VRIJE
UNIVERSITEIT
BRUSSEL

Graduation thesis submitted in fulfilment of the requirements for the degree of
Master of Applied Sciences and Engineering: Computer Science

FLOWDOCS: ONBOARDING SOFTWARE ENGINEERS AT SCALE

Stefan-Calin Crainiciuc

June 2024

Promotors: Prof. Dr. Beat Signer
Advisors: Yoshi Malaise

sciences and bioengineering sciences

Contents

Abstract

As software systems become more complex, developers face challenges in understanding and debugging large codebases, particularly in large-scale projects with many moving parts, leading to communication difficulties between developer teams. In this thesis, we introduce a novel approach to address these challenges by presenting a Visual Studio Code (VSCode, or VSC) extension designed to aid developers in comprehending complex Python codebases and communicating with their peers. The extension employs static code analysis to generate function call graphs and a real-world mapping between code fragments and code maintainers.

The main purpose of our extension is to provide an enhanced visual representation of the code execution flow, helping developers identify code fragments that interact with different microservices and better comprehend the interaction. By evaluating the extension, we can accomplish the objective of our thesis, which is to gain deeper knowledge in the context of developers' understanding of complex codebases, and the methods of communication employed by developers in a large company.

We gathered data about developer issues through a preliminary form, which collected 44 responses, followed by a qualitative interview conducted with 4 developers. We interpreted the data to collect information about the issues developers encounter within our context and relevant feedback how they perceive our extension.

Chapter 1

Introduction

1.1 Problem Statement

The assimilation of new developers in a large company introduces many challenges to both the newcomers and other employees, who may be tasked with helping the newcomer fit in. The challenges range from understanding the newcomers' personality and culture to creating a detailed tutorial to guide them through the code base and getting them in touch with a mentor or instructor. Additionally, when projects have many moving parts, as is the case with microservices, communication becomes a major challenge. The software engineer will have to interact with different developers across teams, this scenario is further exacerbated in the context of global software development, where employees can have different schedules and work in different timezones. In the fortunate case, the newcomer could resort to a spreadsheet which contains the contact information of the team leaders or code maintainers. This solution is a relatively common approach to managing communication in a large company. However, there is a risk that old versions or even conflicting versions of the file circulate in the company, so there is no longer one source of objective truth. We argue that this approach can be improved in terms of efficiency, by merging it into the code analysis and comprehension workflow. We aim to help users to follow data and actions as they move through the codebase, and make it easy for them to identify a human point of contact that can help them with any questions or doubts, for every component, directly from the comfort of their integrated development environment (IDE). More hurdles and solutions for onboarding will be discussed in Section 2.1.2.

To summarise, within the context of onboarding new developers in a large company, we have identified two problems to which we propose a solution. The first problem is the difficulty developers experience while trying to learn, comprehend, and interact with the microservice ecosystem within the context of their new job. The second problem is the difficulty of knowing who to contact in case of a question or remark concerning existing code. The problems can even intersect with each other, thus becoming more difficult to untangle, in the case where a developer does not understand the current code and needs to ask clarifications from more experienced developers.

1.2 Contribution

The first contribution of our thesis is the extensive literature review conducted on the topics of knowledge management and transfer, onboarding in large companies, code comprehension

and code visualisation. We collect a set of common issues experienced in code comprehension by onboarding developers and developer working in large scale GSD companies with complex codebases and architectures. We create a map between these issues and state-of-the-art solutions, and through this we find shortcomings and potential improvements.

We further explore the topic, starting by building prototype solutions. By taking advantage of our constructed map, we are able to efficiently run prototyping iterations. We gain meaningful knowledge on which possible solutions have shortcomings themselves, finding indications on which parts of the solution space should be further explored and which show little sign of being effective at overcoming the identified obstacles.

We take the most promising prototype and develop it in a full-fledged solution, which combines the conclusions drawn from the literature review and the lessons we learned from prototyping. The solution takes the shape of a VSCode extension which aims to better integrate the social and communication-based aspect of working in a large company inside the daily developer's routine, as well as provide a visual aid for understanding microservice workflows.

Finally, we design an evaluation methodology, with which were are able to answer our proposed research questions. With the use of a survey, we gain a large-scale and general idea of the perceived usefulness of our solution as well as confirm the issues identified by the literature. Then, by conducting a set of qualitative interviews we gain valuable information on developer needs when facing code comprehension and communication issues, and we get an in-depth showcase of user interaction with our tool, leading us to identify both strong points and shortcomings of our extension. We gain inspiration on how to further develop our solution to more successfully answer the needs of the developers. As our study focuses on two distinct problems, our proposed solution can also be divided in two features, each aiming to solve one of the identified obstacles.

The first feature has the goal of helping developers better comprehend microservice code. We will refer to this feature "Flow Documentation". This feature aims to find whether visualising function execution as a graph (i.e. a flowchart) within VSCode helps developers better understand code with a microservice architecture.

The second feature targets newly employed developers who have questions or remarks about a piece of code, and will help them connect faster with other developers responsible with maintaining or developing that code. We will refer to the responsible developers as "code maintainers" or "maintainers" and to the second feature as "Maintainer Lookup". This feature aims to assist developers who are new to a large company in connecting sections of code with code maintainers, who have complex knowledge about the code they oversee. "Maintainer Lookup" stands as a replacement for other common ways of finding code maintainers, such as asking coworkers, asking the HR department, or looking up the information in a shared spreadsheet.

Chapter 2

Related Work

We will separate the literature review into two sections, one for each of the features that we have built. The literature review of "Flow Documentation" focuses on the microservice architecture and different code comprehension and visualisation techniques, while the literature review of "Maintainer Lookup" will discuss the issue of onboarding new employees in a company, knowledge management and transfer, interdepartmental integration and global software development.

2.1 Maintainer Lookup

2.1.1 Knowledge Management and Transfer

In his 1966 book "The Tacit Dimension", Michael Polanyi [23] discusses the concept that much of human knowledge is implicit and cannot be fully articulated. Polanyi introduced the idea of "tacit knowledge" to describe this form of understanding, which contrasts with explicit knowledge that can be easily communicated through words and symbols.

In "The Knowledge-Creating Company" [20] and "The knowledge-Creating company: How Japanese companies create the dynamics of innovation" [21], Nonaka et al. explore how Japanese companies, especially those in dynamic industries, maintain competitive advantage through continuous innovation, creating, and managing knowledge. The authors argue that these companies keep the advantage by converting tacit knowledge, which is personal and hard to formalise, into explicit knowledge that can be shared and leveraged across the organisation. This process of knowledge creation is seen as central to their business strategy and operational success.

The authors introduce four ways of knowledge conversion: socialisation, articulation, combination, and internalisation. Socialisation involves sharing tacit knowledge through shared experiences, while articulation converts tacit knowledge into explicit concepts, often through dialogue and reflection. Combination involves integrating different pieces of explicit knowledge to form a more complex understanding, and internalisation is the process by which explicit knowledge becomes tacit again through practice and learning.

Grundstein [11] explores the evolution of knowledge management (KM) and the methods of conversion between explicit and tacit knowledge within an organisation. Grundstein points out that conversion means more than capturing and storing knowledge, as it also includes actively managing and utilising it for organisational benefit. Knowledge conversion involves transforming individual, often tacit knowledge into explicit knowledge that can be shared across the organisation. Grundstein builds on the idea that tacit knowledge, which is deeply rooted in personal experience and context, is difficult to articulate but essential for innovation.

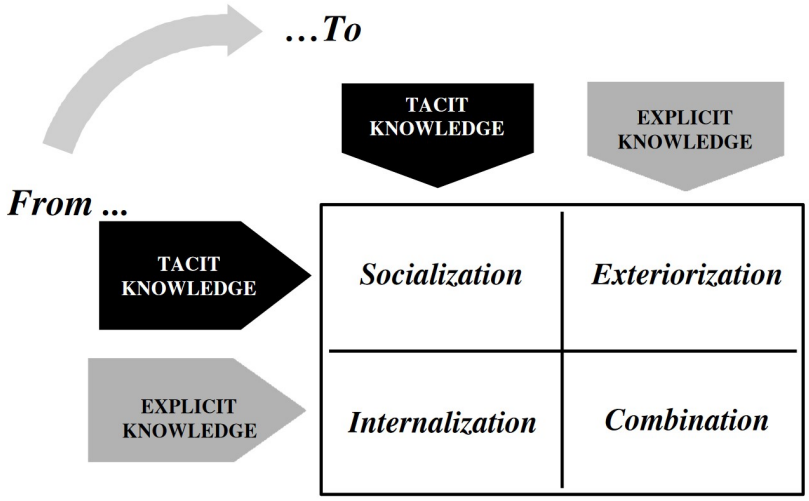


Figure 2.1: The methods of knowledge conversion, Source: "The Knowledge-Creating Company, Oxford Press, 1995"

Argote et al. [2] define knowledge transfer (KT) as the process by which individuals or groups acquire new knowledge or skills from others. They argue that knowledge transfer is essential for success because it enables organisations to adapt quickly in response to changing market conditions. They highlight that effective knowledge transfer has increased the success of interconnected organisations such as franchises. The article also highlights that personnel training and communication are effective methods of improving knowledge transfer within an organisation.

2.1.2 Onboarding of Employees in Large Companies

The process of onboarding integrates new employees into an organisation's culture and processes. Its goal is to help new employees quickly adapt to their roles, understand responsibilities, and become productive team members. Onboarding typically includes orientation sessions, training, mentorship, and resource sharing. For software developers, onboarding also involves technical training to familiarise them with the development environment, tools, technologies, and coding standards. Onboarding is a process that occurs predominantly in larger companies or communities, as larger companies are more likely to have a shared and structured knowledge system and more experienced developers. In smaller companies or startups, a new employee might take over some of the responsibilities of another established employee, in which case a less formal "onboarding" will occur, in the sense that the older employee will transfer some of their work related knowledge to the new employee. The process of onboarding can be guided (with the help of a mentor) or independent (through shared company knowledge and documents). We will focus our attention on the guided onboarding process in a large company or community.

Steinmacher et al. [26] conducted a systematic literature review (SLR), followed by interviews with 36 open source contributors, to identify 58 barriers newcomers face when placing their first contribution in an open source project, including 13 social barriers. The authors claim that the *"newcomers' seamless onboarding is important for online communities that depend upon leveraging the contribution of outsiders"*. While acknowledging that each project has unique challenges, the study aimed to solve this issue by considering different projects and developer profiles.

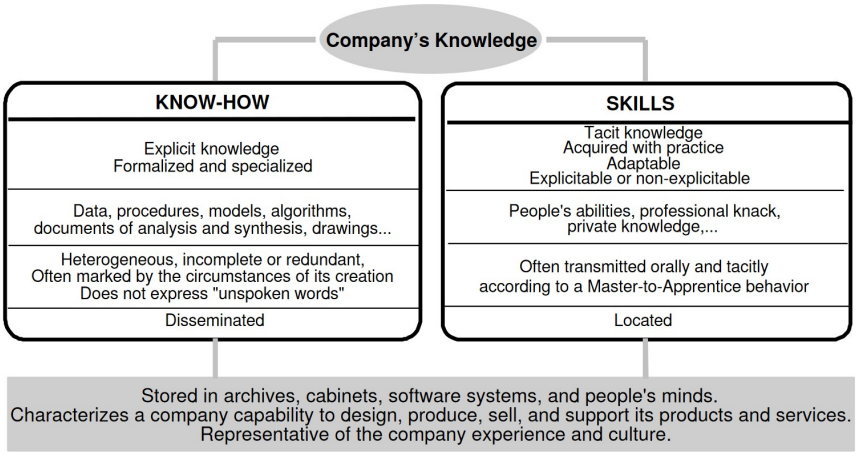


Figure 2.2: The two main categories of company knowledge, Taken from [11]: "From Capitalizing on Company Knowledge to Knowledge Management"

The systematic literature review included studies on all phases of the joining process, not just the first contribution, while interviews focused on barriers faced during the joining process. The authors categorise the 13 social barriers into answer reception issues, newcomer communication issues, orientation issues, and cultural differences. Other technical issues that might be encountered by newcomers are a lack of documentation (design documents, code documentation, project structure documentation), and cognitive problems (understanding architecture/code structure, understanding flow of information). The "Flow Documentation" feature aims to solve the cognitive problems that developers encounter while browsing large codebases, while the "Maintainer Lookup" feature helps with the social issue of providing orientation to newcomers by connecting them with a mentor, which will answer their codebase related questions.

Helic et al. [12] describe Web-Based Training (WBT) systems. Traditional WBT systems have limitations in supporting various training scenarios, which can be addressed through approaches collected from the literature, such as web-based tutoring, mentoring, knowledge mining, profiling, and delivery. The authors have built the WBT-Master tool to incorporate all the mentioned improvements. The tool works with the concept of "Knowledge Cards", which are descriptions of a particular topic or concept, for example "Database Technology" or "Information Systems", and contain information on learning resources for this concept. The knowledge cards are combined into a semantic network and linked through "is a part of" relationships. The authors highlight that the semantic network describes a graph structure as opposed to a hierarchical structure.

Helic et al. evaluated the WBT-Master with three partners over the course of two years, focusing on learning effectiveness analysis, usability analysis, and cost-benefit analysis. Results varied among participants, with some reporting improved learning effectiveness and others noting usability concerns that affect the learning outcomes.

The authors claim that knowledge cards have many benefits to traditional document browsing, such as providing a consistent structure to information, automatic inference of details based on the semantic relationships of the card, reducing repeated knowledge transfer, and acting as an entry point to topic exploration and server resources.

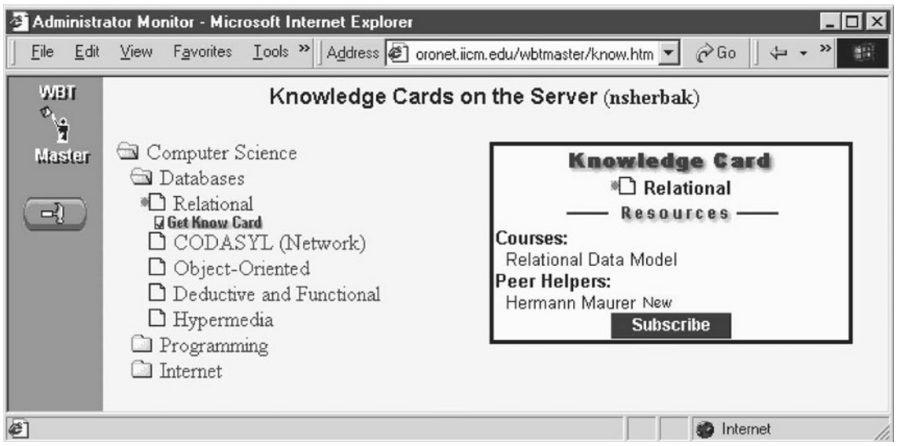


Figure 2.3: A Knowledge card, Taken from [12]: "Knowledge Transfer Processes in a Modern WBT System"

2.1.3 Interdepartmental Integration and Global Software Development

Interdepartmental integration is the process of ensuring that different departments within an organisation work together smoothly. This involves coordinating activities, sharing information, and aligning goals in areas such as marketing, production, finance, and human resources. The goal is for all departments to collaborate towards the organisation's objectives efficiently. This kind of integration makes use of explicit information which is shared across departments.

Kahn [13] conducted a mail survey to examine the effects of interdepartmental integration on product development and management performance. The questionnaire participants were individual department managers from marketing, manufacturing, and R&D departments in the electronics industry. Of the respondents, 177 were marketing managers, 157 manufacturing managers, and 180 R&D managers. The author examined the impact of interdepartmental integration through the lens of collaboration (achieving collective goals, mutual understanding, shared vision) versus interaction (transactions, communication). The study found that collaboration had a significant positive effect on performance in five of six interdepartmental relationships. These results support the importance of collaboration in interdepartmental integration, contrasting previous research which emphasises interaction alone.

Wikis are another way in which tacit information can be collected and disseminated. Buffa [5] summarised their personal experience with wikis in a case study involving two French organisations. The author claims that a company needs a wiki to share knowledge between employees and improve productivity by capturing valuable information on their intranet. The intranet plays different roles in an organisation, serving as a communication tool, and even becoming its memory by improving knowledge sharing between members. However, the main challenge lies in capturing knowledge and making it searchable online. Buffa claims that Wikis can help with knowledge sharing and creativity among employees, but for a wiki to be successful, technical skills are required for software installation and maintenance, as well as social aspects, such as user participation and local guru support.

Taweel et al. [27] have analysed the difficulties encountered in distributed software projects through a case study. They claim that the physical distance between teams reduces the frequency of informal communication between colleagues, which in turn lowers the success rate of projects. They identified two issues, namely inadequate synchronous and asynchronous collab-

oration mechanisms and lack of regular coordination between team members. After observing the evolution of the project, the authors suggest several ways of improving interdepartmental collaboration, including: the need for regular synchronous meetings scheduled in advance, the availability of multiple communication channels, and knowledge management tools.

Mazorodze and Buckley [17] organised a user study with 112 participants from Namibian knowledge-intensive organisations and found that a community of practice is the most effective tool for knowledge transfer in knowledge-intensive organisations, with 40% considering it effective and 27% very effective. A community of practice is a group of people who share a common interest and interact regularly to improve their knowledge and skills. Mentoring is the second most effective solution, with 54% considering it effective and 11% very effective. Other plans, such as storytelling, succession plans, coaching, and knowledge repositories were found less effective. The authors highlight that mentoring provides professional socialisation and improves self-confidence for mentees, while also allowing mentors to reflect on their own practice.

Nidhra et al. [19] discuss knowledge transfer in global software development (GSD), identifying challenges and mitigation strategies. The authors have conducted a systematic literature review containing 35 studies and conducted 8 interviews with industry professionals. The authors argue that insufficient KT can lead to project failure, and delays in communication can cause a loss of important information and context-dependent knowledge that is difficult to transfer. According to one of the interviewees in the study, an additional cost is incurred at the client end for repetitive KT, that is, due to a variety of reasons, the same knowledge has to be transferred multiple times. According to the authors, companies adopt person-to-person communication and collaboration to create and share knowledge.

Sourcing from Betz et al. [3], Nidhra et al. [19] highlight that remote team members, especially from Asia, use emails as their preferred method of communication within their company. Additionally, team members who are not confident with their English language skills prefer asynchronous communication because they get more time to comprehend and compose a response, compared to synchronous methods, such as instant messaging apps or video conferences.

The authors have identified a total of 47 challenges from the systematic literature review, and 29 challenges from developer interviews. Two of these issues are particularly interesting to us, namely **delays due to centralised communication flow** and **loss of information due to centralised communication**, which were identified by both the literature review and interviews. The solutions proposed by the authors for these issues are: client-vendor informal face-to-face meetings, **promoting high volume of communication**, building up team knowledge to reduce single points of failure, and **direct communication of offshore teams with onsite teams**. We chose to focus on these issues while developing the "Maintainer Lookup" feature of our extension. We aim to find out how promoting a high volume of communication and supplying direct communication of offshore team with onsite teams can solve the presented issues. Now let us explore the second section of our research, which focuses on the microservice architecture and different code comprehension and visualisation techniques, in order to build the second feature, "Flow Documentation".

2.2 Flow Documentation

2.2.1 Code Comprehension

Xia et al. [29] try to find out how much time developers spend to comprehend a software program. Previous studies show that program comprehension takes up a large portion of time spent on software maintenance. The authors conducted a large-scale field study with 78 developers in 7 industrial projects, recording 3148 working hours of activity data. Their analysis found that

developers spend up to 58% of their time analysing software, often using web browsers and document editors. The authors challenge the popular assumption that senior developers spend less time on software comprehension, as the results do not show a significant difference between junior and senior developers. The authors claim that the percentages of time spent on software comprehension are also affected by the phase of a project. In the development phase, the team is relatively stable, so the time spent understanding code is shorter, while in the maintenance phase, the teams lose and gain developers, which means that newcomers have to spend more time to understand the code. The authors conclude that code comprehension takes a large portion of developers' time and can be improved by providing targeted tools and support.

Dias et al. [9] have created Hunter, which is a visualisation tool designed to help developers understand JavaScript applications by analysing code dependencies and structure. The interactions in Hunter allow programmers to analyse software metrics and identify components that need refactoring or can be removed (i.e. dead code). Previous studies have shown that developers use development environments for code comprehension tasks. Therefore, the authors designed Hunter as a development environment focused on software comprehension, comparing its capabilities with those of Visual Studio Code. The authors mention that Treemaps, Voronoi treemaps, and tree visualisations have been proven effective for representing hierarchical structures like software systems, though these were not used in the Hunter prototype. The authors have designed a controlled user experiment that included 16 software engineers in order to prove the effectiveness of their tool. Their results show that participants have better accuracy and speed in software comprehension tasks when using Hunter.

