: 978-1-4503-6727-1. DOI: 10.1145/3317550.3321435. [Online]. Available: https://dl.acm.org/doi/10.1145/3317550.3321435.

[76] A. Birrell, M. Jones, and E. Wobber, "A simple and efficient implementation of a small database," *SIGOPS Oper. Syst. Rev.*, vol. 21, no. 5, pp. 149–154, Nov. 1987, ISSN: 0163-5980. DOI: 10.1145/37499.37517. [Online]. Available: https://dl.acm.org/doi/10.1145/37499.37517.

[77] R. Riggs, J. Waldo, A. Wollrath, and K. Bharat, "Pickling state in the javaTM system," in *Proceedings of the 2nd conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 2*, ser. COOTS'96, USA: USENIX Association, Jun. 1996, p. 19.

[78] R. Keller and R. Schauer, "Design components: Towards software composition at the design level," in *Proceedings of the 20th International Conference on Software Engineering*, ISSN: 0270-5257, Apr. 1998, pp. 302–311. DOI: 10.1109/ICSE.1998.671356. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/671356.

[79] A. M. Turing *et al.*, "On computable numbers, with an application to the Entscheidungsproblem," *J. of Math*, vol. 58, no. 345-363, p. 5, 1936, Publisher: Wiley Online Library.

[80] S. Köhler, C. Schindelhauer, and M. Ziegler, "On Approximating Real-World Halting Problems," en, in *Fundamentals of Computation Theory*, M. Liśkiewicz and R. Reischuk, Eds., Berlin, Heidelberg: Springer, 2005, pp. 454–466, ISBN: 978-3-540-31873-6. DOI: 10.1007/11537311_40.

[81] B. Hailpern and P. Tarr, "Model-driven development: The good, the bad, and the ugly," *IBM Systems Journal*, vol. 45, no. 3, pp. 451–461, 2006, Conference Name: IBM Systems Journal, ISSN: 0018-8670. DOI: 10.1147/sj.453.0451. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5386628.

[82] C. Gobert and M. Beaudouin-Lafon, "Lorgnette: Creating Malleable Code Projections," in *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '23, event-place: San Francisco, CA, USA, New York, NY, USA: ACM, 2023. DOI: 10.1145/3586183.3606817.

[83] A. Bragdon *et al.*, "Code bubbles: Rethinking the user interface paradigm of integrated development environments," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10, New York, NY, USA: Association for Computing Machinery, May 2010, pp. 455–464, ISBN: 978-1-60558-719-6. DOI: 10.1145/1806799.1806866. [Online]. Available: https://dl.acm.org/doi/10.1145/1806799.1806866.

[84] A. Bragdon *et al.*, "Code bubbles: A working set-based interface for code understanding and maintenance," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10, New York, NY, USA: Association for Computing Machinery, Apr. 2010, pp. 2503–2512, ISBN: 978-1-60558-929-9. DOI: 10.1145/1753326.1753706. [Online]. Available: https://doi.org/10.1145/1753326.1753706.

[85] A. Goldberg and D. Robson, *Smalltalk-80: the language and its implementation.* USA: Addison-Wesley Longman Publishing Co., Inc., 1983, ISBN: 978-0-201-11371-6.

[86] A. Kay, D. Engalls, A. Goldberg, and D. Robson, "The history of smalltalk," *ACM Sigplan Not.(Mar. 1993)*, vol. 10, no. 155360.155364, 1993.

[87] B. Stefan L. Ram, *Dr. Alan Kay on the Meaning of "Object-Oriented Programming"*, en, Publication Title: https://www.purl.org/stefan_ram/pub/doc_kay_oop_en, Jul. 2003. [Online]. Available: https://www.purl.org/stefan_ram/pub/doc_kay_oop_en.

[88] L. P. Deutsch and A. M. Schiffman, "Efficient implementation of the Smalltalk-80 system," in *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1984, pp. 297–302.

[89] O. Agesen, "Design and implementation of Pep, a JavaTM just-in-time translator," *Theory and Practice of Object Systems*, vol. 3, no. 2, pp. 127–155, 1997, Publisher: Wiley Online Library.

[90] I. Borne, "A visual programming environment for Smalltalk," in *Proceedings 1993 IEEE Symposium on Visual Languages*, Aug. 1993, pp. 214–218. DOI: 10.1109/VL.1993.269599. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/269599.

[91] N. Nikolaidis, K. Flamos, K. Gulati, D. Feitosa, A. Ampatzoglou, and A. Chatzigeorgiou, "A Comparison of the Effectiveness of ChatGPT and Co-Pilot for Generating Quality Python Code Solutions," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)*, Mar. 2024, pp. 93–101. DOI: 10.1109/SANER-C62648.2024.00018. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10621717.