Tampere University

Kristian Skogberg

GENERATING UI CODE FOR SCIENTIFIC COMMAND LINE TOOLS USING LARGE LANGUAGE MODELS

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Dr. Jussi Rasku
Prof. Jyrki Nummenmaa
May 2025

ABSTRACT

Kristian Skogberg: Generating UI Code for Scientific Command Line Tools Using Large Language Models
Master of Science Thesis
Tampere University
Advanced Studies in Software Engineering
May 2025

This thesis explores the current capabilities and limitations of using artificial intelligence (AI) and large language models (LLMs) to generate user interface (UI) code. In the action research component of this thesis, a graphical user interface (GUI) was developed for VeRyPy, a scientific Python library for solving vehicle routing problems. The GUI code was generated using GitHub Copilot and OpenAI's GPT-4o model.

In this action research, the VeRyPy GUI development process was carried out in five iterations, following cycles of planning, action, analysis, and conclusion. In the beginning of the research, the GUI requirements were gathered and structured into user stories, which were then mapped to an iteration plan. In the first iteration, a GUI design was generated using two AI tools: Vercel V0 and Galileo AI. In the subsequent iterations, the GUI features were generated in code according to the iteration plan. The GUI development workflow was documented in detail in the results chapter.

Although AI significantly accelerated especially the early stages of GUI development, it still has notable limitations, such as inability to manage large contexts, occasional unintended code modifications, and challenges in integrating the AI-generated code into existing codebases. Leveraging AI in software development is still a relatively manual process, as it requires writing numerous prompts, reviewing changes, and manual testing to achieve the best results.

Based on the results of the action research, an autonomous UI code generation process utilizing LLMs was proposed. In this process, AI would be used to generate tests prior to generating code for the required features. These tests would be executed whenever new code is applied to the codebase, with the context being updated in the background. Although this process could eliminate some of the manual work involved in AI-powered UI development, having clear requirements, continuous iteration, and validation remain essential aspects of software engineering.

Keywords: code generation, large language model, artificial intelligence, graphical user interface, software engineering

TIIVISTELMÄ

Kristian Skogberg: Käyttöliittymäkoodin generointi tieteellisille komentorivityökaluille kielimallien avulla
Diplomityö
Tampereen yliopisto
Ohjelmistotuotannon syventävät opinnot
Toukokuu 2025

Tässä diplomityössä tutkitaan tekoälyn ja kielimallien nykyisiä kykyjä ja rajoituksia käyttöliittymäkoodin generoimisessa. Diplomityön toimintatutkimuksessa (engl. action research) kehitettiin graafinen käyttöliittymä VeRyPy-nimiseen tieteelliseen Python-kirjastoon, jonka avulla voi ratkaista ajoneuvojen reititysongelmia. Käyttöliittymäkoodi generoitiin GitHub Copilotin ja OpenAI:n GPT-4o-kielimallin avulla.

Tässä toimintatutkimuksessa VeRyPy-käyttöliittymän kehitysprosessi toteutettiin viidessä iteraatiossa, joissa noudatettiin suunnittelun, toiminnan, analyysin ja johtopäätösten syklejä. Tutkimuksen alussa käyttöliittymän vaatimukset kerättiin ja jäsenneltiin käyttäjätarinoiksi (engl. user story), joiden pohjalta luotiin iteraatiosuunnitelma. Ensimmäisessä iteraatiossa luotiin käyttöliittymäsuunnitelma Vercel V0 ja Galileo AI -tekoälytyökalujen avulla. Myöhemmissä iteraatioissa käyttöliittymän toiminnot toteutettiin koodina iteraatiosuunnitelman mukaisesti. Käyttöliittymän kehitysprosessi dokumentoitiin yksityiskohtaisesti tulokset-luvussa.

Vaikka tekoäly nopeutti merkittävästi erityisesti käyttöliittymän kehitystä alkuvaiheessa, sillä on edelleen merkittäviä rajoituksia. Näitä rajoituksia ovat esimerkiksi kielimallien kyvyttömyys hallita suuria konteksteja, satunnaisia ylimääräisiä koodimuutoksia ja haasteita generoidun koodiin integroimiseen olemassa oleviin koodikantoihin. Tekoälyn hyödyntäminen ohjelmistotuotannossa on edelleen melko manuaalinen prosessi, sillä se edellyttää lukuisten syötteiden (engl. prompt) kirjoittamista, jatkuvaa muutosten tarkistamista ja manuaalista testausta parhaan lopputuloksen saavuttamiseksi.

Toimintatutkimuksen tulosten seurauksena muodostui ehdotus autonomisen käyttöliittymäkoodin generointiprosessiin kielimallien avulla. Tässä prosessissa tekoäly generoisi ensin testejä vaadituille toiminnallisuuksille, jonka jälkeen generoitaisiin koodia. Nämä testit suoritettaisiin aina, kun uutta koodia lisätään koodikantaan ja kontekstia päivitettäisiin jatkuvasti taustalla. Vaikka tämä prosessi voisi mahdollisesti automatisoida osan tekoälyavusteisen käyttöliittymäkehityksen manuaalisesta työstä, selkeiden vaatimusten, jatkuvan iteraation ja validoinnin merkitys vaikuttaisi säilyvän oleellisena osana ohjelmistotuotantoa.

Avainsanat: koodin generointi, kielimalli, tekoäly, graafinen käyttöliittymä, ohjelmistotuotanto

USE OF ARTIFICIAL INTELLIGENCE IN THIS WORK

Artificial intelligence (AI) has been used in generating this work:

☐ No
☒ Yes

I hereby declare, that the AI-based applications used in generating this work are as follows:

Application	Version
OpenAI ChatGPT	1.11.2024 – 14.04.2025
GitHub Copilot	v1.263.0 – v1.267.0
Vercel V0	9.12.2024 – 22.12.2024
Galileo AI	8.12.2024

Purpose of the use of AI

Utilizing AI tools was at the core of the action research component of this thesis. At the beginning of the research, Vercel V0 and Galileo AI were used to generate a GUI design for the VeRyPy GUI. During the later iterations, GitHub Copilot and OpenAI's GPT-4o model were used to generate the UI code for the VeRyPy GUI. Details on how these AI tools were used are documented in the results chapter.

Parts of this work, where AI was used

In addition to using AI in the action research of this thesis, ChatGPT was used to search for relevant sources and to identify potential grammatical errors throughout the text. It also helped generate the initial versions of Table A.1 in Appendix A and the iteration plan in Appendix B.

Acknowledgement of risks

I hereby acknowledge, that as the author of this work, I am fully responsible for the contents presented in this thesis. This includes the parts that were generated by an AI, in part or in their entirety. I therefore also acknowledge my responsibility in the case, where use of AI has resulted in ethical guidelines being breached.

PREFACE

I would like to express my gratitude to Dr. Jussi Rasku for his guidance, interest, and role as examiner throughout this thesis. I also thank my second examiner, Prof. Jyrki Nummenmaa, for their valuable time and feedback during the evaluation process. I am grateful to my comfortable office chair at home, which made long hours of writing more pleasant, as well as to my friends and family for their continuous support. Finally, I extend my thanks to GPT-Lab at Tampere University for providing the thesis topic and a license to use OpenAI's latest large language models.

Tampere, 7th May 2025

Kristian Skogberg

CONTENTS

LIST OF FIGURES

GLOSSARY

AI	Artificial Intelligence
AR	Action Research
Context Window	The maximum amount of text (measured in tokens) that a large language model can process at once to generate a response
Copilot	An AI assistant that provides an interface for interacting with large language models to provide suggestions and help users complete their tasks efficiently
GUI	Graphical User Interface
Heuristic	A problem-solving method for finding approximate solutions fast and efficiently
LLM	Large Language Model
UI	User Interface
VeRyPy	A Python library for solving CVRPs using heuristic algorithms
VeRyPy GUI	A graphical user interface for the VeRyPy library
VRP	Vehicle Routing Problem

1. INTRODUCTION

Most if not all industries are changed forever by the adaptation of artificial intelligence (AI) and large language models (LLMs) [1, 2, 3, 4]. Utilizing LLMs and AI tools can greatly increase productivity [5, 6], when some tasks that once took hours or longer can now be completed in a fraction of the time. Moreover, as AI tools and LLMs improve, they are becoming more capable of completing longer and more complex tasks in less time [7]. Therefore, having some understanding of the available AI tools and how to utilize them has become essential to improve efficiency in almost every field. In this thesis, LLMs and AI tools are put to the test by generating a graphical user interface (GUI) for a scientific command line tool.

Scientific tools are usually designed by researchers and scientists, and while they can provide a lot of data and insight, most of the time they are not designed user experience in mind. In other words, usability is not the main focus of scientific tools. Poor usability significantly reduces adoption, as regular users may find it difficult to use these tools. A solution to help especially the non-technical users to use a system more easily is to design and develop a GUI which is an essential part of almost any kind of software today.

This thesis is divided into six main parts. The background chapter introduces key concepts relevant to this thesis, such as requirements engineering, user interface design, large language models, AI copilots, related work, and the vehicle routing problem (VRP). The methods chapter outlines the action research approach used to generate a GUI for VeRyPy, a scientific command line tool for solving VRPs [8], using AI tools. The system under study chapter presents the VeRyPy library and describes the VeRyPy GUI requirements and features. The workflow of the VeRyPy GUI generation process is documented in the results chapter. The findings chapter includes the strengths, current limitations, and overall experience of AI-powered UI code generation observed during the action research. Finally, the discussion chapter compares the findings with related work and proposes a more autonomous UI code generation process as a direction for further research.

The topic for this thesis was provided by GPT-Lab. GPT-Lab is a research hub focused in AI, and it was founded at Tampere University in June 2023. GPT-Lab aims to study how generative AI can be utilized to the fullest in the software engineering industry [9]. GPT-Lab offers solutions for companies looking to integrate AI in their processes through a sandbox environment where businesses can test AI-powered solutions and by connecting businesses with AI researches. Using LLMs to generate UIs autonomously aligns with GPT-Lab's research interests, and this thesis aims to provide valuable insights and ideas about the current state of AI-powered UI generation.

2. BACKGROUND

2.1 Software Engineering Process

Software engineering is a relatively unique field compared to other engineering fields as software systems are abstract and not limited by physical constraints. Well-designed software systems can usually be modified relatively easily, quickly, and cost effectively, even after being deployed to production. Therefore, software development is a very iterative process where changes occur all the time during the software life cycle.

There are countless ways in which different organizations and teams build software, but agile methods are usually present in one form or another. For instance, scrum is a popular agile framework where development is divided into fixed-length iterations or sprints, typically lasting two to four weeks [10]. This structure helps teams prioritize and manage their work effectively. Kanban is another agile method that emphasizes continuous development by using visual tasks to help manage the workflow [11]. Today, most software is developed in iterative cycles, where each sprint begins with prioritizing features from the product backlog, followed by estimating the time required for each task, and concludes with the development of those features [12].

Because software development has such an iterative nature, it is easy to spend a lot of time discussing changes in meetings, which takes time out of implementing the software and its features. In 2024, Rasheed et al. [13] presented an idea for a workflow to improve productivity in software development using AI and LLMs. Their vision is to have multiple AI agents executing tasks and constructing a working software in the background of meetings based on what is discussed in the meeting. This kind of process would accelerate software development, because working code would be generated already during the meeting instead of taking place after the meeting.

2.2 Requirements Engineering

Collecting requirements and specification for a system plays a central role in software engineering. Requirements are descriptions and statements of how the system should behave, and they are outlined at the early stage of development [12]. In other words, collecting and understanding the requirements of a system helps guide the software development in the right direction. To minimize misunderstandings and prevent using resources on unnecessary features, requirements should be clear, unambiguous, and easily understood by all stakeholders.

Epics and User Stories

A common approach to modeling requirements of a system is to present them as epics and user stories. In the software development context, epics are large chunks of work representing the major requirements of a system. Epics are usually broken down into smaller tasks known as user stories. User stories are concise sentences that describe the desired functionality and scenario of a system from the perspective of the end user [14, 12]. User stories typically outline a type of user, their goal of using the system, and the benefit they get after achieving their goal. User stories are usually written in the following Connextra format [14]:

As a [role], I want to [goal], so that [benefit].

Due to the simple structure, user stories are designed to be easily understood by all stakeholders, even without technical knowledge. Developers can design and implement features that correspond to the goals and benefits of different types of user described in the user stories. Furthermore, user stories can be useful reminders for developers to check if the implemented features correspond to the user stories and what kind of features are required but not yet implemented.

Each user story is usually assigned estimates based on their complexity and how much effort implementing them would require. These estimations are known as story points, and they can be a useful way of planning and prioritizing the development of different features.

2.3 User Interface Design

User interfaces (UIs) play a crucial part in software development as they allow users to easily use a system. UIs are closely linked to the usability of a software, despite the fact that UI design can be somewhat decoupled from the software development life cycle [15]. This suggests that UI design may not always be emphasized enough, even though it is a key factor that directly impacts the usability of a system. Incomplete or unclear UIs are difficult to use, which can hinder usability and delay the adoption of the software.

The UI design process consists of multiple tasks for different purposes, but it can be simplified into four key phases [16, 17, 18] as shown in Figure 2.1. In the first phase, potential users are identified along with the tasks they will perform using the UI, as well as the environment in which these tasks will take place. This analysis provides the data needed for the next phase, where the UI is designed. It is crucial to design the interface in a way that helps users complete tasks more easily while boosting their productivity. The third phase involves implementing the UI based on the design. This implementation is typically an interactive prototype, as it is quicker to create and modify than a full implementation in code. Finally, in the fourth phase, the implemented UI is evaluated and validated. Evaluation can be done, for example, by having users test the UI and observing or interviewing them to identify what they found useful and what could be improved in the UI.

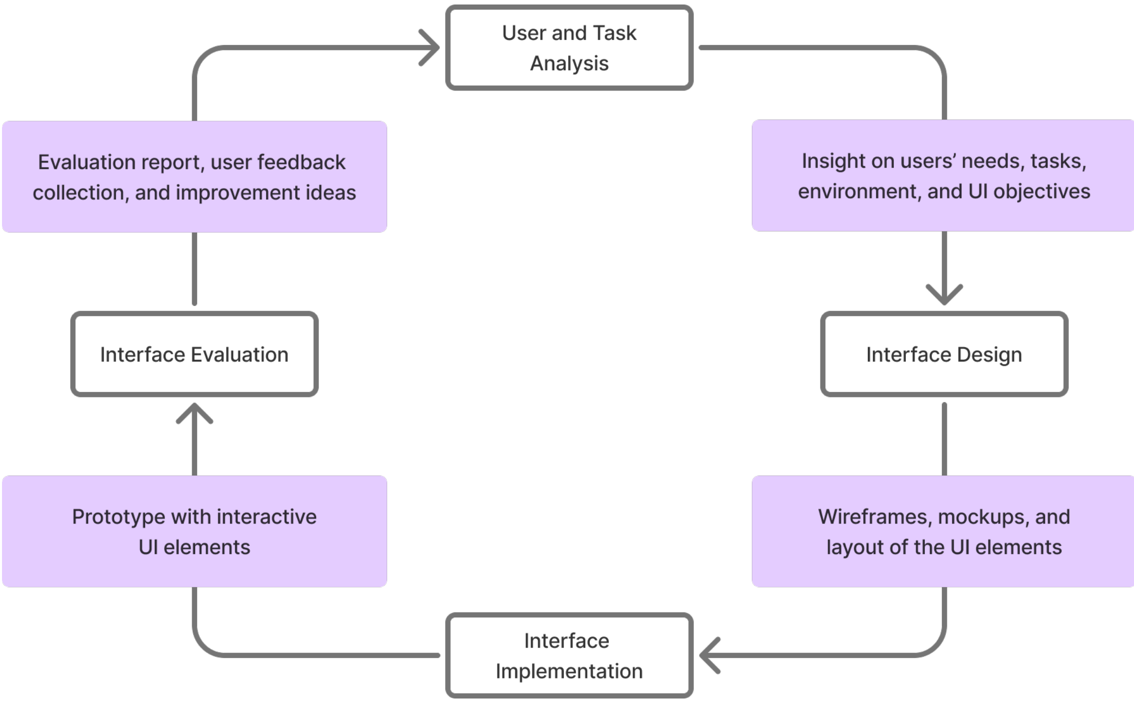


Figure 2.1. *User interface design process, inspired by [16, 17, 18].*

After the evaluation is completed, it is generally recommended to improve the UI through a few iterations of the design process based on user feedback rather than using the initial design as the final design. This claim was supported by Nielsen [19] where they studied iterative UI design and presented four case studies of iterative design. They measured UI design improvement in percentages by considering attributes such as task completion time, errors made, and subjective satisfaction. Across these case studies, Nielsen found that the median improvement of an UI after three versions was 165 % and the average improvement from one version to the next was estimated to be 38 %. Their findings highlight the importance of continuous improvement based on user feedback. Nielsen found that the greatest improvements came from the first iterations, which makes sense considering that early iterations tend to address the most obvious usability issues.

Software engineers often prioritize the implementation of required functionalities and core business logic over increasing usability through intuitive UI design. However, there are signs that this trend is changing where UI design and implementation is emphasized more. Myers and Rosson [20] conducted a survey with 74 responses, where they found that developers spend on average 48 % programming the user interface and that UI development is emphasized in most of the phases of the software life cycle.

A well-designed UI has a great impact on the sales and profitability of a software system. If users are able to accomplish their tasks easily and conveniently using the UI, they are likely to continue using it and potentially even recommend it to others. Lohse and Spiller [21] studied how the user interface of an internet retail store affects traffic and sales in 1999. They found that product list navigation features which reduce the time to purchase products accounted for 61 % of the variance in monthly sales. This suggests that intuitive UI design where users can easily search and browse products would increase sales greatly. If users find UIs confusing or difficult to use, they may choose to switch to a competitor's system, causing the business to lose a potential customer in the process.

Pratama and Cahyadi [22] also studied the effect user interface and user experience have on application sales. They compared the UI and features of two very similar note taking applications for mobile. One of the application has a minimal and appealing UI where the UI of the other application looks more complicated and lacks consistency in some of the UI elements. Even though these applications have very similar functionality, the one with a better appearance has significantly more downloads and higher ratings. While other factors such as marketing and search engine optimization also play a role in download numbers, it can still be said that an intuitive, well-designed UI has a significant impact on the software's popularity and profitability.

2.4 Artificial Intelligence in Software Development

Artificial Intelligence (AI) has a huge impact on how software are developed today. An individual developer can assemble a complete team of AI agents assigned for different tasks such as idea generation, requirements engineering, code generation, and testing. AI has also made it easier to create software for individuals who may lack the otherwise required technical skills in software development.

However, leveraging AI to the fullest requires a certain level of technical knowledge. Writing clear prompts, requesting specific technologies, validating the AI outputs, and integrating them into a codebase is necessary to produce high-quality software. AI can be particularly useful to explore new ideas and in automating repetitive tasks, which in turn frees up more time for developers to focus on problem solving and other higher-level aspects of software development.

AI has already been integrated with some of the most used code editors. For instance, Visual Studio offers IntelliCode AI assistant for C# in Visual Studio 2022 and Python in Visual Studio Code [23]. Pycharm, which is one of the leading code editors for Python projects, also has integrated AI directly within their code editor [24]. Cursor [25] is an example of a trending AI-powered code editor, which has been designed to leverage LLMs in code generation. Even popular design tools such as Figma have integrated AI into their products where users can generate designs using LLMs [26].

2.4.1 Large Language Models

Large language models (LLMs) are the most common application of AI. LLMs are complicated systems trained on vast amounts of data to be able to understand and generate outputs in natural language without requiring a specific input format. LLMs use deep learning techniques to process large datasets and predict text by modeling complex language patterns and dependencies. Examples of some popular LLMs include OpenAI's GPT-4o, DeepSeek's R1 and Google's Gemini.

Due to their broad use cases, most popular LLMs can be used in all stages of software development: from ideas to planning, from code generation to testing, and even for deployment and maintenance [27, 28, 29, 30]. However, while LLMs can be used throughout the entire software development process, they are not always the most suitable choice for every task. Specialized AI tools and models trained for specific purposes, such as code generation, UI design, and requirement analysis, often provide more precise and efficient solutions for those particular tasks.

Context Windows in LLMs

One of the main drawbacks of using LLMs in software development today is their inability to manage large contexts. The context window refers to the maximum number of tokens that an LLM can process simultaneously [31, 32], including tokens used in both input and output. For example, if a model has a context window of 128 000 tokens and 100 000 tokens are used for input, then up to 28 000 tokens remain for output generation. The context window is essentially the working memory of an LLM, and it affects how long the LLM can keep track of the conversation before it starts to forget previous messages [32].

While LLMs generally perform well in short and straightforward contexts, their accuracy tends to decline as context size increases. This issue is particularly evident in code generation, where there are often numerous source files and folders, and understanding the full context is necessary for producing accurate results. For example, using LLMs to generate a function for fetching data is typically manageable, but integrating that data with other components may exceed the model's context window, often leading to hallucinations or unintended results. Due to the limited context window size of LLMs, it is often advised to split complex tasks into multiple smaller tasks where the LLM would complete them one by one. This approach was also suggested in GitHub Copilot's documentation [33].

The current context windows of some commonly used LLMs are shown on a logarithmic scale in Figure 2.2. Among these models, Google's Gemini 1.5 Pro model has a context window of up to 2 million tokens [34], where Anthropic's Claude 3.7 Sonnet and OpenAI's o3-mini models have a context window of 200 000 tokens [35, 36]. Other popular models can manage up to 128 000 tokens, which is about 96 000 English words on average [37].

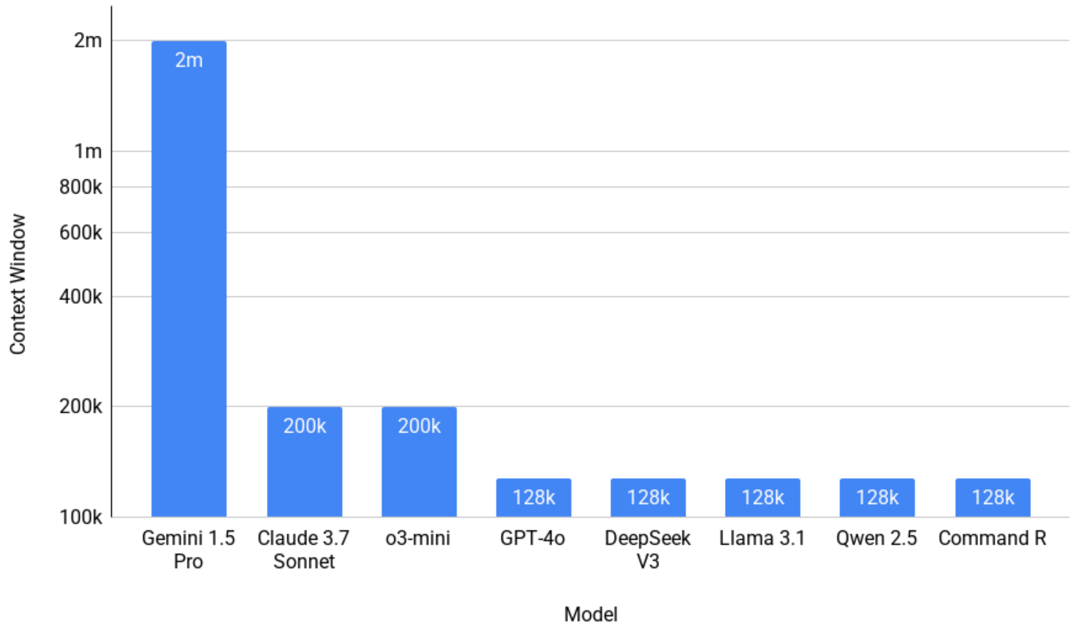


Figure 2.2. *Context windows of commonly used large language models.*

Although context windows exceeding 100 000 tokens may seem substantial, only a portion of this capacity is available for output generation. For example, the GPT-4o model offers a context window of 128 000 tokens, yet it can use a maximum of 16 384 tokens in the output. However, the ratio between the context window size and the maximum output tokens varies significantly across different LLMs. For instance, the o3-mini model provides a context window of 200 000 tokens and can use up to 100 000 tokens for output generation. Similarly, Claude 3.7 Sonnet supports up to 128 000 output tokens while also having a context window of 200 000 tokens.

The context windows of LLMs have increased rapidly in recent years. Figure 2.3 shows the growth of context windows in OpenAI's popular GPT models from 2018 to 2024, visualized with a scatter plot and a trendline. The GPT-1 model was released in 2018 and had a context window of just 512 tokens [38], which increased to 1024 tokens in GPT-2 [39] and 2048 tokens in GPT-3 [40]. Meanwhile, the newer GPT-4o model offers a significantly larger context window of 128 000 tokens. This trend suggests that future LLMs will likely have larger context windows than the previous models.

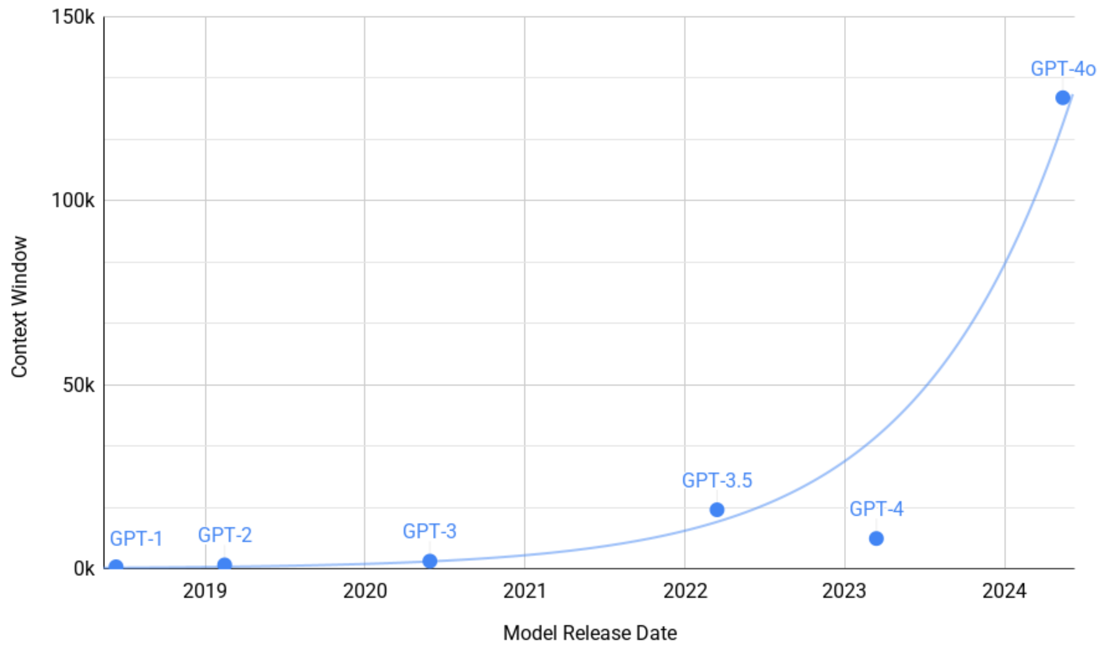


Figure 2.3. *Evolution of context windows of OpenAI's GPT models.*

In practice, the effective context windows of LLMs are often much smaller than what is advertised. In 2024, Hsieh et al. [41] at NVIDIA introduced RULER, a benchmark for evaluating context windows of long-context language models. They found significant performance drops across almost all evaluated models as the context size increased. Additionally, most of the evaluated models fell well below their advertised context windows. For example, the effective context size of OpenAI's GPT-4 and Meta's Llama3.1 (70B) models was evaluated to be 64 000, which is 50 % less than advertised. This suggests that the advertised context windows of LLMs may serve more as a marketing tactic, and relying on them can be misleading when designing processes around LLMs.

Additional Constraints of LLMs

In addition to the limited context windows, LLMs have other major constraints, such as expensive training costs, the quality of the training data set, and the lack of executive power [42]. The quality and accuracy of responses generated by LLMs depend on the quality and extent of their training datasets. Additionally, since generative AI models must be trained before they can be used, their training data will always be somewhat outdated, unless the model is trained again with updated data. In other words, the quality and timeliness of the training data directly impacts the accuracy of AI-generated responses, making high-quality datasets very valuable resources.

Building and training LLMs is expensive. For instance, training the DeepSeek V3 model cost $5.576 million [43], while training OpenAI's GPT-3 model was estimated to cost over

$4.6 million [44]. Furthermore, the training cost for OpenAI's newer GPT-4 model ranges from $63 million to over $100 million [45, 46]. These significant costs are primarily due to the extensive computational resources, such as graphics processing units (GPUs), required to train the models. As LLMs continue to advance, training cost is likely to remain a critical consideration in their development and practical use.

The lack of executive power means that LLMs cannot independently execute actions using their generated responses. For example, while LLMs can generate content such as text, code, or images, it is usually up to the user to manually copy and paste the output into their application of choice. However, AI copilots designed for specific purposes, along with plugins and application programming interfaces (APIs), can bridge the gap between LLMs and action execution. These actions can include, for example, creating and scheduling social media posts [47], improving digital marketing [48], writing emails and autocompleting sentences [49] as well as automating invoice processing [50].

2.4.2 AI Copilots

AI copilots, also referred to as AI assistants or AI tools, are essentially user interfaces for accessing and using LLMs in different applications. AI copilots are typically designed to have chat-like conversational UIs that enable interaction with LLMs using natural languages [42, 51]. This is likely one of the reasons why AI copilots have been rapidly adopted across various industries, as using them is easy and does not require technical knowledge or familiarity with specific syntax.

While OpenAI's ChatGPT is one of the most well-known AI tools used for a wide range of tasks [52], GitHub Copilot is one of the earliest and most widely used AI copilots in code generation, with over 36 million installations on the Visual Studio Code Marketplace [53]. GitHub Copilot, developed by GitHub in collaboration with OpenAI and Microsoft, is trained on a wide range of programming languages and source code from publicly available sources, such as public code repositories on GitHub [54]. GitHub Copilot allows users to select and use LLMs from other companies, such as OpenAI and Anthropic, and it is available in popular code editors such as Visual Studio Code, Visual Studio, and Pycharm. Other examples of AI copilots in code generation include Aider, Cursor, and Supermaven.

Figure 2.4 illustrates how users can interact with LLMs through AI copilots. First, the user provides input in natural language via the AI copilot interface, which is known as a prompt. The AI copilot then sends the prompt to the LLM, where it is processed, and a response is generated. Response generation is a complex process in which a neural network predicts the next word in a sequence based on the given context. Once processed, the generated response is returned to the AI copilot and displayed to the user in the interface.