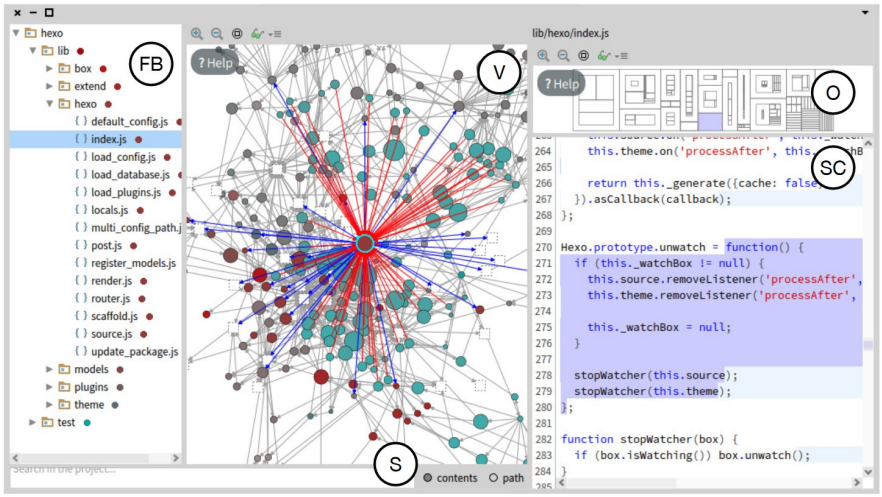


Figure 2.4: Overview of Hunter, Taken from [9]: "Evaluating a Visual Approach for Understanding JavaScript Source Code"

2.2.2 Microservice Architecture

To put the code comprehension information into a more specific context, let us explore the domain of microservice architectures. The microservice architecture is an approach to software development where a large application (i.e. a monolith) is broken down into smaller, loosely coupled services. Each service is designed to perform a specific business function and can be developed, deployed, and scaled independently. These services communicate with each other

through APIs, typically using protocols like HTTP or messaging queues.

Simhandl et al. [25] examine how developers comprehend call hierarchies in microservice and monolith architectures. Forty-two students were recruited for the experiment, which focused on typical interviewing or onboarding tasks when joining a new software development team. Participants were asked to perform feature location and source code modification tasks in both microservice and monolith systems. The results showed that developers comprehend call dependencies faster and spent about 8% less cognitive effort in the monolith variant. They also show that after studying the code for a longer time, the microservice tasks were completed with a better accuracy than the monolith tasks. The results show that participants applied a bottom-up strategy when comprehending microservices and applied a top-down strategy when comprehending monoliths.

Bogner et al. [4] investigated how Microservice API Patterns (MAPs) impact the understandability of service interfaces. The researchers conducted a controlled user experiment with 65 participants, where they present the developers six MAPs, such as: communicating and processing errors in the service, informing the API provider at runtime about the data the client is interested in, helping the user interpret the message content correctly, and increasing communication efficiency by reducing the number of API calls. The authors found that five out of six examined patterns had a significant positive impact on understandability, with effect sizes ranging from small to medium. Their results suggest that familiarity with MAPs is important to combat the increased complexity of the APIs and recommend educational efforts in academia and knowledge sharing among developers. They argue that future research should provide empirical evidence for other MAPs regarding quality attributes such as usability, maintainability, and reliability.

Aksakalli et al. [1] conducted a systematic literature review on deployment and communication patterns in microservice architectures, including 239 articles, 38 of which were selected as primary studies. They have mentioned many industrial issues, one of which is relevant to us. The authors refer to the issue of architectural complexity, which states that as a microservice architecture grows, operational complexity also increases, because more interactions occur between the microservices. The authors claim that managing connections among microservices and deploying thousands of services effectively becomes a concerning issue in terms of operational costs. They claim research should focus on developing management tools to control complexity and ease deployment and communication of microservices.

Zhang et al. [30] investigate the gaps between the developers' vision of microservices and reality through semi-structured interviews with 13 experienced participants from various software industries. The results show that while microservices offer benefits, such as independent development and scaling, they also bring difficulties. Microservice independence allows for easier development, testing, and deployment of services, but can lead to decreased testability and increased complexity in debugging, as encountered by 6/13 interviewees. API management is a significant challenge, as identified by 4/13 interviewees, due to the need for consistent understanding among teams and the lack of effective ways to ensure this consistency. Monitoring microservices is also problematic, as identified by 4/13 interviewees, with developers struggling with automated operation, troubleshooting, and threshold settings. The authors highlight the importance of addressing these pains to improve the adoption and implementation of microservices in industry. We will return to how code comprehension issues can be solved in a microservice architecture in Section 2.2.6.

2.2.3 Source Code Visualisation

Source code and software visualisation can help developers better understand how code is structured and reduce the cognitive load that they experience when analysing unfamiliar code by presenting it in a familiar format. Software visualisation has been thoroughly studied in the past decade, so there exist many software visualisation tools. Many of them focus on dependency analysis, which is a method used to understand and manage relationships between different elements or components within a system or project.

Dependency analysis helps software developers understand the relationships between components and make informed decisions during development. Limited support in integrated development environments (IDEs) forces developers to use standalone tools or sacrifice screen space. Code dependency analysis facilitates understanding of software architecture, debugging, refactoring opportunities, decision-making, and identifying code anti-patterns [10].

Daniel et al. [8] have created a node-link dependency visualisation tool, Polyptychon, and organised a case study to test it on a large Java repository, Netty. They created the dependency graph of Netty and qualitatively analysed the visualisation. Polyptychon is used to identify "problematic" nodes, which create tangles in the graph. The authors do not argue for the efficiency of the visualisation, but only its potential usage for software architecture evaluation. The authors identify that dependency information can be represented as node-link diagrams or matrix-based representations. They argue that node-link diagrams work well with small graphs but become difficult to navigate as the graph grows. Matrix-based representations have better scalability, but have a learning curve and are limited for pathfinding operations.

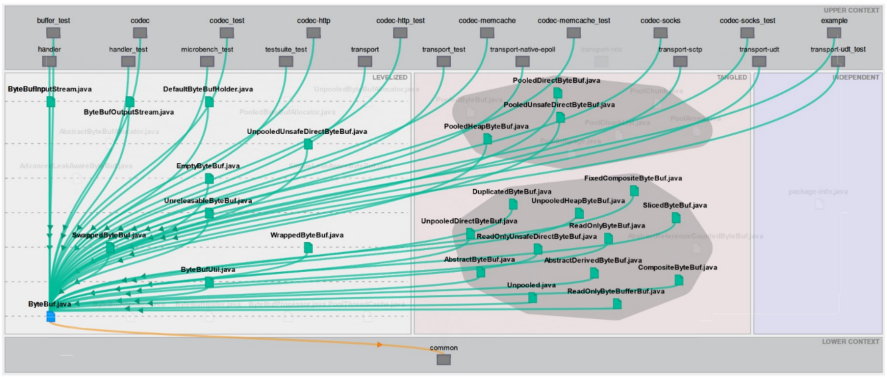


Figure 2.5: Polyptychon visualisation of the dependencies of Netty, Taken from [8]: "Polyptychon: A Hierarchically-Constrained Classified Dependencies Visualisation"

Similarly, Telea et al. [28] identify that large dependency sets are difficult to comprehend. In a small user study, five developers with no prior knowledge on five large-scale libraries were introduced to node link diagrams (NLD) and hierarchical edge bundle (HEB) visualisation methods. The developers used the two visualisation methods to answer generic questions about the systems' components, modularity, polymorphic interfaces, and dead code (a code artefact that is not called during the program execution and can be removed). Specific questions were also asked, such as which interfaces a component calls or provides and where an interface is used. The developers agreed that the hierarchical edge bundle layout is superior to node-link diagrams for navigating large graphs because it shows more data on screen, has less cluttered edges, and provides faster interaction. However, the developers also mention advantages of node link diagrams, including manual layout editing freedom, easier pathfinding, and reduced node proximity issues

compared to the hierarchical edge bundle's circular layouts. With this taken into account, we chose to implement the graph view of the "Flow Documentation" feature using a node-link diagram, as this makes it easier to follow a path, as we expect pathfinding to be the most performed action in software comprehension tasks.

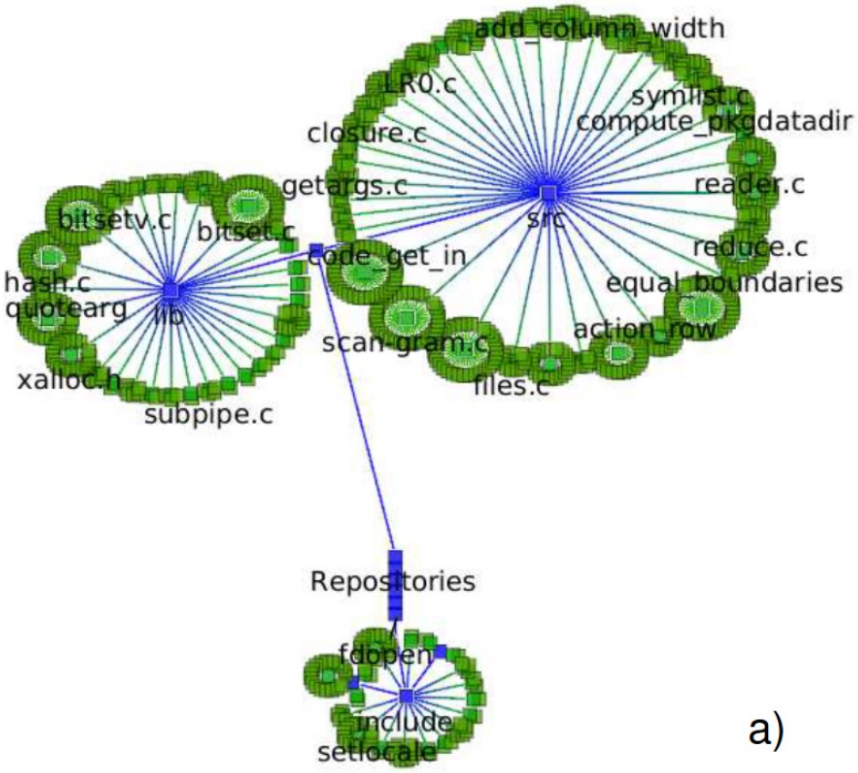


Figure 2.6: Visualisation of the bison call graph using bubble trees, Taken from [28]: "Extraction and Visualisation of Call Dependencies for Large C/C++ Code Bases: A Comparative Study"

According to Lungu et al. [15], software systems are often developed in parallel, with many companies, research institutions, and open source communities working on multiple related projects. These collections of projects form a software ecosystem. A super-repository is defined as a collection of versioned repositories for the projects in an ecosystem, containing information about the evolution of the projects and developers. The authors claim that the importance of super-repositories lies in their ability to provide reliable information about the ecosystem, as documentation can become outdated or inaccurate. They also contain valuable data on social aspects of the ecosystem, such as developer collaboration and movement between projects. The authors claim that visualising inter-project relationships helps identify important projects by revealing those others depend on.

Lungu et al. [15] have created the Small Project Observatory (SPO), which serves as a map into a super-repository. It contains several features for software developers, such as the Developer Collaboration Map (see Figure 2.7), which is a tool that shows how developers collaborate within an ecosystem across project boundaries, in the form of a graph. Nodes represent developer names, and links represent that the two developers collaborated in one or more projects. The edge also encodes which project the developers collaborated on, using the colour attribute. It is constructed

by tracking modifications made by developers in projects. The Developer Collaboration Map reveals moderately coupled communities within an ecosystem where half of contributors work on multiple projects while others work individually.

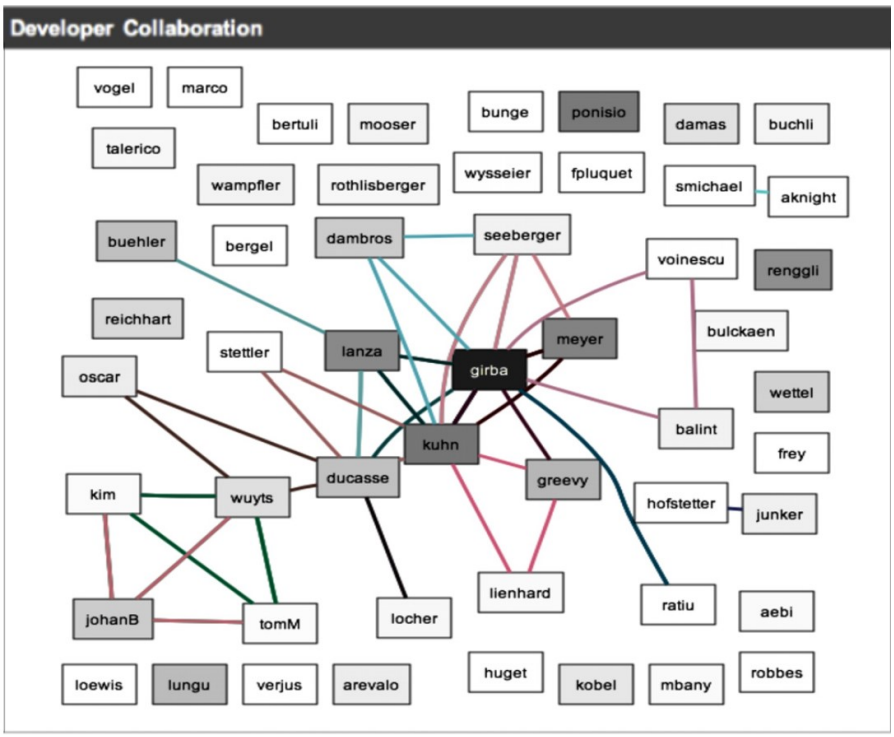


Figure 2.7: The Developer Collaboration Map in the Bern ecosystem, Taken from [15]: "The small project observatory: Visualizing software ecosystems"

The authors have validated the SPO using multiple methods. First, they used the project in open source case studies and discovered it enabled reasoning about project structure and developer collaboration. They also tested it with an industrial partner, who installed and analysed their own projects using SPO. Although the industrial partner had a unique approach to defining projects, Lungu et al. adapted the tool to accommodate their needs and received positive feedback overall. Finally, the authors conducted an experiment with students at the University of Lugano, where they tested their understanding of software ecosystems and usability of SPO by answering questions and rating their understanding on a Likert scale. The results showed that the students succeeded in forming a general idea of the relationships between developers and the project in the ecosystem.

2.2.4 Large Graph Visualisation

In this section, we will explore general graph visualisation techniques and standards.

Purchase [24] organised two controlled experiments with 55 computer science students, to compare certain graph aesthetic metrics in order to see which metrics affect the reader's ability of comprehending the graph. Based on their results, they claim that edge crossing is the metric that influences large graph readability the most. Edge bends and graph symmetry slightly influence graph comprehension, while orthogonality and minimum angle had no effect.

Landesberger et al. [14] conducted a state-of-the-art investigation and discussed the visual techniques for analysing large graphs. While analysing directed and undirected graphs, they also identify the techniques of node-link diagrams, matrix-based representation, and treemaps. The authors claim that node-link diagrams require careful node placement to support readability and aesthetics metrics, such as minimising node and edge overlaps, minimising edge crossings, and equalising edge lengths. In order to do this, they have identified several graph layouts. Force-based layouts simulate mechanical forces and work well for small graphs. Constraint-based layouts extend these with positional constraints. Multi-scale approaches lay out a coarser graph first, by using node aggregation, then refine it. Layered layouts, mainly for directed graphs, place nodes on parallel layers, representing a certain hierarchy or structure.

In addition to graph layout, graph readability can be improved by edge-bundling, the removal of node overlap, the usage of directed edges and arrows, node and edge colouring, and colour and thickness transitions. The authors also claim that preprocessing graph properties can help in visualising complex graphs by using positioning algorithms or highlighting important parts. Preprocessing can be automatic and involves graph filtering and aggregation. Stochastic graph filtering randomly selects nodes and edges, while deterministic filtering uses algorithms based on graph properties, such as edge betweenness centrality or node centrality. Graph aggregation merges nodes and edges to simplify graphs. With graph aggregation, the nodes and edges are merged into a single node, reducing the size of the graph. Similarly, Ma and Muelder [16] claim that extracting subgraphs from the overall graph simplifies the graph according to certain aesthetic metrics.

2.2.5 Dependency Visualisation and Code Exploration

There are several tools used in the industry to analyse and visualise dependencies between projects, using different formats. Doxygen[1] is a documentation generation tool primarily used for creating comprehensive, browsable documentation from annotated source code. It supports a multiple programming languages, including C++, Java and Python. Doxygen can automatically generate documentation in formats such as HTML, LaTeX, and RTF. Doxygen can also generate class and collaboration diagrams using Graphviz, a graph visualisation software (for an example of a diagram, see Figure 3.3). This feature allows developers to visualise the structure and relationships within their code, thus improving codebase comprehension. However, Doxygen has its drawbacks as a graph visualisation tool. The generated diagrams can become cluttered and hard to read for large and complex codebases. The dependency on Graphviz also means that users must install and configure an additional tool, which can add to the setup complexity. While Doxygen excels in static analysis and documentation, it lacks the interactivity and dynamic features found in more modern graph visualisation tools. Doxygen is a great tool for code documentation and basic visualisation, but it may not be the best choice for advanced graph visualisation.

GitHub Copilot[2] provides code suggestions and completions as developers write code, introducing them to new coding techniques, libraries, and patterns. It suggests alternative implementations and context-sensitive suggestions from the code. Copilot also suggests relevant snippets or entire functions based on the project context. This helps developers navigate large codebases and gain a deeper understanding of component interactions.

For example, GitHub Copilot can answer questions such as *"What is the dependency tree of this module?"*. The answer can be a starting point into how different functions, modules, or libraries interact within the codebase. In the future, tool like GitHub Copilot could, in theory,

[1] `https://www.doxygen.nl/`
[2] `https://github.com/features/copilot`

answer the questions that our extension tries to solve. However, currently there are several limitations to Large Language Model (LLM) tools. The attention layer of such models is limited to around 1000 to 2000 tokens, which means that the tool can misremember any information which was provided before the token limit. Additionally, LLM tools follow a nondeterministic answering process, which can lead to false results (hallucinations). The tool can not distinguish between a real fact and a hallucination, meaning that it is very difficult to persuade it that a result is false, once it establishes that it is true. This leads to confusion for the developer and hinders the learning process. We believe that it is worthy to continue investigating LLMs, as once these hurdles are overcome, the tools could provide much faster and accurate responses, which analyse not only the code and its structure, but even comments and file structure, which can lead to more nuanced results. With the context of the current LLM landscape in mind, let us analyse some theoretically possible interactions between a developer and GitHub Copilot.

When it comes to understanding code in a microservice architecture codebase, a developer could ask questions such as *"Can you help me understand how this feature interacts with other parts of the system?"*, *"What are all the places in the codebase where this specific function is called?"*. GitHhub Copilot could answer by providing the feature dependencies, API endpoints, and data flows, and generate a list of the locations where the function is used.

When it comes to finding people who are responsible for a certain piece of code, the user could ask questions such as: *"Who wrote this function and when was it last updated?"* or *"Can you show me the most recent changes to this file/function/module?"*. GitHub Copilot could connect to GitHub in order to identify the author of a piece of code, along with the commit creation date, and display a diff view of the latest commits affecting that file/function/module.

We can overcome the limitations of modern day LLMs by using traditional software analysis visualisation tools, which use either static or dynamic source code analysis. Such tools exist for most programming languages, and we will pick Python as an example.

Sourcetrail[3] is a tool that statically analyses source code in C++, Java and Python. It parses files to extract symbol declarations and references and then identifies dependencies between function calls and variable references. Sourcetrail visualises the dependencies using a flowchart, which allows users to explore the codebase and understand how components interact visually. Users can search for specific symbols or filter them by type, location, or usage context. Sourcetrail integrates with Visual Studio, Eclipse, and JetBrains through extensions which communicate with the main program through websockets.

Pydeps[4] is a Python package that extracts and analyses dependencies between modules in a project. It detects imported modules by searching for import opcodes in compiled Python files. Pydeps creates a graph where nodes are files, and directed edges indicate that the target node imports the start node in the code source. Pydeps can export the graph as a PNG or SVG file or the intermediate graph representation as a DOT file. A developer could then analyse the resulting PNG file to improve their understanding of the import and code structure. As Pydeps analyses compiled Python files, it only detects modules which are imported statically and excludes dynamic imports (e.g. modules imported using the `__import__` function). Pydeps also analyses module-level dependencies only, meaning that it can not tell whether a class or function inherits another class or function, which can be limiting for the developer who tries to understand the codebase.

Pyan[5] is very similar to Pydeps. The main difference between the two modules is that Pyan extracts dependencies at module-level, function-level, class-level, and method-level, while Pydeps is focused on module-level dependencies only.

[3] `https://github.com/CoatiSoftware/Sourcetrail`
[4] `https://github.com/thebjorn/pydeps/`
[5] `https://github.com/davidfraser/pyan`

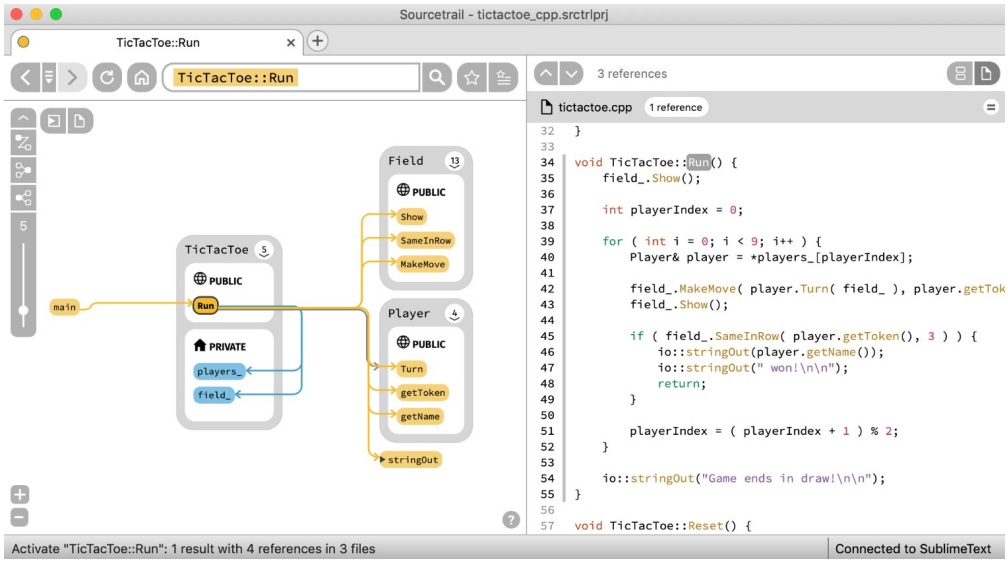


Figure 2.8: SourceTrail Visualisation of C++ Program, Source: `https://github.com/CoatiSoftware/Sourcetrail`

PyCallGraph[6] is a visual profiling tool that tracks function calls and execution times, by overriding function callbacks when entering or leaving functions to gather information on function names, call relationships, and execution metrics. PyCallGraph uses dynamic code analysis to extract dependencies.