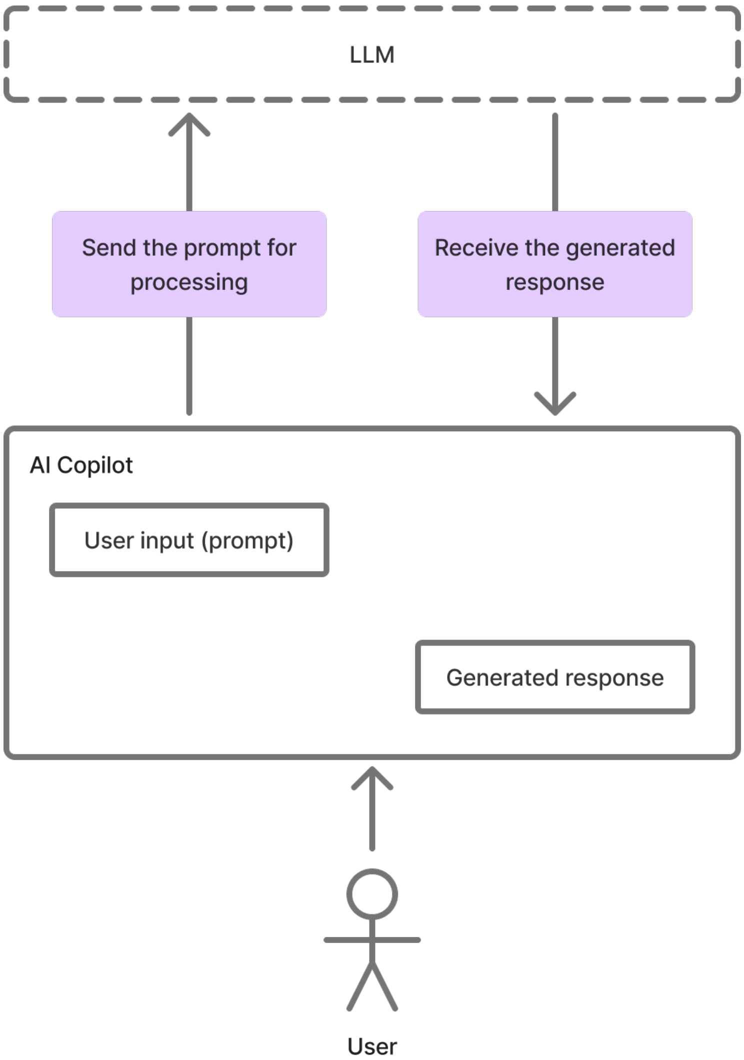


Figure 2.4. *Interaction between user, AI copilot, and LLM.*

2.4.3 Related Work

Numerous studies have explored how LLMs can be used to improve the software development process. Prather et al. [55] studied how novice programmers use GitHub Copilot on typical programming assignments, and they recognized two primary interaction patterns. Some programmers manually typed out Copilot's code suggestions and then adapted the code based on their needs. Other programmers were fast to accept Copilot's suggestions without really understanding them, resulting in more time spent debugging the AI-generated code.

Pandey et al. [56] also studied the efficiency and challenges of using GitHub Copilot in real-world software projects. In their study, a mix of 26 junior and senior engineers used GitHub Copilot in daily tasks within a full stack cloud development environment. They found that GitHub Copilot was particularly useful for generating boilerplate code, debugging, and providing code explanations, suggestions, and examples. Code quality improved significantly when context was provided through files and using descriptive naming in functions and variables. The identified key challenges were the inability to generate code for a unique business logic and distributing the code to multiple files.

Bilgram and Laarmann [57] studied how ChatGPT-4 can be used to help especially people without much UI and UX knowledge to create a working prototype. They found that ChatGPT is able to generate easy-to-follow instructions for performing specific tasks such as designing a UI that opens in a web browser. Additionally, LLMs were proven helpful for guiding the user through the task and fixing errors along the way. They also highlighted the importance of task definition, requirements, and workflow integration which directly affects the outcome of AI-assisted prototyping. Finally, they suggested that organizations and teams should rethink they way they use LLMs in their workflows as it is now much faster to make edits and experiments than to define the requirements and contexts.

One interesting aspect of using AI copilots in software engineering is how much different descriptions and prompts of the same outcome affect the generated code. Mastropaolo et al. [58] did an empirical study where they asked GitHub Copilot to generate 892 Java methods from the original Javadoc descriptions. They generated equivalent descriptions for the same methods and found different code generations in about 46 % of cases. Furthermore, they found that about 28 % of code generated using the equivalent descriptions was incorrect or did not pass some of the tests. This suggests that writing clear prompts plays a crucial role in generating code that produces the desired result.

One of the biggest issues in using LLMs is hallucinations or outputs that are incorrect or contain false or misleading information. Liu et al. [59] studied this problem and they developed a taxonomy for categorizing different kinds of hallucinations in code generations. They found that the selected models (GPT-3.5 turbo, Codellama-7B and DeepSeek-Coder-7B) struggle to identify hallucinations and especially to mitigate them. This highlights the need to develop specialized methods for managing hallucinations in AI-generated code as well as further research in this area.

Liang, Yang, and Myers [60] conducted a large-scale survey with responses from 410 developers. They found that one of the biggest issues in using AI copilots in programming was *giving up on incorporating the outputted code*. Developers reported that the reason for sometimes not using AI-generated code was that *the code does not perform the correct action or it does not meet functional or non-functional requirements*. In other words, the generated code does not always align with the developers intent, and therefore it may sometimes be more convenient to write code manually instead of relying on AI copilots.

Xu, Vasilescu, and Neubig [61] studied the promise and challenges of using in-IDE code generation from natural language by developing a plugin for PyCharm. They found that while developers had a positive experience of using the plugin, *there were no statistically significant gains in any measurable outcome when using the plugin*. They also found several issues of using the plugin such as developers usually had to modify the generated code before it could be used, the code often lacked necessary context, and sometimes the results were not related to the search.

Barke, James, and Polikarpova [62] studied how programmers interact with AI copilots for code generation. They found two distinct interaction modes: acceleration and exploration. In acceleration mode, developers know their goal and use AI to reach it faster. Exploration mode is used when developers are uncertain how to proceed and use AI copilots to explore options or get a starting point. They also found that programmers in acceleration mode are quick to discard AI-generated code, where programmers in exploration mode prefer to accept and modify generated code.

Bajcetic, Draskovic, and Bojic [63] researched how a Python Tkinter GUI can be implemented using ChatGPT 3.5 and Google Bard. They defined a specification for the application in natural language written as scenarios. They found that while the generated code provided a good starting point and covered most of the scenarios, the code was sometimes incomplete and contained errors. They also found that assigning functionalities separately did not result in higher quality code, and specifying all functionalities in one go may be the better approach. This specific finding indicates that, while AI tools may work well in the implementation of GUIs for small-scale applications, a more robust process is needed to integrate AI with the development of more complex systems.

Fakhoury et al. [64] studied how test-driven practices could be integrated with AI-powered code generation. They presented a test-driven interactive code generation workflow that guides user intent by automatically generated tests while also improving the accuracy of AI-generated code. They found that incorporating test generation into code generation significantly improved the accuracy of generated code while also reducing task-induced cognitive load. Furthermore, presenting users with test cases and asking for validation resulted in code that is more aligned with their needs, and users were also likely to correctly evaluate the AI-generated code.

Kolthoff et al. [65] studied how effective LLMs are in detecting the implementation of user stories in GUI prototypes and recommending GUI components. They proposed an approach to validate user stories against a GUI prototype. This user story validation was done by extracting components from the GUI and checking if they validate any user story. In order to obtain the required training dataset, they selected 60 existing GUI prototypes and performed a questionnaire to collect user stories for the already implemented functionalities in the selected GUIs. Although the training dataset collected was small, they still managed to get promising results in the validation of the user story and the recommendation of components.

In addition to the previously highlighted studies and frameworks, there are many other workflows that aim to improve the AI-powered UI code generation process. For instance, Mu et al. [66] introduced a framework to detect ambiguous or unclear requirements and generate clarifying questions before generating code. Wu et al. [67] introduced an automated method for fine-tuning LLMs to generate UI code from textual descriptions. Liang et al. [68] designed a strategy for generating HTML code from UI designs and images. Similarly, Wan et al. [69] introduced an approach for converting webpage screenshots into UI code by dividing the screenshots into smaller segments and generating corresponding HTML and CSS code for each segment. The vast number of tools and workflows built around LLMs suggests a growing interest among researchers in improving the AI-powered code generation process.

To conclude the related work, many recent studies show that LLMs can help with common tasks such as generating code, validating user stories, recommending components, and even creating test cases to improve code quality. While the results are often promising, common challenges remain, such as handling vague requirements, generating incorrect, and incomplete code, or code that does not align with the developer's objective. Overall, these studies highlight the potential of using LLMs to assist in different areas of software development, especially for creating and prototyping UIs.

2.5 The Vehicle Routing Problem

The vehicle routing problem (VRP) is a concept in the field of transportation that has been researched since the 1950s. VRPs are essentially mathematical optimization problems in which the goal is to find the optimal routes for a set of vehicles visiting customers at different locations [70, 71, 72]. VRP generalizes the traveling salesman problem (TSP) while also being more complex with additional constraints such as multiple vehicles, vehicle capacity limits, and route length restrictions.

The vehicle routing problem is visualized in Figure 2.5 on the next page. The dots in the figure represent customer locations, while the lines connecting them form the optimal routes. Each route starts from the depot, visits the nearest customers, and returns to the depot. These routes are typically traversed by vehicles to, for example, deliver goods or pick up customers as efficiently as possible.

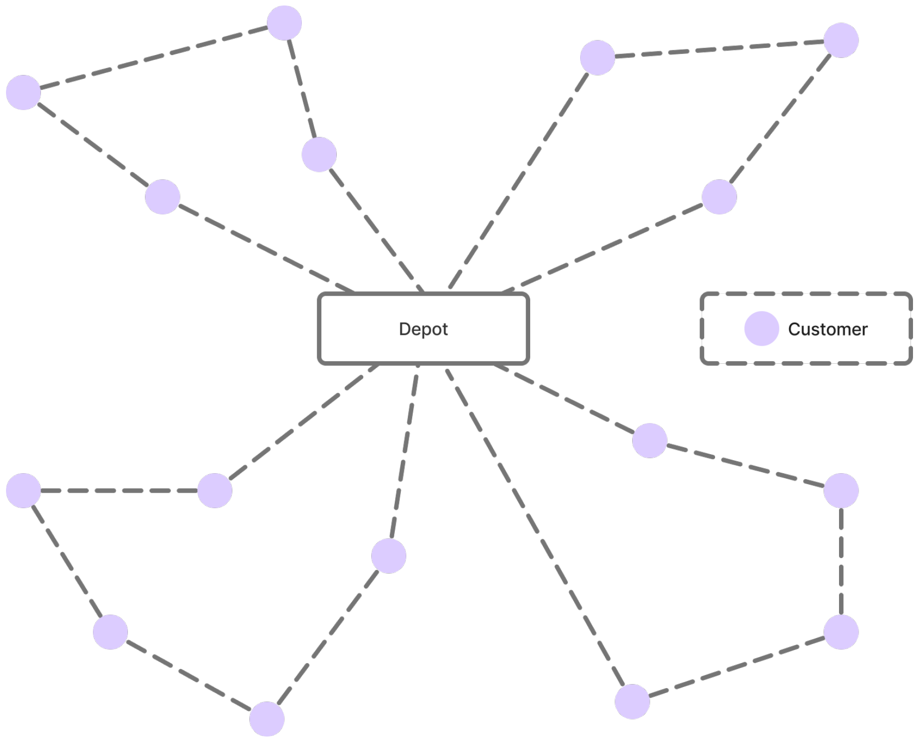


Figure 2.5. *An example of a vehicle routing problem.*

There are a few different variations of the vehicle routing problem. In the capacitated vehicle routing problem (CVRP), all vehicles are identical and share the same capacity and depot [70, 71, 72]. Another variant, the vehicle routing problem with time windows (VRPTW), requires each customer to be visited within a specific time window. Among the different VRP variations, the CVRP is the most widely studied variation of VRP.

2.5.1 Solving VRPs using Heuristics

Calculating the perfect route can be extremely time-consuming and computationally demanding, especially when dealing with multiple vehicles with varying capacities, hundreds of locations, and priority constraints for certain locations. Therefore, using heuristics is beneficial as they enable the computation of optimal or near-optimal routes more efficiently and with fewer computational resources. Heuristics are techniques designed to find not exact but still optimal solutions to problems faster than classical methods.

A well-known heuristic for solving VRPs is the savings algorithm developed by Clarke and Wright in 1964 [73]. In their algorithm, an optimal or near-optimal route is calculated incrementally for vehicles with varying capacities visiting multiple delivery points from a central depot. It starts by assigning each delivery point its own route from the depot. Next, it calculates the potential distance savings from merging two routes. Routes are merged based on the highest savings as long as vehicle capacity and routing constraints are respected. This process continues until no more efficient merges can be made.

2.5.2 Tools for Solving VRPs

There are many tools available for solving VRPs and they can be divided into two categories: open-source and commercial. Open-source tools, such as jsprit [74], VeRyPy [8], VRPy [75] and Google's OR-Tools [76], are commonly used since they are flexible, free to use and support different types of VRPs such as VRPs with time windows or capacity limits. These tools usually use heuristics to find good solutions efficiently.

Commercial tools, such as Gurobi [77] and CPLEX [78], can be used to model VRPs as mathematical optimization problems. These tools can compute exact solutions and are effective in small to medium-sized cases. However, using exact techniques to solve VRPs does not scale well as the problem size or complexity increases [72]. Therefore, exact techniques are used primarily in research settings or as benchmarks.

2.5.3 Scientific User Interfaces for Solving VRPs

There are several scientific tools with UIs created by researchers to solve VRPs. For example, VeRoViz [79] is a Python package which is designed to help visualizing VRPs. In VeRoViz, the first step is to define an area or location coordinates, which are used to generate nodes. VeRoViz uses these nodes to generate a time matrix and a distance matrix. VeRoViz provides a simple solver, but more sophisticated solving is left to the user. The VeRoViz website has a map which can be used to draw a bounding region and coordinates, but the visualization requires installation of the VeRoViz package.

Vroom [80] is another open-source VRP optimization application. Vroom provides a frontend that can be used to add locations to a map and to create a route visiting all locations using real-life roads. On default, Vroom uses one vehicle when solving VRPs, but new identical vehicles can be added in the right-hand side control panel. A breakdown of which vehicles visited which locations can also be seen in the control panel, and each route is drawn in a different color to the map. The route duration, distance, and computing time can be seen in the bottom-left corner of the frontend. However, it is unclear what kind of algorithm is used to solve the VRPs and it is not possible to change the algorithm in the Vroom frontend.

3. METHODS

3.1 Action Research

To select the most suitable method for this research, it is important to understand various research methods. In case studies, a particular subject, entity, or phenomenon are examined in its context within a restricted time frame [81]. Case studies allow researchers to focus on a specific example instead of multiple different cases, and this approach can help to identify core issues for a complex problem. A case study is purely observational, with the goal of gaining insights and better understanding the studied subject.

Design science research (DSR) was also considered as the research method in this thesis. In DSR, an artifact (which is a construct designed to solve a problem within a specific context) is developed to address a particular real-world issue [82, 83]. In other words, DSR focuses on the development of efficient and effective designs over theoretical exploration. However, reviewing the generated UI code is not the primary focus in this research, which is why DSR was not utilized in this research.

Action research (AR) is another commonly used research method. Action and interventions are central to AR, which is typically conducted in iterative cycles of planning, action, evaluating, and reflection [84, 85]. In AR, the researcher actively participates in solving the studied entity or modifying some aspect of it while simultaneously conducting research [81]. Similar to DSR, AR also produces an artifact [83], and AR focuses more on improving processes and is a more action-oriented research method compared to case studies.

Ultimately, a mix of action research and some case study practices was selected as the research method for this thesis. The goal is to get more insights into the process of generating UI code using LLMs, which requires a lot of iteration. Using the selected AI tools requires active participation from the researcher as well as experience in utilizing AI tools and continuous documentation throughout the research. Finally, it is important to study existing applications of using LLMs for generating UIs, which highlights the need for case study methods.

In this action research, the researcher is also the author of this thesis. The researcher is using AI tools to generate UI code while also documenting the experience of AI-powered code generation. Ideally, one researcher would use AI to generate code, while another researcher would observe and interview them to document the process. Therefore, it is important for the researcher to document every step of the VeRyPy GUI generation process in detail to be able to identify the current benefits and challenges of AI-powered code generation.

Action Research Cycle

Figure 3.1 below depicts the cycle used in the action research component of this thesis, consisting of four distinct phases. In the planning phase, the researcher becomes familiar with related work to understand how other researchers have approached UI generation using LLMs. This is important as it provides insight into existing tools, challenges, and best practices. Additionally, the planning phase includes defining user stories and conducting iteration planning for the VeRyPy GUI. The output of the planning phase is knowledge of best practices for using LLMs in UI generation along with the requirements, user stories, and iteration plan for the VeRyPy GUI. These preparations and outputs will be utilized in the next phases of the AR cycle.

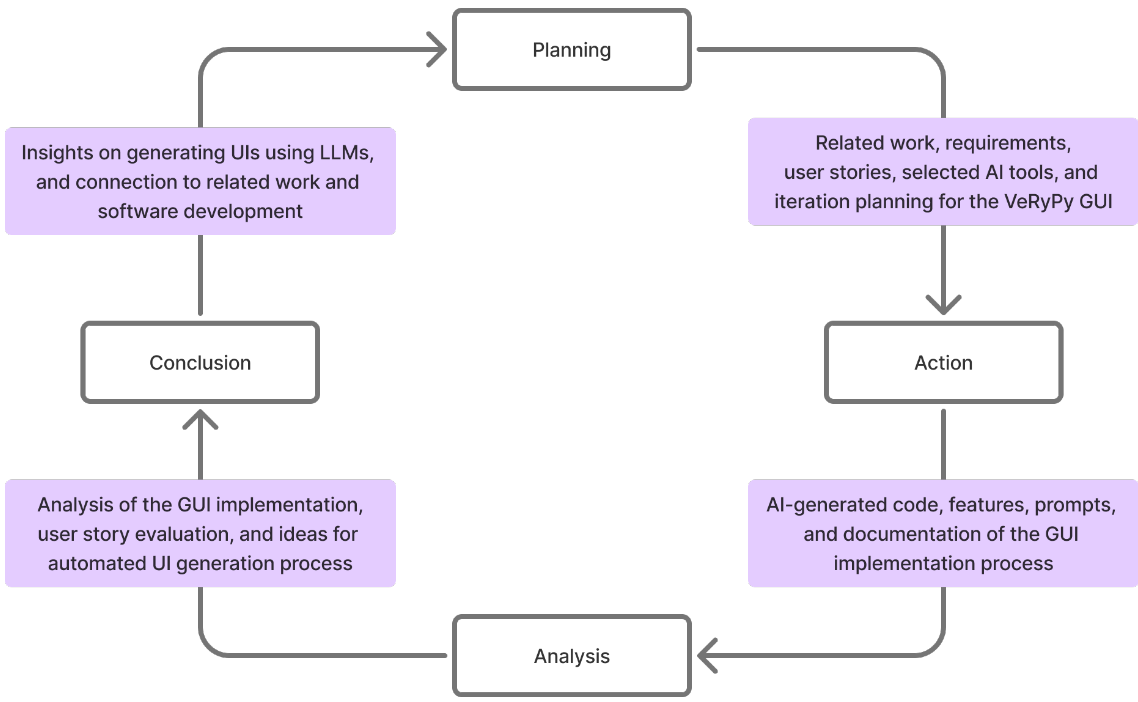


Figure 3.1. *Action research cycle for VeRyPy GUI development.*

The action phase involves implementing the VeRyPy GUI according to the user stories and iteration plan defined in the planning phase. The researcher uses the selected AI tools and methods to implement the GUI in multiple iterations of experimentation and data collection. Therefore, the outputs or artifacts of the action phase include AI-generated code, prompts used, and documentation of the GUI implementation process. It is crucial for the researcher to document the process – i.e. what was done, when, and why – as these insights are needed in the next analysis phase.

In the analysis phase, the researcher reviews the workflow from the previous planning and action phases. The data collected during the action phase is organized to highlight what went well and to identify the challenges and bottlenecks in the code generation process. Additionally, the analysis phase includes validating the user stories and features of the implemented VeRyPy GUI and suggesting ideas for what an automated UI generation workflow could look like.

Finally, the conclusion phase involves compiling the most important insights and highlights from the entire AR cycle to share the acquired knowledge for further research. It is also important to reflect on whether the findings of this AR confirm or conflict with the related work and what improvements or ideas this research can offer in the context of UI code generation in software development.

In this action research, a total of five iterations were carried out to generate the VeRyPy GUI features in stages, and the workflow for each iteration is documented in the results chapter. The number of iterations was fixed during the research to remain within the scope of this thesis. It is also important to note that the study of the related work was done prior to these iterations to gain a better understanding of how other researchers have approached the AI-powered UI code generation process. Furthermore, the most important findings and recurring themes of the VeRyPy GUI generation experience are documented in a separate findings chapter.

Finally, the researcher's knowledge and abilities to utilize AI tools in code generation significantly influence the quality and direction of this research. Having studied Information Technology at Tampere University since 2019 – prior to the widespread adoption of AI tools – the researcher has a lot of knowledge in the software engineering field. The researcher has practical experience with AI tools such as ChatGPT, as well as in designing and implementing user interfaces for both web and mobile platforms. For example, in early 2024, the researcher led a project that heavily incorporated LLMs to develop a gamified website along with multiple personal projects focused on UI design and frontend functionalities.

3.2 Selected AI Tools

There are multiple popular AI tools available from many different vendors. GitHub Copilot and Cursor were identified as the most promising AI tools, and they were compared against each other by reading what developers say about them on online forums like Reddit, Stack Overflow and Medium articles. Developers seem to have mixed feelings for both of these popular AI tools. Some argue that Cursor offers better context awareness, while others claim that GitHub Copilot provides more accurate code generation. In addition to the debate over these tools, the fast pace of improvement in AI tools and the variations in their usage make it challenging to compare them consistently.

Ultimately, GitHub Copilot was selected as the AI copilot to generate code for the VeRyPy GUI while using OpenAI's GPT-4o model in the background. OpanAI's ChatGPT [86] and their GPT-4o model was also used to assist with the user story generation and iteration planning. Vercel V0 AI tool [87] and Galileo AI [88] were used to generate the GUI design for VeRyPy. Table 3.1 below presents the selected AI tools and how they were utilized in this action research.

AI tool	**Usage**
OpenAI's ChatGPT (GPT-4o)	User story generation and iteration planning
Galileo AI	Initial VeRyPy GUI design generation
Vercel V0	Final VeRyPy GUI design generation
GitHub Copilot (GPT-4o)	VeRyPy GUI code generation

Table 3.1. *The selected AI tools and their usage.*

GitHub Copilot and OpenAI's GPT-4o models were selected because they represented the state of the art at the time of writing this thesis. In addition, GPT-Lab at Tampere University, as mentioned in the preface of this thesis, granted the researcher a license to freely use OpenAI's LLMs. GitHub Copilot was also free for students, although it became free for all users of Visual Studio Code (VS Code) during the writing of this thesis [89, 90]. Furthermore, GitHub Copilot is integrated into the VS Code editor, making it convenient for generating features for the VeRyPy GUI. Vercel's V0 was selected based on recommendations from other developers and YouTube videos. Vercel is a well-known vendor that provides developer tools and cloud infrastructure services, such as web application deployment and hosting. Finally, Galileo AI was selected to enable a comparison of GUI designs generated by two different platforms, and it was also recommended by other developers and mentioned in online blog posts.

4. SYSTEM UNDER STUDY

4.1 VeRyPy Overview

In this action research, the system under study is VeRyPy. VeRyPy [8] is a lightweight Python-based open-source library for solving CVRPs. VeRyPy was created by Dr. Jussi Rasku during his PhD thesis conducted in 2019 [91]. VeRyPy contains 15 classical heuristic algorithms, multiple local search heuristics, and some additional constructive heuristics.

Originally, VeRyPy provided a command line interface (CLI) for accessing the implemented heuristic algorithms to solve CVRPs. However, using a CLI is not very intuitive and can be difficult, especially for less technical users. Therefore, a GUI was needed to make VeRyPy more accessible to more people.

Currently, the VeRyPy code repository has been starred by 278 users, forked by 57, and is being watched by 9 users. Figure 4.1 on the next page illustrates the geographic distribution of GitHub users who have starred, forked, or are watching the VeRyPy repository. Users who have starred the repository are represented by orange stars, those who have forked it are shown as dark blue triangles, and users currently watching the repository are marked with gray dots.

Figure 4.1. *Geographic distribution of GitHub users interacting with the VeRyPy repository.*

VeRyPy Codebase

The structure and distribution of the source files in VeRyPy are visualized in Figure 4.2 using a tool created by Wattenberger [92]. The color of the circles represents the file type, while the circle size indicates the file's size. The code is primarily distributed in the *verypy* and *tests* directories with the majority of files being Python files (.py). Most of the code is dedicated to heuristic algorithms and test files.

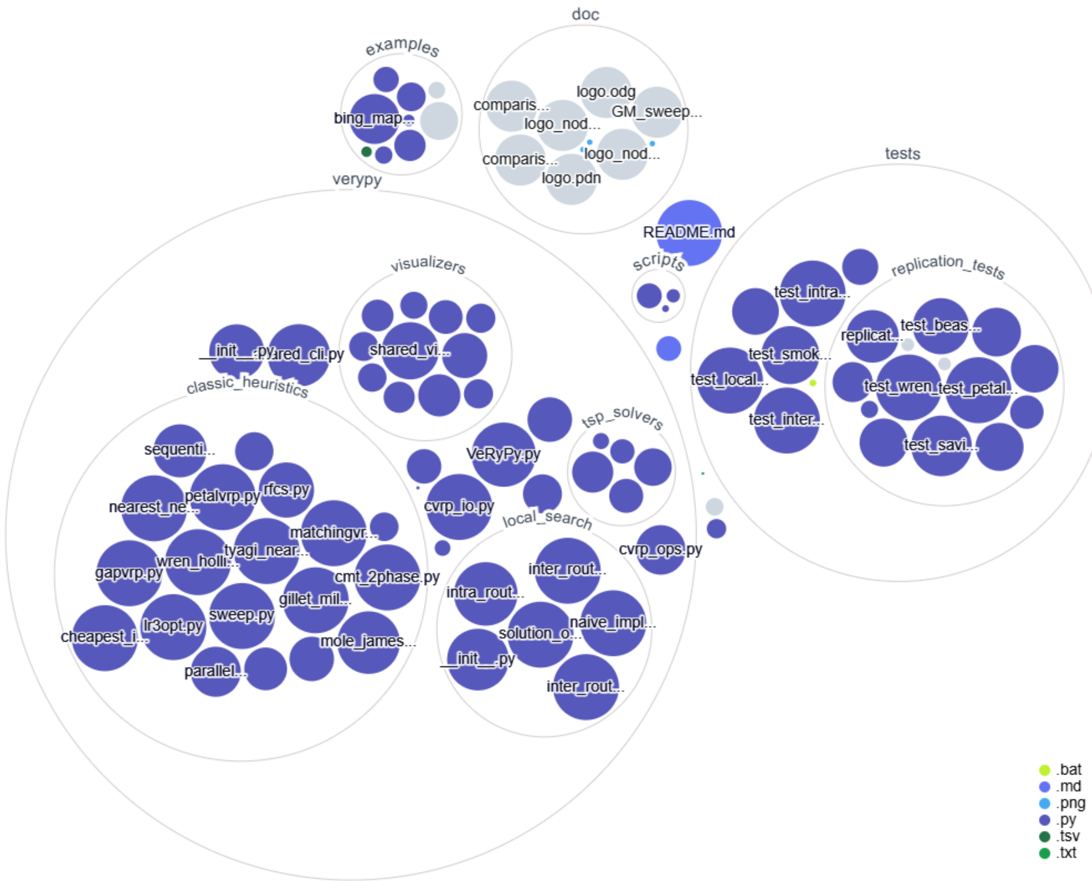


Figure 4.2. *Structure and distribution of source files in VeRyPy.*

Prior to this action research, the lines of code in VeRyPy were calculated using cloc [93], as shown in Figure 4.3 below. Initially, VeRyPy had 13 024 lines of code spread across 83 different files, most of which were Python files. The compact size of VeRyPy reflects its lightweight design philosophy which the GUI should mirror in its implementation.

```
      90 text files.
      83 unique files.
      18 files ignored.

github.com/AlDanial/cloc v 2.04  T=0.62 s (134.9 files/s, 32964.0 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                          79           3095           4027          12837
Markdown                         2            100             37            179
DOS Batch                        1              0              0              5
Text                             1              0              0              3
-------------------------------------------------------------------------------
SUM:                            83           3195           4064          13024
-------------------------------------------------------------------------------
```

Figure 4.3. *Breakdown of VeRyPy lines of code at the start of this action research.*

4.2 VeRyPy GUI Requirements and Features

A set of main requirements for the VeRyPy GUI was identified through discussions with the author of VeRyPy. These requirements are presented as epics in Table 4.1. There are various user roles in the field of routing optimization, ranging from logistics analysts to dispatchers and algorithm developers. For the scope of this research, only dispatchers and algorithm developers were selected as the main users of the VeRyPy GUI. Each of these users approaches VRPs from a different perspective, highlighting the need for diverse use cases and requirements for the VeRyPy GUI.

Table 4.1. *Epics for VeRyPy GUI.*

Role	Epic
Dispatcher	As a Dispatcher, I want to be able to input details about my VRP (e.g. customer locations and vehicle capacities), so that I can model and solve real-world delivery scenarios.
Algorithm Developer	As an Algorithm Developer, I want to select from the heuristic algorithms implemented in VeRyPy, so that I can test and compare different VRP-solving methods.
Algorithm Developer	As an Algorithm Developer, I want to see a visualization and metrics of the routes calculated by the heuristic algorithms, so that I can understand the solutions more easily.

Dispatchers are solving real-world delivery problems by inputting customer locations and vehicle information into the VeRyPy GUI. They represent delivery and logistics companies where their business model requires planning and completing deliveries efficiently. By utilizing the VeRyPy GUI, dispatchers can optimize routes and improve operational efficiency.

Algorithm developers are more technical and research-oriented, focusing on designing methods and algorithms to solve routing problems. They need to experiment and compare different algorithms for solving VRPs by considering their computational time, resources used, and accuracy. Students and researchers studying routing optimizations could also be considered as algorithm developers in this context.

4.3 User Stories and Iteration Planning

The epics presented in Table 4.1 were broken down into 14 user stories, which are listed in Appendix A. OpenAI's ChatGPT and their GPT-4o model were used to assist in defining user stories for the VeRyPy GUI. The intended use of the VeRyPy GUI and the epics were provided to ChatGPT as prompts, instructing it to generate concise user stories. After generating the initial set of user stories, ChatGPT was also used to create a template for the iteration plan, which is detailed in Appendix B.

User stories and iterations that fell outside the scope of the VeRyPy GUI were filtered out, while missing user stories were added to ensure comprehensive coverage of the required functionalities. Additionally, some user stories were refined to align better with the VeRyPy GUI specification.

Each iteration followed an iteration plan that outlined the specific features and user stories to be implemented. The first iteration focused on designing the VeRyPy GUI using the selected AI tools, while the remaining iterations concentrated on constructing the GUI and implementing the required features by generating UI code. The workflow, analysis, and observations of every iteration are documented in detail in the following results chapter.

5. RESULTS

This chapter presents the workflow, observations, and results of a total of five iterations conducted during the action research where a GUI was generated for the VeRyPy library using LLMs and AI tools such as Vercel V0, OpenAI's GPT-4o, and GitHub Copilot. The GitHub Copilot chat logs and the code generated during these iterations are available in the VeRyPy-GUI repository on GitHub [94], which was forked by the researcher from the original VeRyPy repository.

Whenever the term 'Copilot' is mentioned in the following iterations, it refers to GitHub Copilot. Similarly, whenever the term 'VRP' is used, it refers specifically to the CVRP variant. Lastly, the term 'GUI' refers to the VeRyPy GUI.