2.2.6 Dependency Visualisation in a Microservice Architecture

The microservice architecture allows for independent development, deployment, and scaling of services, leading to rapid development and improved performance. However, the decoupling of services can make it difficult to maintain a holistic view of the system and ensure consistency between services. Bushong et al. [6] propose a method of Software Architecture Reconstruction (SAR) that uses static code analysis to recover the data model and interservice communication.

SAR can be accomplished through different methods, such as dynamic, static, or manual analysis of the code. Bushong et al. [6] use static analysis to identify microservice endpoints and calls between individual microservices. This allows them to extract a view of the system architecture before deployment. Their approach also provides an updated view of service APIs and interactions as code changes, which the authors claim is an important feature for monitoring tools.

The authors describe a method for creating a context map for a system composed of multiple microservices. They defined the context map as a visual representation of the relationships between entities and services in a system. The method involves two main steps: extracting bounded contexts from each microservice and combining the bounded contexts into a map for the entire system. Their method also uses standardised formats and annotations to identify metadata about endpoints and calls between services.

To demonstrate their approach, the authors implemented a prototype and applied it to the TrainTicket microservice benchmark, which consists of 41 microservices written in Java, Python,

[6] `https://github.com/gak/pycallgraph`

JavaScript, and NodeJS. The results showed that the tool was able to recover most of the entities, properties, and relationships in the system, but missed some due to ambiguity in the code.

Cerny et al. [7], in collaboration with Bushong, further developed the idea asking if it is feasible to use static code analysis to visualise microservice interaction in a large codebase. The authors use the SAR (Software Architecture Reconstruction) approach, similarly to Bushong et al. [6], which now involves three phases: extraction, construction, and manipulation. In the extraction phase, information is collected from each microservice. The construction phase compiles the information, creating Component Call Graphs (CCGs). The manipulation phase combines all CCGs into a single complete graph.

Cerny et al. [7] have implemented Prophet, a tool that statically analyses Java-based microservices using the Spring Boot framework. The authors conducted two case studies to test their tool, namely The Teacher Management System (TMS), which contains three microservices and TrainTicket, which contains 41 microservices. Prophet can represent the TMS project structure nicely, but the TrainTicket graph is convoluted and hard to read as there are more microservices, which communicate more frequently. The authors have also experimented with alternative visualisations within three-dimensional space and augmented reality (AR). This solves some graph complexity issues, as it visualises data in three dimensions instead of two.

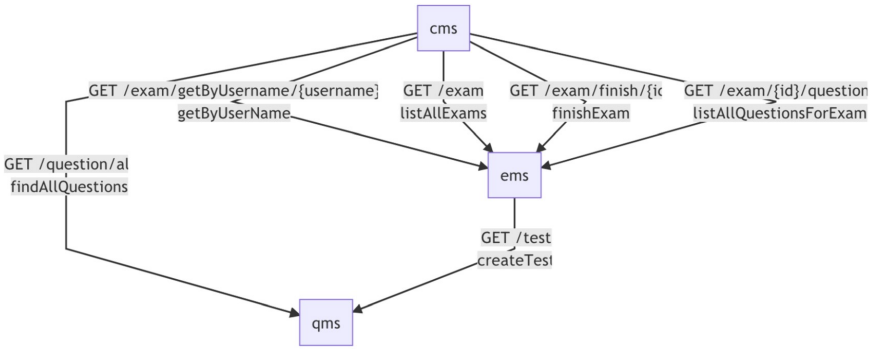


Figure 2.9: Sample service view extracted from the TMS benchmark, Taken from [7]: "From Static Code Analysis to Visual Models of Microservice Architecture"

The authors acknowledge several limitations in Prophet. The prototype tool is restricted to Java-based microservices using the Spring Boot framework and cannot detect event-driven communication between microservices. The study did not examine the way in which microservices evolve, which can predict software architecture degradation. Finally, the authors explored 2D, 3D, and AR visualisations, leaving room for further exploration in the virtual reality domain.

2.3 Literature Review Contribution

Following from the research we conducted, we have identified improvement opportunities in the field of newcomer onboarding and microservice architecture comprehension. Newcomer onboarding suffers from the common problem of finding a mentor for new employees in a large company, as previously identified by Steinmacher et al. [26] and Nidhra et al. [19]. In this thesis, our aim is to provide a solution to ease this issue, while also improving the inderdepartmental collaboration, which Kahn [13] identified as a way to improve product success in a company. We will achieve this goal by providing a solution integrated within the newcomer's IDE and workflow, in the form of a VSCode feature called "Maintainer Lookup". The features will use several techniques which

improve knowledge management and transfer, as identified by Nonaka [20] and Helic et al. [12], such as knowledge articulation and combination, web-based knowledge delivery and knowledge cards.

Xia et al. [29] have described the issue of code comprehension taking up to 58% of the developer's time, which we believe can be lowered using various software visualisation techniques. Simhandl et al. [25] have further analysed the issue of code comprehension in the microservice architecture vs. the monolith architecture. The authors claim that, while initially developers spend more time understanding microservices, after allowing some comprehension time, they perform modification tasks more accurately. We have also analysed the work of Cerny [7] et al., Bushong [6] et al., which aims to ease the task of microservice architecture comprehension by visualising the interaction between microservices using a graph. The authors have conducted a case study in which they show the usability of their tool on projects with both a large and small number of microservices. Their results seem promising for smaller scale projects, and we claim that their tool can be improved for large-scale projects. Their tool, Prophet, provides support for data exploration, but not enough for data modification, which is an important part of a software developer's routine. We acknowledge that the increasing visualisation complexity is inevitable when working with a large codebase, but it can be simplified by a couple of methods extracted from the literature. First, it is not always necessary to interact with the entire graph at once, so it can be simplified by looking at a smaller portion of a code (i.e. a business flow, as described in Section 4), which improves the readability of the graph according to Ma and Muelder [16]. We can also improve the readability of the graph by reducing the edge crossing using a force-directed graph algorithm, as identified by Landesberger et al. [14]. Additionally, by making the graph more interactive, certain nodes can be expanded/contracted on demand by the user, which can contract the size of the graph. Finally, using colours, we can separate the nodes in the graph by function or by the microservice that they belong to, making it easier for the user to distinguish between related and unrelated services, as identified by Landesberger et al. [14]. We use visualisation techniques which are particularly suitable for pathfinding operations, as identified by Telea et al. [28] and Daniel et al. [8]. As Freire et al.[10] highlight, the limited support of dependency analysis and software comprehension in IDEs forces developers to use standalone tools. We challenge this belief, by creating an IDE extension and comparing it to other standalone tools, such as Hunter (developed by Dias et al. [9]), SPO (Lungu et al.[15]), and Prophet (Bushong et al. [6] and Cerny et al. [7]). We chose to use a static analysis component in our library, following the exploration of both static and dynamic analysis of Bushong et al.[6]. The "Flow Documentation" static analysis of the code requires programming language-specific libraries and tools, but does not make the entire feature language dependent.

As the study of Cerny et al. [7] is closely related to the issues we want to solve with our "Flow Documentation" feature, we will integrate the proposed alternatives to create a more user-friendly experience, which will hopefully help developers better understand codebases with a microservice architecture.

2.4 Research Conclusion

In this chapter we have identified several ongoing issues in the state-of-the-art research relating to code comprehension tasks, and a need for developers to connect to a mentor when onboarding in a large company. We will detail how we tackled these issues in Chapter 3, and describe our solution in Chapters 4 and 5.

We are surprised to find so little research in the field of integrated solutions for connecting developers to mentors, since it is such an important issue in the context of large companies, which

can determine project success and newcomer integration. Similarly, there is a lack of comparative research on whether interactive visualisations offer benefits in terms of content comprehension, compared to static visualisations.

When it comes to support for IDE integrated tooling, we have only found knowledge management tools, such as Foam and Obsidian (later described in Chapter 3), but not for code analysis tools, which are implemented as separate application in the state-of-the-art (Doxygen, SourceTrail, Prophet [6], Hunter [9], SPO [15], Polyptychon [8]). While acknowledging that integrated tooling has less dedicated screen space [10] as a pitfall, we will focus our efforts to advance the research in the field of integrated software analysis and visualisation. We will also focus on overcoming issues related to microservice interaction visualisation in Prophet [6], by using several techniques identified by Landesberger et al. [14].

Chapter 3

Prototyping

3.1 Methodology

3.1.1 Research Questions

We design the prototyping, solution and evaluation chapters of our thesis with the goal of answering the following research questions (RQs):

- RQ1: What challenges do software developers face when trying to understand codebases with a microservice architecture and locating maintainers?
- RQ2: How do developers perceive the usefulness of FlowDocs for understanding codebases and locating maintainers?
- RQ3: How does FlowDocs influence developers' performance and accuracy in different tasks?

Research question 1 aims to confirm the validity of the issues we identified in Chapter 2. RQs 2 and 3 aim to demonstrate the effectiveness of our extension in solving these issues.

3.1.2 Design Science Research Methodology

The Design Science Research Methodology (DSRM) process model, first introduced by Peffers et al. [22] is a structured approach to conducting research in the field of Information Systems, particularly aimed at creating and evaluating IT artefacts (or prototypes) that address identified problems. The authors have defined DSRM as an iterative approach composed of six steps: problem identification and motivation, inferring the objectives of a solution, design and development, demonstration, evaluation, and communication. The process is illustrated in Figure 3.1.

With DSRM in mind, we started exploring the field of code comprehension with the goal of identifying common issues of developers in the state-of-the-art approaches. We will describe the problems we identified, our approach to solutions and how it fits within the DSRM process model.

3.2 Iterations

The first issue we wanted to look at is the need for better code comprehension tools, which was identified by Xia et al. [29], Telea et al. [28], and Dias et al. [9]. The state-of-the-art

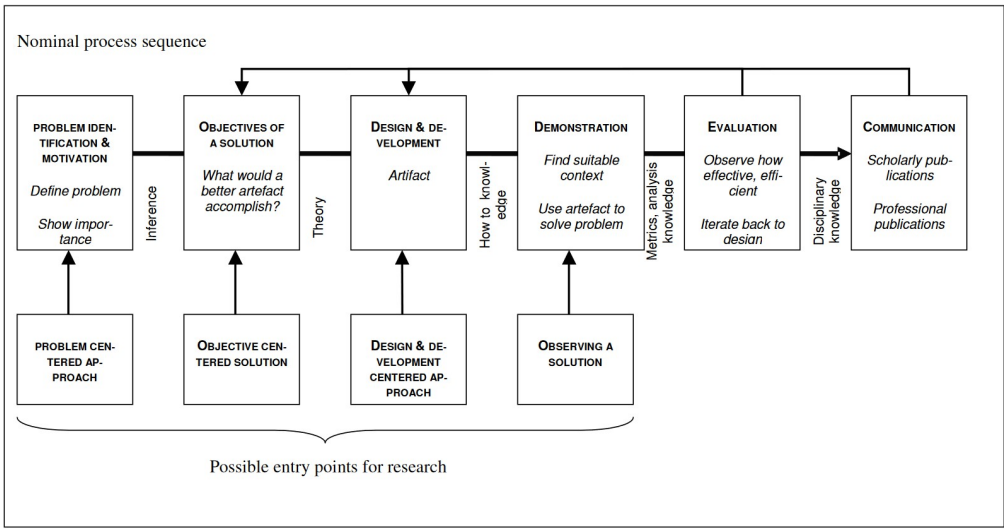


Figure 3.1: The Design science research process model, Taken from [22]: "A design science research methodology for information systems research"

literature highlights the large amount of time which developers spend while analysing and trying to comprehend existing code. We tried to solve this issue with two separate solutions, one involving knowledge management tools, such as Foam[1] and Obsidian[2], and the other involving code documentation and graph generating tools, such as Doxygen[3].

Foam works primarily with MarkDown files and enhances them with a number of useful shortcuts and reference symbols, such as: [[folder/path/to/reference.md]] - adds a link in the current file which points to reference.md, [[#tag]] - attaches the keyword 'tag' to your current file, and special properties, such as the note title and type. All these features help the user with keeping their information organised and up to date. The links, tags and special properties in a MarkDown file can then be compiled into a graph representation and displayed in a separate view, in order to visually and interactively represent the connection of information in the user system (see Figure 3.2).

We tried to extend this functionality to code with the purpose of generating code flow graphs. Each file or function would be converted into a note, and links would be created between functions that call each other and files which import each other. This solution has the objective of enhancing code comprehension by creating these navigable reference graphs with the help of Foam. We implemented a static analyser of Python code, as will be described in Section 5.4.2. The analyser had the purpose of generating an intermediate graph representation from Python code, which would then be passed to Foam for parsing and displaying. The process of generating the intermediate graph structure turned out to be quite convoluted. The Python files had to be annotated with the [[link]] delimiters, so that the corresponding file nodes could be clicked in the graph. While this prototype idea can work with larger projects, we think that it is not worth pursuing since the added annotations clutter the code file with information which makes code harder to follow and comprehend.

Separately, we also looked at how code documentation tools, such as Doxygen, can help de-

[1] `https://github.com/foambubble/foam`
[2] `https://obsidian.md/`
[3] `https://www.doxygen.nl/`

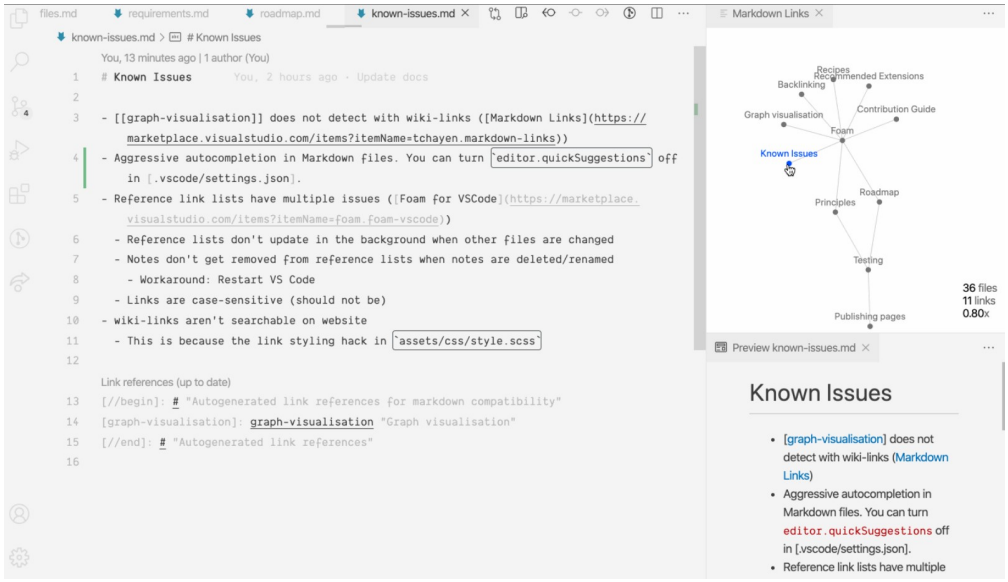


Figure 3.2: Foam Graph, Source: "`https://foambubble.github.io/foam/`"

velopers with the task of code comprehension. While the diagrams generated by Doxygen are quite comprehensive, they tend to become overly technical, with class names and scopes that span a large chunk of the visual graph. We hypothesise that the overabundance of information, along with the fact that generated graphs and documentation are static and do not allow user interaction, inhibits the developer's code comprehension efforts. The graph interactivity can be improved in several ways. For example, the user can view the graph in parallel with the code that they are trying to analyse. The user can pan and zoom the graph to focus on the information that they want to capture. The nodes of the graph can be clicked to either see the codes to which they refer, or they can highlight the interactions between that specific function or part of the code with the other functions that it calls, thus making the graph smaller and easier to understand. The edges can be clicked to open the line of code where the function call is located. We propose a solution, which uses Doxygen to produce the intermediate graph representation of a piece of code (that is, in DOT notation), then instead of rendering the graph with GraphViz, it reverse engineers the DOT file into a machine-readable format, such as JSON, and passes it to a custom graph renderer (as it will be described in Subsection 5.4.5). The graph labels can be simplified and displayed in a dynamic graph, which the user can use to navigate the codebase. While this sounds good on paper, we were confronted with the issue of simplifying node labels, which turned out to be nontrivial. For example, let us consider the node `std::stack<const DocNodeVariant*, std::deque<const DocNodeVariant*>>`. How should we go about simplifying the structure? Removing all the template parameters and keeping only `std::stack` does not convey sufficient information to the reader, and many nodes will have the same label, further confusing the developer. Removing access modifier and reference information turns the label into `std::stack<DocNodeVariant, std::deque<DocNodeVariant>>`, which is more understandable, but still occupies too much space on the screen. This is also a relatively simple example, since the namespace only consists of one level `std::`, while real world code contains many nested namespaces, which makes it even more difficult to label. We consider that if this issue is overcome, there is potential in creating interactive graphs from Doxygen DOT files, which

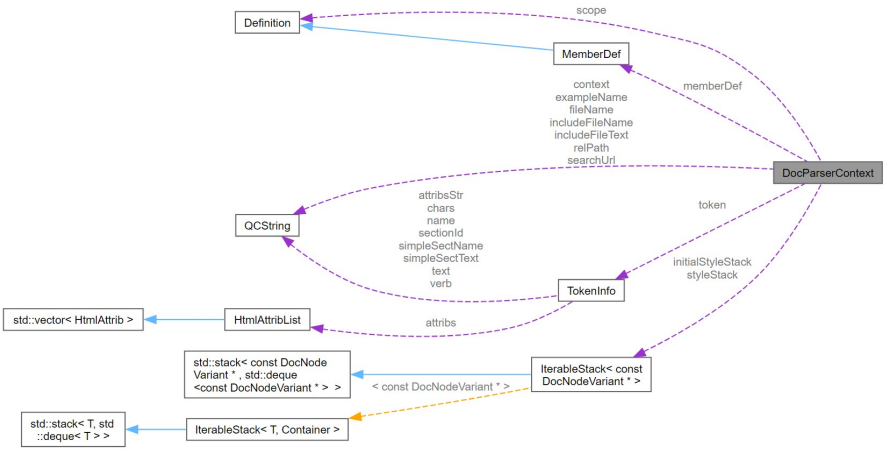


Figure 3.3: Doxygen Graph, Source: "`https://www.doxygen.nl/`"

can help the developer better comprehend the code.

After building the Python code analyser and the JSON graph renderer, we began to explore whether the "Flow Documentation" feature could be adapted for debugging purposes. The idea is that the interactive graph can also serve as a valuable tool for developers diagnosing and solving unintended behaviour in their project. We offered the hypothesis that that by tagging a function containing unintended behaviour as the starting point of a flow, developers could generate a flow call graph to trace which other functions or external services are involved during program execution.

To test this hypothesis, we have created a prototype which implemented the debugging helper features (see Figure 3.4). We can right-click on the nodes of the graph to either place a breakpoint in the code where the start of the function or view the function definition. We can also right-click on the edges in the graph to place a breakpoint at the line (or lines) of code in which the function call (from the start to the end node) is located or reveal the lines in the editor. The breakpoints present in the system are highlighted in the graph by colour, either by a red outline on a node or a red edge. The interactive graph, with clickable nodes and edges for adding breakpoints and starting debugging sessions, seemed promising.

We evaluated our prototype by conducting a small study with three developers who regularly fix bugs and unintended behaviour. This initial feedback phase revealed that our extension did not integrate well with their usual debugging workflows. These developers emphasised that their debugging process relies heavily on line-by-line code analysis, rather than starting with an overview and then delving into the details. One developer pointed out that the default Python debugger, along with the stack-trace viewer, is more efficient for locating the exact line of code where a bug occurs. This method proved to be equally effective for identifying bugs in external services, as placing a breakpoint after a service call and analysing the return variable was straightforward and easier to understand.