5.1 Iteration 1: GUI Design

5.1.1 Planning

Refer to Iteration 1: GUI Design in Appendix B for the iteration 1 plan including the user stories and features implemented in this iteration.

The plan of the first iteration is to generate a GUI design for the VeRyPy GUI before generating any code. I will use Galileo AI and Vercel V0 to generate two GUI designs and select the one that best fits the VeRyPy GUI requirements. Once the GUI design looks good, I will convert it to code to prepare it for the next iterations where the VeRyPy GUI features corresponding to the user stories will be implemented in code. It is important to write prompts that clearly describe the main requirements of the VeRyPy GUI to get the most accurate results.

5.1.2 Action

In the beginning of this iteration, I used Galileo AI to generate a GUI design for the VeRyPy GUI. Galileo AI can generate GUI designs separately for mobile and web. First, a mobile design was generated, followed by a web design. The initial GUI design for mobile can be seen in Figure 5.1 and 5.2, while the initial GUI design for web is shown in Figure 5.3 and 5.4. I was unable to share the GUI designs generated in Galileo AI as URLs.

I used the following prompt in Galileo AI: *Design a minimal and easy-to-use user interface for a single page web app for a VRP solver library. The UI should allow users to easily input their VRP problem and select from a dropdown menu a heuristic algorithm and visualizing the result. The UI should allow users to view some statistics of the algorithm result and also be able to playback the solution in stages*. (Note: The playback feature was filtered out at the end of this action research to remain within the scope of this thesis.)

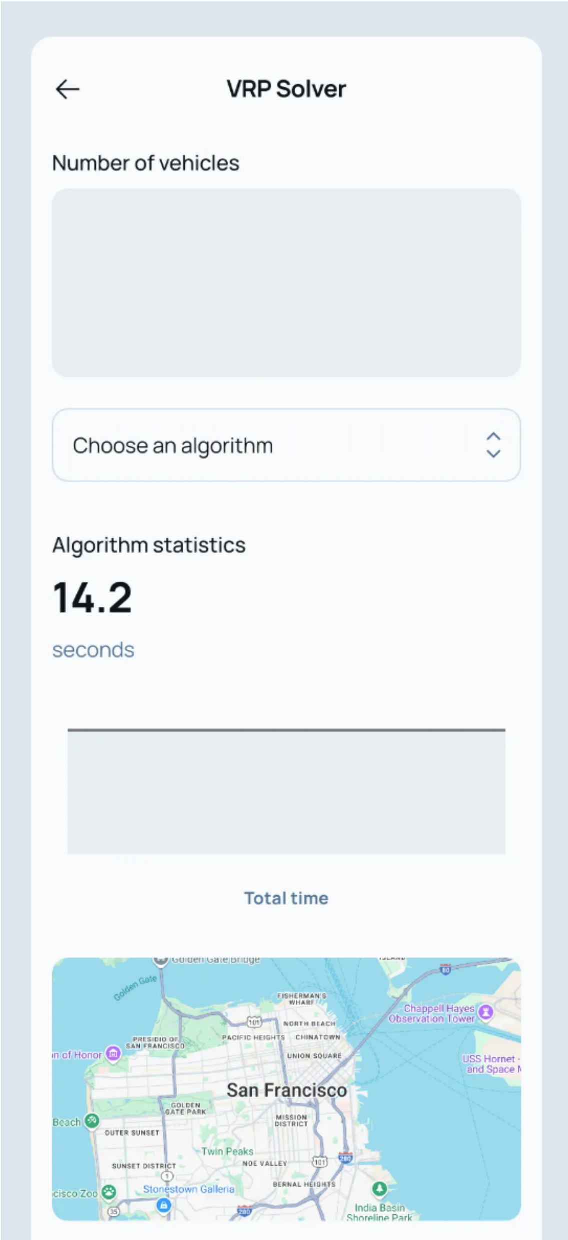


Figure 5.1. *VeRyPy GUI design for mobile generated using Galileo AI (1/2).*

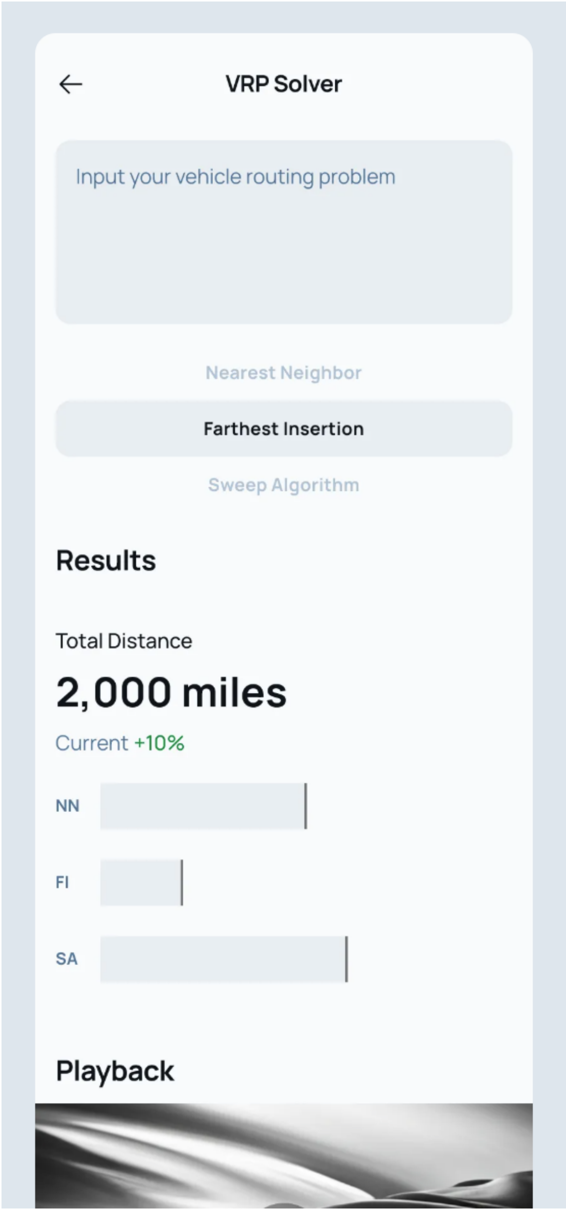


Figure 5.2. *VeRyPy GUI design for mobile generated using Galileo AI (2/2).*

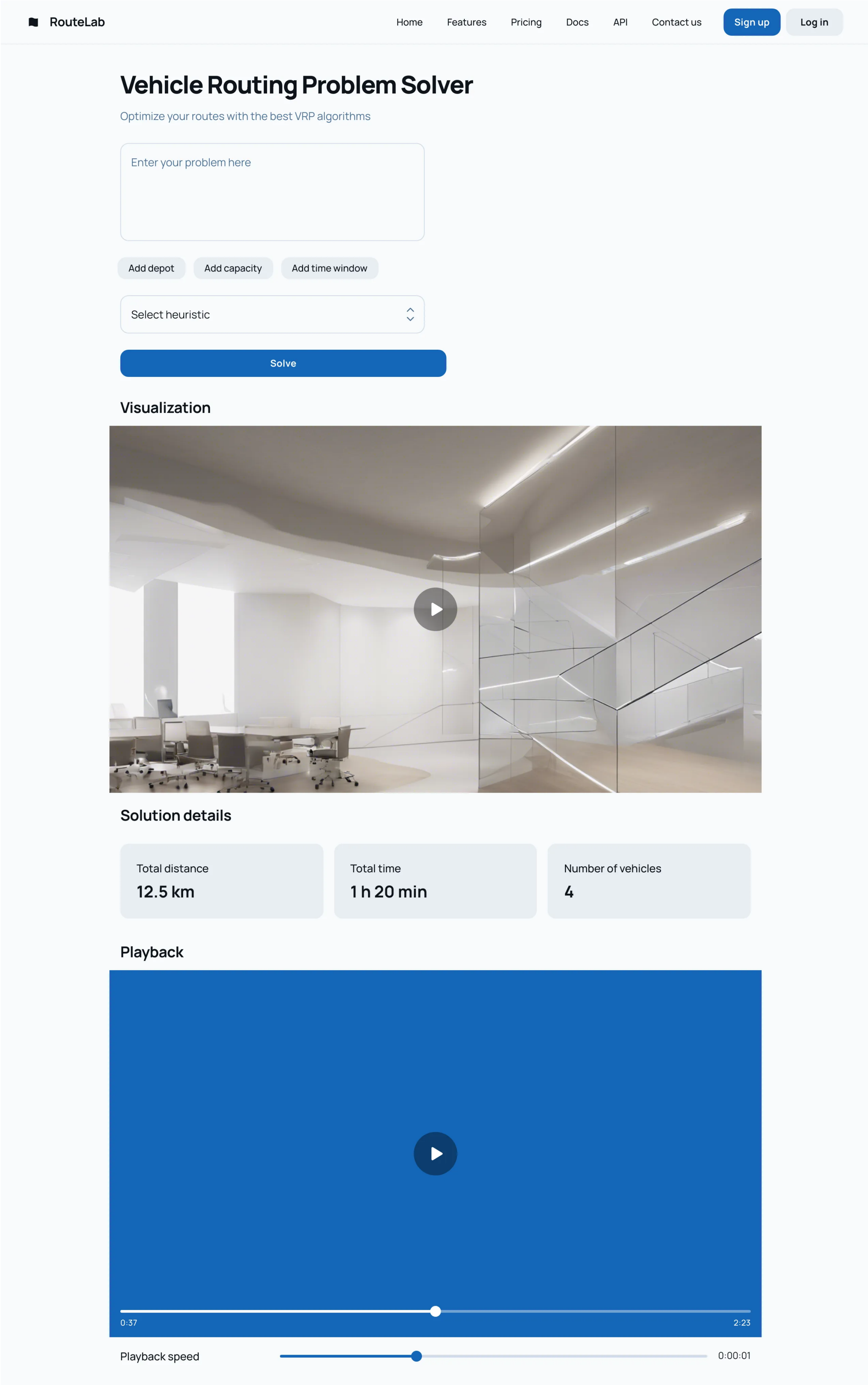


Figure 5.3. *VeRyPy GUI design for web generated using Galileo AI (1/2).*

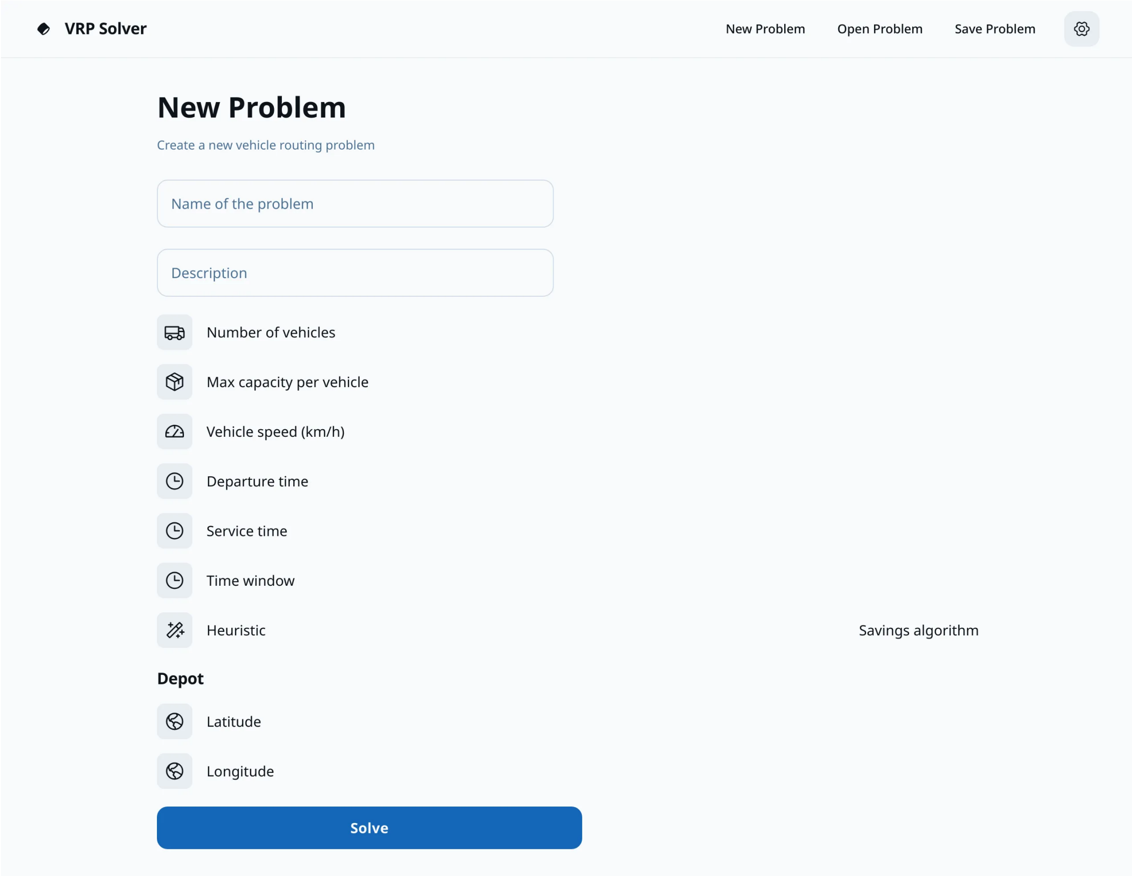


Figure 5.4. *VeRyPy GUI design for web generated using Galileo AI (2/2).*

Next, I used V0 by Vercel to generate another GUI design for the VeRyPy GUI with the same prompt I used in Galileo AI. I was able to generate a really good GUI design after the first attempt, but it still required some fine-tuning. I wanted to move the algorithm metrics on top of the visualization element instead of having them at the bottom. I also wanted to allow the user to input their problem either by uploading a `.vrp` formatted file or to input details about their problem manually. The final GUI design generated in V0 can be seen in Figure 5.5 on the next page, and the V0 chat logs are presented in the references [95].

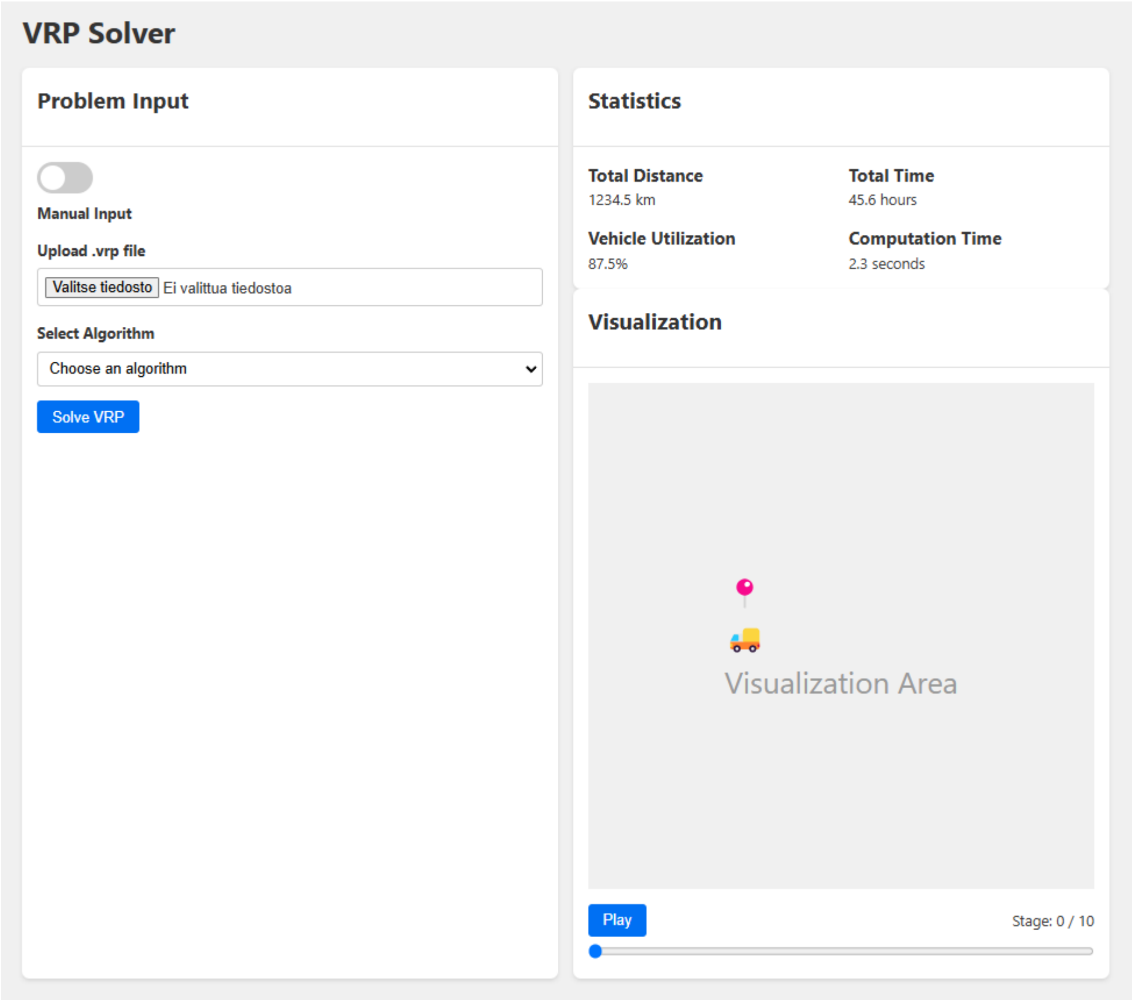


Figure 5.5. *The final VeRyPy GUI design generated using V0 by Vercel.*

Figure 5.6 below shows that it is possible to integrate code generated in Vercel V0 into existing projects using Shadcn, a library designed to manage custom GUI components. While this feature is convenient, it would add another dependency to the project, which is against VeRyPy's requirements to minimize external dependencies.

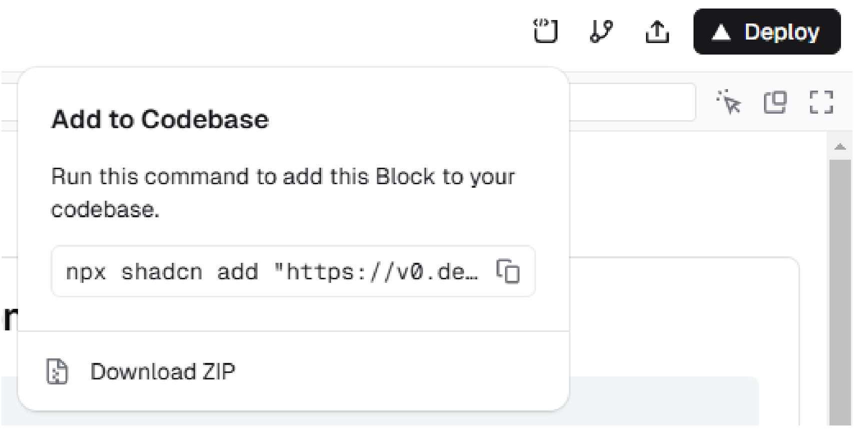


Figure 5.6. *Button for adding code generated in Vercel V0 to an existing codebase.*

To prepare the GUI design for the next iterations, I wrote a prompt in the V0 chat instructing it to convert the UI code into HTML and CSS. V0 successfully generated the code without altering the original design much. However, I noticed that the button for toggling between manual input and file input for the VRP no longer worked, as I was unable to display the input fields while entering VRP details manually. This issue occurred because the HTML and CSS code lacked any interactive code, but it can easily be added in the next iteration, where I will use these HTML and CSS files as a starting point to generate code for features of the VeRyPy GUI.

5.1.3 Analysis

Galileo AI offers a way to convert the generated GUI designs to both code and Figma files for further prototyping. When I clicked the 'code' button, the UI was converted into HTML code which uses Tailwind CSS, a popular CSS library, for styling. The HTML code can then be exported as plain code for further development. Additionally, it is possible to create a Replit project via Galileo AI, which is a nice way to quickly render the CSS in the cloud without having to run the HTML page locally. This is convenient, especially for less technical users and designers, as the generated HTML document can be set up and run without requiring technical skills.

The GUI designs generated by Galileo AI were clear and minimal, as requested in my prompt. The applied Tailwind CSS styling improved the GUI look a lot, and it followed good GUI design principles, such as using enough padding between the UI elements and using varying font weights to guide the user to focus on the most important elements.

Although the GUI design generated by Galileo AI was impressive, it was a bit too extensive for the VeRyPy GUI. For example, multiple different pages and a navigation bar are not needed in the VeRyPy GUI as the focus is to generate a minimal but intuitive GUI for solving VRPs without any additional content. However, the GUI designs generated by Galileo AI could easily be refined with additional prompts.

While generating the VeRyPy GUI design using Vercel V0, it actually generated the GUI code that was used to preview the GUI design. This makes sense as it can be easier to train LLMs using machine-readable content, such as code, compared to images or screenshots of different GUIs. The generated GUI code can then be compiled into the GUI design using existing compilers. In other words, AI tools focusing on GUI design generation may just be UI code generators, but instead of displaying the code, they display the result after compiling the generated code. This also enables users to convert GUI designs to code as the code already exists.

Initially, Vercel V0 generated code in TypeScript and using Next.js, a popular React framework for web applications. The code included custom UI components, which were imported at the top of the file and abstracted away from the user. Although the use of these stylized components improved the look of the GUI a lot, this also makes the code less modular because the developer may not have these components created or installed and therefore requires extra manual work to get the code running locally.

V0 was able to convert the UI code from TypeScript, Next.js, and some custom UI components to HTML and CSS, which looked almost exactly the same and aligned with the minimal dependencies requirement of the VeRyPy GUI. However, this step required an extra prompt that specifically describes this conversion, as well as knowledge of these web development techniques and languages. In V0, it was also possible to view the previous versions of the generated GUI in code, which was useful in case there was a need to return to earlier designs.

Overall, the generated GUI designs followed a standard website design where the navigation bar is at the top of the page. The app logo was placed on the left side, and the navigation links were located on the right side. There is a lot of padding on the left and right sides of the page, which helps the user to focus on the center of the screen. Responsive web design was also taken into account in the code, making the HTML page usable even on smaller screen sizes. This was great to see, considering I did not specifically mention the responsive design in the prompt.

5.1.4 Conclusion

Both Galileo AI and Vercel V0 provided great results in GUI design generation. Both of these AI tools provided ways to convert GUI designs into code. This feature makes it more convenient for developers to incorporate these AI tools into their GUI design workflows because it is easy to add the code generated using these tools to existing codebases.

The GUI design generated in Vercel V0 was more in line with the vision I had for the VeRyPy GUI. Also, V0 was a bit faster compared to Galileo AI, but the current server load for both of these tools was also likely affecting the performance. However, it was difficult to implement business logic and add functionalities to the UI elements within these tools. Also, it is currently not possible to import an existing code repository into either of these tools. However, it is possible to upload files to V0, but it is not very convenient if a project consists of multiple or hundreds of files and folders.

Overall, Vercel V0 and Galileo AI offer promising solutions for quickly creating mockups and GUI designs without requiring technical expertise. The generated designs featured consistent elements and followed good design practices, such as clear labels and enough padding between the UI elements. However, issues may arise when connecting business logic to the UI, especially with existing codebases that may not align with the technologies used in the AI-generated code.

5.2 Iteration 2: Basic GUI and Problem Input

The goal of iteration 2 is to generate a basic structure for the VeRyPy GUI and to allow users to input their vehicle routing problem along with parameters relevant to the VRP. One of the most important aspects of this iteration is selecting the framework and technology stack used to implement the VeRyPy GUI, as subsequent iterations will build upon this foundation.

5.2.1 Planning

Refer to Iteration 2: Basic GUI and Problem Input in Appendix B for the iteration 2 plan including the user stories and features implemented in this iteration.

The plan for this iteration is to first ask GitHub Copilot for suggestions regarding the implementation technology for the VeRyPy GUI. However, I intend to use basic Python modules to implement the GUI, as the heuristic algorithms and the VeRyPy codebase are written in Python, and the GUI needs to utilize these existing functions and algorithms to solve VRPs. Additionally, using basic Python modules supports the minimal dependency requirement of the VeRyPy GUI.

I also prefer to use a web-based architecture so that the VeRyPy GUI could be deployed as a web application in the future. Flask would be a suitable lightweight framework for Python-based web applications, but it is still too heavy for VeRyPy and does not align with the minimal dependency requirement. Therefore, the VeRyPy GUI will likely be a small-scale web app with a Python backend and a frontend built using HTML, JavaScript, and CSS.

After selecting the technology, I will construct the basic GUI by utilizing the HTML and CSS code generated by Vercel V0 in the first iteration. It is crucial to establish and test the connection between the frontend and the backend. The frontend must be able to pass data to the backend in order to call the heuristic algorithms implemented in VeRyPy, and the solution must be returned to the frontend.

5.2.2 Action

First, I initiated a new chat with GitHub Copilot within Visual Studio Code and asked for recommendations on the most suitable technology for implementing the VeRyPy GUI. Copilot suggested Tkinter, Flask, and Python's built-in HTTP server module. I chose the basic HTTP server module for generating the VeRyPy GUI, as it meets the minimal dependency requirement and is well-suited for a simple GUI. Copilot then generated the server code using Python's HTTP server module. I created a new file called `server.py` and pasted the generated code there.

Next, I copied and pasted the HTML and CSS code (generated by Vercel V0 in the previous iteration) into the GitHub Copilot chat and asked Copilot to use them as the foundation for the VeRyPy GUI. Initially, Copilot combined the HTML and CSS into a single file, just as V0 had done previously. However, separating the HTML and CSS into distinct files is beneficial for cleaner code, particularly as new features are added to the GUI. I instructed Copilot to move the CSS to a separate file, but this required the understanding that separating CSS and HTML is not only possible but also considered good practice in web development. As a result, I manually created the necessary files (`index.html` and `styles.css`).

To test the GUI and ensure it was working correctly, I started the server using the `python server.py` command, as suggested by Copilot. The server started successfully and was running on port 8000. I then accessed the GUI by navigating to `localhost:8000` in the Google Chrome web browser. The GUI appeared exactly as it had in V0's preview window.

I reviewed the implementation of the GUI. Copilot had generated a Python-based backend using native Python modules (such as `http.server`), while the frontend was structured with HTML and CSS. However, the frontend lacked interactive functionalities, such as event listeners for button clicks and input field interactions, which were also missing in the code generated by Vercel V0 in the first iteration. In other words, the foundation of the VeRyPy GUI was a full-stack web-based interface, with a Python backend and an HTML-based frontend. To enable interactivity, JavaScript would be required to handle user interactions and update the UI elements.

I wrote a prompt asking Copilot to add interactions to the HTML elements. In response, Copilot generated event listeners in JavaScript within the `index.html` file. To keep the code organized, I created a new file called `scripts.js` and moved the generated JavaScript code there.

Next, I prompted Copilot to validate the input fields when the user clicks the 'solve' button in the VeRyPy GUI. I also requested Copilot to display the selected algorithm and other parameters from the HTML form in the server logs to ensure the data is being correctly sent from the frontend to the backend. After reviewing the code generated by Copilot, I applied the necessary changes to the `scripts.js` and `server.py` files.

I asked Copilot to parse the information from the example VRP file in `.vrp` format (`E-n51-k5.vrp`), which was provided in the VeRyPy library, by attaching it as context for Copilot. Copilot was able to parse the `.vrp` formatted file correctly without any issues. I then asked Copilot to add a reset button to reset the input field values. Additionally, I requested Copilot to move the scripts from the HTML file to a separate `scripts.js` file, similar to how the CSS was moved to the `styles.css` file. I had to create the `scripts.js` file manually.

When I continued the project a few days later, I noticed that nothing happened when the 'solve' button was clicked after correctly inputting a VRP. I suspected the problem lay in the event listener linked to the 'solve' button. I highlighted the event listener and used the 'Review using Copilot' feature to ask Copilot to review it. Copilot suggested some useful quality-of-life changes, which are displayed in Figures 5.7 and 5.8. Among these suggestions, Copilot recommended changing `locations` to `locations.trim()` and `response.text` to `response.json`. While these suggestions were helpful, they did not resolve the issue where nothing happened after clicking the 'solve' button.

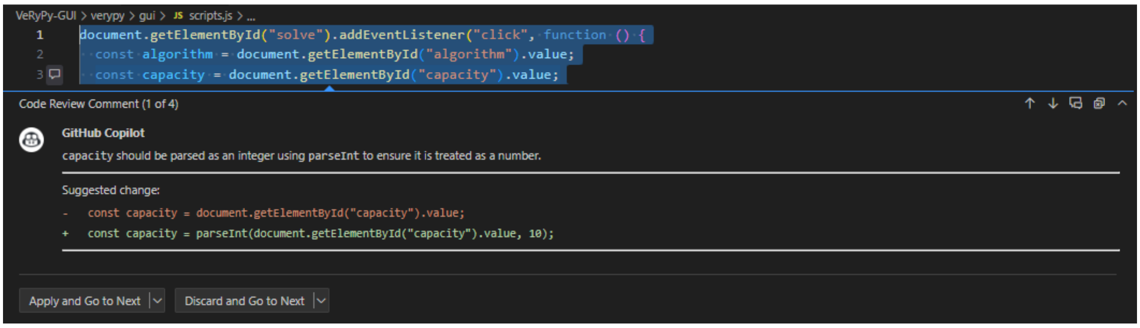


Figure 5.7. *Code review suggestions by GitHub Copilot (1/2).*

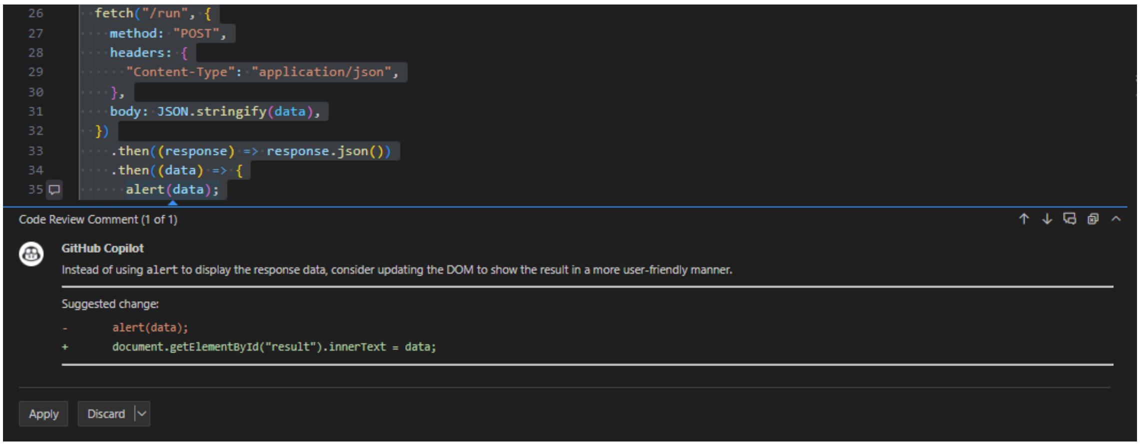


Figure 5.8. *Code review suggestions by GitHub Copilot (2/2).*

I asked Copilot in chat to fix the 'solve' button issue. It generated some changes that overrode the previously accepted quality-of-life improvements. Copilot suggested modifications to the `server.py` and `scripts.js` files, but they did not resolve the issue. I received an error in the Google Chrome DevTools console, which I copied and pasted into Copilot. It briefly explained the error and suggested some fixes, but those did not solve the issue. I tried again, but the same error occurred. After pasting the error again, Copilot suggested fixes similar to those it had proposed earlier. However, this time, it recommended a small change in the `server.py` file that affected how the server was started, as shown in Figure 5.9. After restarting the server, the 'solve' button was finally working as expected.

Figure 5.9. *Prompt for fixing an issue with the 'solve' button and a partial response.*

After the GUI was working again, I highlighted the `server.py` file and asked Copilot to review it using the 'Review using Copilot' feature. Initially, Copilot suggested removing some redundant logging configurations. It also recommended changing the way the VRP file was parsed, opting for a line-by-line approach instead of reading the entire file into memory. Following this, Copilot added a comment to the `do_GET` method. Additionally, it proposed error handling for the `do_POST` method to account for missing `Content-Length` headers. Copilot also added error handling for the `json.loads` function in cases where the request body is not valid JSON. Finally, it suggested modifying `server.py` to allow handling multiple requests concurrently.

I applied each change to the `server.py` file as suggested by the 'Review using Copilot' feature. However, I noticed a few errors related to the use of return statements outside of functions. In other words, some issues arose in the code generated by Copilot. When I hovered over these errors, a menu called 'Fix using Copilot' appeared. Copilot provided a brief description of how it would address the issue, as shown in Figure 5.10 below. After clicking Accept, Copilot generated a fix, which I accepted. Following this, the code compiled without errors.

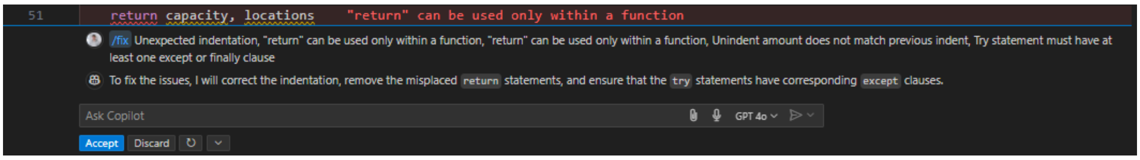


Figure 5.10. *GitHub Copilot suggestion for fixing an error.*

I moved to the `scripts.js` file to review it with Copilot again. First, it suggested moving a function inside an event listener to a separate function, ensuring that it would not be executed every time the script runs. Then, it proposed adding error handling for `parseInt` and `parseFloat`. Copilot also recommended updating the UI element instead of using an alert, as it is more user-friendly. It then noticed an error logged to the console that was not visible to the user, so it suggested displaying the error message to the user via an alert. Additionally, it suggested error handling for invalid vehicle capacity and locations. I applied all the changes, but encountered an error where `.then` was called twice in succession inside the fetch request to the `/run` endpoint. I resolved this by using the 'Fix using Copilot' feature, though it could have been easily fixed manually as well. After testing the application with these adjustments, it worked exactly as before, with no visible difference in functionality.

Next, I switched to the `index.html` file and highlighted the page title. I asked Copilot to change it to 'VeRyPy VRP Solver.' It made the change quickly, as expected, though this could have easily been done manually. Requesting such small changes through the chat might be overkill, as it would have generated the entire HTML file.

5.2.3 Analysis

When selecting the technology stack, Copilot suggested multiple options, including Tkinter, Flask, and Python's built-in HTTP server module. Because LLMs have been trained with a lot of data, and due to the popularity of using Tkinter and Flask in Python projects, it will assume it is the most likely way to build other similar Python projects. While Flask is a great framework for Python-based web applications, it does not align with the minimal dependency requirement of the VeRyPy GUI. However, if the VeRyPy GUI was more complex, had multiple pages, and minimizing external dependencies was not a requirement, then Flask could have been selected as the framework. On the other hand, Tkinter would align with this requirement, but generating the GUI as a web application was my personal preference.

Integrating the code from Vercel V0 to the VeRyPy GUI codebase required some manual steps. Initially, Copilot placed both HTML and CSS code in a single file, similar to V0's output. Separating them required explicit prompting and knowledge that separating them is possible and generally recommended. The same issue came up when moving JavaScript code into a separate `scripts.js` file where Copilot was able to generate the code but it did not automatically create the necessary file. There was a button in the Copilot window to create a new file that would include the contents of the generated code, but it did not work for some reason. These observations indicate that while Copilot can accelerate code generation, developers must still guide structural decisions to maintain clean and scalable code.

Reviewing and improving the `server.py` file demonstrated Copilot's strengths and weaknesses in optimizing existing code. Copilot was able to identify redundant logging, suggested improvements to file parsing, and introduced better error handling for HTTP methods. However, applying these improvements also introduced errors, such as incorrect return statements outside functions. Although Copilot provided fixes for these errors through the 'Fix using Copilot' feature, it does not eliminate the careful review of the generated code. In other words, the generated code sometimes introduced new errors that require additional debugging.

Similarly, reviewing `scripts.js` with Copilot revealed useful improvements, such as better error handling and user-friendly notifications in the GUI. However, at one point a code generation led to a double `.then()` call in a fetch request. This indicates that while Copilot can suggest useful improvements at the function level, it may also introduce side effects.

I noticed that most of the time, Copilot generated entire files regardless of where the change was applied in the code. This behavior slowed down the code generation process, as it would be more efficient if only the modified parts of the code were generated instead of the entire file. Additionally, using Copilot for trivial or minor changes may be unnecessary, since manual adjustments are often quicker.

5.2.4 Conclusion

During the iteration 2, the VeRyPy GUI design generated in the previous iteration was integrated into the existing VeRyPy codebase. The VeRyPy GUI uses a Python backend (`http.server`), while the frontend is built with HTML, CSS, and JavaScript (`index.html`, `styles.css`, and `scripts.js`). The frontend interacts with the backend through standard HTTP requests.

The code generated by GitHub Copilot worked relatively well, requiring only minor clarifications and adjustments. This was expected, given that my prompts were both concise and clear. Selecting the appropriate technology required prior knowledge of various approaches for implementing a GUI that would meet the requirements of the VeRyPy GUI. Additionally, multiple prompts were used to build and refine the GUI, highlighting the iterative nature of software development, regardless of whether AI is involved.

Troubleshooting with GitHub Copilot proved somewhat challenging. Since Copilot lacks visibility into error messages and runtime behavior, especially in full-stack applications, I had to manually describe and explain the issues encountered while interacting with the GUI. Resolving problems, such as the failure to redirect the user to the /**run** endpoint after clicking the 'solve' button, required several attempts. However, fixing syntax errors was much easier and more convenient, as they were clearly highlighted in red by the code editor and quickly detected by Copilot.

5.3 Iteration 3: Algorithm Selection and Execution

The goal of the third iteration is to integrate the existing heuristic algorithms from VeRyPy into the GUI. Once this step is completed, users will be able to select any of the implemented algorithms to solve their VRPs. The solution will be displayed in a text-based format within the GUI, with visualizations to be added in the following iteration.

5.3.1 Planning

Refer to Iteration 3: Algorithm Selection and Execution in Appendix B for the iteration 3 plan including the user stories and features implemented in this iteration.

I studied the existing VeRyPy codebase to understand how to call the implemented heuristic algorithms for solving VRPs. The algorithms are located in the *verypy/classic_heuristics* directory as separate files. Each algorithm file contains a function ending in `_init`, which can be called from other files to utilize the algorithms. Additionally, there are some helpful examples in the *examples* directory, particularly the `single_solve_example.py` file, which provided valuable insights into how the algorithms should be used.

I will ask Copilot to find the implemented heuristic algorithms from the VeRyPy codebase, testing how well it can understand the codebase without being provided the exact path to the relevant files. If Copilot cannot find the algorithms, I will either provide the path to the directory containing the algorithms or attach the algorithm code files as context.

Next, I plan to ask Copilot for the best and most efficient way to integrate these algorithms into the VeRyPy GUI, while considering scalability and modularity. It's crucial to import the existing algorithms in a manner that respects potential future modifications, such as new algorithms or changes to algorithm parameters or naming conventions. Once the selected algorithms are integrated correctly into the GUI, adding additional algorithms will be more straightforward.

5.3.2 Action

I noticed that there is already a function for parsing `.vrp` files implemented in VeRyPy. In the previous iteration, I asked Copilot to generate the same parsing logic by providing the `E-n51-k5.vrp` file as context, which resulted in JavaScript code in the `scripts.js` file. I wanted to remove this redundancy and use the already implemented VRP parser instead. I asked Copilot to do this refactoring. First, it suggested creating a separate `api.py` file using Flask, but this approach was against the requirements of the VeRyPy GUI. I then asked it to add the endpoint to the `server.py` file instead. I tested these changes, and they worked perfectly on the first try.

I noticed that I need to extract vehicle constraints, customer demands, and the distance matrix from the uploaded `.vrp` file. I asked Copilot to modify the `server.py` file and `scripts.js` file to extract these attributes, as they are used in the provided example files for solving VRPs. I applied the changes generated by Copilot and tested the GUI, which worked without any issues.

I moved on to integrating the implemented algorithms into the VeRyPy GUI. This time, I added the codebase as context in Copilot chat. I asked Copilot to search for the implemented heuristic algorithms in this codebase and to provide a quick summary of how they can be used. I had to verify my GitHub account and provide read access to my public and private repositories before I could proceed. Copilot searched for "heuristic algorithms" and provided a nice summary of the implemented heuristics in VeRyPy, along with how they can be called and what parameters are required. The prompt and the partial response can be seen in Figure 5.11 on the next page.

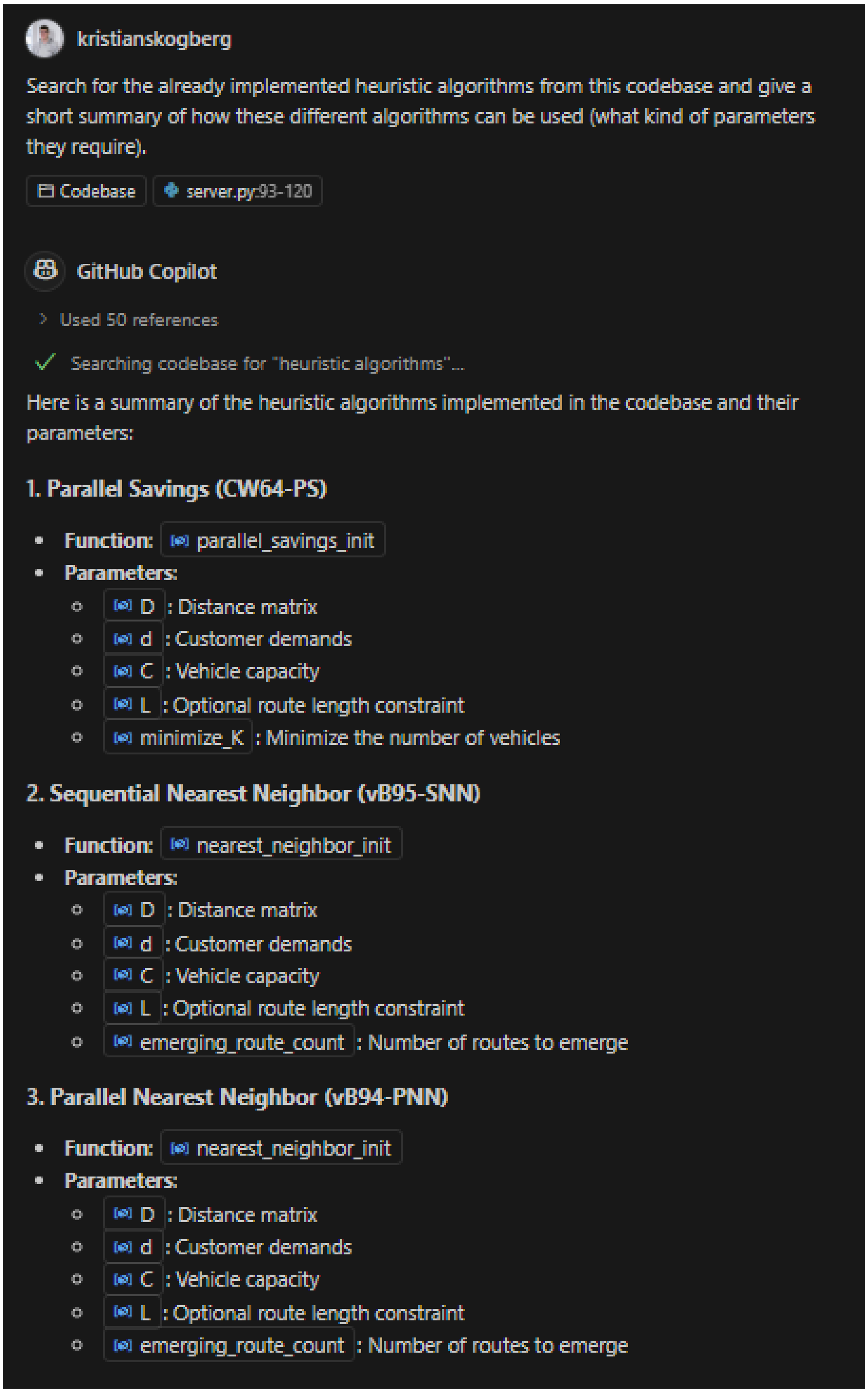


Figure 5.11. *Prompt for searching heuristic algorithms from the VeRyPy codebase and a partial response.*

Then, I asked Copilot what would be the most optimal way to integrate these algorithms into the VeRyPy GUI. It suggested creating a configuration file for the algorithms and using that file to populate the algorithm selection menu. It also suggested changes to the `server.py` and `scripts.js` files. I applied all the changes as they seemed good. However, after refreshing the web page, I noticed that the algorithm selection menu was empty. I restarted the server, but the same issue persisted. I described the situation in Copilot chat, and it suggested some debugging steps, but they did not resolve the issue. I attached the `scripts.js` and newly created `algorithms.json` files as context, and Copilot suggested concrete debugging steps, such as checking the folder structure and the network tab in Google Chrome DevTools.

In the network tab, I found that no request was being made to get the `algorithms.json` file, which I explained to Copilot. It suggested ensuring that the server is configured to serve static files. I added the `server.py` file as context and asked it to check if any modifications were needed. Copilot generated code for the `server.py` file, but it did not include the necessary changes. I copied and pasted the code into the `server.py` file and restarted the server, but the same error persisted. Unable to fix the issue, I reverted the changes and asked Copilot to add the algorithm selection as `<select>` elements directly to the HTML file.

I tested the GUI and encountered a problem when clicking the 'solve' button. I copied and pasted the error into Copilot, and it generated some code for the `server.py` file to fix the issue. However, I noticed that it extracted some old parameters from the request, which had been changed earlier during this iteration, but Copilot did not notice this. I asked Copilot to update the parameters and provided the code with the updated parameter names. Additionally, I noticed that some previous suggestions from Copilot for the `scripts.js` file had removed the event listener of the 'Upload file (.vrp)' button. This change was not updated for me because it was stored in my cache, even though it was not present in the code. Once I noticed this, I disabled the cache and refreshed the web page, and now I had the newest changes. I attached the `scripts.js` and `server.py` files to Copilot and asked it to add this functionality back into the code. Then, I asked Copilot to change `location` to `distance_matrix` in `server.py` to avoid misinterpretations.

There were some issues when the server returned the contents of the parsed `.vrp` file that were not present previously. I asked Copilot to fix this, and it suggested some changes. After two attempts, I was able to resolve the issue. I asked Copilot to display the distance matrix exactly as it is formatted in the `.vrp` file in the GUI, and this was successful on the first attempt. After I fixed the caching issue, I asked Copilot to generate a feature for reading or parsing algorithms from the `algorithms.json` file and to populate the algorithm selection element in `index.html` with this algorithm data. This time, I was able to get it working on the first attempt since the Google Chrome cache was disabled.

I tested the GUI and noticed that there was an error regarding updating an HTML element that did not exist. This change originated from modifying the code using the 'Review using Copilot' feature. I removed this code manually because it was much faster than writing a prompt and waiting for Copilot to remove it.

Another error occurred when solving the VRP using the selected algorithm in the `server.py` file. This error is likely due to how `.vrp` files are formatted in the GUI before being passed to the server. I asked Copilot to create separate input fields for the distance matrix and customer demands instead of grouping them into one `<textarea>` element.

I noticed that the GUI currently formats the `.vrp` file two times. I wanted to change the logic so that it only reads the `.vrp` file once, right before solving the VRP. I asked Copilot to refactor this. Copilot generated code for the `server.py` file, but trying to solve the VRP resulted in an error with list indexes being tuples instead of integers or slices. I asked Copilot to solve this, but the error persisted. Then, I added the provided example usage of the `.vrp` parser function as context and asked Copilot to follow this example. Copilot still did not follow the example line by line, even though it was very short, and it still generated some extra code for it. I removed the extra code lines so that it looked just like the example.

The code generated using Copilot tried to access the `.vrp` file and also some other fields from the GUI. I asked Copilot to only use the `.vrp` file to make debugging easier. Now, the errors disappeared, but I was still unable to see the solution in the server logs. It seems as if the server does not read the selected algorithm function correctly. I asked Copilot to add the Parallel Savings algorithm to the `algorithms.json` configuration file, as it was also used in the example. However, I was still unable to see any logs of the solution.

I noticed that Copilot kept suggesting the same change, which I had rejected multiple times. This change can be seen in Figure 5.12 below. I rejected this change again.

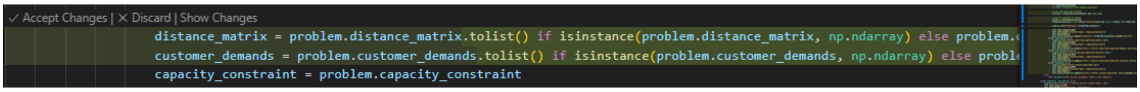


Figure 5.12. *A code change GitHub Copilot kept suggesting multiple times.*

Copilot suggested some changes in debugging the algorithm module import part. I applied the changes and restarted the server. Now, I was able to see the logs of the solution in the `server.py` terminal when selecting the Parallel Savings algorithm. I tested this with the other algorithms, but I got a module import error in the server terminal. I noticed that this algorithm does not exist in VeRyPy (at least not with this name). I attached the VeRyPy codebase and the `algorithms.json` config file as context and asked Copilot to add only the existing algorithms to this JSON file, which can be imported as modules. Table 5.1 contains notes on which algorithms I was able to run successfully on the first attempt after applying Copilot's initial code suggestion.

Heuristic Algorithm	Execution
Nearest Neighbour	Success
Parallel Savings	Success
Simulated Annealing	Failure
Sequential Savings	Success
Paessens Savings	Success
Gillett-Millet Sweep	Failure
Wren-Holiday Sweep	Failure
Tyagi Nearest Neighbour	Failure
Route First, Cluster Second	Failure
Petal VRP	Failure
Matching VRP	Failure
CMT Two Phase	Failure

Table 5.1. *Heuristic algorithms and their execution success rate applying Copilot's initial code suggestion.*

I found that the reason why these failing algorithms could not be imported was related to the naming. For example, the Gillet Miller Sweep algorithm should be called `gillet_miller_init` instead of `gillet_miller_sweep_init`, which was used in the `server.py` file. I asked Copilot what would be the best way to fix this, this prompt and the partial response can be seen in Figure 5.13. Copilot suggested adding import path and function name attributes to the `algorithms.json` file and reading them in the `server.py` file, which was an okay approach for this kind of situation without having to hard code anything. I noticed that Copilot added the import path and function name to only about half of the algorithms, so I asked Copilot to add them to the remaining algorithms.

Figure 5.13. *Prompt for importing heuristic algorithms and a partial response.*

I tested the GUI again and was able to import the rest of the algorithms successfully. However, there were import errors in two algorithms (Petal VRP and Matching VRP): *Error importing algorithm module: No module named 'gurobipy'*, which made sense because these algorithms used the Gurobi library to solve VRPs. This was something that was not specified by me in the prompt, and Copilot did not take this kind of import issue into account during code generation. Additionally, two algorithms (Gillet Miller Sweep and Wren Holliday Sweep) required a 'points' parameter, unlike the rest of the algorithms, causing an error when they were selected and used for solving VRPs through the GUI.

As I was testing the GUI, I noticed that the 'reset' button was not working properly. I also found an error in the `scripts.js` file, as it had an event listener for adding the location manually, which had been removed by Copilot previously. I removed this line manually, and now the 'reset' button worked properly.

I noticed the metrics used in the `solve_from_data_dict.py` file, such as `elapsed_t` and `K`. I wanted to use the same metrics in the server while solving VRPs because metrics will need to be calculated and displayed in one of the user stories. I added the codebase, `solve_from_data_dict.py`, `algorithms.json`, and `server.py` as context and asked Copilot to calculate the same metrics. Copilot added the changes to the `server.py` file, and I restarted the server. There was an error while importing some of VeRyPy's helper functions, as Copilot tried to import them from the wrong path. I asked Copilot to correct this, and it did. However, I got a new error related to how Copilot tried to serialize the solution data to JSON in the server. I asked it to fix this, and it used another method for sending the solution data as a response, which worked.

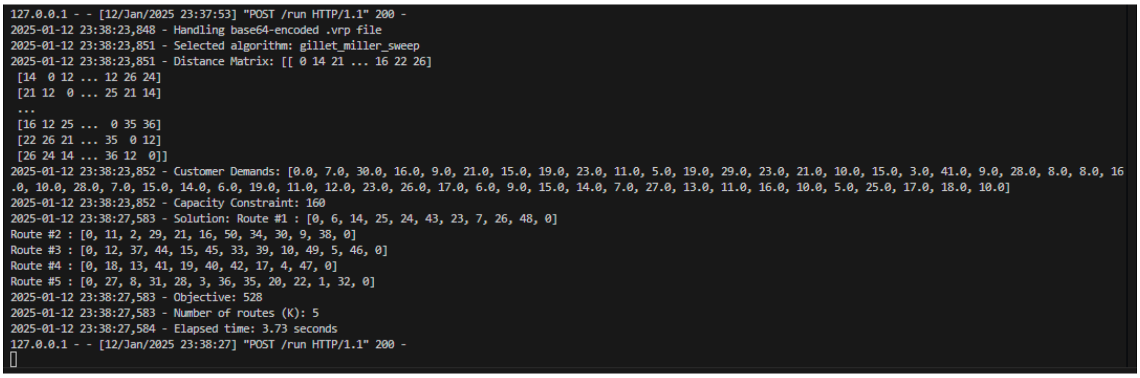
I found that the sweep algorithms require a 'points' parameter. I asked Copilot to update the parameters for each algorithm in the `algorithms.json` config file while attaching the codebase as context. I applied these changes. Then I asked Copilot to use these parameters for the selected algorithm in the `server.py` file. The changes looked good, so I applied them, but I noticed that Copilot tried using an attribute of the VRP parser function which did not exist (`node_coords`), as shown in Figure 5.14 below. I modified this code line manually because it was faster than explaining this to Copilot and waiting for it to generate a fix. I restarted the server, tested the GUI, and now I was able to use the sweep algorithms correctly as well.

```
problem = cvrp_io.read_TSPLIB_CVRP(file_path)
os.remove(file_path)

distance_matrix = problem.distance_matrix
customer_demands = problem.customer_demands
capacity_constraint = problem.capacity_constraint
points = problem.node_coords if hasattr(problem, 'node_coords') else None
```

Figure 5.14. *A code snippet generated by GitHub Copilot.*

I tested solving the `E-n51-k5.vrp` file with every algorithm. I was able to solve it successfully with most of the algorithms, except for GAP VRP, Petal VRP, and Matching VRP, as they required Gurobi to be installed, which I did not have on my machine. The algorithm solution logs in the server terminal are displayed in Figure 5.15 on the next page.



```
127.0.0.1 - - [12/Jan/2025 23:37:53] "POST /run HTTP/1.1" 200 -
2025-01-12 23:38:23,848 - Handling base64-encoded .vrp file
2025-01-12 23:38:23,851 - Selected algorithm: gillet_miller_sweep
2025-01-12 23:38:23,851 - Distance Matrix: [[ 0 14 21 ... 16 22 26]
 [14  0 12 ... 12 26 24]
 [21 12  0 ... 25 21 14]
 ...
 [16 12 25 ...  0 35 36]
 [22 26 21 ... 35  0 12]
 [26 24 14 ... 36 12  0]]
2025-01-12 23:38:23,852 - Customer Demands: [0.0, 7.0, 30.0, 16.0, 9.0, 21.0, 15.0, 19.0, 23.0, 11.0, 5.0, 19.0, 29.0, 23.0, 21.0, 10.0, 15.0, 3.0, 41.0, 9.0, 28.0, 8.0, 8.0, 16
.0, 10.0, 28.0, 7.0, 15.0, 14.0, 6.0, 19.0, 11.0, 12.0, 23.0, 26.0, 17.0, 6.0, 9.0, 15.0, 14.0, 7.0, 27.0, 13.0, 11.0, 16.0, 10.0, 5.0, 25.0, 17.0, 18.0, 10.0]
2025-01-12 23:38:23,852 - Capacity Constraint: 160
2025-01-12 23:38:27,583 - Solution: Route #1 : [0, 6, 14, 25, 24, 43, 23, 7, 26, 48, 0]
Route #2 : [0, 11, 2, 29, 21, 16, 50, 34, 30, 9, 38, 0]
Route #3 : [0, 12, 37, 44, 15, 45, 33, 39, 10, 49, 5, 46, 0]
Route #4 : [0, 18, 13, 41, 19, 40, 42, 17, 4, 47, 0]
Route #5 : [0, 27, 8, 31, 28, 3, 36, 35, 20, 22, 1, 32, 0]
2025-01-12 23:38:27,583 - Objective: 528
2025-01-12 23:38:27,583 - Number of routes (K): 5
2025-01-12 23:38:27,584 - Elapsed time: 3.73 seconds
127.0.0.1 - - [12/Jan/2025 23:38:27] "POST /run HTTP/1.1" 200 -
```

Figure 5.15. *Algorithm solution logs in `server.py` terminal.*

I looked at the GUI and noticed that the naming of the distance matrix should actually be coordinates. I highlighted this part of the code in `index.html` and used the 'Modify using Copilot' feature to change the naming to coordinates and applied the change. I also had to update this change in the `scripts.js` file since Copilot did not do this automatically.

I noticed a function for calculating the VRP solution's feasibility in the `cvrp_ops.py` file. I attached this file and the `server.py` file as context for Copilot and asked it to use this in `server.py`. I restarted `server.py` and noticed that Copilot had once again overwritten the points attribute, so I had to manually revert this line.

While I was testing to solve the VRP provided in the `E-n51-k5.vrp` file, I noticed that the sweep algorithms took a few seconds to solve the VRP. From my frontend development background, I know that it is always a good idea to display loading indicators to the user while heavy calculations are being done. I asked Copilot to change the text of the 'solve' button to 'solving...' while the VRP is being solved, and also to disable all input fields during the solving process and enable them after the VRP has been solved. The prompt and the beginning of the response can be seen in Figure 5.16 on the next page. I applied the code and had to restart the server to update the changes in the `scripts.js` file to get it to work. Now, the GUI is more user-friendly, as it displays the loading indicator during the solve process.

kristianskogberg

Solving VRPs with some algorithms can take some time. I want you to change the text of the solve button to "solving..." while the VRP is being solved. Also, disable all buttons and input fields while the VRP is being solved. Finally, enable the buttons after the VRP has been solved. Do these changes without breaking anything else.

JS scripts.js:166-179

GitHub Copilot

› Used 1 reference

To change the text of the solve button to "solving..." while the VRP is being solved, disable all buttons and input fields during the solving process, and enable them after the VRP has been solved, we can update the event listener for the solve button.

Updated JS scripts.js)

Update the event listener for the solve button to handle the changes:

```
document.addEventListener("DOMContentLoaded", function () {
  fetch("algorithms.json")
    .then((response) => {
      if (!response.ok) {
        throw new Error(`HTTP error! status: ${response.status}`);
      }
      return response.json();
    })
    .then((algorithms) => {
      const algorithmSelect = document.getElementById("algorithm")
      algorithms.forEach((algorithm) => {
        const option = document.createElement("option");
        option.value = algorithm.value;
        option.textContent = algorithm.name;
        algorithmSelect.appendChild(option);
      });
    })
    .catch((error) => {
      console.error("Error loading algorithms:", error);
    });
});

document.getElementById("vrp-file").addEventListener("change", fun
  const file = this.files[0];
  const reader = new FileReader();

  reader.onload = function(event) {
    const content = event.target.result;
    const lines = content.split("\n");
```

Figure 5.16. *Prompt for improving the VeRyPy GUI usability and a partial response.*

I also found that it would be good to disable the hover effects while the VRPs are being solved. I asked Copilot to do this, and it suggested a few lines of changes to the `scripts.js` file using a custom CSS class for disabling the hover effects. It also generated the CSS class, but it did not specify to which file it should be added. Copilot was unable to find the `styles.css` file, even when it was in the same directory as the `scripts.js` file. I opened the `styles.css` file and applied the newly generated CSS class there, restarted the server, and I was able to see the changes correctly.

Before moving on to the next iteration, I wanted to restore the input fields for adding new coordinates, which Copilot had previously removed. I reviewed the `E-n51-k5.vrp` syntax and noticed that the number of coordinates matches the number of customer demands. I asked Jussi Rasku about this, and he confirmed that the number of customer demands must be equal to the number of coordinates. I had overlooked this rule in the previous iteration.

Next, I asked Copilot to generate input fields for manually adding coordinates and customer demands to the GUI. I provided the `scripts.js` and `index.html` files as context. In my prompt, I requested the generation of the input elements first, with error handling to be addressed in the next step. I also specified where these input fields should be placed in the GUI. After applying the changes, I noticed that the new input fields were larger than expected and took up a lot of space in the GUI. Additionally, the current design and implementation made it difficult to edit the already entered coordinates and customer demands. I then wrote a prompt to revert the changes (removing the manual coordinate and customer demands input fields) and to make the `<textarea>` elements editable, as this would make modifying the existing data easier.

I applied the changes to the `index.html` file and also made modifications to the `scripts.js` file, but there were actually no changes made to the `scripts.js` file. I wrote a new prompt asking Copilot to specifically remove the previously added code related to the manual coordinates and customer demands fields, and this time it worked. Then, I asked Copilot to implement error handling that checks if the number of rows in the coordinates and customer demands `<textarea>` elements are the same before attempting to solve the VRP. After applying the code and testing the GUI, it worked as expected. I am now quite satisfied with the GUI, and the current version can be seen in Figure 5.17.