Similarly, we experimented with the "Affected Files" feature (Figure 3.5), which was designed to display a tree view of files potentially containing bugs. Our hypothesis was that this view could help developers isolate bug locations more efficiently by narrowing down the list of files or modules to inspect. The files can be displayed in a list, where the user can see all of the files and modules that might contain the bug. The nodes can be clicked to open the function definition in the code, or right-clicked so that they are focused and highlighted in the graph. However,

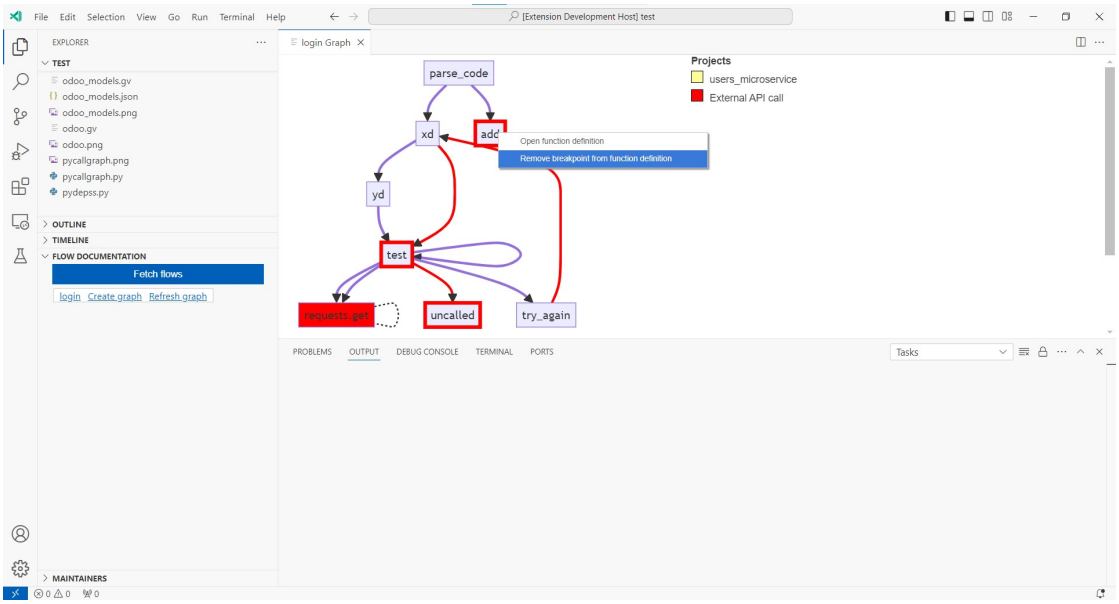


Figure 3.4: Using Flow Graphs as a Debugging Helper

the feedback from our study indicated that this feature did not seamlessly integrate into their debugging processes. While it received neither positive nor negative feedback, it was clear that it did not add significant value to their workflow.

The work of Simhandl et al. [25] shows that microservices are generally more difficult to understand at first sight than monolithic codebases. Our final and most advanced prototype is focused on making microservice codebases more understandable to any developer, including newcomers. The prototype will be described in Sections 4, 5 and evaluated in Section 6.

3.3 Prototyping Conclusions

Throughout this iterative development process, each round of feedback guided us to a more refined view of the goals of our extension. We learned that if we want users to better understand code structure, we have to provide several features. The graph visualisation method can condense the code in an entire project in an intuitive way for the user. The visualisation has to be interactive, such as Foam's graph, because static graphs, such as Doxygen's, tend to get lost in the documentation and get relatively low attention from developers. We learned that our solution has to be integrated in the existing tooling, or at least should communicate smoothly with a popular IDE, in order to lower the cognitive effort of developers to switch between different software for the same task. As it stands, the solution should not interfere with the traditional debugging workflow, as it does not comply with the line-by-line approach that many developers take.

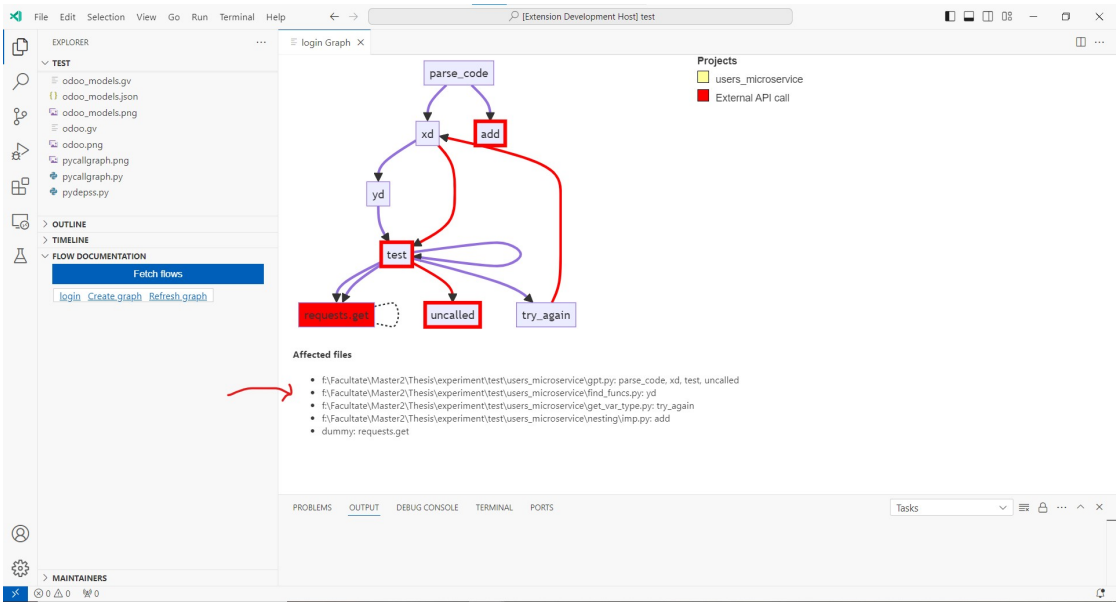


Figure 3.5: Using Affected Files to Identify the Scope of the Bug

Chapter 4

Solution

In this chapter, we will discuss our proposed solution to the issues, as described in Section 1.2. We will do so based on two uses cases which will each serve as an illustration of one of the features. We will provide two use cases as it allows us to better describe the features and showcase their usefulness. Even though the two features have stand-alone benefits it is important to note that they complement each other and can and be used together.

4.1 Concepts

In order to provide the reader with understandable use cases we first need to introduce some concepts.

API Endpoint or **Endpoint**: API endpoints are the point of contact where an application or microservice receives HTTP requests and returns responses.

Route: a line/piece of code which points to another service, typically an API endpoint. The route has the purpose of remotely executing a function or procedure (typically belonging to another microservice).

Business Flow or **Flow**: an action, which is composed of a sequence of physical operations, which have been abstracted and implemented with code. For example, thinking about a physical store, cashing out products at the counter can be split into the following operations: a person approaches the counter, identifying the owner of a cart, locating their cart, scanning the products in their cart, calculating the total sum of the products, and proposing a method of payment for those products. In the digital space, we can think of each of these steps as individual operations, and encode them into functions, all of which compose the `buy-cart` flow.

Route-to-Project map or **RTP map**: A map which holds a set of associations (i.e. records) between a route and the project/microservice that it is part of. The map also contains information about the functions defined in the project (with the goal that they can be uniquely identified by project and file), and information about the identified business flows.

Maintainer-to-Code map or **MTC map**: A map which holds a set of associations between the maintainers of a project and the folders, files or pieces of code that they maintain.

We identify use cases for two types of users which could benefit from our application: a developer being onboarded in a large company, and a developer working with a microservice codebase. The onboarding developers are interested in understanding the codebase, implementing new features, and fixing potential bugs in the system. The microservice developer is interested in comprehending the interaction between the microservices in the codebase, implementing features, or changing existing features.

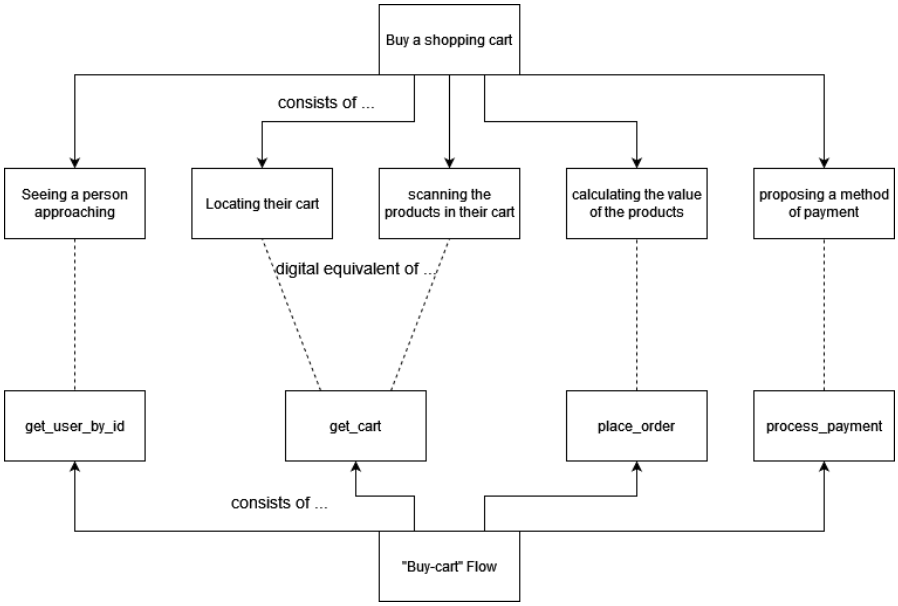


Figure 4.1: An example of a business flow

4.2 Onboarding Use Case

When working in a multi-team environment, some cross-cutting problems will arise, which means developers from different teams will need to collaborate. In a dynamic environment, it may not always be clear who is responsible for a certain project / piece of code. This is where our application can help, by offering a way to access this knowledge, directly from within the IDE. The developer can select a piece of code, right-click and select the "Find Maintainer" context menu action, which will trigger a search within the extension for any records that match the code, then return the maintainers, in the form of "Knowledge Cards" [12].

The "Find Maintainer" functionality gives the user the contact information of the code maintainer. This action is especially useful when integrated with the developer's daily activities. Let us assume that the developer is tasked with finding the root cause of a system problem. After starting a debugger and going through the lines of code, they notice a call to an external route in the code, which returns a faulty result.

After checking that the input for the call is correct but the returned result is false, they conclude that the issue is in the external service. By performing a "Find Maintainer" check, they notice that the service is maintained by another team, and their contact information. Now, they have an idea who to contact in order to either get more information about the expected behaviour or share the relevant traceback details, in order to delegate the bugfixing task to the appropriate team.

The maintainer-to-code map is able to display information that is either scraped from the project structure or an external service, such as Git, or manually created by a system administrator. Existing records can be edited or appended as needed, which encourages information reuse. The intermediate maintainer-to-code map should be stored in a readable format, and should be easy to share, version and synchronise. If users prefer to browse manually the existing map, which is displayed as "Knowledge Cards", they should have the freedom to do so.

Moreover, the newcomer can also focus on a specific line of code which contains an exter-

nal API call and either match it with the existing record or create a new association with a known point of contact. This new association can then be shared with colleagues to encourage information reuse and help repeated communication in the future.

4.3 Microservice Architecture Use Case

According to Bushong et al. [6], a developer working with microservices is interested in understanding the holistic overview of the system in order to modify it in the future. We provide a way for the developer to quickly see how microservices interact with each other and whether they are tightly coupled or not, thus identifying a potential future problem. The developer can define the area of code that they want to analyse or comprehend better, by adding a marker/comment at the root of the execution tree. The app then analyses the code starting from that marker, keeping track of function and module references and creates a graph structure, which displays the functions which would be called during the code execution.

While analysing the code, calls to internal routes are extracted and mapped to function definitions which can be explored recursively as nodes in the graph, while external routes are displayed as dummy nodes, with basic information attached. This graph is then simplified and displayed, where nodes in the graph indicate microservice functions and directed edges indicate calls from one function to the other. With this graphical view of the software, we argue that it is easier to gain a holistic view of the system and deduce which components are highly connected/coupled and require special attention in the future.

The graph can also be useful for the software developer as it allows them to get a better general picture of the code execution order. This is more effective when getting acquainted with an unfamiliar piece of code, which generally happens during onboarding, but can also happen when changing teams or simply working on a different part of the project.

4.4 FlowDocs Walkthrough

Following the onboarding use case described in Section 4.2, let us follow how a developer might use our tool and the “Maintainer Lookup” feature to connect with a more experienced developer within their company. The developer starts their day by noticing a new task in their pipeline. They are tasked to solve a bug concerning a wrong output of a function. While debugging to understand where the code logic goes wrong, they reach a folder called `stock_account`, and within the code they notice a comment: *“According to Belgian form 281.50”* (Figure 4.2). The developer does not have knowledge about this specific form, so they need to gather more information from the accounting department. To do so within VSCode, they right-click on the file which contains the code logic and select the ”Find Maintainer” action (Figure 4.3). They could also opt to right-click on a file in the explorer view to get a feel for who manages the entire `stock_account` folder, not specifically the file which contains the code. This opens the VSCode sidebar which contains an interactive view with three entries: `stock`, `account`, and `stock_account` (Figure 4.4). They figure out that someone who is knowledgeable about the domain must be either under the account or `stock_account` entries. They expand the `stock_account` entry (since the code logic is in the folder called `stock_account`), and find a list of code maintainers, then expands one of the maintainer headers to find the contact information of the product owner, which should know more information about the specific accounting rule (Figure 4.5). Finally, they can contact the product owner and receive relevant information about the code.

Following the microservice architecture use case described in Section 4.3, let us follow how a developer might use our tool and the “Flow Documentation” feature to understand the microser-

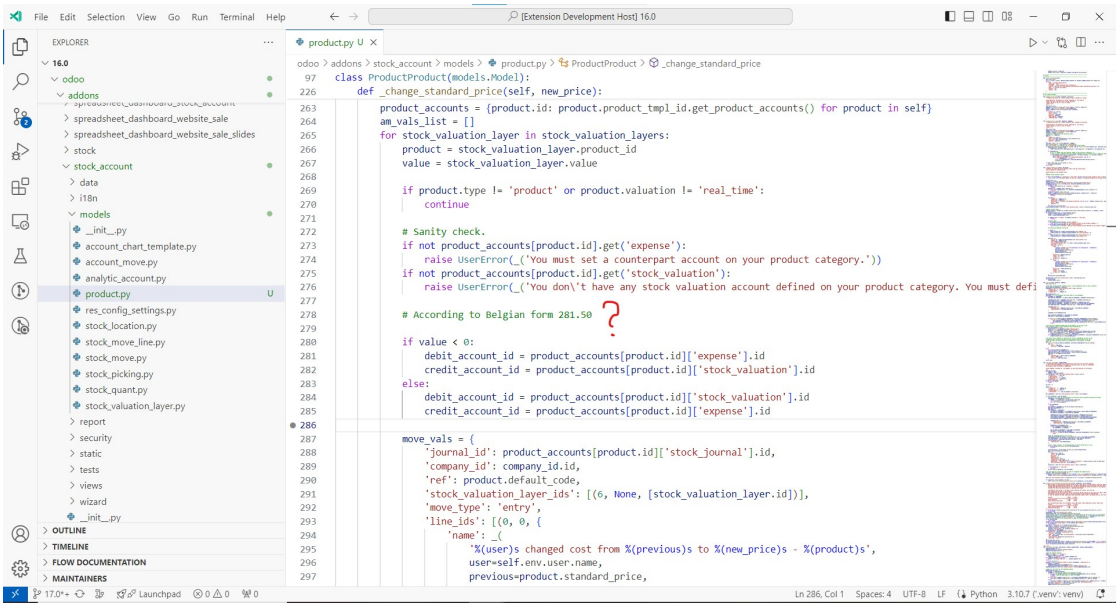


Figure 4.2: A situation where a developer needs additional information

vice codebase faster with the use of flow graphs. The developer starts their day by noticing a new task in their pipeline, which requires a code change in the payment section of the `buy_cart` business flow. Within VSCode, the developer opens the "Flow Documentation" sidebar which reveals a list of flow markers and their location in the code (Figure 4.6, (1)). They then click on the "Create Graph" button, which opens a new window which contains two main aspects. First a graph, where the nodes represent functions in the graph or calls to external APIs, and edges represent that the source node makes a call to the target node within the code (Figure 4.6, (2)). The `buy_cart` function is found at the top of the graph, as it represent the beginning of the business flow. The nodes are coloured according to the legend on the right (Figure 4.6, (3)), which means that a node's source code is located within the microservice with the same colour in the legend. Each microservice is given a different colour, according to the editor theme (light/dark) and the colours are maximally distinct. If a node makes multiple API calls, the corresponding nodes are ordered vertically, giving a sense of timeline progression or order within the code. The user can interact with the graph by either clicking on a node, which then opens the function definition in the code (Figure 4.7), or by clicking on an edge, which opens the line of code where the source node calls the target node (Figure 4.8). The user could also take the alternative approach of manually browsing through the code (or even starting a debugger session) until they reach an API call that they are not familiar with. Then, they could highlight that line of code, right click on it, then click "Highlight in Flow Graph", which switches to the graph view and temporarily animates the node containing the target API function code (Figure 4.9). If they choose to highlight a piece of code that is not tied to any API call, then the graph will highlight the function which contains that line of code. This will directly show them the function that they are interested in, and they can start modifying the code there. Alternatively, without browsing the code, the developer can just look at the graph, visually see what is the flow of information. As the graph in Figure 4.6 reveals, we sequentially make calls to the user, cart, order, and payment microservice. They remember that the flow concerns the payment microservice code, so they jump to those functions by clicking the payment nodes on the graph and continue

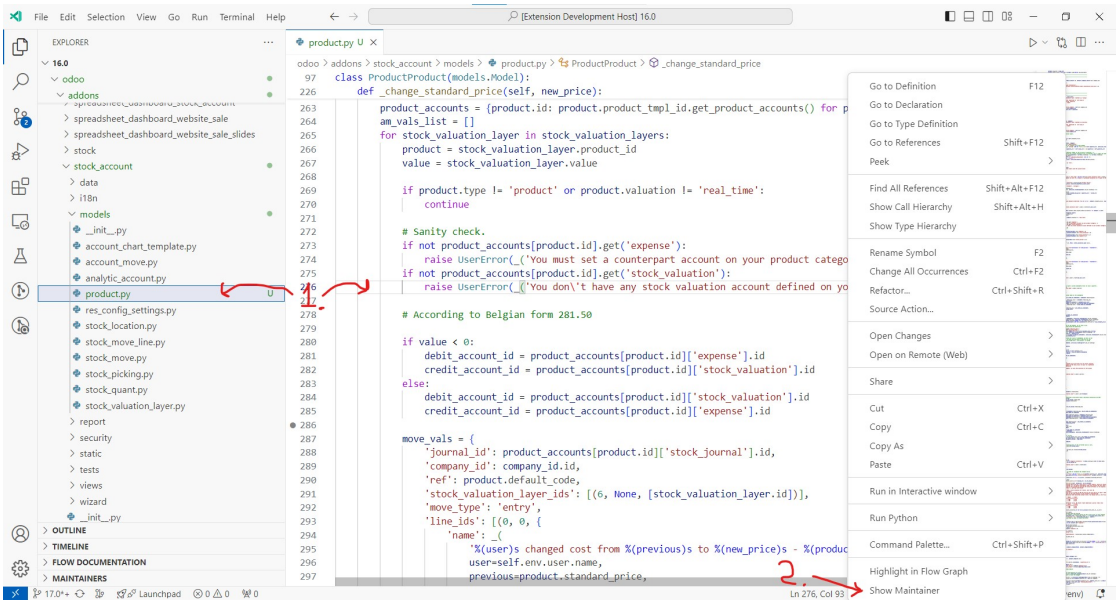


Figure 4.3: A developer right-clicks on a file (1) and clicks on "Show Maintainer"

exploring the codebase from there.

4.5 Extension Details

When the extension is loaded, it can already start analysing the current working folder for routes, in order to generate or update the RTP and MTC maps. After this, the user has the option to generate a workflow graph or highlight a piece of code and find its maintainer. Generating the flow graph does not update the mapping by default, using an internal cache instead. However, the user can opt to refresh the graph, which reanalyses the project for new routes and updates the mapping. The "Find Maintainer" action reads only from the MTC map.

Our extension uses graphs, but VSCode is not specifically designed for graph views, thus the navigation does not have instant feedback at times, especially when analysing large codebases. We believe that this is a necessary trade-off (as identified by Freire et al. [10]), as developers can see the graph without leaving the comfort of their IDE, rather than having an independent and more feature-rich program, like Hunter[9], Prophet[6] or SourceTrail[1].

The function call graphs generated by our extension outline the flow of information, by analysing calls to external services. They also provide a visual representation that fits the mental model of the developer by displaying a flowchart. The graphs offer the option of hiding functions which do not interact with other projects, in order to simplify the structure of the code and focus on the problematic sections which might generate a system fault.

In terms of scope, the application should be language agnostic, which means that it should support all major programming languages (C++, Java, JavaScript, Python, etc.). The route and flow extraction is easier to abstract through regular expressions, potentially just checking the file extension type. The maintainer-to-code map is stored as a JSON file, which makes it easy to edit, share, and store.