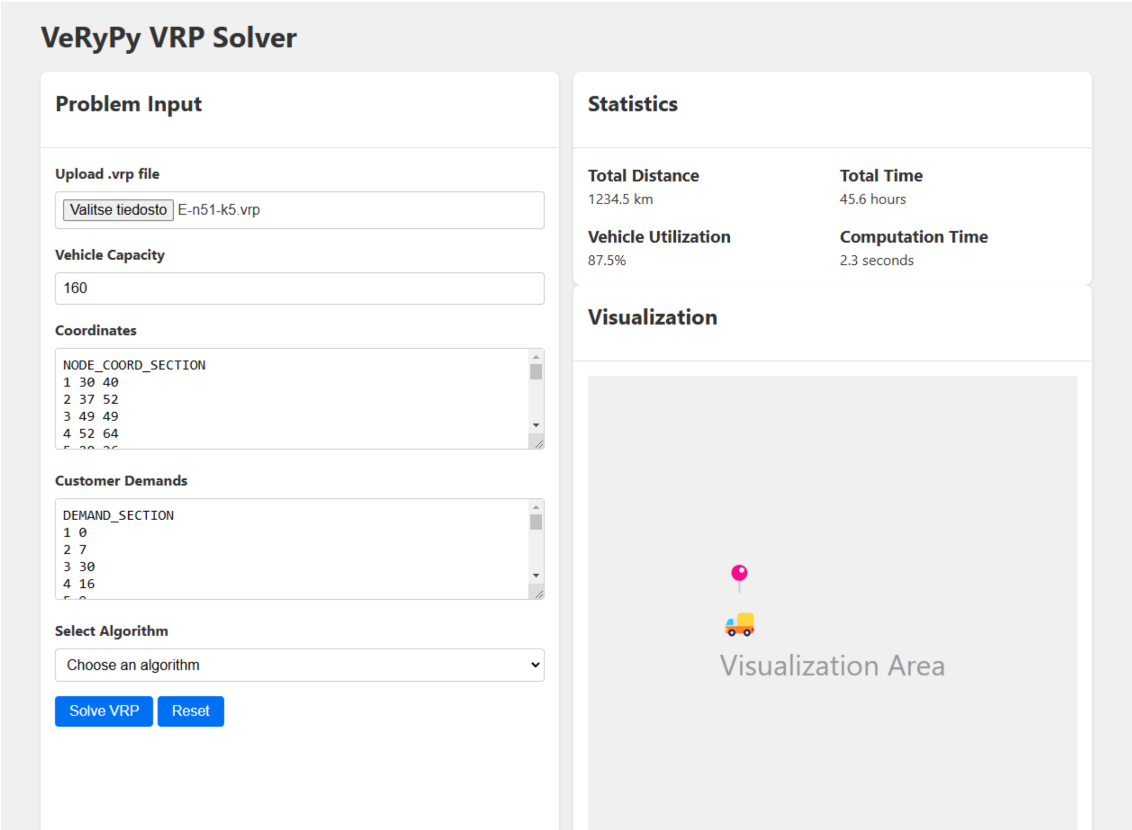


Figure 5.17. *Current state of the VeRyPy GUI.*

I found more VRP examples in `.vrp` format from a public repository on GitHub [96]. I thought it would be useful to test the GUI with other `.vrp` formatted files. I downloaded a few `.vrp` files and imported them into the GUI one by one for testing. I discovered that the `.vrp` files with the `EDGE_WEIGHT_TYPE` set to `EXPLICIT` contained the `EDGE_WEIGHT_SECTION` attribute instead of the `NODE_COORDS_SECTION` attribute, which is currently used in the GUI to display the coordinates of the uploaded `.vrp` file.

I thought it would be better to use the already implemented `get_algorithms` function from VeRyPy to import all algorithms into the GUI, instead of relying on the separate `algorithms.json` config file that was previously generated. I asked Copilot to make this change and provided the relevant files as context: `(_init_py)`, `index.html`, `server.py`, and `algorithms.json`. Copilot suggested creating a new endpoint in the server to fetch all algorithms and then retrieving this data when the DOM is loaded. I started the server and tested the GUI, and I was able to see the algorithms in the selection element.

However, when I tried solving the `E-n51-k5.vrp`, I encountered an error in the server terminal related to the algorithm function parameters. I explained the situation and pasted the error message into Copilot chat. I initially thought the fix would only require changes in the `server.py` file, but Copilot generated a full `index.html` file and `scripts.js` file as well. This led to a response limit error, which is shown in Figure 5.18, and the response was never completed.

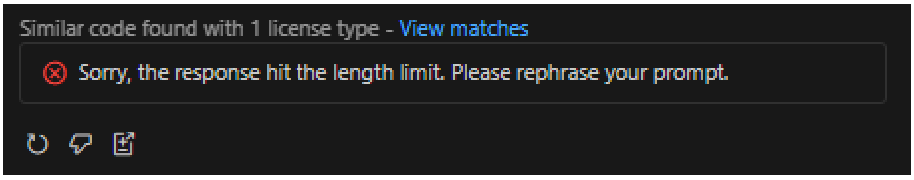


Figure 5.18. *GitHub Copilot response limit error.*

I rephrased the prompt so that Copilot would only modify the `server.py` file. After applying the changes, I noticed that Copilot once again suggested an unnecessary modification to how the coordinate points are retrieved from the parsed problem variable. I discarded the change to this line and added a comment to indicate that this variable should not be modified, hoping Copilot would understand this in future code generations. I also observed that the example file always used the same parameters for calling each algorithm, so I wrote a prompt to implement this approach in `server.py` as well.

The generated code looked good, so I applied it. However, I noticed that Copilot had again changed how the coordinate points were retrieved, despite my comment to avoid modifying this variable. I discarded this change once more and tested the GUI. To my surprise, the algorithm population in the HTML element was not working. Upon further inspection, I found that Copilot had removed the endpoint it had just generated in the `server.py` file. I wrote a prompt to reintroduce this endpoint and added both the server and `scripts.js` files as context. After applying the changes and restarting the server, everything worked again, and I was able to solve the VRP using the selected algorithms.

Copilot had used the algorithm name to populate the algorithm element in the HTML DOM, but I wanted it to use the algorithm description instead, as it was easier to read. I highlighted this part in `server.py` and used the 'Modify using Copilot' feature to make the change. Copilot added a new description attribute when fetching all algorithms. I then moved to `scripts.js` and manually modified a line from `algorithm.name` to `algorithm.description`. After restarting the server and testing the GUI, the changes were applied correctly. I deleted the previously generated `algorithms.json` configuration file since it was no longer needed and committed the changes. The updated algorithm selection menu can be seen in Figure 5.19.

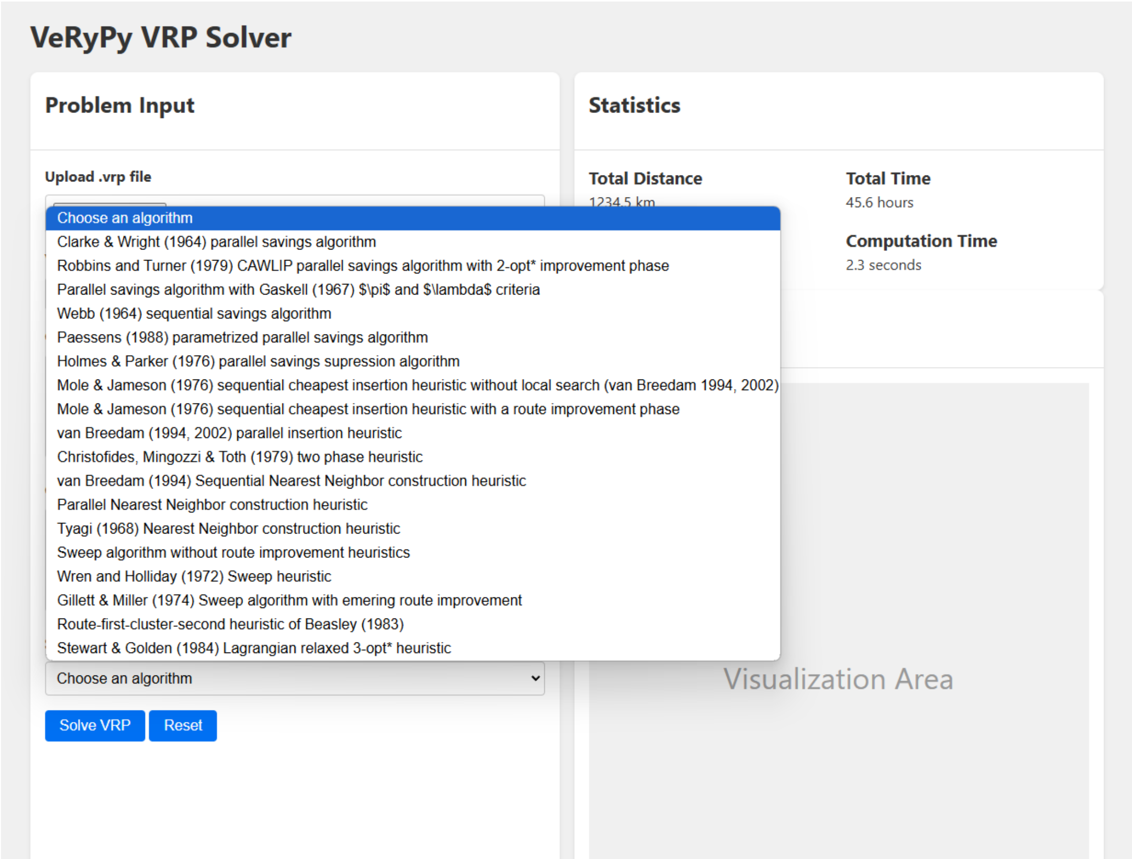


Figure 5.19. *Heuristic algorithm selection menu in the VeRyPy GUI.*

All algorithms can use the following parameters: 'L', 'single', and 'minimize K'. I wrote a prompt for Copilot to add these parameters to the GUI and provided the `index.html`, `scripts.js`, and `server.py` files as context. The current state of the GUI with these new algorithm parameter input fields can be seen in Figure 5.20 on the next page.

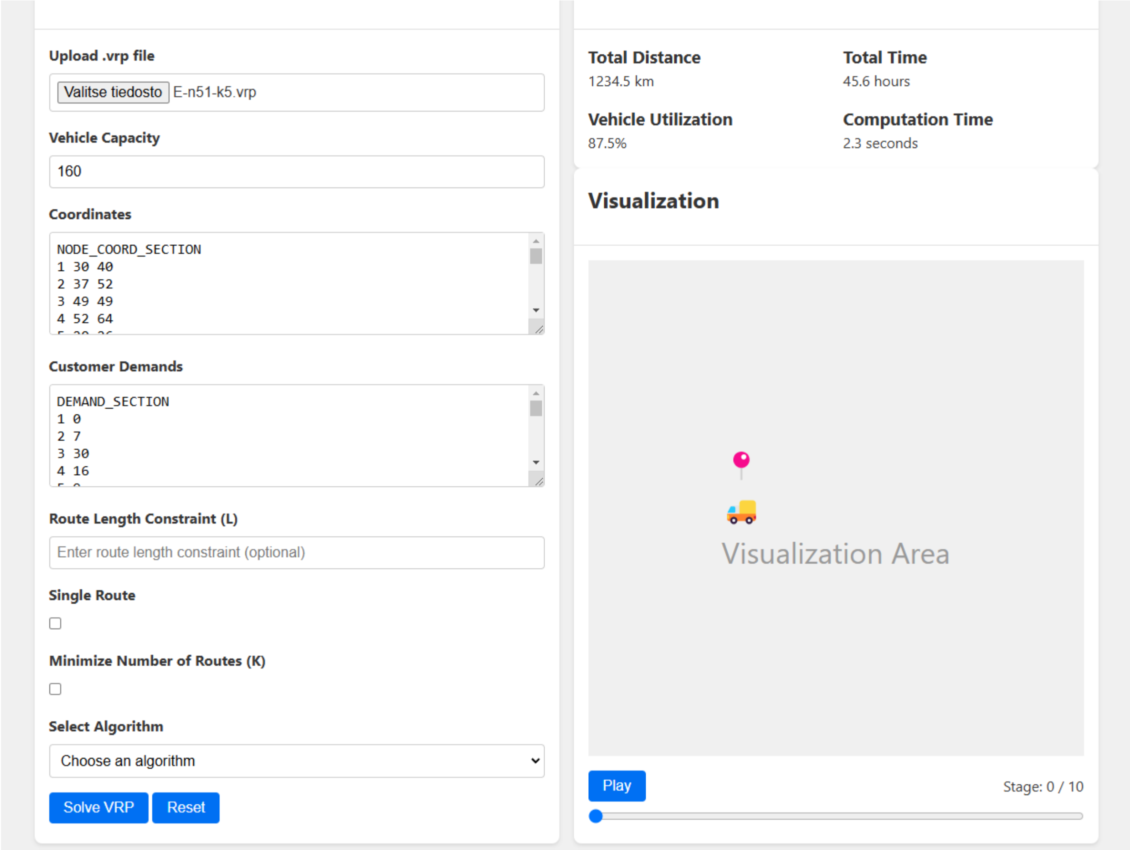


Figure 5.20. *VeRyPy GUI with algorithm parameter input fields.*

I noticed that Copilot reverted the small change for the algorithm description attribute, which was done in the previous prompt's output. I discarded this change but applied the other suggested changes. This suggestion can be seen in Figure 5.21 below.

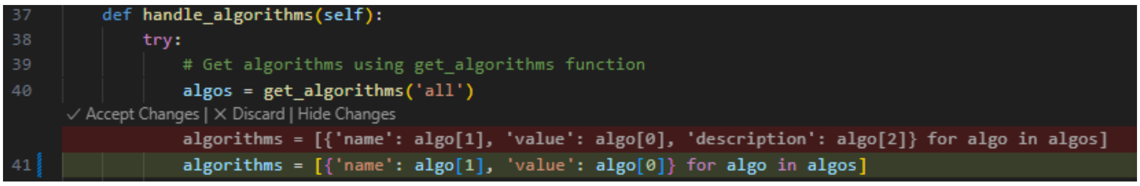


Figure 5.21. *A code suggestion by GitHub Copilot.*

I tested the GUI, and it seemed to work, but no logging was added for the newly generated parameters. I asked Copilot to generate logging, but it began adding input fields in the `index.html` and `scripts.js` files, which were unrelated to the prompt (Figure 5.22). I discarded the changes and rewrote the prompt to specify that logging should only be added to `server.py`. However, Copilot generated the same output again. I then modified my prompt to: *Add logging to all parameters in the server.py file and do not modify other files.* This time, Copilot understood and added the logging without touching other files. The task could have been done much faster manually.

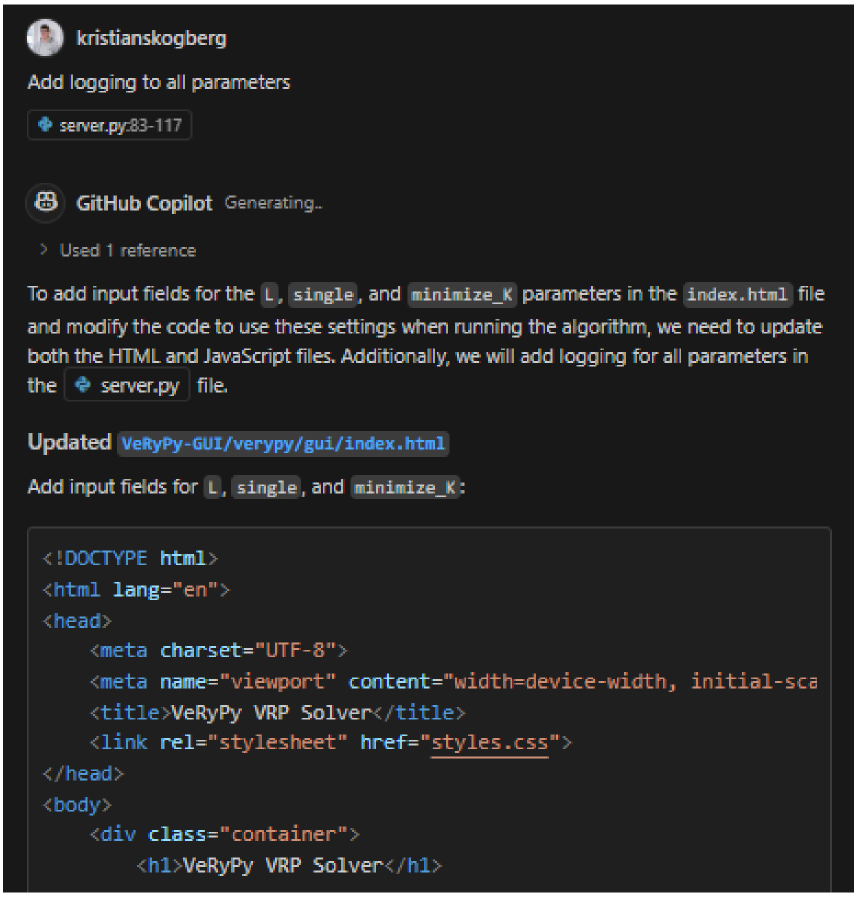


Figure 5.22. *An example of misinterpretation by GitHub Copilot.*

Next, I wanted to move the labels and input elements for the `single` and `minimize_K` parameters, which are used in heuristic algorithms, to the same row in the GUI. First, I highlighted these elements in `index.html` and used the 'Modify using Copilot' feature. It generated some inline CSS, but it did not change anything, most likely because the CSS in `styles.css` may override the inline CSS. I attached `index.html` and `styles.css` as context and asked Copilot to make the adjustments. I applied the changes to `index.html` and `styles.css`, but it did not move these fields to the same row, and Copilot also moved them to a separate location, which was not mentioned in the prompt. I undid these changes by pressing Ctrl + Z and decided to come back to this later.

Each algorithm uses a capacity constraint limit (`C`) as a parameter, which has been set to `None` so far. I wrote a prompt to add a new input field for this value and use it when calling the algorithm functions. I applied the changes to `index.html` and `scripts.js` files, but I noticed that applying the changes took a little more time than before (approximately 15 seconds). I had hit the response length limit again when Copilot was generating code for the `server.py` file, so I wrote a new prompt asking it to modify just the `server.py` file and add the vehicle capacity constraint there. However, I noticed that I had already implemented the vehicle capacity constraint in the code previously, but it was under a different variable name (`capacity` instead of `C`), so I undid the changes using Ctrl + Z.

I wanted to revisit the idea of moving the input checkboxes for single iteration and minimizing the number of routes options in the GUI next to each other instead of being on separate rows. I highlighted them and used the 'Modify using Copilot' feature to instruct moving them next to each other. Copilot quickly generated some inline CSS, and it worked as I expected. I noticed that there was some margin at the bottom of the label elements due to the rule set in `styles.css` for label elements. I overrode this by manually adding inline CSS to these two label elements, setting the bottom margin to `0`. I refreshed the page, and now the position of these elements looked good.

While I was testing the GUI, I noticed that the server called the algorithm functions with the initial capacity, which was retrieved from the uploaded `.vrp` file. So, when I changed the vehicle capacity value in the GUI, the change was not passed to the algorithm function. I looked at the code and fixed it manually because it was faster than describing the situation to Copilot and waiting for it to generate and apply the code. I tested the GUI and now the changes in vehicle capacity were successfully reflected in the algorithm function calls.

Before completing the action phase of this iteration, I wanted to fine-tune the GUI slightly by removing some margin and adding a gap between the statistics and visualization elements. I attached `styles.css` and `index.html` as context to Copilot chat and wrote a prompt regarding this. I applied the generated code, restarted the server, and it looked okay, but there was still some margin under the `h2` elements. Additionally, Copilot added a blue background for the card headers, which I had not mentioned in the prompt. I decided to fix these manually, as it was faster than using Copilot. The current GUI version after iteration 3 can be seen in Figure 5.23 on the next page.

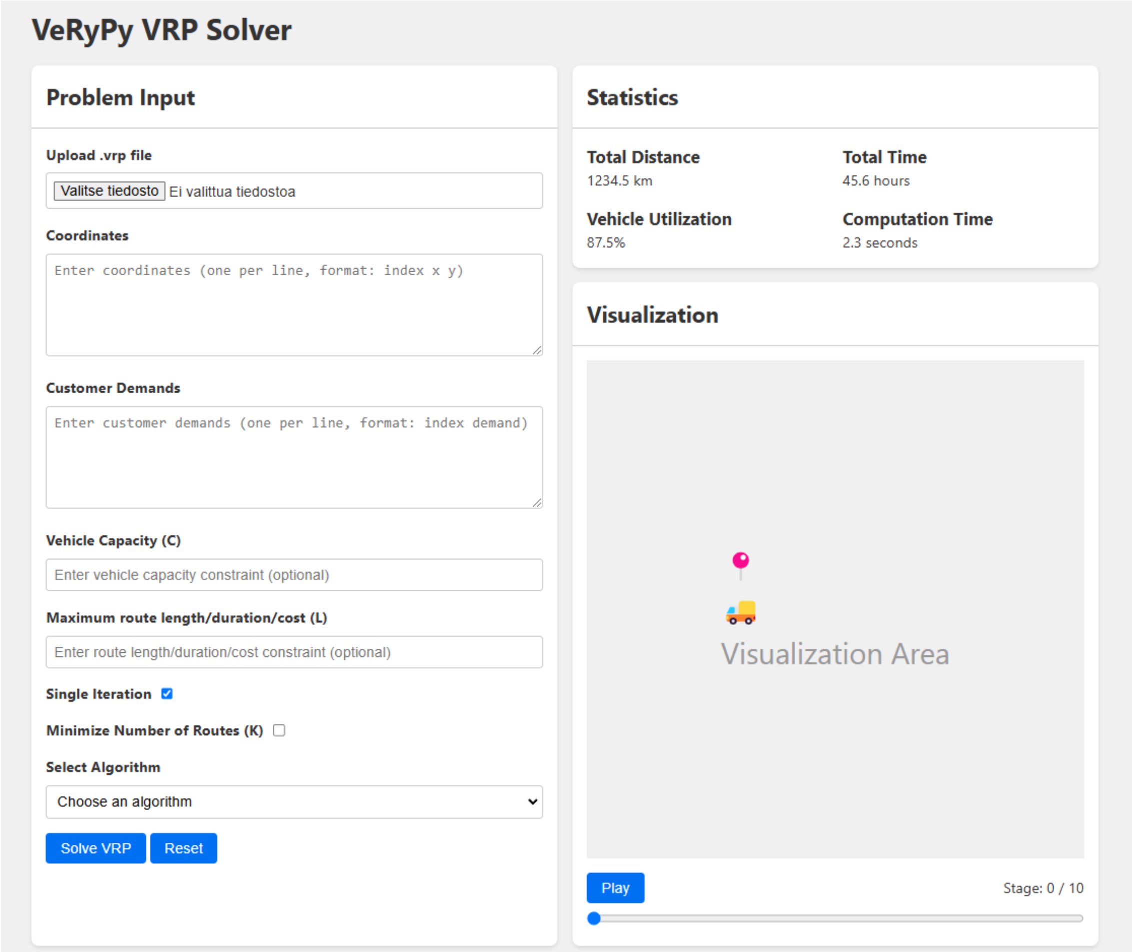


Figure 5.23. *VeRyPy GUI after iteration 3.*

5.3.3 Analysis

At the beginning of this iteration, Copilot struggled to recognize the existing architecture of the VeRyPy GUI. When prompted to refactor the `.vrp` file parsing logic, it suggested creating a separate `api.py` file using Flask, even though the `server.py` file already contained the necessary endpoint logic. While I was able to identify this issue and correct the approach, a less experienced developer might have accepted Copilot's suggestion, leading to unnecessary complexity and a deviation from the project's requirements. Although Copilot can generate functional solutions, they do not always align with the intended design unless explicitly mentioned in the prompt.

Another major issue I encountered was browser caching, which prevented certain changes from appearing in the GUI. Copilot did not detect the issue or offer meaningful debugging steps. This was expected, as Copilot seems to operate solely on source code and lacks visibility into runtime behavior, browser settings, and other environmental factors. This limitation highlights a key drawback of AI-powered coding tools: they cannot directly observe the rendered output or the full development environment.

Clear and precise prompts played a crucial role in this iteration. Adding relevant code files as context significantly improved Copilot's responses and helped maintain consistency. While it is possible to provide the entire codebase as context, I found that selectively including only the relevant files was sufficient. Although Copilot automatically attaches the currently open file as context, I had to manually attach other relevant files for each prompt.

In some cases, it was faster to make small edits manually rather than waiting for Copilot to generate these changes. Copilot also occasionally reverted manually implemented changes between prompts, even when they were unrelated to the newly generated code. Additionally, it failed to remove obsolete code in situations where it introduced an updated version of a function, such as the event listener for inputting the customer locations. This behavior of reverting prior modifications suggests that Copilot lacks either some part of the context or an understanding of which parts of the code have become redundant.

5.3.4 Conclusion

During iteration 3, the existing heuristic algorithms of VeRyPy were integrated into the GUI. Users can now select from any of the implemented algorithms to solve VRPs. The solution and routes are currently visible only in the server logs, but they will be displayed in the GUI in the next iteration.

5.4 Iteration 4: Solution Visualization, Metrics, and Export Options

The goal of iteration 4 is to generate the VRP solution visualization and display it in the GUI. This iteration also includes displaying the solution metrics within the GUI. Additionally, functionality for exporting the visualization and metrics is implemented during this iteration.

5.4.1 Planning

Refer to Iteration 4: Solution Visualization, Metrics, and Export Options in Appendix B for the iteration 4 plan including the user stories and features implemented in this iteration.

First, I will display the solution metrics in the VeRyPy GUI. Currently, the metrics are shown in the server logs, so I need to push them to the frontend and update the corresponding DOM elements. I expect this part of the iteration to be relatively straightforward since the metrics are already present in the server logs, but the GUI needs to be updated to reflect the solution metrics.

After the solution metrics are updated in the GUI, I will proceed with generating the solution visualizer. I expect this to be somewhat time-consuming and will likely require some iteration and fine-tuning with Copilot. While Copilot may suggest a visualizer library for this task, it should be achievable without relying on any external packages.

Once the visualizer is complete, I will implement functionality for exporting the solution metrics and visualization. I anticipate this to be straightforward for Copilot to generate, as the data is already present in the DOM.

5.4.2 Action

At the beginning of the action phase of this iteration, I wrote a prompt outlining the plan for the iteration and attached the codebase as context. I anticipate that Copilot will lose this context after a certain number of prompts and code generations. This prompt and the partial response can be seen in Figure 5.24 on the next page.

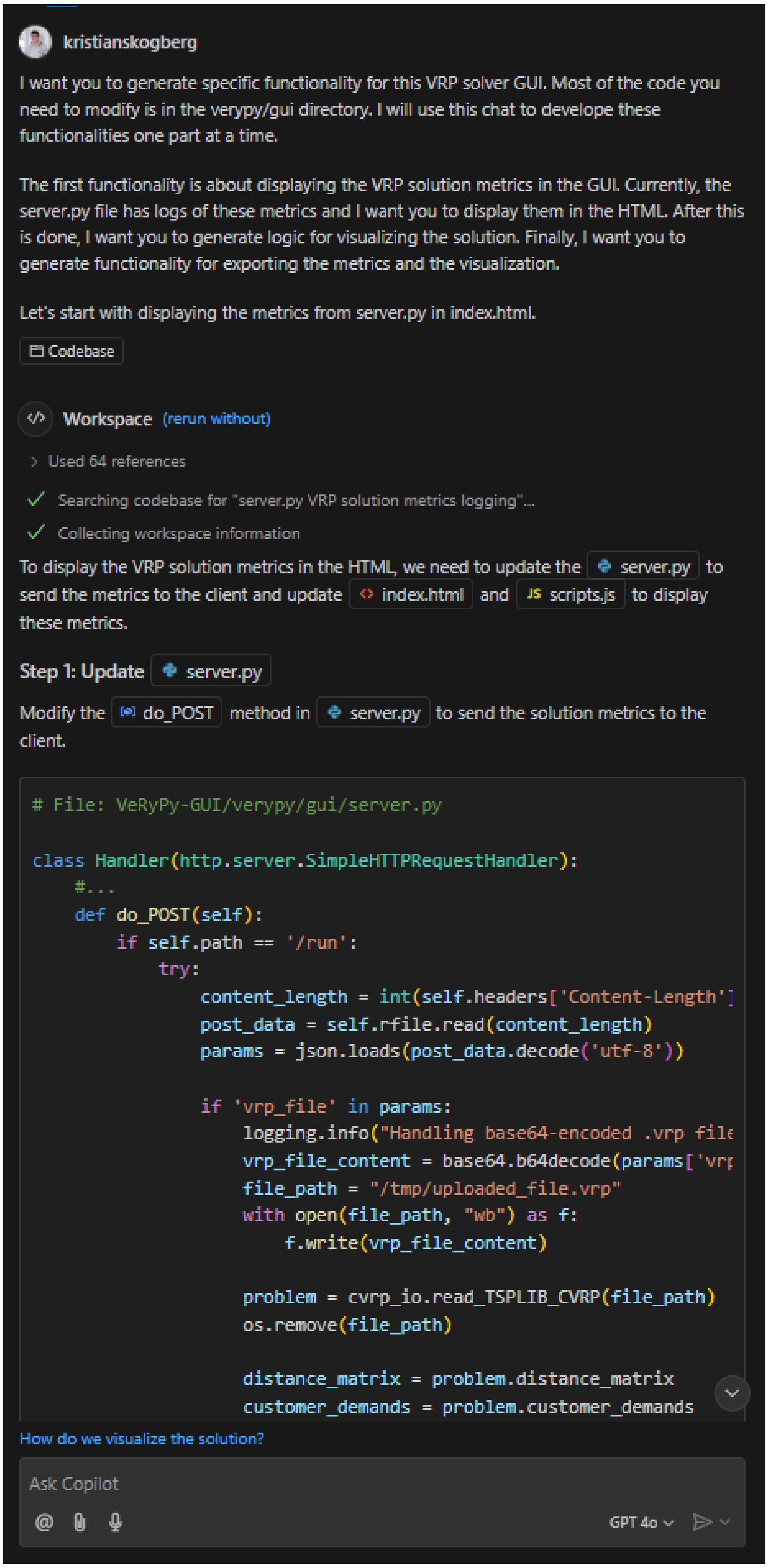


Figure 5.24. *Prompt for displaying VRP solution metrics in the GUI and a partial response.*

The code suggestion for displaying the metrics in the GUI seemed good. I applied the changes to `server.py`, but it took over a minute to apply. The file size of `server.py` decreased from 187 rows to 157 rows as Copilot removed some of the logging. Applying changes to the `index.html` file (about 100 rows) was much faster. However, when applying the code to the `scripts.js` file (currently 232 rows), I received an error: *Failed to apply code block*. I clicked 'apply code' again, and after approximately 2 minutes, the changes went through. After applying them, I restarted the server and tested the GUI. The metrics were displayed correctly in the GUI after solving the `E-n51-k5.vrp`, as shown in Figure 5.25.

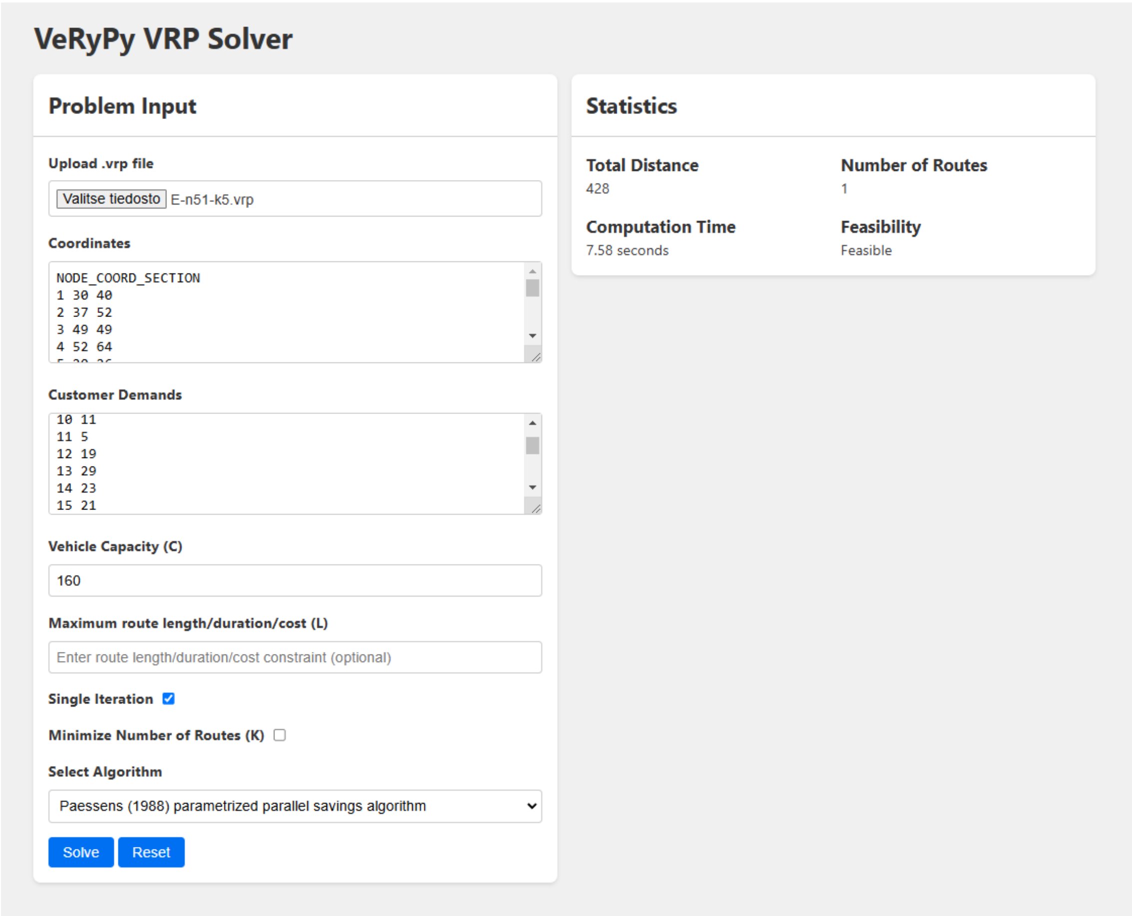


Figure 5.25. *Initial version of displaying VRP solution metrics in the VeRyPy GUI.*

Copilot had rounded the elapsed solution time to 2 decimal places, but I changed it manually to 4 decimal places. I also removed the alert that appeared after solving the VRP and changed 'Statistics' to 'Solution Metrics' in the `index.html` file. After restarting the server, these small changes were updated.

I added the logging of the parameters and solution back to `server.py` manually. I noticed that the number of routes was always one for some reason. Upon reviewing the logs, I found that capacity was always set to None. I examined `scripts.js` and discovered that

Copilot had altered the capacity parameter in the data object before sending the `POST` request to the server. I fixed this manually.

The function responsible for calculating the feasibility of the solution returns an array of three boolean values: the first boolean is the covering feasibility, the second is the capacity feasibility, and the last is the route cost feasibility. I wrote a prompt explaining this to Copilot and attached the `scripts.js` and `index.html` files, as these were the only files needing modification in this case. Applying the code to `index.html` was fast, but applying the code to `scripts.js` took almost 3 minutes. After applying the changes and restarting the server, I tested the GUI, and Copilot had added these feasibility metrics separately, as requested in my prompt. However, I noticed that Copilot had also reverted some code lines that I had manually changed or deleted after the last code generation. I reverted these changes manually.

I noticed that solving a VRP using different algorithms and settings did not reset the metrics while the VRP was being solved. This was not ideal in terms of usability, as the GUI displayed metrics from solving a VRP using previous settings and only updated them after the VRP had been solved. I wrote a prompt asking Copilot to change the metric values to empty while solving the VRP. I applied the changes to `scripts.js`, and it worked well, although it took some time for Copilot to finish the code generation. I decided to edit the empty values to three dots because I thought it looked better in the GUI. I also changed the initial values of the metrics from 'N/A' to '-' as a personal preference. The current VeRyPy GUI with the feasibility metrics can be seen in Figure 5.26 on the next page.

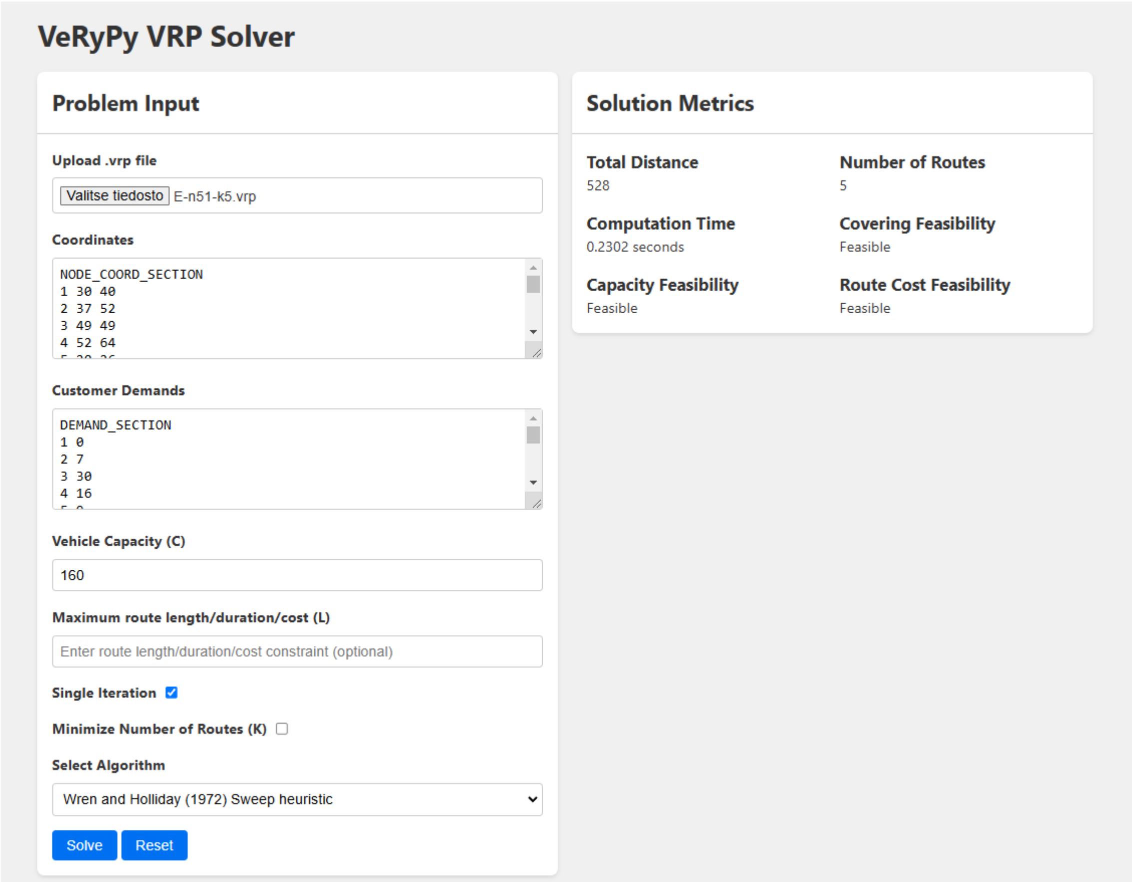
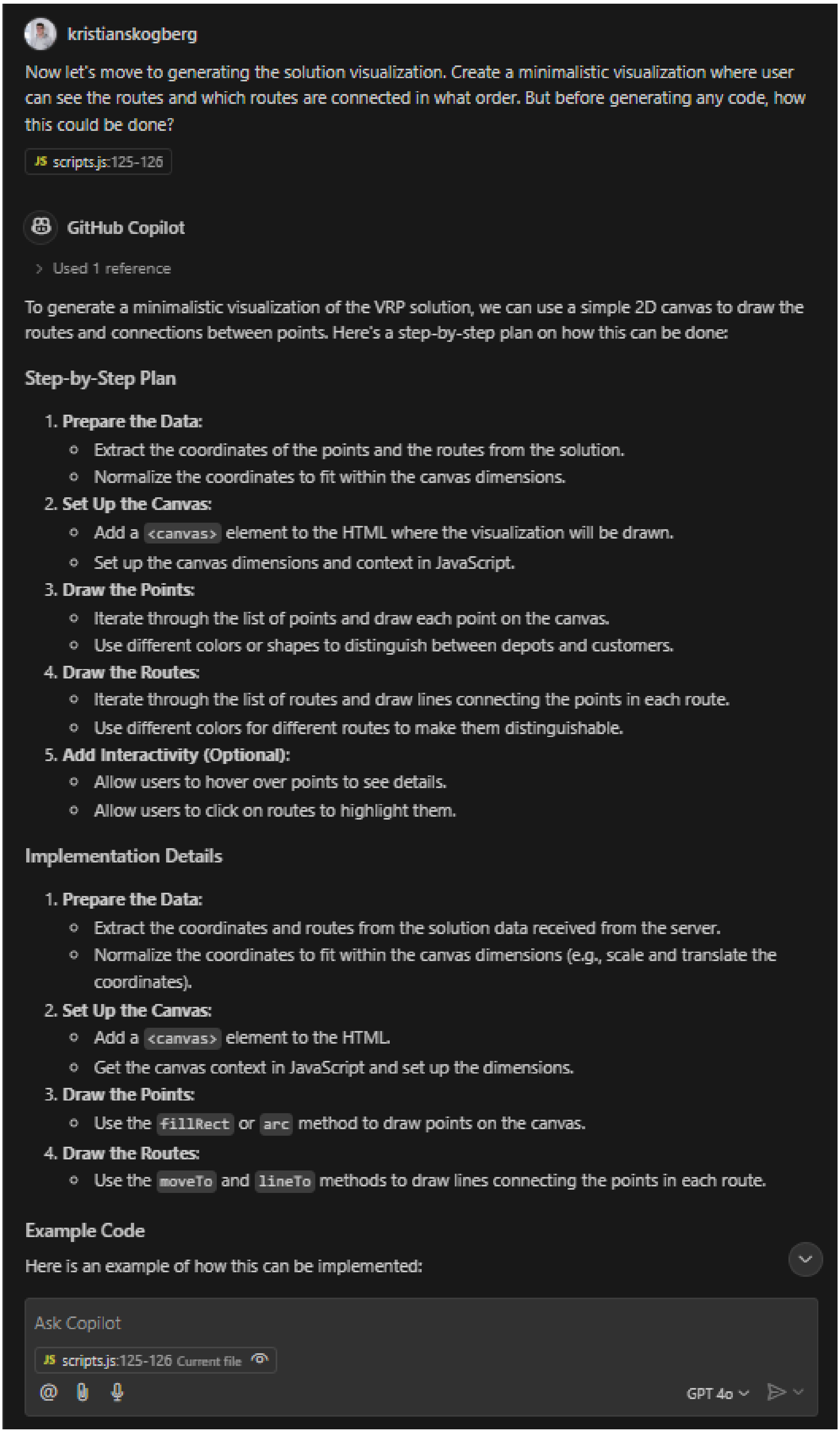


Figure 5.26. *Feasibility metrics added to the solution metrics element in the GUI.*

Now that the metrics are added to the GUI, I moved on to generating the visualization as mentioned in the planning phase of this iteration. I wrote a prompt to Copilot asking for ideas on how the solution visualization could be implemented. This prompt can be seen in Figure 5.27 on the next page.



kristianskogberg

Now let's move to generating the solution visualization. Create a minimalistic visualization where user can see the routes and which routes are connected in what order. But before generating any code, how this could be done?

JS scripts.js:125-126

GitHub Copilot

› Used 1 reference

To generate a minimalistic visualization of the VRP solution, we can use a simple 2D canvas to draw the routes and connections between points. Here's a step-by-step plan on how this can be done:

Step-by-Step Plan

1. **Prepare the Data:**
   - Extract the coordinates of the points and the routes from the solution.
   - Normalize the coordinates to fit within the canvas dimensions.
2. **Set Up the Canvas:**
   - Add a `<canvas>` element to the HTML where the visualization will be drawn.
   - Set up the canvas dimensions and context in JavaScript.
3. **Draw the Points:**
   - Iterate through the list of points and draw each point on the canvas.
   - Use different colors or shapes to distinguish between depots and customers.
4. **Draw the Routes:**
   - Iterate through the list of routes and draw lines connecting the points in each route.
   - Use different colors for different routes to make them distinguishable.
5. **Add Interactivity (Optional):**
   - Allow users to hover over points to see details.
   - Allow users to click on routes to highlight them.

Implementation Details

1. **Prepare the Data:**
   - Extract the coordinates and routes from the solution data received from the server.
   - Normalize the coordinates to fit within the canvas dimensions (e.g., scale and translate the coordinates).
2. **Set Up the Canvas:**
   - Add a `<canvas>` element to the HTML.
   - Get the canvas context in JavaScript and set up the dimensions.
3. **Draw the Points:**
   - Use the `fillRect` or `arc` method to draw points on the canvas.
4. **Draw the Routes:**
   - Use the `moveTo` and `lineTo` methods to draw lines connecting the points in each route.

Example Code

Here is an example of how this can be implemented:

Ask Copilot

JS scripts.js:125-126 Current file

GPT 4o

Figure 5.27. *Prompt for generating the VRP solution visualization and a partial response.*

Copilot suggested creating a canvas HTML element for the visualization, which I expected. Copilot generated code for the `index.html` and `scripts.js` files but not for the `server.py` file. I wrote another prompt asking Copilot to modify the server to work with this solution and attached it as context. I applied the changes to `server.py` and `scripts.js` and tested the GUI. I encountered an error related to reading data in the map, but the GUI still produced the visualization, which looked great. The initial visualization can be seen in Figure 5.28 below.

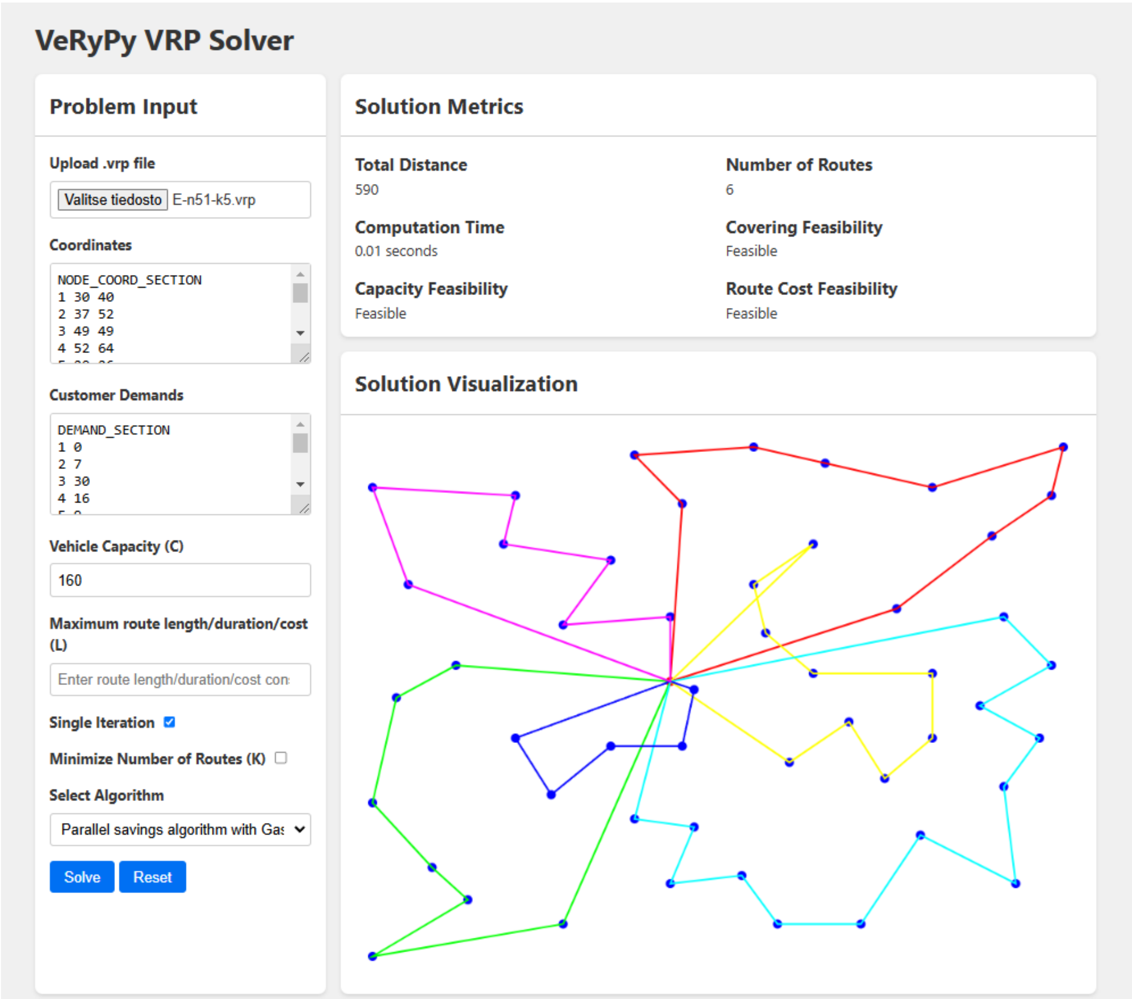
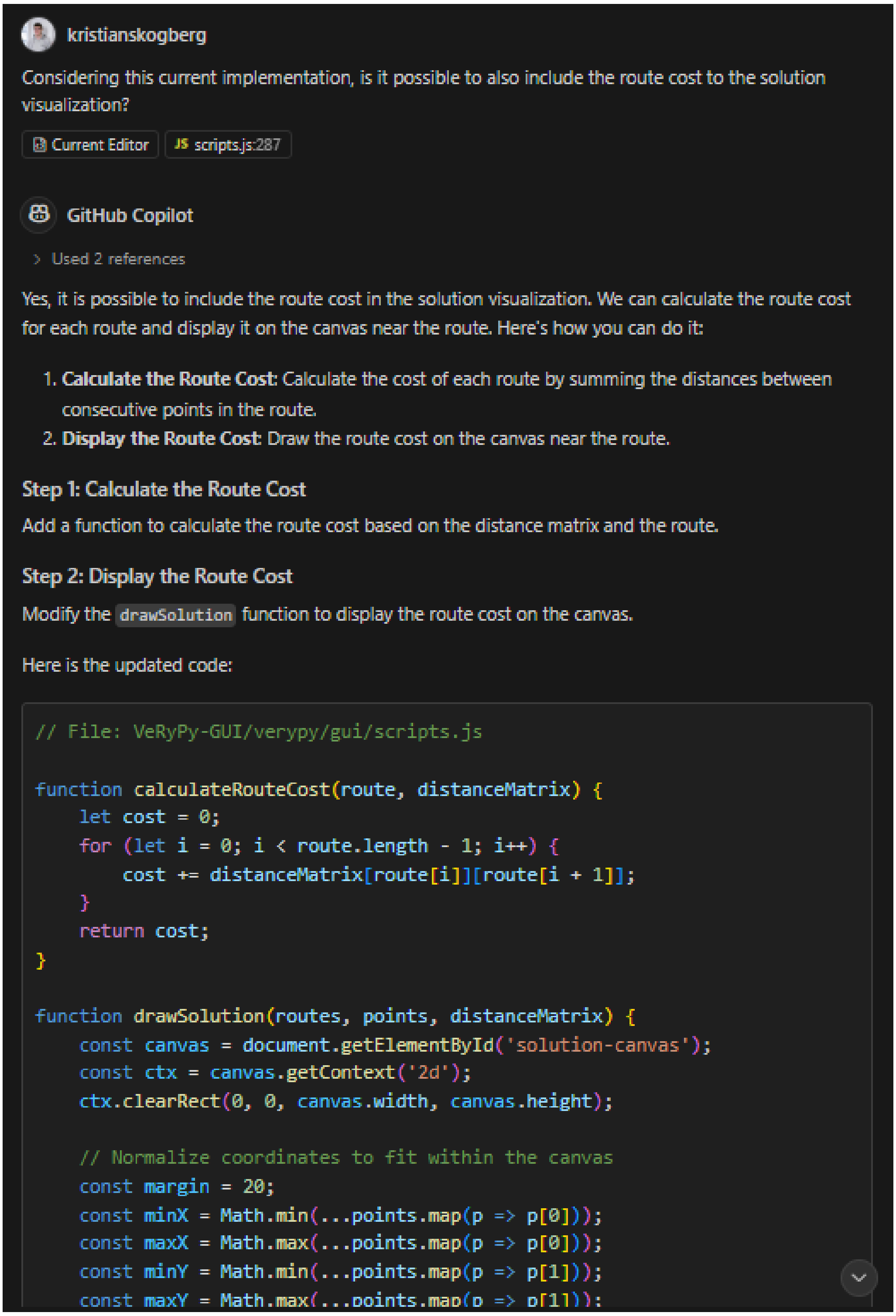


Figure 5.28. *Initial VRP solution visualization in the VeRyPy GUI.*

The current implementation in `server.py` performs some text formatting for the route to display it more conveniently in the server logs. I asked GitHub Copilot to remove the text formatting, as it was likely causing errors when drawing the visualization, and the routes array should be used instead. After applying this change, the error related to reading data in the map disappeared. However, I noticed that some routes were drawn in yellow, and light colors are not ideal for usability due to the white background. I highlighted the `drawSolution` function in `scripts.js` and used the 'Modify using Copilot' feature to avoid using light colors when drawing the visualization.

I also wanted to display the route cost in the visualization. I wrote a prompt in Copilot chat about this and attached the code files that were currently open in the editor (`scripts.js`, `index.html`, and `server.py`) as context. This prompt and the partial response can be seen in Figure 5.29 below. I applied the code but noticed that Copilot had reverted the small changes I had made manually between this prompt and the previous one.



kristianskogberg

Considering this current implementation, is it possible to also include the route cost to the solution visualization?

Current Editor | JS scripts.js:287

GitHub Copilot

› Used 2 references

Yes, it is possible to include the route cost in the solution visualization. We can calculate the route cost for each route and display it on the canvas near the route. Here's how you can do it:

1. **Calculate the Route Cost:** Calculate the cost of each route by summing the distances between consecutive points in the route.
2. **Display the Route Cost:** Draw the route cost on the canvas near the route.

Step 1: Calculate the Route Cost

Add a function to calculate the route cost based on the distance matrix and the route.

Step 2: Display the Route Cost

Modify the `drawSolution` function to display the route cost on the canvas.

Here is the updated code:

```
// File: VeRyPy-GUI/verypy/gui/scripts.js

function calculateRouteCost(route, distanceMatrix) {
    let cost = 0;
    for (let i = 0; i < route.length - 1; i++) {
        cost += distanceMatrix[route[i]][route[i + 1]];
    }
    return cost;
}

function drawSolution(routes, points, distanceMatrix) {
    const canvas = document.getElementById('solution-canvas');
    const ctx = canvas.getContext('2d');
    ctx.clearRect(0, 0, canvas.width, canvas.height);

    // Normalize coordinates to fit within the canvas
    const margin = 20;
    const minX = Math.min(...points.map(p => p[0]));
    const maxX = Math.max(...points.map(p => p[0]));
    const minY = Math.min(...points.map(p => p[1]));
    const maxY = Math.max(...points.map(p => p[1]));
```

Figure 5.29. *Prompt for adding route cost to the VRP solution and a partial response.*

However, applying the modified code to `scripts.js` broke the visualization functionality, which was working previously. The current GUI is presented in Figure 5.30 below. I also encountered an error while solving the `E-n51-k5.vrp` file.

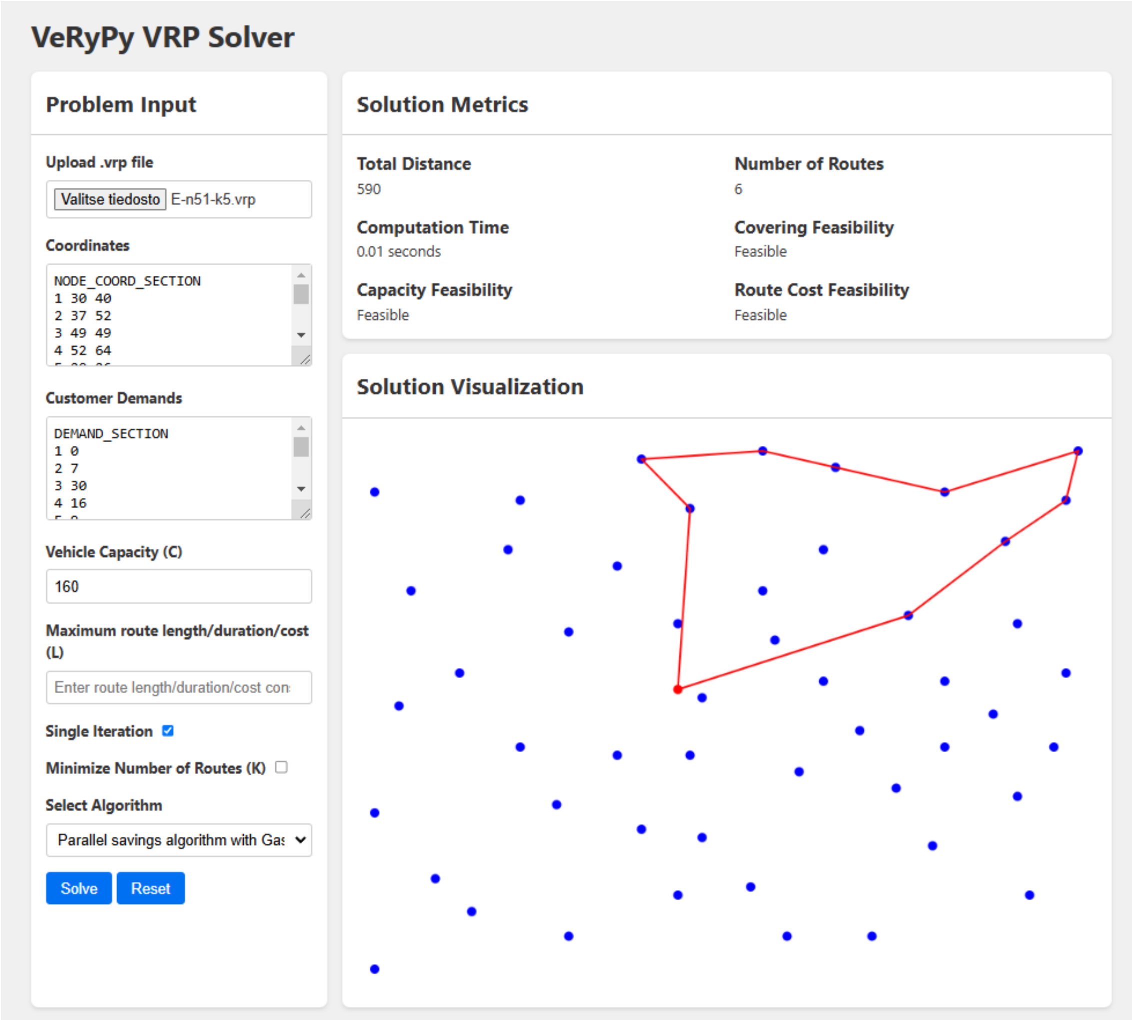


Figure 5.30. *Error in the solution visualization in the VeRyPy GUI.*

I asked Copilot to fix the visualization and copied the error message into the Copilot chat. Copilot suggested checking if the distance matrix exists in the `server.py` file. It did not exist, as Copilot had not modified the `server.py` file during the previous prompt. I applied the code, restarted the server, and encountered another error because the distance matrix is a NumPy array, which is not JSON serializable. I copied and pasted this error to Copilot and applied the generated code to both the `server.py` and `scripts.js` files. Now, it was working properly, but the route costs were placed at the same location in the visualization, making them impossible to read.

I wrote a prompt asking Copilot to display the route costs in a table and attached the `index.html`, `scripts.js`, and `server.py` files as context. I applied the code first to the `index.html` file and then to the `scripts.js` file. After restarting the server and testing the GUI, Copilot had added a card element for the route costs below the visualization, and the route costs were correctly populated in the table. The table lacked CSS styling, so I asked Copilot to create some CSS for it to improve its appearance. After applying the CSS, the route costs table looked great, as shown in Figure 5.31.

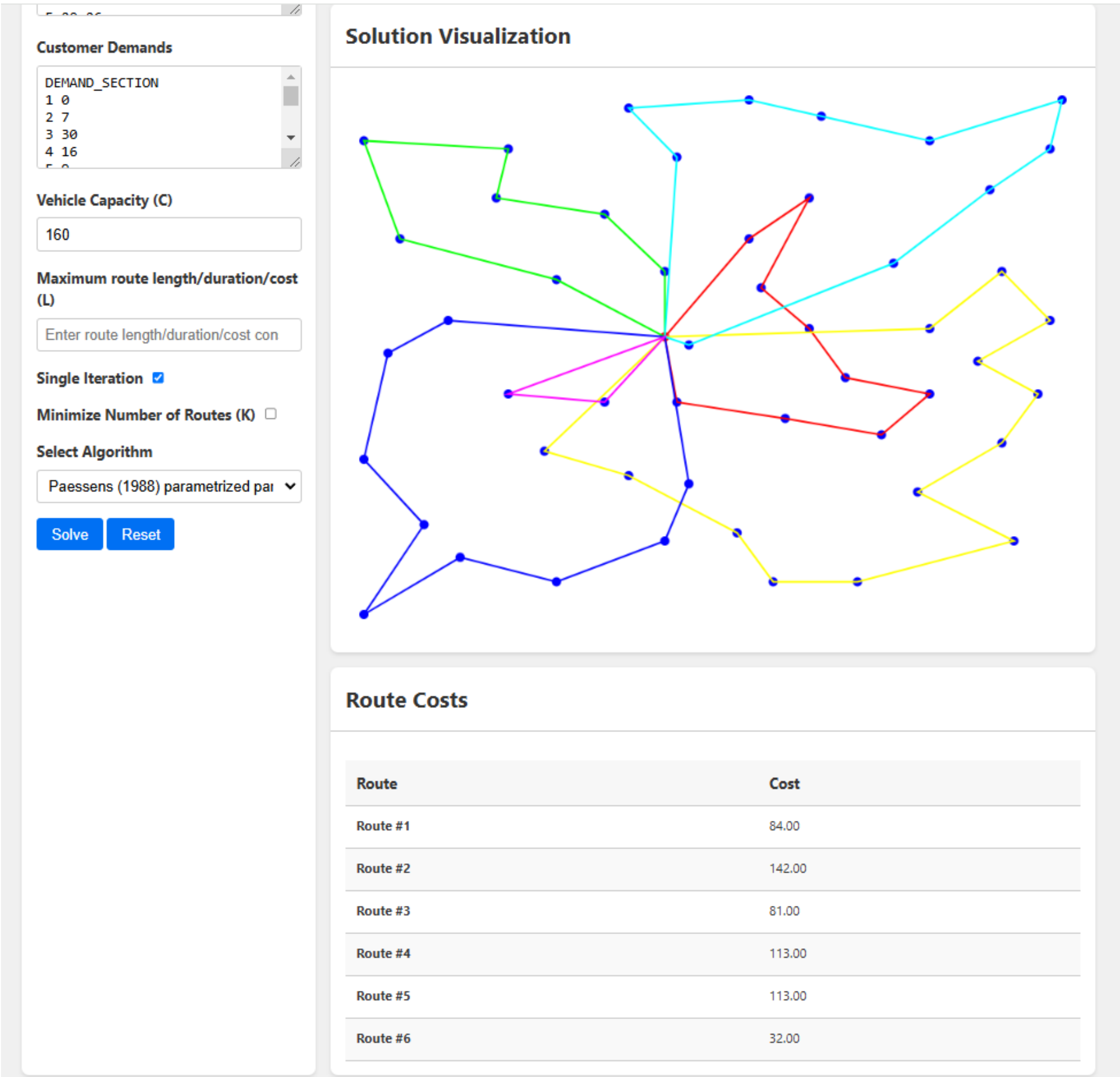


Figure 5.31. *Added an element for route costs to the VeRyPy GUI.*

Next, I asked Copilot to display the route data in the route costs table instead of showing only the route number. Copilot suggested adding a new column to the route costs table to display the details of each route. I applied the code, and the route details were now correctly shown in the route costs element, as illustrated in Figure 5.32.