[1] https://github.com/CoatiSoftware/Sourcetrail

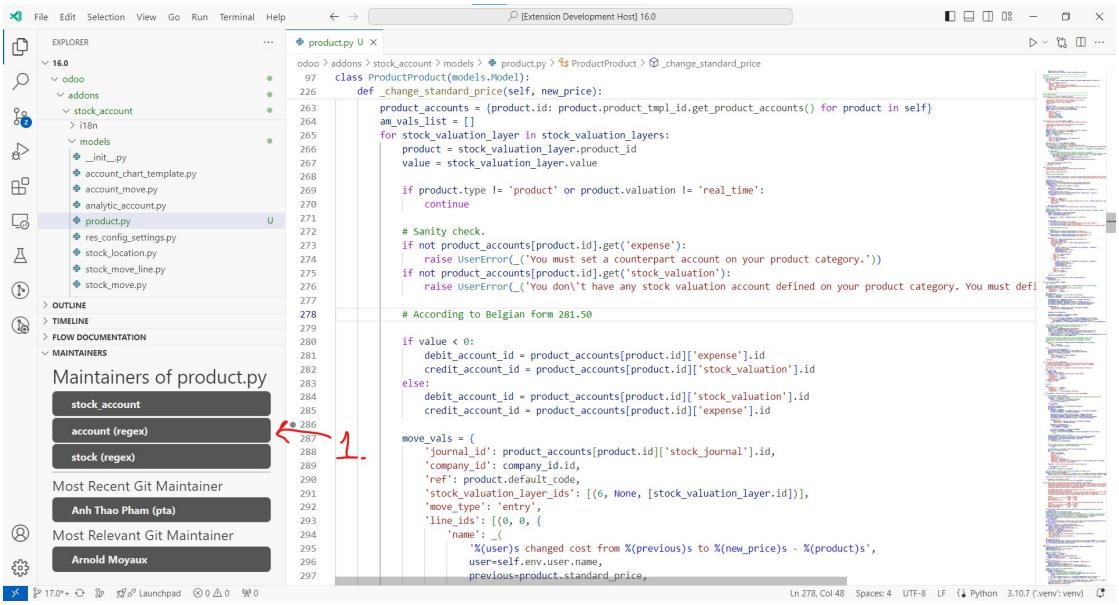


Figure 4.4: A list of maintainers

4.6 Additional Remarks

The application should be a helper for everyday tasks of both onboarding developers and microservice developers, and help them while they perform software comprehension and modification tasks.

The extension also analyses and documents API calls made to external services, such as those using the Python `requests` library. These calls are automatically linked with other local project components, enabling developers to discover potentially incorrect behaviour beyond their internal code. System administrators can manually associate calls with a human point of contact to facilitate future developer communication.

The "Flow Documentation" feature integrates with developers; daily workflows and provides a concrete solution to discovering and comprehending large Python codebases. The flow diagrams generated by the extension help developers comprehend the microservice architecture of their codebase faster and without needing to resort to external documentation. In this sense, the newcomer can generate a call graph and identify external calls to other project components while still being connected to their integrated development environment (IDE).

The "Maintainer Lookup" feature provides an integrated way of looking for code experts within the developer's IDE. It is a mode of articulating tacit knowledge (*"Who are the people responsible with a piece of code?"*) into explicit knowledge (a shared list of code maintainers). The feature provides a higher level of interactivity than a company spreadsheet or document, while still allowing the user to freely communicate with the code maintainer, if they choose so. We use the concept of "Knowledge Cards", introduced by Helic et al. [12] to display the maintainer information in a structured manner and to facilitate maintainer-to-maintainer relation exploration.

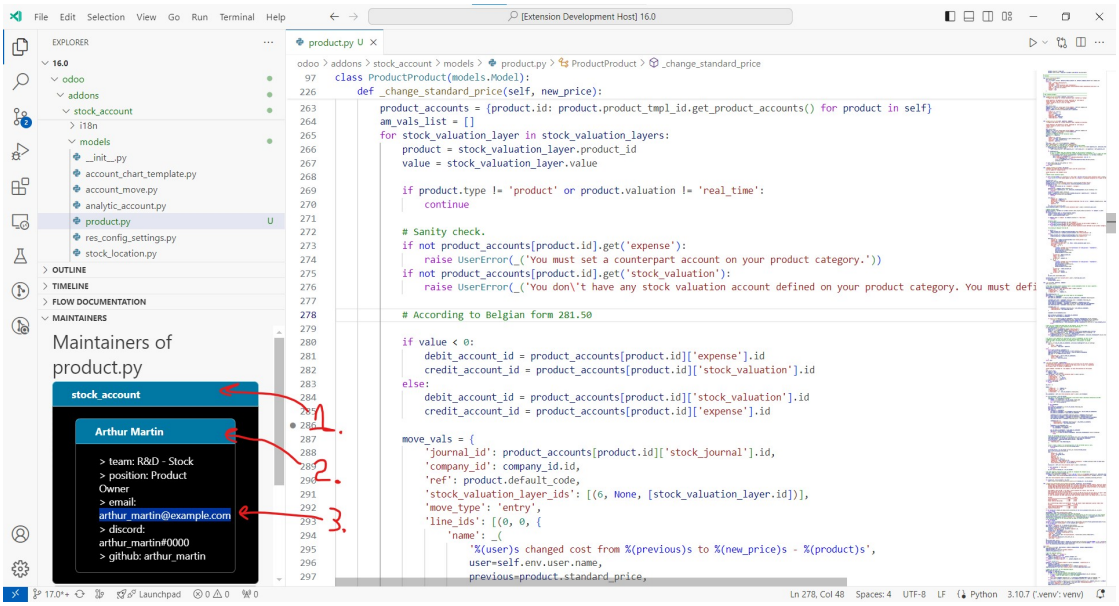


Figure 4.5: The detailed information of a maintainer

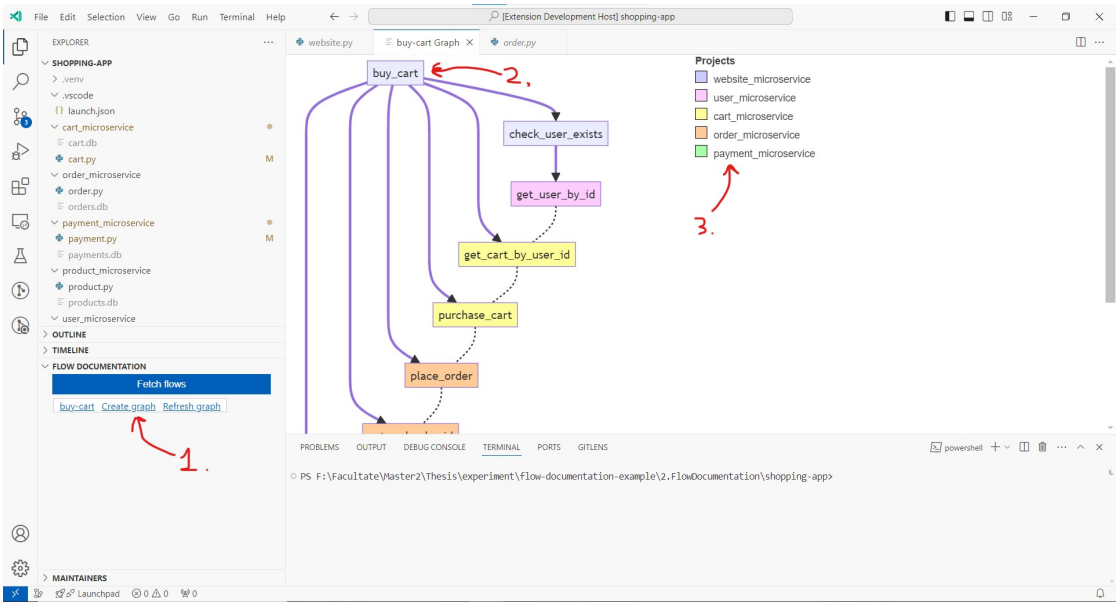
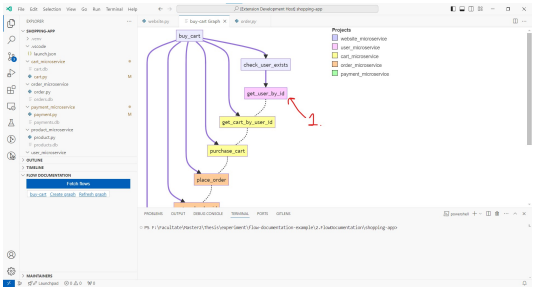
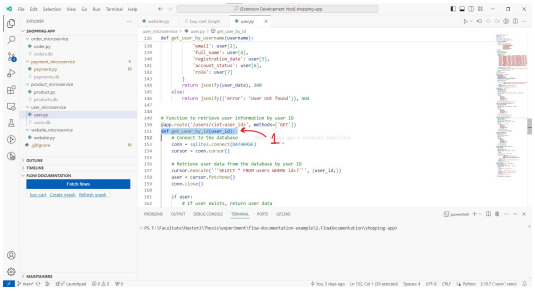


Figure 4.6: An example of a flow graph. (1): List of flow markers, (2): Flow Graph, (3): Flow Graph Legend

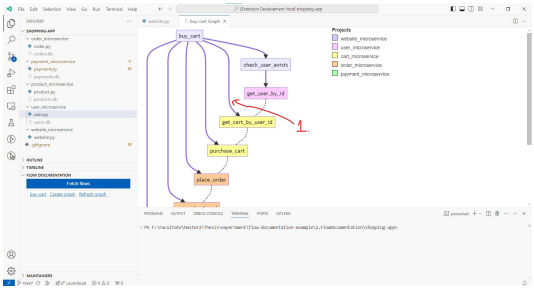


(a) User clicks on node (1) ...

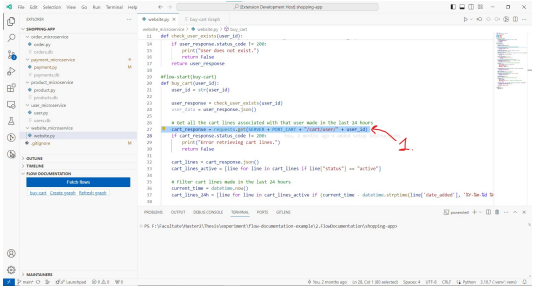


(b) ... and the function is revealed (1)

Figure 4.7

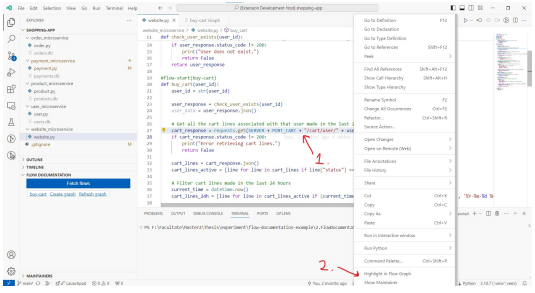


(a) User right-clicks on edge (1) ...

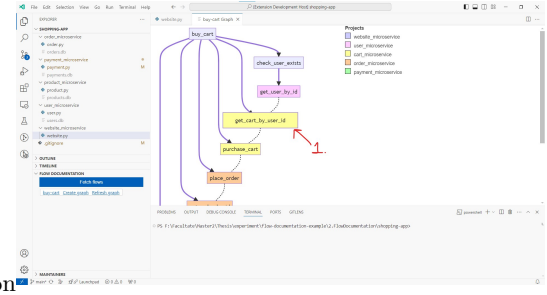


(b) ... and the call to target node is revealed (1)

Figure 4.8



(a) User right-clicks on code line (1), then clicks on "Highlight in Flow Graph" (2) ...



(b) ... and node in graph is highlighted (1)

Figure 4.9

Chapter 5

Implementation

In Chapter 4 we have presented a conceptual solution to the issues that we discovered and introduced in Chapter 2. Now, we will focus on the technical details of our VSCode extension.

First, because of time and resource constraints, we chose to focus on and provide support for the Python programming language. This means that currently, the static code analyser described in Section 5.4.2 will only recognise projects containing Python code. However, the extension is modular and allows for future extension in terms of supported programming languages.

Our extension started as one integral project which aimed to help developers which work with microservices better comprehend and navigate the codebase. Sometimes, developers do not have access to the entire codebase, as some services' source code might be only accessible to a specific team or department, mainly because of security reasons. While still using the extension, they could then connect with people who maintain the inaccessible code and either request explanations or report an endpoint issue to the concerned team. However, while developing, we encountered several issues with code compatibility, in the sense that working with microservice code was only suitable for a small number of projects, while looking up maintainers of code was a broad issue which impacted many large-scale projects. So, we have observed that we could reach our extension's goal more effectively by splitting it in two separate features. The features were introduced previously: "Flow Documentation" and "Maintainer Lookup".

5.1 Overview

In this section, we will describe the basic interaction between the FlowDocs components. FlowDocs is composed of two features, "Flow Documentation" and "Maintainer Lookup"(Figure 5.1). The "Flow Documentation" feature begins its interaction in the VSCode backend, where it parses the Python files in the open VSCode workspace, identifying routes, flows and functions, as described in Subsection 5.4.1. The identified flows are displayed as a list in the sidebar, where the user can decide to create a flow graph. If the user creates a flow graph, then a Python script is started, which recursively analyses the Python codebase and extract abstract syntax trees, starting from the base flow function (see Subsection 5.4.2). The AST are then compiled into an intermediate graph structure (see Subsection 5.4.4) and returned to the backend. Then, the flow is rendered into a flow graph by the VSCode frontend (see Subsection 5.4.5). The "Maintainer Lookup" feature works similarly. The VSCode backend requests the latest maintainer map JSON (see Subsection 5.5.1), which is parsed and aggregated based on maintained concept and maintainer name. The aggregated maintainer information is then sent to the VSCode frontend and rendered into a list (see Subsection 5.5.2).

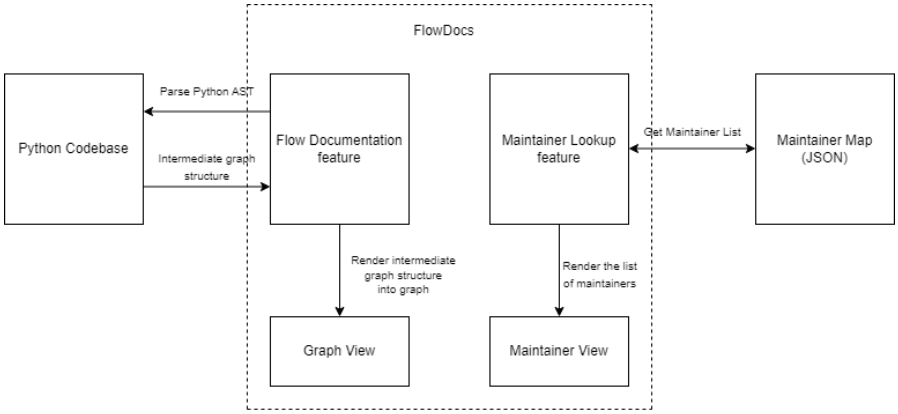


Figure 5.1: The FlowDocs Architecture

5.2 The VSCode API

Visual Studio Code (VSCode) is an integrated development environment (IDE) developed by Microsoft. VSCode is a popular tool used by many developers worldwide, with an active contributor community. The Visual Studio Marketplace contains user created VSCode extensions and plugins which enhance the developer's experience while coding and provide integration with other services and products, such as GitHub. We have chosen to implement our app as a VSCode extension because of its popularity and extension support.

The extension makes use of the VSCode Application Programming Interface (API), which is a set of tools and functionalities provided by Microsoft to allow developers to extend and customise the VSCode editor. The extension uses a NodeJS back-end to handle the intensive computations, and for creating and displaying panels with which the user will interact. The front-end uses the VSCode Webview API to display the graph in an interactive and responsive manner to the user. For example, the user can click a node in the graph to open the function definition in the code. The VSCode API implementation allows for asynchronous communication between the back-end and the front-end through the `postMessage` and `onDidReceiveMessage` functions.

5.3 VSCode Manifest

VSCode extensions must have a manifest file called `package.json` at the root of the extension, which contains details about the extension, as well as the default configuration for the extension's settings (see Listing 1). Some notable settings include the `activationEvents` field, which triggers the initial computation of the route-to-project map (RTP), when there exists at least one Python file in the workspace, and the `commands` field, which defines globally available functions exported by our extension. This includes the "Create Graphs" command, which is a one-click solution for generating the graphs of all flows found in the project, and the "Find Maintainer" command, which allows for contextual search of the maintainer-to-code (MTC) map. Extra configuration settings include the MTC map location and extra workspace locations on the user's device, which should always be scanned for changes.

```
{
  "activationEvents": [
    "workspaceContains:**/*.py"
  ],
  "contributes": {
    "views": {
      "explorer": [
        {
          "type": "webview",
          "id": "flow-documentation.flowsView",
          "name": "Flow Documentation"
        }
      ]
    },
    "commands": [
      {
        "command": "flow-documentation.createGraphs",
        "title": "Create Graphs"
      }
    ],
    "configuration": {
      "title": "FlowDoc",
      "properties": {
        "flow-documentation.extraFolders": {
          "type": "array",
          "default": [".", "C://Code/demo"],
          "description": "List of extra folders to scan."
        }
      }
    }
  }
}
```

Listing 1: Extract of the manifest file

5.4 Flow Documentation Feature

In order to extract and display a flow graph to the user, the extension needs to apply the following steps:

- Extract routes, functions, and flow markers from the source files
- Extract the abstract syntax tree (AST) of the Python source code
- Recursively analyse the AST
- Identify and extract routes
- Convert the AST into a graph-like structure
- Interpret the graph-like structure and display it to the user

In the following sections, we will detail each step individually.

5.4.1 Extracting Routes, Functions and Flow Markers

When opening the VSCode editor, our extension scans through the open workspaces for specific keywords, which are then stored in the RTP map. Each keyword consists of a regular expression, which is used to either identify a route, a python function or a flow marker. After the scan is complete, every identified object is stored in a global context object, provided by the VSCode API. This creates a cache which persists through multiple sessions, in order to limit the resource consumption of this process when it is repeatedly applied to large codebases. This step is located in the NodeJS back-end using TypeScript.

5.4.2 Recursive AST Code Retriever

After refreshing the RTP map, our extension launches a Python program with the flow name and the RTP map as parameters, which will parse the Python code's Abstract Syntax Tree (AST). This can be achieved using Python's internal meta-analysis library `ast`. The `ast.parse` function takes as input the syntactically correct Python code and returns the corresponding abstract syntax tree. We also make use of the `ast.NodeVisitor` class, which allows us to visit an AST and add custom logic to nodes (see Listing 2). In order to create a dependency map of the flow, first we specify a starting point for the AST, which is usually a function name, and then we recursively visit the tree, keeping track of imported modules, functions, and API calls (for future extraction and parsing).

The final AST goes through a final processing step which serves the purpose of simplifying AST trees by removing nodes that do not contain function calls or references. The `SimplifyAST` class inherits from `ast.NodeVisitor`, takes an AST as input, and returns a smaller AST, which only contains nodes relevant to the graph. After executing some performance analysis, we have concluded that this module does not greatly impact the graph generation time, so at this point in the project it does not bring any added value. Perhaps this will change when creating the graph of more intricate projects.

5.4.3 Route Identification and Extraction

While visiting the function AST in the previous step, we keep track of AST nodes which contain routes. We took a heuristic approach to identifying the nodes, by selecting a popular Python library which creates and manages HTTP requests, namely the `requests` library. The route nodes are marked with an additional indicator, which will be useful in the final step when we match the routes to the functions before displaying the graph.

5.4.4 AST to Graph Conversion

After the AST is extracted and simplified, it is converted to an intermediate graph-like structure that will be parsed by the extension front-end in the next step. Listing 3 shows an example of such a graph representation. The nodes contain information about the represented function name and other meta information of the function, including file path, module, line number, file name, etc. The edges, in addition to the start and end node, contain information about which is the code line number which contains the function call.

The example graph of Listing 3 will contain one node called `parse_code`, and one directed edge, which goes from `parse_code` to itself. The intermediate graph structure is then returned from the Python program and to the NodeJS back-end.

```
class ImportVisitor(ast.NodeVisitor):
  # visits the entire file AST

  def visit_Import(self, node):
    self.update_imported_modules(node)

  def visit_FunctionDef(self, node):
    fw = FunctionVisitor()
    new_imports, new_functions = fw.visit(node)

    for import in new_imports:
      iv = ImportVisitor()
      newer_imports, newer_functions = iv.visit()
      self.update_imported_modules(newer_imports)

class FunctionVisitor(ast.NodeVisitor):
  # visits the actual function AST

  def visit_Call(self, node):
    # ...

walker = ImportVisitor()
all_imports, all_functions = walker.visit(tree)
```

Listing 2: Simplified snippet of the recursive AST generation function

5.4.5 Graph Interpretation and Display

The app front-end makes use of the VSCode Webview API, which allows extensions to create fully customisable views within Visual Studio Code. There are other view types, such as the ListView and the Treeview, which are more optimised for performance, but do not suit our user interaction needs, as they cannot display graphs. A Webview can render almost any HTML content in its frame and communicates asynchronously with the extension using messages.

The graph viewer is a Webview which contains the graphical representation of a business flow. The nodes in the graph represent functions in a microservice, and the directed edges connect the nodes where the start node calls the end node in the code. We can consider that the start function in the flow represents the "root" node in the graph, but it is important to note that the graph is not a tree since it can contain cycles (e.g. generated by recursive functions). The back-end converts our intermediate graph structure into MermaidJS style notation, which is then parsed by the Mermaid library[1], and inserted into the HTML code.

While implementing, we have encountered several situations in which we have to first generate and render the Mermaid graph, then target specific nodes and edges to add custom behaviour to them. Currently, this is a working approach, but if the custom features increase in complexity in the future, we will consider using a more powerful graphing library, such as `d3.js`[2], which allows the injecting of custom behaviour in the graph before rendering it.