Route Costs

Route	Cost	Details
Route #1	112.00	0 -> 2 -> 20 -> 35 -> 36 -> 3 -> 28 -> 31 -> 26 -> 8 -> 0
Route #2	93.00	0 -> 5 -> 38 -> 49 -> 9 -> 16 -> 11 -> 32 -> 1 -> 22 -> 0
Route #3	109.00	0 -> 14 -> 25 -> 13 -> 41 -> 19 -> 40 -> 42 -> 0
Route #4	140.00	0 -> 17 -> 37 -> 44 -> 15 -> 45 -> 33 -> 10 -> 39 -> 30 -> 34 -> 50 -> 21 -> 29 -> 0
Route #5	46.00	0 -> 18 -> 4 -> 47 -> 12 -> 46 -> 0
Route #6	90.00	0 -> 24 -> 43 -> 7 -> 23 -> 48 -> 6 -> 27 -> 0

Figure 5.32. *Added route details to the route costs element in the VeRyPy GUI.*

I wrote a prompt where I compiled several small changes that I had previously made between prompts but which Copilot had reverted. My observation was that Copilot sometimes overlooks manual edits or those made using the 'Modify using Copilot' or 'Review using Copilot' features. I applied the generated code, which successfully addressed most of the minor changes I had listed. However, Copilot did not detect the removal of certain alert messages, so I manually copied and pasted the updated `scripts.js` file. After doing this, I noticed that some existing event listeners were missing—likely due to reaching the GPT-4o output limit. I reverted the pasted changes by pressing Ctrl + Z.

I also wanted to display the route utilization rate for each route in the GUI. I wrote a prompt describing this feature and applied the code Copilot generated to the `index.html` and `scripts.js` files. Copilot created a function in `scripts.js` to calculate utilization rates using customer demands. However, since customer demands were not previously returned from `server.py`, the calculation failed. I wrote another prompt asking Copilot to fix this, likely by modifying `server.py`, and after applying the code, the error was resolved—but the GUI showed `NaN` for every utilization rate.

To debug, I added manual logging and discovered that the calculation relied on a capacity attribute from the server response, which was missing. I prompted Copilot to include this attribute in the response data object in `server.py`. This time, Copilot did not revert the manual logging I had added earlier. After applying the changes and restarting the server, the route utilization rates were calculated and displayed correctly in the GUI.

Next, I wanted to display the corresponding colors from the visualization alongside the route names in the Route Metrics table to help users more easily associate each table entry with its respective visualized route. I applied the generated changes to the `scripts.js` file, tested the GUI, and was satisfied with the outcome. I also renamed the 'Route Costs' card element to 'Route Metrics' in the `index.html` file, as the element now includes more than just cost information. The updated GUI is shown in Figure 5.33 below.

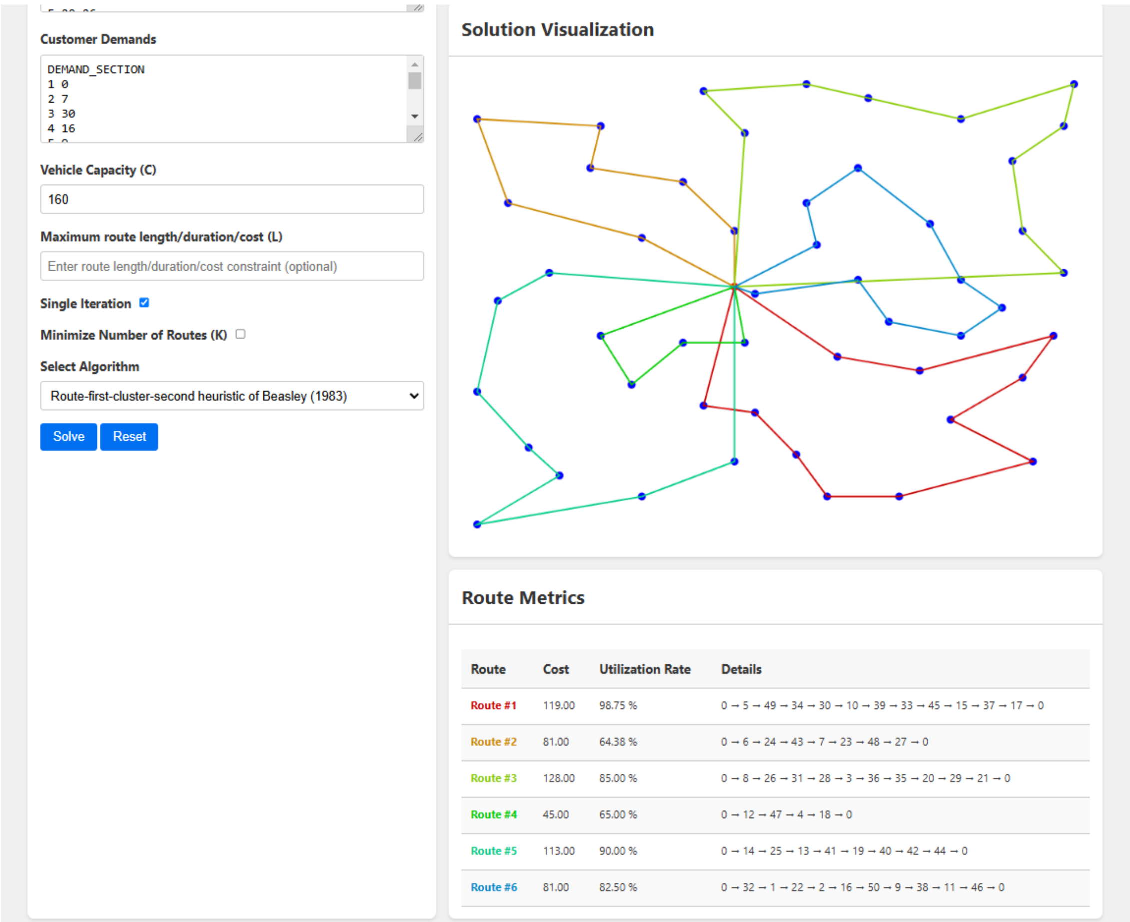


Figure 5.33. *Improved solution visualization and route metrics in the VeRyPy GUI.*

Now I am quite happy with the visualization and metrics parts of the GUI so I will move to generating the exporting functionality for them. I wrote a prompt for Copilot about what would be the best way to add the export functionality to the GUI without having to install any external dependencies. Copilot generated code where new export buttons were added to the `index.html` file as well as corresponding functions in `scripts.js` file. I applied the changes and tested the GUI. Exporting the visualization as an image (`.png`) was working perfectly, but exporting the metrics as `.csv` resulted in an empty `.csv` file with just the table headers from the route metrics table. I described this situation to Copilot and asked it to fix them.

I applied the changes, but nothing was changed in the `scripts.js` file. I wrote a prompt asking Copilot to check how the route metrics are stored in the table and modify the `exportMetrics` function based on that. I applied the code which had a minor change in `exportMetrics` function but it still did not work. I added logging manually and found that the issue is related to how the CSV content is being encoded. I described the situation to Copilot and applied the changes and now exporting the metrics as CSV worked correctly.

I extended the `exportMetrics` function to include overall solution metrics such as total distance (cost) and computational time. After applying the generated code, I verified that the export function worked correctly. To improve clarity and traceability, I then modified the default filename of the exported metrics file to include the selected algorithm and the local timestamp. This change required a few rounds of prompt refinement. The final contents of the exported metrics file are shown in Figure 5.34 below.


```json
{
  "algorithm": "Clarke & Wright (1964) parallel savings algorithm",
  "solutionMetrics": {
    "totalDistance": "580",
    "numRoutes": "6",
    "computationTime": "0.0176 seconds",
    "coveringFeasibility": "Feasible",
    "capacityFeasibility": "Feasible",
    "routeCostFeasibility": "Feasible"
  },
  "routeMetrics": [
    {
      "route": "Route #1",
      "cost": "98.00",
      "utilizationRate": "76.25 %",
      "details": "0 -> 6 -> 24 -> 43 -> 7 -> 23 -> 48 -> 1 -> 32 -> 27 -> 0"
    },
    {
      "route": "Route #2",
      "cost": "123.00",
      "utilizationRate": "100.00 %",
      "details": "0 -> 8 -> 26 -> 31 -> 28 -> 3 -> 36 -> 35 -> 20 -> 2 -> 22 -> 0"
    },
    {
      "route": "Route #3",
      "cost": "147.00",
      "utilizationRate": "100.00 %",
      "details": "0 -> 10 -> 49 -> 9 -> 50 -> 29 -> 21 -> 34 -> 30 -> 39 -> 33 -> 45 -> 15 -> 0"
    },
    {
      "route": "Route #4",
      "cost": "55.00",
      "utilizationRate": "65.00 %",
      "details": "0 -> 12 -> 5 -> 38 -> 16 -> 11 -> 46 -> 0"
    },
    {
      "route": "Route #5",
      "cost": "117.00",
      "utilizationRate": "97.50 %",
      "details": "0 -> 14 -> 25 -> 13 -> 41 -> 40 -> 19 -> 42 -> 44 -> 37 -> 17 -> 0"
    },
    {
      "route": "Route #6",
      "cost": "40.00",
      "utilizationRate": "46.88 %",
      "details": "0 -> 18 -> 4 -> 47 -> 0"
    }
  ]
}
```


Figure 5.34. *Solution metrics exported as a JSON file.*

Lastly, I wanted to move the 'Route Metrics' into the same card element as the `Solution Metrics` to create a more unified layout. I also wanted to reposition the export buttons into the header section to improve visibility and accessibility. I wrote a prompt asking Copilot to make these changes without affecting any other part of the GUI, but Copilot repeatedly removed the visualization element from `index.html`. After encountering this issue multiple times, I decided to make the necessary adjustments manually in the `index.html` and `styles.css` files. The current state of the VeRyPy GUI after iteration 4 can be seen in Figure 5.35 below.

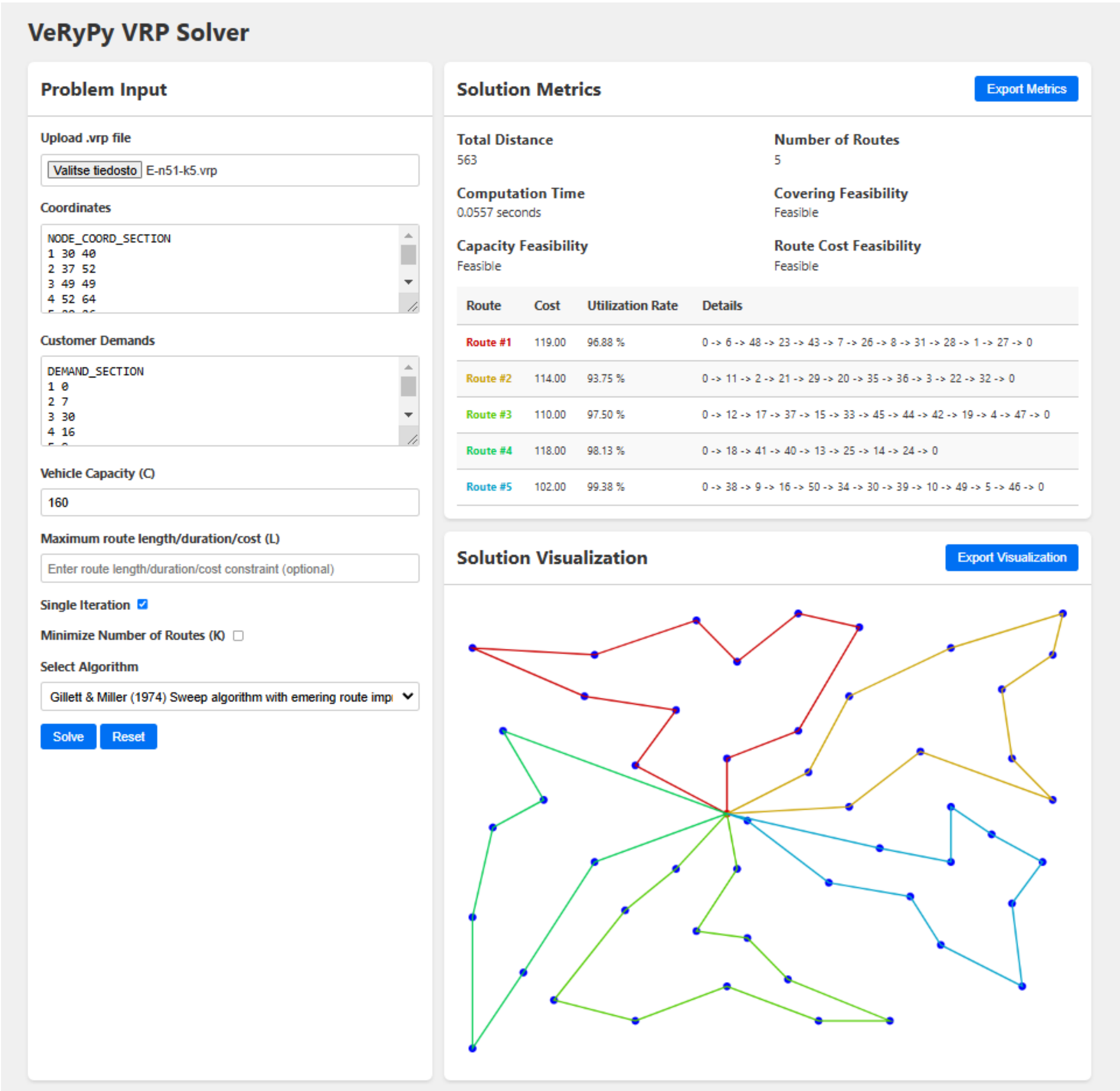


Figure 5.35. *VeRyPy GUI after iteration 4.*

5.4.3 Analysis

One of the key observations of iteration 4 was that Copilot's ability to generate code efficiently depended heavily on the size of the modified files. Applying changes to smaller files such as `index.html` was quick, but modifying larger files like `scripts.js` and `server.py` sometimes took several minutes. The context window of GPT-4o, as well as the overall service load of the API, are likely the most significant factors affecting Copilot's performance. This raised concerns about Copilot's scalability, particularly when working with larger or more complex codebases.

Furthermore, Copilot frequently reverted manual changes when generating new code, indicating a lack of persistent memory between prompts. It also struggled to maintain an accurate understanding of the evolving state of the VeRyPy GUI across multiple iterations. To address these limitations, I had to carefully review Copilot's output and manually reapply lost changes, slowing down the AI-powered development process.

When implementing the VRP solution visualization, Copilot suggested using an HTML `<canvas>` element, which worked well for this task. However, it failed to account for some usability aspects, such as color contrast and the readability of the route visualization. Initially, the colors included light shades that blended into the white background of the VeRyPy GUI. I had to manually adjust these colors and provide additional prompts to improve visibility. This demonstrated a broader limitation of AI-generated code, where Copilot sometimes lacked awareness of certain accessibility and usability principles. Similarly, while Copilot successfully generated a table for displaying route metrics, the table initially had no styling. Even though Copilot managed the functional aspects well, I still had to refine the aesthetics and user experience to make the interface more intuitive.

Maintaining consistency across different parts of the application was another challenge. When generating code for route utilization rates, Copilot failed to recognize that the server response lacked the necessary data. This caused errors, which I identified manually and described in prompts to Copilot for fixing. Additionally, JSON serialization errors arose because Copilot did not account for the fact that the server response included a NumPy array, which is not automatically JSON serializable. Since this issue was not specified in my prompt, Copilot did not anticipate it. Although Copilot generated code relatively quickly most of the time during this iteration, it did not seem to validate whether those changes would integrate smoothly into the GUI. As a result, I spent some time debugging and refining the AI-generated code.

Implementing the solution export feature further emphasized the necessity of manual testing and validation. Copilot successfully generated the function for exporting the visualization, but exporting metrics as a CSV file initially failed due to encoding issues. Since Copilot could not independently detect or resolve this issue, I tested and debugged it manually. Additionally, Copilot removed the visualization component when modifying the layout. This reinforced the importance of carefully reviewing AI-generated code.

5.4.4 Conclusion

Iteration 4 revolved around implementing the VRP solution visualization, solution metrics, and export options features for the VeRyPy GUI. In most cases, Copilot was able to provide valuable assistance, which significantly reduced the amount of code that needed to be written manually. It generated code with good structure, but sometimes the code did not function as intended. On the other hand, Copilot struggled with maintaining consistency across this and previous iterations, likely due to the limited context window of Copilot and/or GPT-4o. The frequent need for manual corrections indicates that, while Copilot is a powerful tool, it is not yet capable of fully replacing human oversight in constructing a working GUI.

5.5 Iteration 5: Improved Problem Input

Iteration 5 focuses on improving the problem input section of the VeRyPy GUI. The initial version of this section was generated in Iteration 2 and fine-tuned during Iteration 3. In this iteration, the usability of the problem input section is improved so that users no longer have to manually add indexes for coordinates and customer demands, as these will now be managed automatically. This enhances the usability of the VeRyPy GUI and reduces the likelihood of users encountering errors while entering or modifying their problem.

Another improvement to the problem input section is the ability to enter coordinates in geographical format using longitude and latitude. This could be implemented using a toggle button or a dropdown menu, which would default to Cartesian coordinates. The user can then switch between the two input formats as needed. Additionally, since VeRyPy supports reading and solving `.tsp`-formatted files, support for importing such files will also be added to the GUI during this iteration.

At the end of this iteration, users should be able to input details about their problem more easily, either manually, by uploading a `.vrp` or `.tsp`-formatted file, or by modifying the content of an uploaded file. Users should also have the option to input coordinates in longitude and latitude. Lastly, input validation will be improved.

5.5.1 Planning

Refer to Iteration 5: Improved Problem Input in Appendix B for the iteration 5 plan including the user stories and features implemented in this iteration.

The plan is to first improve the existing coordinate inputs and the input fields for customer demand so that the indexes are automatically managed in the code. This likely requires changes only to the `scripts.js` file, as it is responsible for the frontend functionality. In other words, the indexes for coordinates and customer demands will still exist but will be hidden in the GUI and inaccessible to the user.

After the indexes are hidden and managed successfully, I will test the GUI by inputting problem details manually. I expect there may be some errors, as the GUI has primarily been tested by uploading the `E-n51-k5.vrp` file rather than inputting the VRP details manually.

Finally, I will guide Copilot to generate the functionality for toggling between regular coordinates and geographical coordinates using longitude and latitude. The GUI will require new input fields for this functionality, as well as event listeners for these fields. This should be easy to add to the existing `scripts.js` file, but it may require some modifications to the event listener of the 'solve' button.

The changes and improvements to be generated in this iteration are not complex and should be straightforward to implement. However, I expect that Copilot may attempt to revert some unnecessary changes, as observed during the previous iterations, and it may generate code that could break the GUI. For these reasons, I must review each change carefully and discard any unnecessary or code-breaking modifications.

5.5.2 Action

I created a new chat for this iteration and copied the first paragraph of the planning chapter into the Copilot chat as a prompt. Copilot suggested updating the `drawSolution` function as well as the event listeners to process indexes automatically. I applied the code to the `scripts.js` file and tested the GUI by uploading the previously used `E-n51-k5.vrp` file from VeRyPy.

I noticed that the indexes were correctly hidden from the GUI, but changes made to the coordinates or customer demands manually in the input fields were not reflected in the solution and resulted in errors. The reason for this is that the current implementation assumes that the user has uploaded a `.vrp` file, as the server uses it to calculate the distance matrix and points. I wrote a prompt about this, and Copilot generated code for the `scripts.js` and `server.py` files.

I applied these changes, restarted the server, and tested the GUI. I input some random VRP data and encountered a new error while running the algorithm, which I had not seen before. I copied and pasted this error message into Copilot, and it generated new changes, which I applied. However, I encountered a different error, which I copied and pasted into the Copilot chat. After testing the GUI, I found that the solution worked when there was one or two coordinate pairs, but it did not work when there were more pairs.

I thought about the best way to solve this problem. I wanted Copilot to generate a temporary `.vrp` file using the provided parameters (contents of the input fields) from the frontend. I attached the relevant files as context and wrote a prompt regarding this, which is presented in Figure 5.36 on the next page.

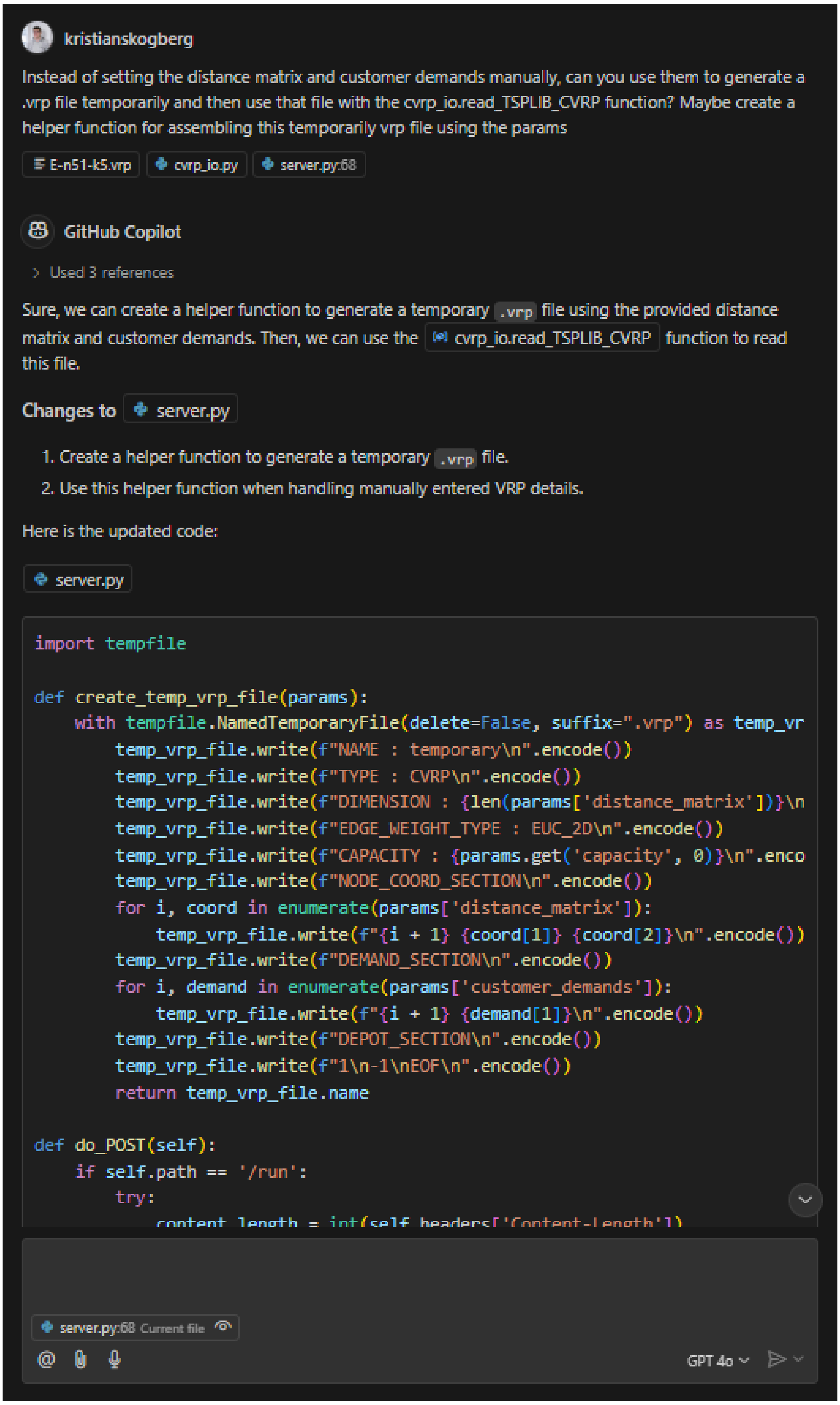


Figure 5.36. *Prompt for generating a function to create a temporary `.vrp` file and a partial response.*

Copilot suggested new code for a function that generates a temporary `.vrp` file using parameters from the HTTP request sent by the frontend (specifically `scripts.js`). I applied these changes and restarted the server. After testing the GUI, I found that it was working correctly when adding coordinates and customer demands manually, and I did not encounter the previously mentioned errors. However, if the user uploads a .vrp file and then modifies the coordinates or customer demands, these changes are not reflected in the solution.

I explained this situation to Copilot and asked it to fix the issue by always generating a temporary .vrp file on the server. Copilot generated changes to the server.py and scripts.js files, but I encountered a response length limit error when generating changes to the scripts.js file. I applied the changes to server.py and wrote a new prompt to remove any redundant code in the scripts.js file. After applying the changes to scripts.js and testing the GUI, I noticed that Copilot had used the 'os' module in server.py but had not imported it. I wrote a prompt about this, applied the code, and the GUI started working better. I was able to upload a .vrp file, change coordinates and customer demands, and these changes were now considered when running the algorithm. I was also able to input a problem manually.

Next, I wrote a prompt about allowing users to upload also .tsp formatted files and attached the index.html file as context. After testing the upload of a .tsp file, I noticed an error in server.py. However, this error was not displayed in the frontend. I wrote a prompt about this, and once again encountered the response length limit error while generating code for the scripts.js file. I applied the code generated for the server.py file, but it did not include any changes. I wrote a new prompt to update the event listener of the 'solve' button to display error messages from the server. Copilot generated code only for the event listener of the 'solve' button in the scripts.js file, which was much faster and more efficient since there was no need to modify other functions. After testing the GUI, I found that errors occurring on the server were now properly displayed in the frontend.

I tested the GUI with a .tsp file and noticed that it did not include capacity or customer demands. I examined the cvrp_io.read_TSPLIB_CVRP function and realized that capacity or customer demands may be null, and therefore not mandatory parameters to solve a VRP. I described this to Copilot, and it adjusted the create_temp_vrp_file function in the server.py file. I then wrote a new prompt asking Copilot to adapt these changes to the scripts.js file as well. I encountered the same error again, and I pasted the error message into the Copilot chat.

I debugged the code in server.py by adding some logging in the run endpoint, where the solution was calculated. I found that the error might be in the read_TSPLIB_CVRP function. I asked Copilot to add some logging to the create_temp_vrp_file function so that I could preview how the temporary .vrp file was being formatted. I noticed that there were three NaN rows in the NODE_COORD_SECTION, and these were also reflected in the coordinates input field in the GUI. I described this to Copilot and applied the changes to the `scripts.js` file. After testing the GUI, I noticed that it no longer populated the capacity and customer demand fields from the `E-n51-k5.vrp` file. I undid the changes by pressing Ctrl + Z.

I wrote a prompt to populate the capacity and customer demand input fields if they are included in the uploaded `.vrp` or `.tsp` file. I applied the changes and noticed in the server logs that the fields were populated correctly in the frontend, but the customer demands were passed as an array of null values to the server. I described this issue to Copilot and applied new changes to the `scripts.js` file. However, I noticed that Copilot had removed the colors from the route names in the route table, changed the algorithm selection to use the algorithm value instead of the description, and altered some of the logging lines. Copilot had also reverted the change I made earlier during this iteration regarding removing the indexes in the coordinates and customer demand fields.

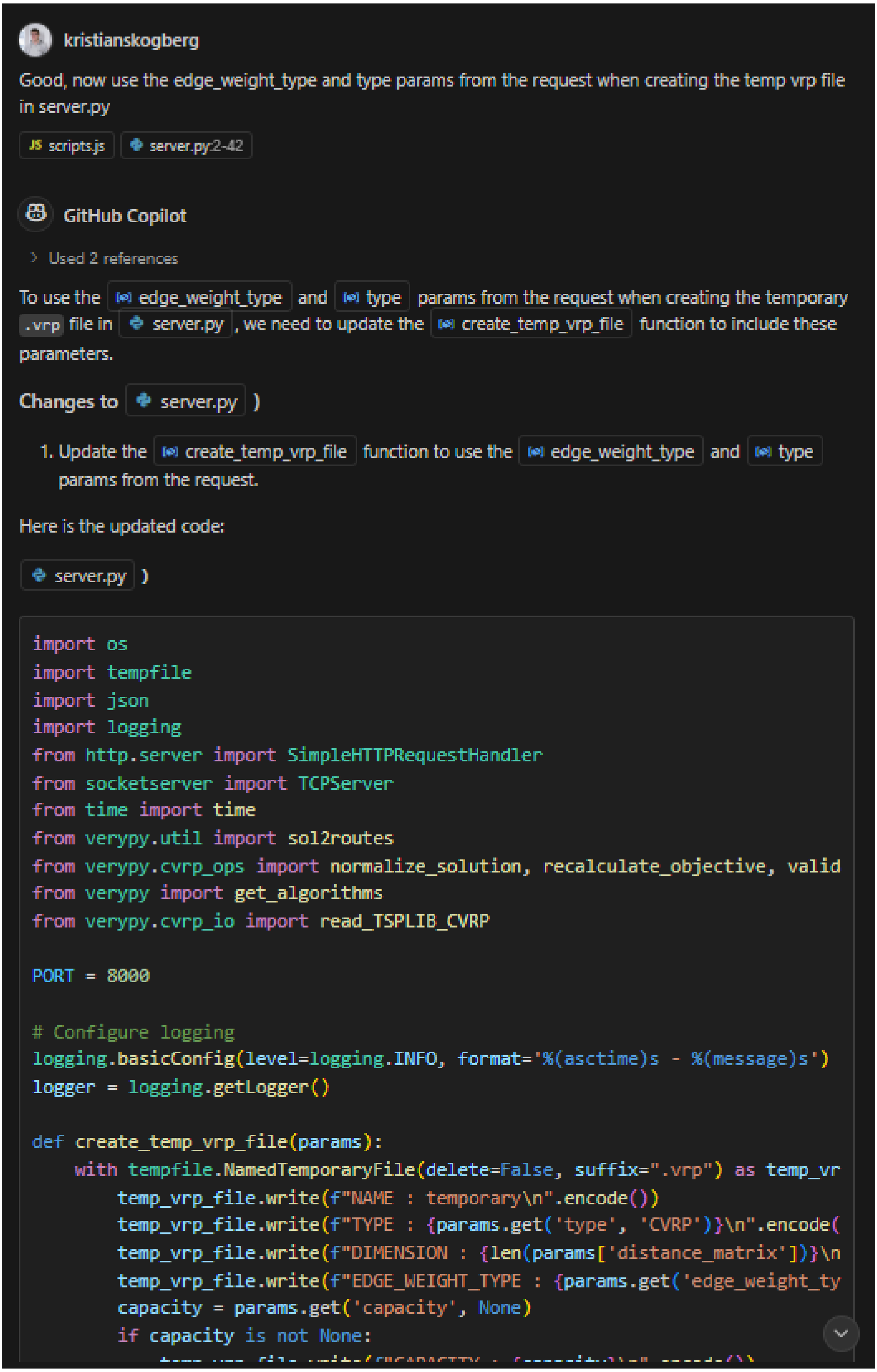
I wrote a prompt to remove the indexes from the coordinates and customer demand fields, as well as to use the algorithm description in the algorithm selection element. I applied the code, and it fixed the indexes and algorithm names. However, I encountered an error regarding indexes on the server while generating the temporary `.vrp` file based on the parameters from the frontend.