[1] https://mermaid.js.org/
[2] https://d3js.org/

```
{
  "graph": {
    "nodes": {
      "function_1": {
        "module": "gpt",
        "file": "C:\\Code\\main.py",
        "func_name": "parse_code",
        "id": 0,
        "lineno": 47,
      },
      ...
    },
    "edges": [
      {
        "start_node": "parse_code",
        "end_node": "parse_code",
        "call_lines": [
          56,
        ],
      },
    ],
  },
}
```

Listing 3: Intermediate Graph Representation

The Flow Viewer

The flow viewer is a panel displayed on the sidebar of VSCode. This is where developers can see the flows present in the project. It contains a button which retrieves all the flow marker locations in the open workspaces and displays them in a list view. Once the flows are retrieved, the user can choose to either generate or refresh the flow's graph representation (which triggers an RTP map recalculation) and view it in the graph viewer.

5.5 Maintainer Lookup Feature

5.5.1 The Maintainer-to-Code Map

In order for users to create and share maintainer information, visualised as "Knowledge cards", they must be stored consistently. We chose to create a `JSON` mapping, as it is easy to share across multiple users. As an alternative, we could have used a database, which would satisfy the ACID principles, but we argue that this method introduces a large overhead and provides too little benefit for our current purposes.

Listing 4 shows a sample record in the MTC map. The record consists of a `contact` object, which contains the contact details of a maintainer, and the `maintains` object, which contains a list of all relative paths in a project that are maintained. The MTC map can be easily shared between users, by exporting and importing a JSON file, through the extension's settings page. The map can be created and maintained by a system administrator and then shared to the other

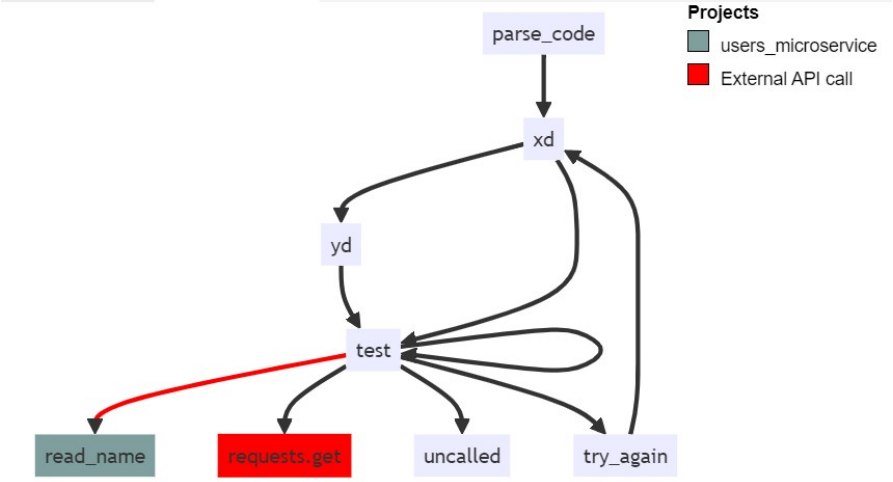


Figure 5.2: A Flow Graph Example

Figure 5.3: The Flow Viewer Sidebar

employees in the company.

5.5.2 The Maintainer Viewer

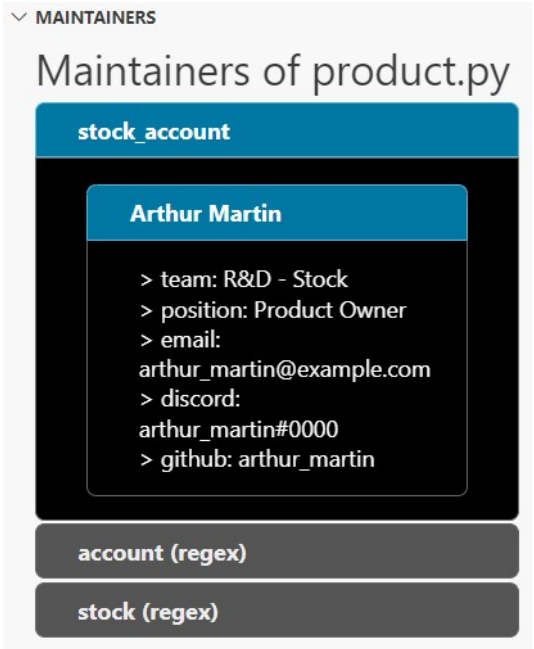
When initialising the extension, it searches for `maintainerMap.json` file in the user's workspace or project, which contains the MTC map. The MTC map could be uploaded to the company's file sharing software, from which the developers can download it and use it locally. If the map is found, then a sidebar is displayed, which contains the list of the most active contributors to the codebase. This is the default view, which can already show the user who is the contributor who might know most about the codebase. A high number of contributions does not necessarily correlate with overall codebase knowledge, but is a good approximation.

The user can modify the information displayed by using the "Find Maintainer" action on a file or folder. Then, the maintainer viewer displays all maintainers of the user-selected code. The way in which the MTC map is interpreted allows for partial path matching and regex matching, which means that the user will see maintainers not only of the selected file, but also the parent folders, or regex matched paths. So, the maintainer view can show multiple related maintainers, in order of relevance.

```
{
  "contact": {
    "name": "Antoine Dubois",
    "team": "R&D - Accounting",
    "position": "Developer",
    "email": "antoine_dubois@example.com",
    "discord": "antoine_dubois#0000",
    "github": "antoine_dubois"
  },
  "maintains": [
    {
      "type": "folder",
      "regex": true,
      "path": "account"
    },
    {
      "type": "file",
      "path": "accounting/models/invoice.py"
    }
  ]
},
```

Listing 4: Maintainer-to-Code map record example

(a) The Default Maintainer Viewer Sidebar



(b) The Maintainer Viewer Sidebar looking at a specific file

Figure 5.4: Different states of the Maintainer Viewer Sidebar

Chapter 6

Evaluation

6.1 Experiment Design

Our user study follows the norm of evaluating code visualisation software, which is usually conducted with a user study (Telea et al. [28], Dias et al. [9], Bogner et al. [4]) or a case study (Daniel et al. [8], Lungu et al. [15] and Cerny et al. [7]).

We have adapted the experiment design of Dias et al. [9], which is similar to our use case, in the sense that both projects implement a visualisation tool for source code with the aim of easing software comprehension for developers. The tools are tested through a user study in terms of both design quality and developer efficiency. One main difference is that Hunter is a standalone application, whereas our tool is an extension for VSCode.

Merino et al. [18] organised a systematic review of the literature on software visualisation methodologies and created a framework to define the scope of experiments in software visualisation. They describe this framework as *"a starting point for researchers who are planning to evaluate a software visualisation approach"*. We can describe the scope of our experiment through the framework proposed by Merino et al. [18] as such:

"Analyse <the FlowDocs visualisation tool> executing in a <machine using VSCode> to support <code comprehension and modification tasks> using a <node-link diagram>, as well as <code maintainer location tasks> using a <list view> displayed on a <standard computer screen> for the purpose of <comparing functionalities available in Visual Studio Code> with respect to <effectiveness> in terms of <user performance> and <user experience> from the point of view of <software developers>."

We have selected software developers who are familiar with VSCode to participate in our experiment. Each developer was tasked with solving a set of tasks using VSCode (and additionally their installed extensions), and VSCode with the FlowDocs extension. Within our experiment, the two features of our extension, namely "Flow Documentation" and "Maintainer Lookup", were tested separately by our participants. Every participant was able to select which feature they wanted to experiment with.

6.2 Participants

To start recruiting developers for our interview, we shared a link to a Google Forms survey (see Section 6.3) to see who might be interested in participating in online interviews. Every person who was interested in "FlowDocs" and met the prerequisites was invited to take an interview to

further discuss it. We share this link with three main groups:

Odoo employees: As a former Odoo employee, I have faced the issue of not knowing who is the maintainer of a certain file multiple times, so we assume that other employees would be a good fit for testing the "Maintainer Lookup" feature.

Vrije Universiteit Brussel (VUB) students: We also posted the form link on the VUB Discord server to involve students which are following the Bachelor or Master in Applied Sciences and Engineering: Computer Science programme. There, we should find software developers with different cultures and ideas, so the group of participants is more diverse.

Social media: Besides reaching out to specific groups, we also shared the survey link on the social media platform, Reddit. The survey was shared on different programming related subreddits, such as r/surveyexchange, r/compsci, r/opensource etc., with the purpose of exposing "FlowDocs" to the community. The form stayed open for about two weeks, until we gathered enough respondents. In this way, we hoped to attract an even more diverse group of software developers, outside the Belgian community.

6.3 Preliminary Form

A preliminary Google Form was used to select participants. The first set of questions aims to select developers who are comfortable using VSCode and have some work experience. The questions (*"Have you worked for a company where the codebase was structured using a microservice architecture?"* and *"What is the largest size of a software development team (or department) you have been a part of across all your previous roles?"*) played the role of a heuristic to estimate which people might be interested in which feature, although they did not fully determine our selection process.

The following two sets of questions describe one feature each. The set starts with a small description of the feature and its intended use case, followed by a video which describes a possible scenario where this feature might be useful, ending with a mix of Likert scale questions about the first impression on the usability of the tool and long answer questions about general impressions. Each set contained the following targeted questions:

- *"How frequently have you encountered the same issue described in the video?"*: aims to answer RQ1 if participants encounter these issues in their day-to-day work routine. The question is graded on a Likert scale, and answer values 3-5 suggest that participants face this issue regularly.
- *"Do you think you might like to use Flow Documentation in the future?"*: aims to answer RQ2 and if participants believe that FlowDocs improves their workflow. The question is graded on a Likert scale, and answer values 3-5 suggest that participants find our tool useful.

The form conclusion ends with an invitation for the participants to test the extension by themselves in the extension's GitHub repository[1] and join us for an interview where we can talk more in-depth about how they use our extension and answer RQ3.

6.4 Interview Design

After selecting developers who have completed the form and are also familiar with VSCode and visualisations, we have invited them to an interview in which we aim to answer RQ3.

[1] https://github.com/blox-dev/flow-documentation-example

After confirming the participant's consent, we recorded video and audio from interviewees' laptops for analysis and timing measurements. In the case where they did not consent to having the conversation recorded, we noted task times, main ideas and suggestions in a text file. Each task was read aloud by an experimenter. The participants answered verbally and had unlimited time to complete tasks, with the option to ask questions at any point. During the interview, we validate RQ3 by presenting the users with two different kinds of tasks.

In the first task the user will have to locate a function which is supposed to be modified in the provided sample code, either with or without using the "Flow Documentation" feature. We are interested to see how does a user navigate a microservice-based architecture and how the "Flow Documentation" feature helps them navigate the code (RQ3).

The second task is a maintainer finding task, where the user will have to find the person responsible for a certain file or folder, either with or without using the "Maintainer Lookup" feature. A maintainer spreadsheet is provided to simulate a company communication resource. Similarly, our goal is to see how a user navigates this task in general and whether using the extension helps them perform the task faster and more accurately (RQ3).

With both tasks, we measure the time that it takes for participants to complete them and the accuracy of their answers. Additionally, we test our assumption that developers are actually facing this issue regularly (RQ1) by the way in which they solve the issue without using our extension. After solving the tasks, we have an open set of questions to see how the participants feel about using FlowDocs. Here is the set of qualitative questions that we used:

- Do you think that the tasks we presented are representative of the problems that you encounter at your job? Why / Why not?
- Do you think that browsing the flow graph is easy?
- Do you think that finding maintainer information is easy?
- Did using either feature feel natural to you or would you go about solving the task differently?
- Do you think that you had enough information to finish the task? If not, what other information would you require?
- Did you feel that looking for information was easy?
- Did you ever feel confused while using FlowDocs?
- Did you find it more enjoyable/easier to solve the tasks using FlowDocs?

6.5 Methodology

For each of the two features, we have two exercises from which we randomly select one exercise to be accomplished with plain VSCode (VSC) and another exercise with VSCode and the feature. We split the participants in 4 groups, using the "counterbalanced within-subjects design" depending on the order of the tasks and the order of medium (plain VSC or VSC with FlowDocs). We will discuss the advantages of counterbalanced within-subjects design before justifying our decision of experiment design.

- Maintainer Lookup (ML) first, then Flow Documentation (FD)
  - VSC, VSC+ML, VSC, VSC+FD

  - VSC+ML, VSC, VSC+FD, VSC

- Flow Documentation (FD) first, then Maintainer Lookup (ML)
  - VSC, VSC+FD, VSC, VSC+ML
  - VSC+FD, VSC, VSC+ML, VSC

Since the same participants are exposed to both conditions (feature and plain VSCode), personal variables such as skill level, familiarity with VSCode, and problem-solving abilities are controlled. This means any observed differences in performance or experience are more likely to be due to the features themselves rather than individual differences between participants. Participants are directly comparing their experiences with and without the features, meaning that the collected data shows how each feature impacts their task-solving process.

By changing the order of participants using plain VSC and VSC with a feature, we address potential order effects that can bias the results. For example, the learning effect happens where participants improve their performance because they become more familiar with the tasks through repeated practice. If participants use plain VSCode first, they might perform better when they later use VSCode with the feature, not necessarily because the feature is more effective, but because they have become more skilled at the tasks through practice. Fatigue effects represent another potential order effect. Participants may become fatigued or lose concentration as they progress through tasks.

To address these order effects, we use counterbalancing. This involves varying the order in which participants accomplish the tasks. By splitting participants into two groups, with one group using plain VSCode first and the other using the feature first, learning and fatigue effects are balanced out. Additionally, we consider the implications of the fatigue effect by splitting the groups in two once again, the first group will try the "Flow Documentation" feature first, and the second group will try the "Maintainer Lookup" feature first.

We have considered the use of other experiment designs, such as "between-subjects design" and "pairwise design", before settling on the current approach. The between-subjects design does not require us to take into account the previously mentioned order effects. We could have used this design by splitting the participants in 3 groups: plain VSCode, VSCode with "Maintainer Lookup" and VSCode with "Flow Documentation". Each group will be given the same set of tasks to perform, without the need for task counterbalancing. These benefits also come with a couple of disadvantages. Firstly, individual differences between participants in different groups can introduce variability, making it harder to detect the true effect of the features. Also, each condition requires a separate group of participants, which means that we would need many more answers participants to draw a meaningful conclusion. Given that we would not be able to collect many interviews within the given time frame, we decided to not use the between-subjects design.

Pairwise design implies participants being matched in pairs based on relevant characteristics. Participants are matched and then randomly assigned to different conditions, meaning comparisons are made between different but matched individuals. This approach can control for some individual differences but not as effectively as comparing the same individuals across conditions. Although participants are matched on key characteristics, there may still be unmeasured differences that can introduce variability and affect the results.

6.6 Threats to Validity

Odoo employees may be biased when using the "Maintainer Lookup" feature because the task involves using the Odoo codebase. However, since the feature has a broad enough developer

reach, it can be implemented in other projects without much hassle, and thus we believe that the tasks represent the broader industry standard.

Odoo employees already know some of the people who manage different parts of the software. We managed this issue by finding a part of code of which they were not sure about. The Odoo codebase is too large for anyone to memorise. Additionally, this is information that an employee needs from time to time, so it is unlikely that they remember every maintainer.

The project for the "Flow Documentation" feature only contains 5 microservices and does not reflect industry standards for a microservice codebase. This issue was not mitigated in the current iteration of the experiment. In Section 6.9, we will describe how one of the participants has changed their behaviour because of the small size of the project, and how we plan to address it in the next iteration.

In an attempt to reach a representative sample of real world developers we selected developers from different Reddit communities, which should have different backgrounds and experiences. Participants may also not have the same degree of familiarity with VSCode and the extension. We solved this by selecting users who are already adequately familiar with VSCode from the preliminary form.

To ensure valid results from the interviews, we guarantee that the interviewees' answers will be anonymous and confidential. We were available to clarify any misunderstandings during interviews to ensure clarity. The question structure was reviewed before conducting the interviews, and participants had some experience with either large codebases or microservice architecture.

6.7 Preliminary Form Results

The preliminary form described in Section 6.3 has received 43 responses, one of which was removed because of the respondent not having any programming experience. The respondents occupy different positions in the software development industry, with an average of 6.1 years of experience in the field (median 6). 25 out of 42 participants reported a score of familiarity with VSCode of 3 or higher (mean 2.85, median 3), while 21 said that the use data visualisations while coding (mean 2.42, median 2.5). 88.1% of developers (37/42) have previously worked with a microservice architecture, while 78.6% have worked in a company containing 250 or more software developers.

We will interpret the results of the two separate features, where all scaled questions are rated on a scale from 1 to 5 (mean 3). Participants who responded to the qualitative questions with a score of 4 or 5 will be considered as having a positive experience. We will not include 3 as a positive score, as that indicates neutrality.

Regarding the "Flow Documentation" feature, 36 out of 42 participants encountered a similar scenario as described in our microservice architecture use case (Section 4.3). 15 out of 42 participants rated the intuitiveness and user-friendliness of the feature with a score of 4 or above (mean 2.95, median 3), and similarly, 15 participants thought that the feature was easy to use (mean 3.11, median 3). Finally, 19 participants said that they might like to use "Flow Documentation" in the future (mean 3.28, median 3).

Regarding the "Maintainer Lookup" feature, 28 out of 42 participants encountered a similar scenario as described in our onboarding use case (Section 4.2). 13 out of 42 participants rated the intuitiveness and user-friendliness of the feature with a score of 4 or above (mean 3.16, median 3), and similarly, 14 participants thought that the feature was easy to use (mean 3.09, median 3). Finally, 21 participants said that they might like to use "Flow Documentation" in the future (mean 3.30, median 3.5). Out of the current 43 preliminary form respondents, four have booked an appointment for the qualitative interview.

6.8 Qualitative Interviews

Following from the qualitative interviews, we can determine two main approaches that developers take to locate the source of an issue in the codebase: **debugging**, and **lookup techniques**.

Debugging means that a developer locates the initial function or flow that produces a bug, creates a breakpoint at the beginning of the function, starts a debugging session, then follows the code line-by-line, checking after every line that the result they expect from executing the code is the same as the result that they get. Whenever the result differs, then they place a breakpoint at that line, and then recursively debug it. This technique is more time-consuming, but it makes sure that the developer does not miss any potential interaction or bug source in the codebase, as every line of code is checked.

Alternatively, a developer can also follow a **lookup** strategy to identify the source of an issue. This means that the developers locates the initial function or flow that produces a bug, then tries to find relevant variables or function names which are called within the flow. They reason about the which variables or functions are probably responsible for the bug, then place a breakpoint in different "hotspots", such as: the variable definition, the line of code where a variable is modified, the function definition, the line of code where the function is called, the line of code where a function result is returned, etc. Then, they start a debugging session, skip through code which they did not consider a hotspot, and only stop at the designated hotspots, where they similarly reason whether the result they expect at the time of execution is the result they get. This technique is less time-consuming than traditional debugging, as the developer does not have to check every line of code, but sometimes it misses the correct bug source.

Developers tend to employ a mix of the debugging and lookup methods. For example, they might start with the lookup technique and look for hotspots, but then continue debugging for a couple of lines of code after it, to get a more solid grip on how variables change over time, before skipping to the next hotspot. Similarly, a developer who starts traditional debugging, might lose interest in the current bit of code that they are following, so they skip ahead and add a breakpoint to the next hotspot, from which they resume the debugging workflow.

We will now describe the individual interviews and how the participants reacted to the tasks and our extension. We will interpret the qualitative feedback and improvement opportunities in Section 6.10. We will describe the interviews one by one. In order to remove duplicate information, if two participants gave the same piece of feedback, a small mark will be added after describing the feedback, as such: (+ P2, P3). If this particular annotation is present after a piece of feedback from participant 1, it means that participant 2 and participant 3 gave the same feedback as participant 1. All participants have completed the preliminary form and tested both the "Flow Documentation" and the "Maintainer Lookup" features. The makeup of the individual interview sections will be as follows. First, we present relevant background information collected about the participant, then we present relevant observations made during task execution, followed by remarks and feedback given by the participant. Last, we present the relative performance increase or decrease in completing the given task with or without the use of the extension. We repeat this structure for both the Flow Documentation and Maintainer Lookup features, using the same order in each interview.

Though P1 does not work with microservices directly, they have worked with database integration before. In their work environment calls are made directly from the application to different databases, after which the results are combined and used within the application.

P1 does not normally use a debugging feature in their regular workflow. They attributed this to the fact that their technology stack has limited debugging possibilities. Their usual approach to locating the piece of code containing the bug is the following: They separate the code in chunks after which they add a log statement in each chunk. After running the code, they can

and see how the state of the relevant variables is changing. If all the values are as expected they conclude that the faulty behaviour happens later in the code. They then follow the execution flow until the bug is found. P1 also remarks that they do not use VSCode in their day-to-day job environment.

When tasked with the function identification task, they did not approach the problem through a debugging perspective, line by line, but rather started locating variables useful to the task, looking for where they are defined, modified, and used (+ P2, P3). They found the variable relevant to the task in the main file, but expressed their confusion as they could not find where the variable was being overwritten. Later in the interview they said that this was the moment when they realised that a microservice might be involved. P1 then opened the graph without looking further into the main file. First, they clicked on the node containing a main file function. This led to some confusion as they expected to be directed to the microservice file and not the main file. After returning to the graph they clicked on the correct node containing the function within the microservice file. After which they located the problem and implement the correct logic to solve the problem.

When asked about the experience of using the Flow Documentation feature they said that with the graph, *"I skipped the process of having to find the bug location [...], and thinking more on a concept-scale [...], I knew my problem was about users, so looking at the node names I instantly saw where the problem could be located"*, as well as *"It really helps me locate the issue"*. P1 explains that before we do debugging, we think about code in a conceptual way and that this can be improved through an overview of the code.

When asked about how they experienced navigating through the graph they said that browsing the flow graph is easy, but they would like to see the whole graph at once on the screen. Related to this P1 expressed that they expected the graph to be zoomable, instead of the window scrolling down (+ P2). P1 also said they would like to see which is the structure of data objects being returned from API calls and passed to the next one. Furthermore, when looking back at the graph they remarked that they did not see the legend on the right (+ P2). The fact that API calls happen sequentially in the graph was also not immediately obvious to them (+ P4). P1 also said that they forgot about the sidebar tabs, which is one of the VSCode features. As a closing remark, P1 said they were *"amazed at the speed of which the file is opened when clicking on a node"*. P1 completed the task 79.5% **faster** when using Flow Documentation compared to when they were not using the feature.

Within the company P1 works at, the developers mainly handle code maintaining issues through their project manager (PM). These project managers usually know who is writing what code, so they can connect the developers involved with each other. P1 then told an anecdote in which they were contacted by a PM about a specific piece of code, which was not written by them. The confusion arose because two developers had worked on the same file and the PM did not remember which developer wrote which specific code section.

Within P1's current team they usually know who to contact because of its small size. P1's team is the only development team in the company, and consists out of 5 developers. Even though a convention exists where the developer who creates a script, adds their name at the top, P1 still feels they have to mentally keep track of who is responsible for a specific piece of code, saying that: *"I have to rely on my knowledge, remembering who did what"*. P1 is more interested in finding the person who is most knowledgeable about a certain concept than the person who is the responsible for a piece of code within the company structure. They argue that project managers and product owner, even though they overlook a lot of code, might not be the most knowledgeable of a concept they are faced with in their programming task.

When opening the maintainer record during the task P1 received two matches, one being 'account' and the other being '/security'. They assumed person listed under 'account' occupied

a higher position. When asked why, they pointed out that there were many folders in the main structure of the project whose names start with 'account'. They assumed that the person listed under '/security' was a developer maintaining this specific folder. Additionally, they point out that 'security' was the name of the direct parent folder of the file that was being inspected. P1 said they would choose to contact the person listed under '/security' as they seemed more directly involved with the code. P1 also added that seeing the name of the maintainer in the title card instead of the GitHub username made them perceive the maintainer as more approachable, which according to themselves, was another factor in P1's decision. They also noted that they were confused by the word "regex" next to the matched routes (+ P2, P4). They said that they would like to have more clarification on what the keyword actually means. They exemplify, if the keyword would be 'account', does this refer to a concept (within the business and programming logic), a folder (with in the project file structure) or a department (within the company).

P1 did not immediately notice the section of the knowledge card containing the information of git maintainers. They said that the git maintainer section removes their concern of information staleness. They mentioned that the displayed git information was overwhelming to them. The commit hash made them feel that the rest of the information displayed was not worth reading. The participant remarked that they were confused about the meaning of the 'account' keyword, and added that because of the reduced size of the explorer they had to scroll up before being able to see the root folder, which was called 'account'. They remarked that the sidebar felt "a bit crowed" when actively using both the maintainer tab and file explorer (+ P2, P3, P4).

P1 closed the maintainer tab and clicked on the "show maintainer" context menu item, which did not return a visual response. The participant stated that they expected this action to open the maintainer tab again or for the results of their search to be present after opening the maintainer tab manually after having clicked the menu item.

When asked about idea on how to improve the feature P1 said that they would like the contact information to be clickable, which would then allow them to more easily navigate to their preferred communication channel and set up to contact the person (+ P3, P4). P1 feels like the position (job title) of the maintainer should be more visible (+ P2, P4). They also suggested that each maintainer could add a personal note to their card showing their availability or a list of frequently asked questions. Concerning the user interface, they suggest splitting the card into two sections. One containing information about the person (name, team, position) and the other containing the communication information (GitHub, email, phone number). When talking about the user interface P1 mentioned that they found the used colour scheme to be attractive. Finally, P1 thinks that it would be useful to them to be able to add their own maintainer information, and that they could imagine using the feature. P1 completed the task 47% **faster** when using Maintainer Lookup compared to when they were not using the feature.

P2 has interacted with microservice architectures during their studies, but has never worked with a large scale microservice application. When asked about debugging, P2 mentioned that they accomplish this sort of task daily, as they work as a bug fixer. They tell us that working with business use cases and large codebases is very relevant to their day-to-day job. Since the codebase P2 works with has its own object relational mapper and the classes have a custom inheritance scheme, it is not usually straight forward to see which functions are actually called. In these cases, P2 starts a debug session, to see which function gets called, even though it might take a long time to accomplish.

They approached the function identification task by opening the microservice file directly. Just as P1, they did not approach the problem through a debugging perspective, line by line, but rather started locating variables useful to the task. When asked why they did not follow the same debugging approach as in their work context, they said it was because they felt the scale of the project that we provided did not require this. They felt it was easier to just look at the

code and read through it, while in a larger project this might be unfeasible, because of the large quantity of code. They noted that with the flow graphs, it was straightforward to see which functions are being called, without needing to go line by line in the codebase to locate them.

When asked about their experience while interacting with the Flow Documentation feature, they felt like it was easy to interact with the graph and that they liked that clicking on a function takes you to the function definition. They remarked the graph is easier to follow than reading code, and that it gives a high-level overview.

They said that if it were to use the extension frequently with the same project, they would like to be able to manually edit the graph. They imagine doing so by reordering the nodes in the graph or adding notes to certain parts of the graph. They would also like to be able to hide certain nodes that they are not interested in. They suggest this could be done by clicking on the corresponding item in the legend. If for example, they would be part of the payment team, they would hide nodes which are not related to payment.

P2 notes that for languages like C++, having the entire function signature would be more useful, compared to only having the function name. This could be useful in case of overloading. Another possibility that they mentioned is to keep displaying only the name as the node label, and to display more information like the signature and docstrings when they hover over the node. They like the simplicity of the graph view, as additional information does not distract from the main feature, which is the graph. If the graph is too tall, it should be split in several smaller graphs, or nodes should be collapsible, in order to get the whole graph on the screen. They think that the content provided is sufficient for the task at hand (+ P1, P3, P4). P2 completed the task 23.3% **faster** when using Flow Documentation compared to when they were not using the feature.

P2 works in a company that has over 500 developers, who are part of different teams. When it comes to finding maintainers, P2 asks their senior team members about whom they should reach out to (+ P3, P4). The response P2 can expect is either the name of a team or the name of a specific person within that team. Even if they have access to a company spreadsheet which contains information about product owners and project managers, they still rely on their teammates' help. They stated that it feels easier and more accurate to them, even if it takes more time to get a response. When requested to elaborate, P2 proceeds by presenting a situation they had found themselves in, in which they noticed a potential issue with using the spreadsheet. After being hinted by code references to the concept of an "XML report" in their codebase, they looked to contact the maintainer responsible for this concept. They did not find the manager of "XML reports" in the spreadsheet. However, they had the prior knowledge that the "XML report" concept is a subpart of the "Document" concept, which did have a manager in the spreadsheet. They add that the company spreadsheet would be much more useful if it contained all the concepts present in the codebase, not only the generic ones, even though acknowledging that this would increase the spreadsheet size considerably, and reduce the readability.

P2 was dealing with the issue of finding maintainers almost daily when first employed, for up to 2 months. At the time they were working with modules that were very often interconnected. Now, their work has a narrower scope, and they say that they encounter this issue very rarely (+ P3, P4). P2 mentions that they also find additional information on who to contact through an internal company app. This app contains all employees of the company and for each employee the following information is present: GitHub username, email, phone number, job position, manager, coach and work location. The participant remarks that the information may not be up-to-date, as the employees have to manually edit this information while according to P2, they might not have the incentive to do so. P2 states that they search for code maintainers while exploring the codebase. They would check who wrote a certain piece of code with the *git blame* feature of the *GitLens* extension. According to P2, the *git blame* feature is useful when one person

implemented the function in question. However, when a function is often edited by multiple people it becomes more difficult to find the right person (+ P1, P4). P2 states that either it takes more time or it is impossible to find the developer who first wrote the function because of successive commits on certain lines, which are either added or removed, which causes the git tool to lose track of original code author.

After using the Maintainer Lookup feature of the extension P2 said that they would like to change the path name, which is the header of the matches after look up, with the maintainer name and position. The position in particular is very important to them, as it is the first thing they want to check (+ P3). When asked for suggestions on how to improve the git related knowledge card the participant said that they felt clicking the git maintainer should highlight the lines of code which they edited. P2 also mentions the visual history feature of *GitLens*, which gives a visual intuition of who made the latest commits on a file, and the largest commit, by using a bubble chart. As for the information on the cards, P2 says that seeing the commit title is more important than the git author name. P2 mentions liking the "one-click" usability of the Maintainer Lookup. Finally, the participant noticed that when the sidebar was really thin, the information displayed became almost unreadable because of the limited space (+ P4). As an alternative, P2 suggests using tooltips to show the complete information only while hovering over the displayed maintainer name or team. They also suggest adding a "copy all" button to the knowledge cards. P2 completed the task 60% **slower** when using Maintainer Lookup compared to when they were not using the feature.

P3 worked with microservice in their university courses, where they had to create separate microservices and scale them horizontally and vertically as part of the course project description. They stated that they feel confident with their understanding of microservices. They also drew a parallel between microservice concepts and their work codebase. The work environment has a "micropackages" structure where many tools call the same packages, the only difference being that calls are performed locally rather than over a network. When P3 is asked to modify something in their codebase at work, they start by first searching for the function they intuit to be relevant by name or signature. They noted that their intuition is based on a combination of previous experience and general code knowledge. They then place a breakpoint on one or several functions to start a debugging process. They follow a lookup approach not because the tool they use has limited debugging powers (XCode), but because the code structure is inheritance-heavy, and debugging would take a lot of time to reach the relevant function which contains the code logic.

While performing the task using the Flow Documentation feature, P3 did not notice all the buttons in the sidebar (create graph, refresh graph). They clicked on the flow graph name which they assumed would start the flow generation, which confused them slightly, as that button only redirected them to the flow definition. P3 suggested adding a two-step flow when using the feature: the first step would be to click on the flow name, which would reveal the buttons "create graph" and "refresh graph", which could be clicked to actually open the graph.

P3 also mentioned that they would find it useful to have a similar functionality as in XCode (as well as VSCode) where the user can go to a function definition by using the "Ctrl + Left Click" shortcut. P3 did not notice that a similar functionality exists in Flow Documentation, which resides in the context menu option "Highlight in Flow Graph". After the interviewer brought attention to this functionality, they acknowledge its usefulness, but said that they would like to see the use of the same "Ctrl + Left Click" shortcut. They felt Flow Documentation was very intuitive and easy to use. They underline this by saying: *"The graph legend was very useful in perceiving the code as modular, but interconnected"*. P3 felt the graph design was very intuitive as well, knowing what every call means and clicking on the buttons was described as a "cool" functionality. The participant expressed that they found the graph was very responsive.

P3 completed the task 33.3% **slower** when using Flow Documentation compared to when they were not using the feature.

P3 works in a company where developers are split in two teams, both managing the different mobile implementations of the same app, one for Android and one for IOS/Apple. The code is structured as a monorepository, so almost every module is shared with other teams and is designed to be modular and reusable. It happens quite often that P3 needs to contact other developers or engineering managers, who are the first point of contact, or product designers/managers/owners, who are the second point of contact. In practice, it happens more often that they contact product owners than engineering managers. P3 estimates that they contact developers 60% of the time and product owners/engineering managers 40% of time.

In the case of contacting developers, P3 uses *git blame* to actually find the developer, then by using their GitHub username, P3 can find them on Slack and contact them. When it comes to contact product managers P3 can only contact his own product manager for questions. If the problem that he has to solve reaches a common library/package or a feature of another app, P3 will contact their manager, who will then reach out to the other product manager responsible for the concerned common library. This process can delay communication by two to three days, during which their task is blocked. Questions aimed for developers usually concern code reusability. On the other hand, they stated that they have to talk to other maintainers/product managers almost every day.

P3 started by reading through the default intro message to find out how to operate the extension. While completing the task, they verbally expressed that they did not understand the difference between "Most recent Git Maintainer" and "Most Relevant Git Maintainer".

During the task when P3 was using plain VSCode, they identified several keywords within the code file one at a time. Some keywords did not exist in the spreadsheet, so they had to switch back and forth between VSCode and the spreadsheet multiple times, in search of the correct keyword. After completing the task using the Maintainer Lookup feature, P3 stated that they find the feature very easy to use and intuitive, saying *"At first glance, I knew who is maintaining this feature or product, so I could contact them directly"*. According to P3 the tool was very intuitive. They said that they would add the team name and position to the title card of the maintainer, so that they do not have to click on each individual card to find the product owner, or the relevant person in the team. They would like the maintainers to be split in two groups: technical and non-technical people. P3 also mentioned that the "Show Maintainer" button is not very accessible, so a separate button always visible in the header would be a better solution according to them. Another suggestion given by P3 was a "minimap", with the list of maintainers, with toggled visibility. They also suggested that the sidebar could update when a user changes file. P3 completed the task 49.3% **faster** when using Maintainer Lookup compared to when they were not using the feature.

P4 currently works within a company which integrates different hotel services into one shared database. They say that they feel familiar with using microservices, as they have to interact with different websites and external API's almost every day. They often do not have access to the external services API, so when they have an issue with external code, P4 has to contact the different service providers and ask them to fix the issue. This process is usually time-consuming, for both locating who to contact, and also waiting for a reply and, eventually, a fix in their code. This process usually takes between one and four weeks.

P4 employs a traditional debugging technique when tasked to identify and fix an issue in the code. They place a breakpoint at the start of the function, the analyses each code line, checking if the results that they expect are actually returned. When this is not the case, they place a breakpoint at that specific line and continue debugging from there.

When performing the faulty function location task without the Flow Documentation feature,

	VSCode	VSCode + FD	VSCode	VSCode + ML
P1	83s	17s	42s	22s
P2	30s	23s	8s	19s
P3	50s	75s	75s	38s
P4	37s	66s	25s	19s

Table 6.1: Task time completion (in seconds)

they debug the code line by line and reach the faulty API within 20 seconds. They confirm that the function returns false results, then look for a similar route name in the microservice code. Comparatively, when using the Flow Documentation feature, P4 creates the graph, then browses through the code names. They mention that they were confused when seeing so many nodes which were unrelated to the task at hand. After clicking on several nodes which seemed relevant to them and placing breakpoints at the start of each function, they resume the debugging process. They spend more time in the debugging stage, as several breakpoints are actually irrelevant to the issue that we described. P4 said that they were becoming frustrated by the fact that *"some nodes in the graph did not lead to anything"*. They suggest that the graph should be smaller and only contain nodes that are relevant to the task. P4 completed the task 43.9% **slower** when using Flow Documentation compared to when they were not using the feature.

P4 works in a company which has over 500 developers. They encounter the issue of having to locate a relevant person quite frequently, as they are still a new employee in the company. When they have a question about a piece of code, they use *git blame* to locate the last developer who modified the code. When they have a question about the functionality of a feature, P4 goes through a spreadsheet of maintainers, provided by their company. They describe the spreadsheet as being very descriptive, where every module of their code is handled by either a senior developer or a product owner. If neither *git blame* nor the spreadsheet provide sufficient information, P4 asks a colleague about who should they contact about a certain feature or how to fix the problem. They mention that this happens very rarely, in about 10% of cases.

P4 started the maintainer identification task without our feature, and they identified a relevant keyword in the file they were analysing, which lead them to the correct maintainer in the spreadsheet. They mentioned that they perceived the task as relatively easy. While using our feature, they identified a list of two maintainers, one of which was a product owner, and the other a team leader. Based on this difference, they picked the product owner as the person they would contact next. P4 completed the task 24% **faster** when using Flow Documentation compared to when they were not using the feature.

6.9 Experiment Results Discussion

We have gathered plenty of feedback from both the preliminary form and the qualitative interviews. We will start by interpreting the results of the preliminary form first, where the feedback is less detailed, as the participants did not experiment with the tool for themselves. We will then move on to the more detailed feedback collected from the qualitative interviews where participants had the freedom of experimenting with the tools by themselves, and through completing microservice architecture comprehension and maintainer identification tasks.

The preliminary form gives us insight about how users have previously experienced and solved the issue of finding the relevant function within a microservice architecture codebase. The most common approach developers employ is debugging or asking a colleague, while some users prefer searching the codebase for the endpoint route name and manually linking them

with an exposed API endpoint (through different ways: grep2win, *double shift* in IntelliJ, *Find Usages* in VSCode), asking for an explanation from the developer who last modified the code, browsing through the GitHub pull request section to see which files have been added or changed, or exploring the code documentation.

Similarly, people who have experienced the issue of not knowing who to contact to get more information about a piece of code have tried different approaches as well. The most common approach is using *git blame* (through git or other extensions which incorporate it), a company spreadsheet which contains maintainer information, or reaching out to other teammates, developers or project managers, through direct communication or instant messaging apps.

From the participants responses, we can conclude that a majority of users have sometimes encountered both issue (36/42 encountered the microservice issue, 27/42 encountered the maintainer issue), meaning that we reached our target audience when sharing the form. Since the majority of users try to solve the microservice issue through debugging, we have to ask if the "Flow Documentation" feature is helpful to them. As we discovered during prototyping (Chapter 3), our extension did not integrate well with developer's usual debugging workflow. These developers emphasised that their debugging process relies heavily on line-by-line code analysis, rather than starting with an overview and then delving into the details. The prototyping feedback is now confirmed, as 3 out of 4 participants in our qualitative interview did not follow a debugging procedure while solving the tasks, and all of them found the feature useful, while one programmer debugged the code, and found the feature less useful. This leads us to a key takeaway from our experiment:

> The "Flow Documentation" feature is most effective when combined with a lookup approach to code modification.

P2 did not follow a lookup approach to solve our task, but did mention that they do usually follow a debugging approach in their company setting. They go on to explain that before starting the task, they assessed the size of our showcase code as being relatively small, and they thought following a lookup approach would yield simpler results. This explanation leads us to consider using a larger microservice codebase in future experiments, where the developers would be more inclined to follow their usual approach, although this can force other developers into a style of problem-solving which might not be their preferred way. For example, due to the mentioned limitations of their company work environment, P3 prefers to avoid debugging as it is usually faster to employ a lookup approach.

From the interviews of P2 and P4, we can suggest that the maintainer identification task is predominantly done when a developer first joins their company, and it decreases in frequency as they acquire more experience. This happens because the more time a developer spends in a team, department, or company, they get to know more people and learn who is responsible for what code. As we learned from the interviews, many companies have different approaches to internal communication. All interview participants except P1 note that they use *git blame* to find the developer who is responsible for the code. Additionally, all participants ask their teammates for more information either as a first attempt to find the code responsible (P1, P2, P3) or as a backup to *git blame* (P4). This is also reflected by the responses from the preliminary form, where 10 out of 42 developers say they use *git blame* or similar tools to find the responsible, while 14 out of 42 developers said they prefer in-person communication. When we questioned the interview participants why do they ask their colleagues for information as a main strategy of finding a code maintainer, the answers varied. P1 does it because they feel more at ease with communicating with their colleagues rather than using software tools. P2 asks their teammates because of commodity. In P3's company, they are restricted to contacting their engineering manager when having a question about a feature, even if the feature is under another team's supervision. P4

does not use it as a first strategy, but they ask their colleagues for more information when *git blame* and the company maintainer spreadsheets do not provide sufficient information. This indicates that we should not ignore the developer's human connection needs when designing our tool.

> "Maintainer Lookup" should not aim to replace human communication, but rather make the process smoother and more effective by providing additional maintainer information.

All interview participants except P1 felt like the Maintainer Lookup feature improves their knowledge on the field. P1 brought up a very good argument for in-person communication during the interview, noting that: *"By using it [the Maintainer Lookup feature] I get an answer to who is the maintainer, but currently I don't get the reasoning behind the answer"*.

P1 further explains that when they encounter code written by another developer, they might ask the manager, who usually speaks their thoughts out loud. So P1 could additionally find out that the code piece is linked to a certain feature, a certain view or a certain project, and who is maintaining all of these components in the codebase. This lead us to understand the importance of linking maintainers not necessarily to a file or a folder in the codebase, but rather to a concept. To explain this, let us consider the example of a Python file which manages the rendering of financial data on the frontend. Within this file, we can already identify three concepts: accounting information, frontend rendering, and Python code. Each of these concepts can have different maintainers, which should be accounted for in the maintainer to code map.

> "Maintainer Lookup" should link maintainers to code concepts, rather than files or folders.

The "Maintainer Lookup" feature can be further improved in terms of functionality and developer reach. Currently, we assume that the last person who edited a certain file or the person who contributed the most commits within the last month is the current maintainer of that file. We could scrape additional information from their GitHub profile, such as their public email address, and social media links to include as contact information on the maintainer card. This would make setting up the extension less demanding of the system administrator. However, judging from the qualitative interviews' feedback, P2 mentioned that a similar feature exists in the *GitLens VSCode extension*. The *GitLens* extension allows the user to view the latest commits in the current file either in a list view or visually with a bubble timeline graph (Figure 6.1). Then, the user can achieve even more accurate conclusions than by using the "Maintainer Lookup" git feature. However, *GitLens* does not offer the possibility to see real world information about the maintainers, and, in this case, it can only recognise developers that contributed to the code. In this sense, the git functionality might draw the user away from the intended use of "Maintainer Lookup", which is locating experienced people within the company, including not only developers, but also product owners, administrators and project managers.

Several common design improvements were suggested by participants in the qualitative interviews, which could improve their perception of the presented information. For example, the "Flow Documentation" graph legend can be highlighted more (P1, P2), and the graph could have more interaction: zooming, panning (P1, P2), node rearranging and shuffling (P2). Several participants noted that the "Maintainer Lookup" list view gives the same level of importance to information about the maintainer position, team, and name as contact information, such as email, phone number, GitHub username. P1, P2 and P3 perceived that the first type of information as more important than contact information, so they argue that it should be better highlighted. Additionally, P1 and P2 note that they have to work with a limiting space while using the "Maintainer Lookup" feature, within the VSCode sidebar, while P2 and P3 suggest that

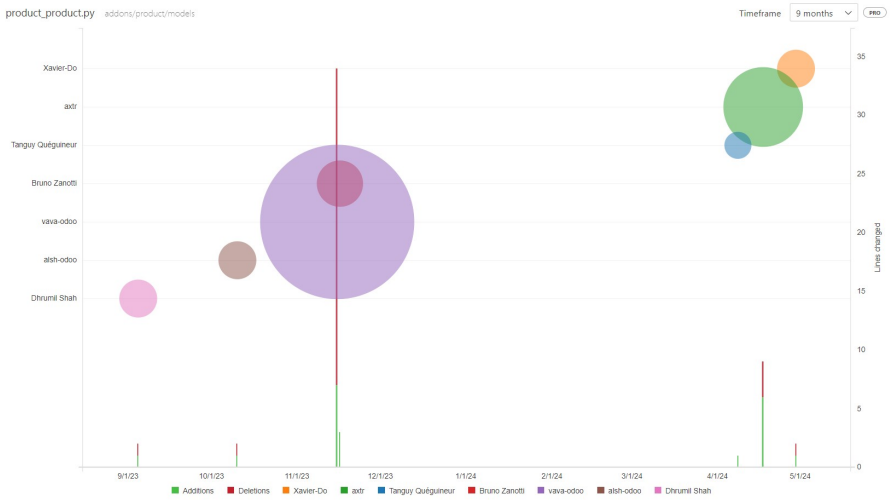


Figure 6.1: GitLens bubble timeline graph

the maintainer information can be displayed similarly to the flow graph, in a separate window, a minimap, or a button in the file editor header.

All participants have noticed that the workspace is cluttered while using the extension, especially visible while using the "Maintainer Lookup" feature. P2 mentioned that this is not an issue that bothers them, as they still find it more useful to have everything integrated within the IDE, than having to switch between windows. During the interview, the other participants had to change the size of the sidebar in order to fit their needs, which can be seen as a disadvantage while working, as there is less visible space for the code itself. Although they did not explicitly mention this a negative point, we still need to take this into account when designing an integrated tool, leading us to our final remark:

Developers need to have enough screen space while coding.

We hypothesise that this issue can be mitigated by using a bigger display screen, or using a multiscreen setup. We believe that this issue should be further investigated in research.

6.10 Research Questions Answers

To reiterate our initial purpose, we have designed the prototyping, solution and evaluation chapters of our thesis with the goal of answering the following research questions (RQs):

- RQ1: What challenges do software developers face when trying to understand codebases with a microservice architecture and locating maintainers?
- RQ2: How do developers perceive the usefulness of FlowDocs for understanding codebases and locating maintainers?
- RQ3: How does FlowDocs influence developers' performance and accuracy in different tasks?

After analysing the literature, we found that developers spend a considerable time of their code modification tasks by trying to comprehend the codebase. Also, when working with mi-

croservice architectures, they initially spend more time comprehending the microservice interactions before trying to modify the code. From the interviews of P2, P3 and P4, we notice that the issue of debugging large codebases is present in industry code, because of the large scale of the codebase and the code inheritance structure which limits the developer's IDE function locating capabilities. The developers often have to rely on a lookup approach, by identifying relevant functions or variable names, placing breakpoints by intuition or experience, and only then starting a debugging session.

Similarly, in the context of newcomer onboarding, the literature reveals issues about delays in communication and loss of information between developers due to a centralised communication flow. The interview with P3 confirms this issue concerning centralised communication. At the same time, from the interviews with P2 and P3, we can see developers trying to practice direct communication with their team members, and by using tools like *GitLens* for locating maintaining developers. P2 mentions that the limitations of a spreadsheet to find maintainers discourages them from using it, and pushes them towards relying on their teammates to get maintainer information. The limitations of spreadsheets are the speed of browsing through them, as we can find out from P1, the inability to capture all code concepts and having to rely on personal knowledge of concept interconnection, as P2's interview reveals.

> We can answer RQ1 by concluding that the issues we identified in the literature review affect a considerable portion of our experiment participants.

After interpreting the results of the qualitative interviews, we can give an answer to RQ2. Regarding the "Flow Documentation" feature, P1 highlights that some developers interpret code in a conceptual way, rather than seeing it as individual lines of code. Using "Flow Documentation", P1 skipped the process of having to find the bug location, and could follow their conceptual way of thinking about code. P1, P2, P3 said that the "Flow Documentation" graph is easy to interact with, and it is easier to follow the graph than reading the code, and it gives a high level overview of the code. P4, as they use plain debugging to understand code logic, did not find that "Flow Documentation" improved their workflow. P1 and P3 separate code into logic pieces based on intuition, which is itself based on their understanding the code. The expert developers will solve a task with the information that they gathered through prior experience. A newcomer, in contrast, does not have sufficient experience and have to analyse the code line by line to comprehend the codebase. P2 said that it is easier to read a graph than it is to read code, which leads to faster code comprehension, by avoiding the analysis of many lines of code which do not contribute to advancing the code logic. On the other hand, people who are more experienced, could benefit from the graph visual aid (enhances mental map), and fast user interface features, which P3 said would be essential to make their task faster. P1, P2 and P3 consider that the graph enhances their lookup workflow. By contrast, P4 argues that the debugging workflow does not benefit from the conceptual map put forward by the code graph overview.

Regarding the "Maintainer Lookup" feature, the speed of user interaction is greatly appreciated by all participants, who highlight the usefulness of being a click away from the information in their own IDE. At the same time, we see that users have the desire to customise what information is displayed and how. Different developers have different communication approaches and, as such, different information and communication channels are interesting to them. We appreciate that we could further explore this field, by analysing the different developers' communication needs, and also research how developers think about code conceptually rather than through folders and files.

> We can answer RQ2 by concluding that the participants perceived the "Maintainer Lookup" as useful, while the "Flow Documentation" feature is useful only when developers employ lookup techniques for code comprehension.

While using the "Flow Documentation" feature, P1, P2, and P4, who quickly navigated the flow graph, were faster at completing the interview task. P3 did not intuitively know how to use the tool; we observe that they took longer to notice the buttons, which resulted in them completing the task slower with the feature than without. This explains the large increase in P3's time when using the tool compared to the other interviews. The accuracy of completing the tasks for everybody involved was 100%. There is an indicator that the tool can help with speeding up the function locating task, which is a task often accomplished by developers within their daily jobs, but it is also clear that the learning curve of the tool should be smoothed out by adding more visual indications of progress and tooltips.

While using the "Maintainer Lookup" feature, the participants generally had a faster task completion time. P2 is the only participant who performed slower with the feature. When completing the task without the feature, they identified the maintainer almost instantly since the name of the file they were looking at pointed directly to one of the entries in the spreadsheet. When using the feature, the maintainer list displayed multiple maintainers, which made them reconsider their initial choice. All participants approached the task by searching for a certain concept or keyword in the file they were investigating, in order to increase their chances of finding a maintainer. P3 even identified a keyword which did not have anyone associated with it in the spreadsheet, which increased the spreadsheet lookup time further. When using the "Maintainer Lookup" feature, the participants could identify at least one of the concepts in the sidebar and chose this person as the most relevant maintainer. P1 picked the most relevant maintainer by looking at the file hierarchy and picking the closest parent of the file they were analysing, while the other participants picked a maintainer based on the contact position. This means that they picked the person who had the most important perceived role in the company, which in our experiment was also the intended answer. Our interview results indicate that developers tend to search the right person to contact either based on previous interactions with other developers or project managers, or with the centralised point of communication. We believe that more effort should be spent analysing whether the increased amount of possible maintainers for a file also increases the success rate of choosing and contacting the most appropriate person for the task, considering the code concepts that they manage.

> We can answer RQ3 by concluding that FlowDocs has the potential to improve developers' speed when performing comprehension and maintainer identification tasks, as long as it can account for individual differences in communication styles and conceptual understanding of code.

6.11 Limitations

Before talking about future plans, we must acknowledge the limitations of our extension and the user study. The "Flow Documentation" flow graphs still suffer from the same issue of overcrowding when analysing large-scale microservice codebases, though to a lesser extent than the Prophet tool[6]. The "Maintainer Lookup" feature can only extract limited maintainer information from git, such as the username and email, so developers will have to mostly rely on contact information added manually by a system administrator.

The user qualitative interview includes four participants. Their actions were tracked in a virtual environment and over a short period of time (an interview lasts between 45 minutes and

one hour), which does not typically represent a real world situation or task that they might have to perform on the job over a longer period of time. Tracking user interaction over a longer period of time could be beneficial for analysing the "Maintainer Lookup" feature. In this way, we could encounter how often do developers need to perform the lookup task in a real-world environment and also track people from different teams/countries, in a GSD environment. The tasks presented during the "Flow Documentation" feature analysis only cover a small architecture codebase, which again might not represent a real industry use case and can only give us limited information about user comprehension in a large distributed project. The microservice architecture allows users to develop smaller projects individually and gives them the freedom to choose any technology stack that they desire. The "Flow Documentation" feature only targets Python microservices and can detect endpoints generated with the Flask[2] framework. This is only a small sample of what the microservice architecture can offer.

6.12 Future Work

To address the limitations, we propose several improvements that can be achieved in the future.

The "Flow Documentation" feature can be functionally extended by adding support for detecting endpoints in multiple frameworks, and by statically analysing code implemented with different programming languages. This is made easier by the modularity of the code and the intermediate JSON flow graph representation, which eases the interaction between parsing the code (language specific) and displaying the flow graph (language agnostic). This is important because even within a single project, there can be microservices written in different languages, and analysing just some of them gives an incomplete image of the microservice interaction.

The flow graph feature can also benefit from several improvements. When a developer creates the graph, they could view only the interaction between microservices, while the specific functions can be abstracted away, so they can get a sense of how interconnected their microservice is with other microservices. Additionally, two participants in the qualitative interview have theorised that the graph can become complex and hard to follow when there are many microservices, so hiding some user-selected nodes can mitigate this issue. After they get this general picture, they could switch to the current functional view, where all the details of the function call stack are exposed. Considering the research of Landesberger et al. [14], we could also experiment with node aggregation and filtering based on certain metrics, such as grouping nodes that belong to the same microservice, or filtering nodes that are a certain path length away in the path from the start node. Similarly, force-based graph approaches could be implemented in the interactive graph in order to improve readability metrics such as minimising node and edge overlaps, minimising edge crossings, and equalising edge lengths.

Some questions in the future work of Bushong et al. [6] remain relevant to our extension as well. Microservices change structure over time and this can lead to bottlenecks and architectural degradation. The tool could take into account the evolution of a microservice and prevent the mentioned issues by comparing microservice versions and general trends of microservice interaction. Dynamic analysis of the code can be considered, as it can give detailed information on code execution and potential bugs present in microservice interactions, even if it does not generate the complete tree of possible paths of execution and interaction.

Zhou et al. [31] have created a benchmark system for microservices which includes 24 microservices related to business logic and a high level of interaction between them. The benchmark is developed using different programming languages (Java, NodeJS, Python, Go) and databases

[2] `https://flask.palletsprojects.com/en/3.0.x/`

(MongoDB, MySQL). We could use their code to test our "Flow Documentation" feature, once we extend it to other programming languages other than Python.

Let us reiterate the meaningful future directions of research, as they were introduced in this chapter. We believe that more research should be conducted in the field of developers' communication needs, in order to understand how developers perceive the hierarchy in a company. *Do they perceive product owners, project managers, or otherwise other middle management employees as being more knowledgeable about a certain feature of their codebase, or do they trust fellow developers more? Does code knowledge relate to product or conceptual knowledge?*

This also creates the question of how do developers approach code modification tasks. Are traditional debugging and lookup debugging the only methods developers employ? How can we encourage developers who use these methods to combine their regular workflow with other tools and extensions in the most effective way?

Additionally, we should explore how developers think about their code. Is it a conceptual process, where they think in terms of concepts, such as "Documents" and "Accounting information", or do they interpret code based on location and context, such as the folder and file hierarchy? Drawing from the experience of P2, we can ask: *"Does the size of a codebase influence the way in which developers perform code modification tasks?"*

These questions can further lead the development of understanding how developers think and interact with large and small codebases, creating new opportunities for effective tools and extensions.

Conclusion

In this study, we have explored the issue of onboarding new developers in a large-scale company, with many independently moving projects or microservices. We have split the issue into two smaller issues, which are interlinked, but can be viewed separately as well. In Chapter 2, we performed a literature study to obtain information about the fields of knowledge management and transfer, onboarding new employees in large companies, global software development, code comprehension, dependency visualisation and microservice architectures.

Based on our findings, we create several hypotheses and follow the design science research methodology in order to develop conceptual and code prototypes to further explore the realm of possible solutions. In Chapter 4 we describe an ideal tool which will solve the identified problems, by helping developers understand the microservice interaction and their code base more quickly, and get in code maintainers in their company. In Chapter 5, we describe the technical implementation of our VSCode extension, detailing each of the individual features: "Flow Documentation" and "Maintainer Lookup".

In Chapter 6, we organise a preliminary survey with software developers, then follow it with 4 qualitative interviews to gain a deeper understanding of the problems developers face, as well as their solutions to the problems. Following the evaluation, within the context of software comprehension, we conclude that understanding large codebases still poses problems to software developers. As codebases become more complex and interlinked, the tools that are available within and outside the developer's IDE become more limited. We identify that certain developers approach codebase exploration in different ways. Some view code through a factual lens, following the folders, files, and line of code hierarchy, and employ traditional debugging techniques, while others view code through a conceptual lens, following a hierarchy of concepts, and employ lookup techniques. We try to fix their issues with "FlowDocs" and achieve moderate success with developers who employ lookup techniques when modifying code. Similarly, in the context of developer communication, we find that developers use a combination of tools and in-person communication to find information in their company. We notice that this difference arises from the way in which companies either enforce rules and hierarchy, or leave developers to use their own means of communication. We aim to aid developers in following their preferred means of communication, by providing additional contact information of maintainers. In this context, "FlowDocs" was perceived positively and improved the experience of our interviewees when looking for a code responsible. We believe more research should be conducted in the field of developer comprehension of codebases in general and whether that is impacted by the way in which they perceive the code structure.

Bibliography

[1] Işıl Karabey Aksakalli et al. “Deployment and Communication Patterns in Microservice Architectures: A Systematic Literature Review”. In: *Journal of Systems and Software* 180.1 (Oct. 2021), p. 111014.

[2] Linda Argote et al. “Knowledge transfer in Organizations: Learning from the Experience of Others”. In: *Organizational Behavior and Human Decision Processes* 82.1 (May 2000), pp. 1–8.

[3] Stefanie Betz, Andreas Oberweis, and Rolf Stephan. “Knowledge Transfer in IT Offshore Outsourcing Projects: An Analysis of the Current State and Best Practices”. In: *2010 5th IEEE International Conference on Global Software Engineering*. Princeton, USA, Aug. 2010, pp. 330–335.

[4] Justus Bogner, Pawel Wójcik, and Olaf Zimmermann. “How Do Microservice API Patterns Impact Understandability? A Controlled Experiment”. In: *arXiv preprint arXiv:2402.13696* (2024).

[5] Michel Buffa. “Intranet Wikis”. In: *Proceedings of the IntraWebs Workshop 2006 at the 15th International World Wide Web Conference*. Edinburgh, Scotland, May 2006, pp. 811–817.

[6] Vincent Bushong et al. “Using Static Analysis to Address Microservice Architecture Reconstruction”. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Melbourne, Australia, Nov. 2021, pp. 1199–1201.

[7] Tomas Cerny et al. “From Static Code Analysis to Visual Models of Microservice Architecture”. In: *Cluster Computing* 27.3 (Apr. 2024), pp. 1–26.

[8] Donny T Daniel et al. “Polyptychon: A Hierarchically-Constrained Classified Dependencies Visualization”. In: *2014 Second IEEE Working Conference on Software Visualization*. Victoria, Canada, Sept. 2014, pp. 83–86.

[9] Martin Dias et al. “Evaluating a Visual Approach for Understanding JavaScript Source Code”. In: *ICPC 2020: Proceedings of the 28th International Conference on Program Comprehension*. Seoul, Republic of Korea, July 2020, pp. 128–138.

[10] Dussan Freire-Pozo et al. “DGT-AR: Visualizing Code Dependencies in AR”. In: *2023 IEEE Working Conference on Software Visualization (VISSOFT)*. Bogotá, Colombia, Oct. 2023, pp. 95–99.

[11] Michel Grundstein. “From Capitalizing on Company Knowledge to Knowledge Management”. In: *International Conference on Systems Thinking in Management* 2.1 (Apr. 2000), pp. 42–53.

[12] Denis Helic, Hermann Maurer, and Nick Scerbakov. "Knowledge Transfer Processes in a Modern WBT System". In: *Journal of Network and Computer Applications* 27.3 (Aug. 2004), pp. 163–190.

[13] Kenneth B Kahn. "Interdepartmental Integration: A Definition with Implications for Product Development Performance". In: *Journal of Product Innovation Management* 13.2 (Oct. 1996), pp. 137–151.

[14] Tatiana von Landesberger et al. "Visual Analysis of Large Graphs." In: *Eurographics (State of the Art Reports)*. Norrköping, Sweden, May 2010, pp. 37–60.

[15] Mircea Lungu et al. "The Small Project Observatory: Visualizing Software Ecosystems". In: *Science of Computer Programming* 75.4 (Apr. 2010), pp. 264–275.

[16] Kwan-Liu Ma and Chris W Muelder. "Large-scale Graph Visualization and Analytics". In: *Computer* 46.7 (July 2013), pp. 39–46.

[17] Alfred H Mazorodze and Sheryl Buckley. "A Review of Knowledge Transfer Tools in Knowledge-Intensive Organisations". In: *South African Journal of Information Management* 22.1 (Oct. 2020), pp. 1–6.

[18] Leonel Merino et al. "A Systematic Literature Review of Software Visualization Evaluation". In: *Journal of Systems and Software* 144 (Oct. 2018), pp. 165–180.

[19] Srinivas Nidhra et al. "Knowledge Transfer Challenges and Mitigation Strategies in Global Software Development-A Systematic Literature Review and Industrial Validation". In: *International Journal of Information Management* 33.2 (Apr. 2013), pp. 333–355.

[20] Ikujirō Nonaka and Hirotaka Takeuchi. "The Knowledge-Creating Company". In: *Harvard Business Review* 85.7/8 (July 2007), p. 162.

[21] Ikujirō Nonaka and Hirotaka Takeuchi. *The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation*. Oxford University Press, Sept. 1995.

[22] Ken Peffers et al. "A Design Science Research Methodology for Information Systems Research". In: *Journal of Management Information Systems* 24.3 (Jan. 2007), pp. 45–77.

[23] Michael Polanyi. *The Tacit Dimension*. The University of Chicago Press, May 2009.

[24] Helen C. Purchase. "Effective Information Visualisation: A Study of Graph Drawing Aesthetics and Algorithms". In: *Interacting with Computers* 13.2 (Dec. 2000), pp. 147–162.

[25] Georg Simhandl, Philipp Paulweber, and Uwe Zdun. "Developer's Cognitive Effort Maintaining Monoliths vs. Microservices-An Eye-Tracking Study". In: *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. Los Alamitos, USA, Dec. 2023, pp. 339–348.

[26] Igor Steinmacher et al. "Social Barriers Faced by Newcomers Placing their First Contribution in Open Source Software Projects". In: *Proceedings of the 18th ACM conference on Computer supported cooperative work & social computing*. New York, USA, Feb. 2015, pp. 1379–1392.

[27] Adel Taweel et al. "Communication, Knowledge and Co-ordination Management in Globally Distributed Software Development: Informed by a Scientific Software Engineering Case Study". In: *2009 Fourth IEEE International Conference on Global Software Engineering*. Limerick, Ireland, Aug. 2009, pp. 370–375.

[28] Alexandru Telea et al. "Extraction and Visualization of Call Dependencies for Large C/C++ Code Bases: A Comparative Study". In: *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. Edmonton, Canada, Sept. 2009, pp. 81–88.

[29] Xin Xia et al. “Measuring Program Comprehension: A Large-scale Field Study with Professionals”. In: *IEEE Transactions on Software Engineering* 44.10 (July 2017), pp. 951–976.

[30] He Zhang et al. “Microservice Architecture in Reality: An Industrial Inquiry”. In: *2019 IEEE International Conference on Software Architecture (ICSA)*. Hamburg, Germany, Mar. 2019, pp. 51–60.

[31] Xiang Zhou et al. “Benchmarking Microservice Systems for Software Engineering Research”. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. New York, USA, May 2018, pp. 323–324.