When I was applying the changes to the `server.py` file, I noticed a misunderstanding. Copilot assumed that the coordinates and customer demands included the index in the array (for example, coordinates like [30, 40] were assumed to be [1, 30, 40]). However, this was not the case, as Copilot had updated how the indexes were managed in the frontend. I described this issue and asked Copilot to adjust the `server.py` file. I applied the code, and now the solution was working properly again, except that the route table no longer included any of the details for each route.

I decided to simply copy the `drawSolution` function from the VeRyPy-GUI repository and replace the current version with it, as it had been working perfectly fine before. After testing the GUI, I noticed that Copilot had also reverted the `exportMetrics` and `exportVisualization` functions to their initial versions, which had been modified during the previous iteration. I copied these functions from the VeRyPy-GUI repository and used them to replace the corresponding functions in the `scripts.js` file.

The GUI now supports uploading both `.tsp` and `.vrp` files. Since these file types use different edge weight types, the GUI must account for these differences when generating temporary `.vrp` or `.tsp` files. I described this situation and asked Copilot to first modify the `scripts.js` file, as I knew it would hit the response length limit if Copilot tried to modify both the `scripts.js` and `server.py` files in the same output. I applied the changes carefully, as Copilot tried to revert the changes I had just manually reverted. Copilot also assumed that I had HTML elements for storing the edge weight type and type attributes in the `index.html` file, which I did not have. I explained this, and Copilot generated the corresponding HTML elements for the `index.html` file.

Now that Copilot had generated changes to the `scripts.js` file to extract and pass the edge weight type from the uploaded `.vrp` or `.tsp` file, I moved on to adapting this change in the `server.py` file. The prompt I used and the partial response are displayed in Figure 5.37 on the next page.


kristianskogberg

Good, now use the edge_weight_type and type params from the request when creating the temp vrp file in server.py

JS scripts.js | server.py:2-42

GitHub Copilot

Used 2 references

To use the edge_weight_type and type params from the request when creating the temporary `.vrp` file in server.py, we need to update the create_temp_vrp_file function to include these parameters.

Changes to server.py

1. Update the create_temp_vrp_file function to use the edge_weight_type and type params from the request.

Here is the updated code:

server.py

```
import os
import tempfile
import json
import logging
from http.server import SimpleHTTPRequestHandler
from socketserver import TCPServer
from time import time
from verypy.util import sol2routes
from verypy.cvrp_ops import normalize_solution, recalculate_objective, valid
from verypy import get_algorithms
from verypy.cvrp_io import read_TSPLIB_CVRP

PORT = 8000

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(message)s')
logger = logging.getLogger()

def create_temp_vrp_file(params):
    with tempfile.NamedTemporaryFile(delete=False, suffix=".vrp") as temp_vr
        temp_vrp_file.write(f"NAME : temporary\n".encode())
        temp_vrp_file.write(f"TYPE : {params.get('type', 'CVRP')}\n".encode(
        temp_vrp_file.write(f"DIMENSION : {len(params['distance_matrix'])}\n
        temp_vrp_file.write(f"EDGE_WEIGHT_TYPE : {params.get('edge_weight_ty
        capacity = params.get('capacity', None)
        if capacity is not None:
```

Figure 5.37. *Prompt for modifying the* `create_temp_vrp_file` *function and a partial response.*

I applied the changes and noticed that there were still some issues when running algorithms using data from uploaded `.tsp` files. To guide Copilot in the right direction, I added some logging and discovered that the server returned null in the 'points' attribute when a `.tsp` file was uploaded in the GUI. I remembered seeing a function in the `VeRyPy.py` file that was used to generate missing points if needed. Instead of describing the situation to Copilot, I copied and pasted this function call into `server.py` as it was quicker, given that it was only a few lines of code. I restarted the server and tested the GUI, and now the visualization worked for an uploaded `.tsp` file. However, there was an error when generating the route table, which appeared in Google Chrome DevTools console window, as shown in Figure 5.38 below.

Figure 5.38. *Error displayed in Google Chrome DevTools when calculating utilization rate for routes.*

I added some logging to the `calculateUtilizationRate` function in `scripts.js` and found that customer demands and capacity were null in the example `.tsp` file I had used to test the GUI. To fix this, I used the 'Modify using Copilot' feature as I did not want to risk describing this specific change in Copilot chat, since it might revert some of the changes I made. Copilot typically generates the entire code file, even if the change is relevant to just one function. I accepted the small change, tested the GUI, and now it was fully working with the uploaded `.tsp` file. The current state of the VeRyPy GUI, including the TSP visualization, is shown in Figure 5.39.

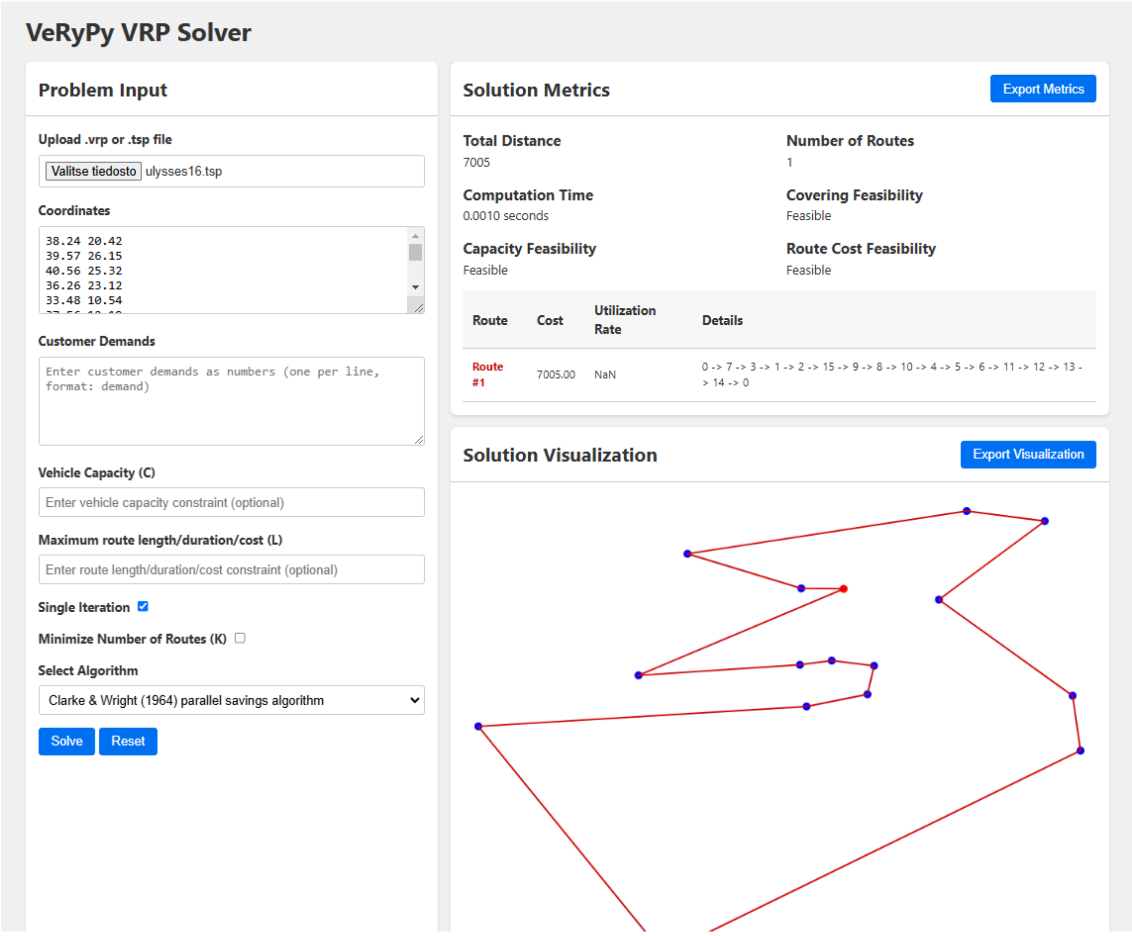


Figure 5.39. *VeRyPy GUI and visualization for solving a TSP.*

I tested inputting a VRP manually without uploading any files and encountered an error. Upon checking the server logs, I discovered that the temporary `.vrp` file had no values for the type and edge weight type, as these fields are initially null unless the user uploads a `.vrp` or `.tsp` file. To fix this, I added default values for these fields in the `index.html` file. Now, if the user has not uploaded any files and inputs a VRP manually, the edge weight type will default to `EUC_2D` and the type will default to `CVRP`. I implemented this fix manually by adding the appropriate value properties to these input fields in `index.html`. After testing the GUI, it is now able to solve a manually inputted VRP.

Next, I wanted to make the type and edge weight type input fields visible in the GUI. I attached the `index.html` and `scripts.js` files as context and wrote a prompt for this modification. I applied the code to the `index.html` file, but Copilot hit the response length limit while generating code for the `scripts.js` file. I copied and pasted the previous prompt, but this time I asked Copilot to only modify the `scripts.js` file. However, Copilot still generated changes for the `index.html` file as well. Upon reviewing the changes, I realized there was no need to modify the `scripts.js` file, as it already contained the necessary logic to pass the edge weight type and type parameters to the server.

I wanted to update the validation logic. If the type is TSP, then there should not be capacity or customer demand values. If the type is CVRP, then the number of coordinates and customer demands must match. I wrote a prompt about this and asked Copilot to only modify the event listener of the 'solve' button, as shown in Figure 5.40 below. However, Copilot still regenerated the `index.html` file and attempted to generate the entire `scripts.js` file, even though the changes were needed in just one function. As a result, Copilot hit the response length limit again.

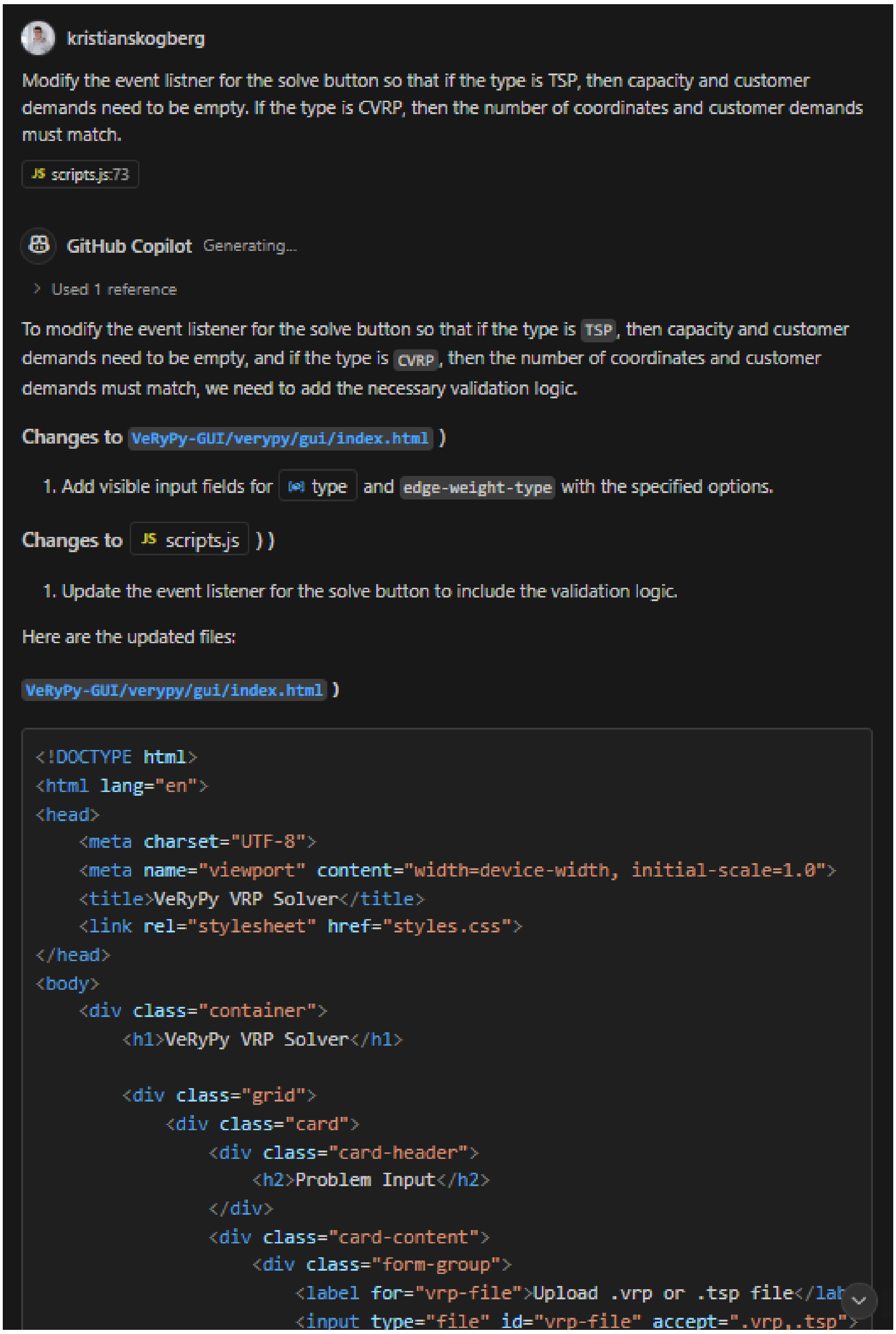


Figure 5.40. *Prompt for improving TSP and CVRP validation and a partial response.*

I was able to solve the response length limit issue by highlighting the event listener function of the 'solve' button and using the 'Modify using Copilot' feature to make changes to only the highlighted code. Copilot added an `if-else` code block that displayed different alerts based on which problem type was selected in the GUI. I applied the code and tested the GUI, and now it displays the alerts accordingly.

Next, I wanted to add functionality to the frontend where the GUI only displays the relevant input fields based on the selected problem type. In other words, the capacity and customer demand input fields should be hidden if the user selects 'TSP' as the problem type, as they are not used for TSPs. I wrote a prompt, and Copilot was able to generate an output without hitting the response limit. However, it made some small unnecessary changes which I had to discard. One of these changes can be seen in the image below. Copilot also attempted to remove some of the validation done before sending the VRP parameters to the server, which I had to decline. Additionally, Copilot tried to revert some random changes I had made many prompts ago, such as the algorithm naming.

I reviewed each change carefully instead of just accepting them, and had to discard a couple of small but unnecessary changes, such as the one displayed in Figure 5.41. After testing the GUI, I confirmed that it was now correctly displaying and hiding the necessary elements based on the selected problem type.

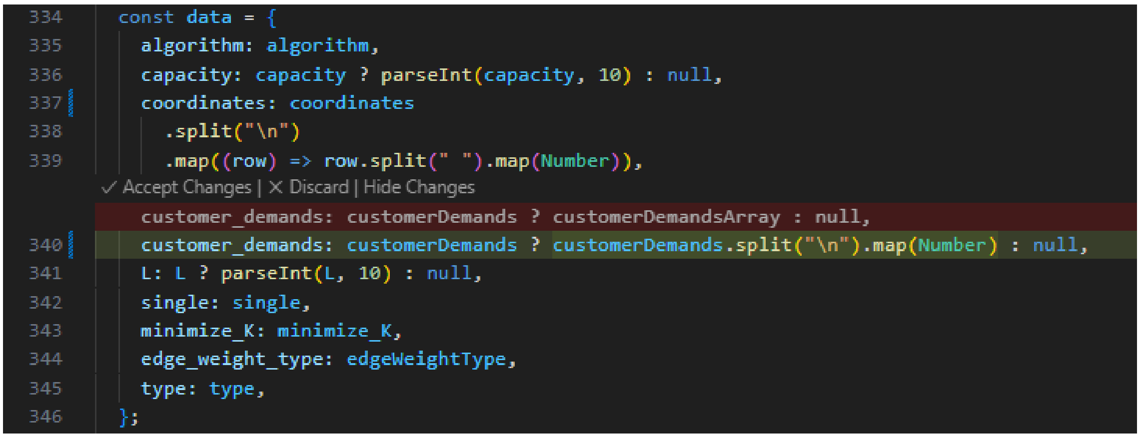
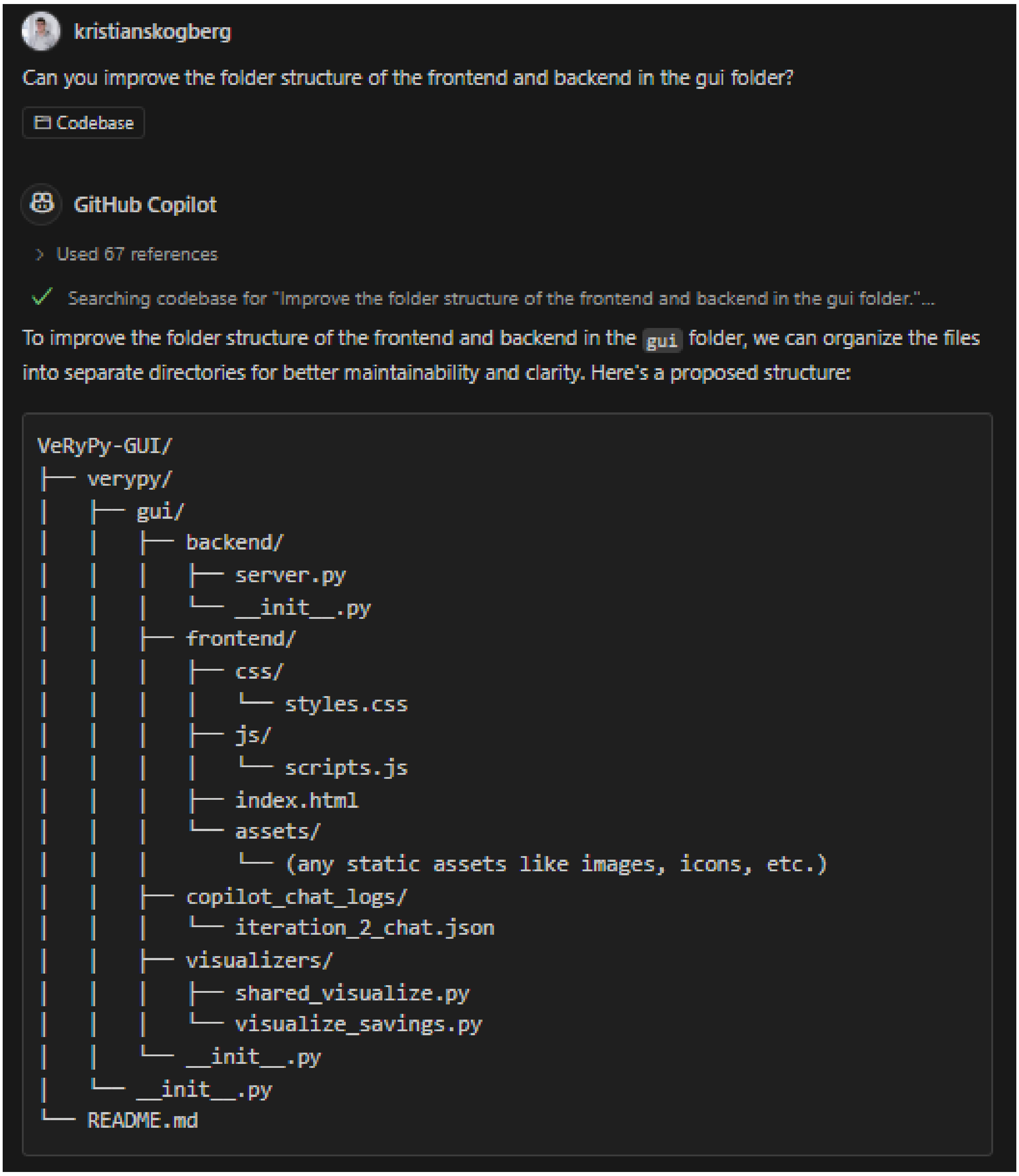


Figure 5.41. *An example of an unnecessary code change suggested by GitHub Copilot.*

I wanted to improve the folder structure of the code files related to the GUI. I wrote a prompt about this, and Copilot suggested creating separate folders for the frontend and backend, as well as additional folders for JavaScript files, CSS files, and other assets within the frontend folder. However, I believe it is sufficient to keep all frontend-related files in a single 'frontend' folder without further subdividing them into separate folders. Copilot did not create these folders for me but provided a clear folder structure, which can be seen in Figure 5.42 on the next page.



kristianskogberg

Can you improve the folder structure of the frontend and backend in the gui folder?

Codebase

GitHub Copilot

Used 67 references

Searching codebase for "Improve the folder structure of the frontend and backend in the gui folder."...

To improve the folder structure of the frontend and backend in the `gui` folder, we can organize the files into separate directories for better maintainability and clarity. Here's a proposed structure:

```
VeRyPy-GUI/
├── verypy/
│   ├── gui/
│   │   ├── backend/
│   │   │   ├── server.py
│   │   │   └── __init__.py
│   │   ├── frontend/
│   │   │   ├── css/
│   │   │   │   └── styles.css
│   │   │   ├── js/
│   │   │   │   └── scripts.js
│   │   │   ├── index.html
│   │   │   └── assets/
│   │   │       └── (any static assets like images, icons, etc.)
│   │   ├── copilot_chat_logs/
│   │   │   └── iteration_2_chat.json
│   │   ├── visualizers/
│   │   │   ├── shared_visualize.py
│   │   │   └── visualize_savings.py
│   │   └── __init__.py
│   └── __init__.py
└── README.md
```

Figure 5.42. *Improved folder structure suggestion by GitHub Copilot.*

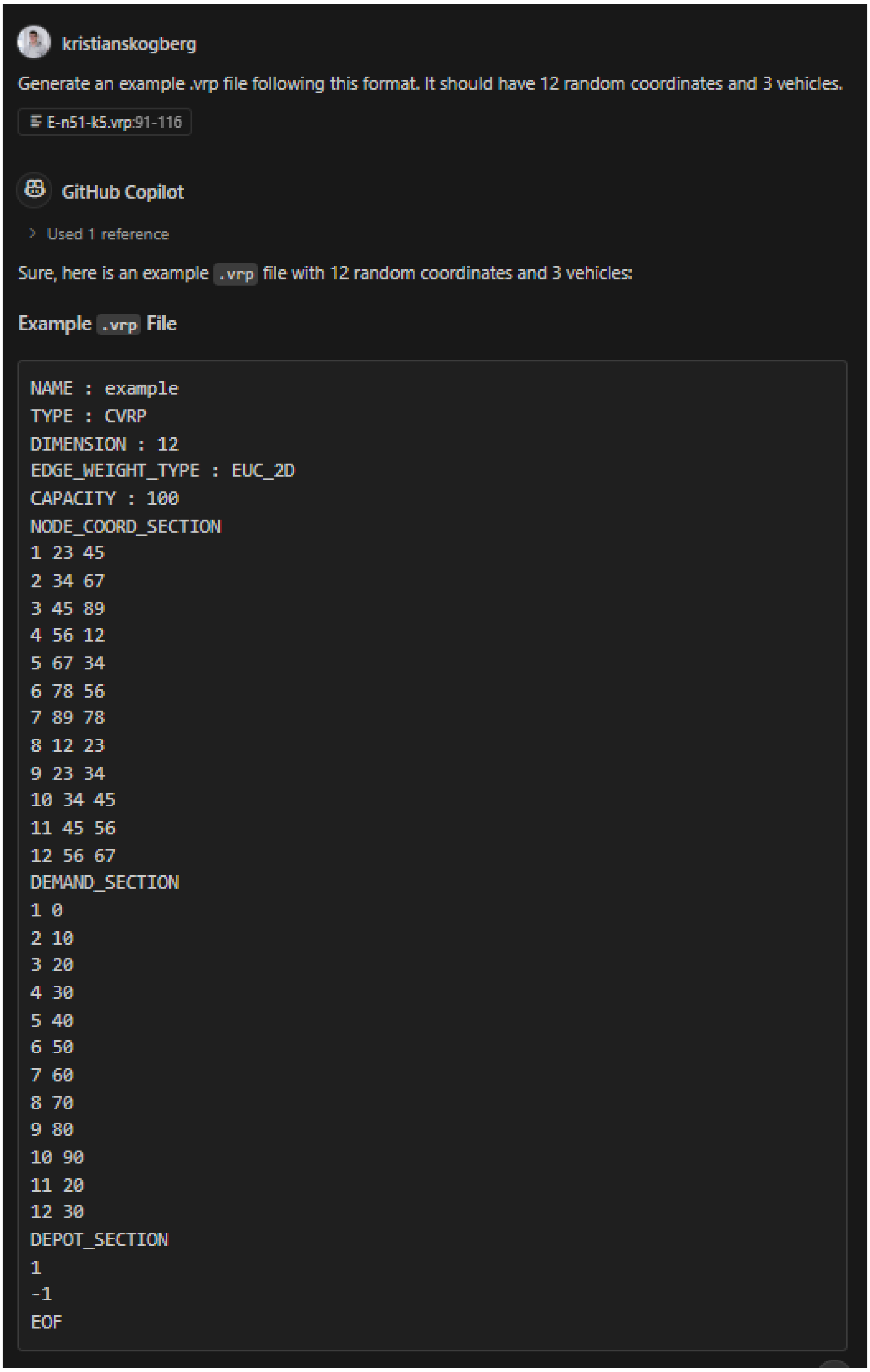
I manually created the 'frontend' and 'backend' folders and moved the server.py file to the backend folder, along with the rest of the GUI files to the frontend folder. I then asked Copilot to update the paths in the server.py file to match the new folder structure. I expected a simple one-line change related to how the server reads the index.html file, but Copilot generated several changes that would have reverted some of the recent modifications I had made in the server.py file, so I had to discard them. After testing the GUI, I discovered that the server could not locate the index.html file in the frontend folder. I wrote a few debugging prompts and, after a couple of attempts and some manual fixes, I was able to resolve the import issues in the server.

I wanted to update the placeholder text for the coordinates input element based on the selected problem type. If 'TSP' is selected, the coordinates should be inputted in latitude/longitude format; otherwise, they should be in x/y format. I wrote a prompt for this, and Copilot generated code for the relevant event listeners, which was great since it only modified the necessary parts of the scripts.js file instead of regenerating the whole file. After applying the changes and testing the GUI, I noticed that the placeholder text was not updating according to the selected problem type. I realized it was a caching issue. Once I restarted the server and refreshed the website, the placeholders updated correctly.

I wanted to add additional validation for the coordinates and customer demands input fields to ensure they contain numbers in each row. I wrote a prompt for this, and Copilot generated code for only the relevant part of the `scripts.js` file. It added two simple `for loops` to check that each row of these input fields contained numbers. However, I noticed that Copilot created new variables for validating coordinates and customer demands instead of using the existing variables, which made the code harder to read.

I wrote a prompt asking Copilot to improve this and remove the duplicate variables. While applying the changes, I found that Copilot had removed the coordinates property from the `POST` request parameters, which would break the GUI functionality. It also reverted some changes I had made manually, so I discarded all of Copilot's changes. In the end, I decided to clean up the variable names manually, as it was quicker and easier than making further adjustments.

I thought it would be a good idea to add a new example VRP to the GUI, as this would enhance both its usability and user experience. I asked Copilot to generate a new `example.vrp` file with 12 random coordinates. The output is shown in Figure 5.43 on the next page.



kristianskogberg

Generate an example .vrp file following this format. It should have 12 random coordinates and 3 vehicles.

E-n51-k5.vrp:91-116

GitHub Copilot

Used 1 reference

Sure, here is an example `.vrp` file with 12 random coordinates and 3 vehicles:

Example `.vrp` File

```
NAME : example
TYPE : CVRP
DIMENSION : 12
EDGE_WEIGHT_TYPE : EUC_2D
CAPACITY : 100
NODE_COORD_SECTION
1 23 45
2 34 67
3 45 89
4 56 12
5 67 34
6 78 56
7 89 78
8 12 23
9 23 34
10 34 45
11 45 56
12 56 67
DEMAND_SECTION
1 0
2 10
3 20
4 30
5 40
6 50
7 60
8 70
9 80
10 90
11 20
12 30
DEPOT_SECTION
1
-1
EOF
```

Figure 5.43. *Contents of the `example.vrp` file generated by GitHub Copilot.*

Next, I wrote a prompt to create a new button in the GUI. Clicking this button should load the `example.vrp` file and populate the input fields with its content. Copilot successfully generated the new button with the label 'Use Example VRP' in the `index.html` file, as well as the event listener in the `scripts.js` file. I applied the code and tested the GUI; clicking the 'Use Example VRP' button correctly populated the input fields. I moved the

'Use Example VRP' button to the top of the problem input section manually.

I wanted to add another button for using an example TSP, so I asked Copilot to generate an example `.tsp` file using random geographical coordinates. I created the file manually and pasted the generated content into it. Then, I wrote a prompt asking Copilot to add a new button for using the generated `example.tsp` file, placing it next to the 'Use Example VRP' button. I applied the code to the `index.html` file and noticed that Copilot did not revert the manual change I had made to the position of the 'Use Example VRP' button. I tested the GUI, and clicking the newly generated 'Use Example TSP' button successfully solved the `example.tsp` on the first attempt.

It would be useful if the GUI had an input field for changing the depot node. Currently, the first coordinate pair is used as the depot, and it is hardcoded in the `server.py` file. I wrote a prompt about this request in the Copilot chat and attached the `index.html` and `scripts.js` files as context. I did not attach the `server.py` file yet because I anticipated that Copilot would likely hit the response length limit.

As I was reviewing the changes in the `scripts.js` file, I noticed that Copilot had reverted the change in the naming of the coordinates attribute in the `POST` request, as shown in Figure 5.44. This change was made several prompts ago. Copilot had added useful validation for the depot node input field. In the end, I discarded these changes and manually copied the new lines into the `scripts.js` file.

```
const data = {
  algorithm: algorithm,
  capacity: capacity ? parseInt(capacity, 10) : null,
✓ Accept Changes | ✕ Discard | Hide Changes
  coordinates: coordinates
    .split("\n")
    .map((row) => row.split(" ").map(Number)),
  customer_demands: customerDemands ? customerDemandsArray : null,
  L: L ? parseInt(L, 10) : null,
  single: single,
  minimize_K: minimize_K,
  edge_weight_type: edgeWeightType,
  type: type,
  distance_matrix: coordinateLines.map(row => row.split(" ").map(Number)),
  customer_demands: customerDemandLines.map(Number),
  L: L ? parseInt(L, 10) : null,
  single: single,
  minimize_K: minimize_K,
  edge_weight_type: edgeWeightType,
  type: type,
  depot_node: depotNode ? parseInt(depotNode, 10) : null
};
```

Figure 5.44. *An example of a code change that GitHub Copilot attempted to revert.*

Next, I wrote a prompt to incorporate the depot node parameter in `server.py` and attached the file to the chat. Copilot reverted multiple changes, which I discarded, and I only accepted the modification to use the depot node parameter. I tested the GUI, and changing the depot node successfully affected the solution and the visualization as expected.

However, Copilot had not modified the event listeners for the VRP and TSP examples, nor for the file upload element, although this was not mentioned in my prompt. I asked Copilot to integrate the new depot node input field into the relevant event listeners. After applying the code and testing the GUI, I noticed that the depot node was always `-1` instead of `1` when importing a `.vrp` file. I explained this issue to Copilot, and it suggested a simple fix, which resolved the problem. The current state of the VeRyPy GUI is displayed in Figure 5.45 below.

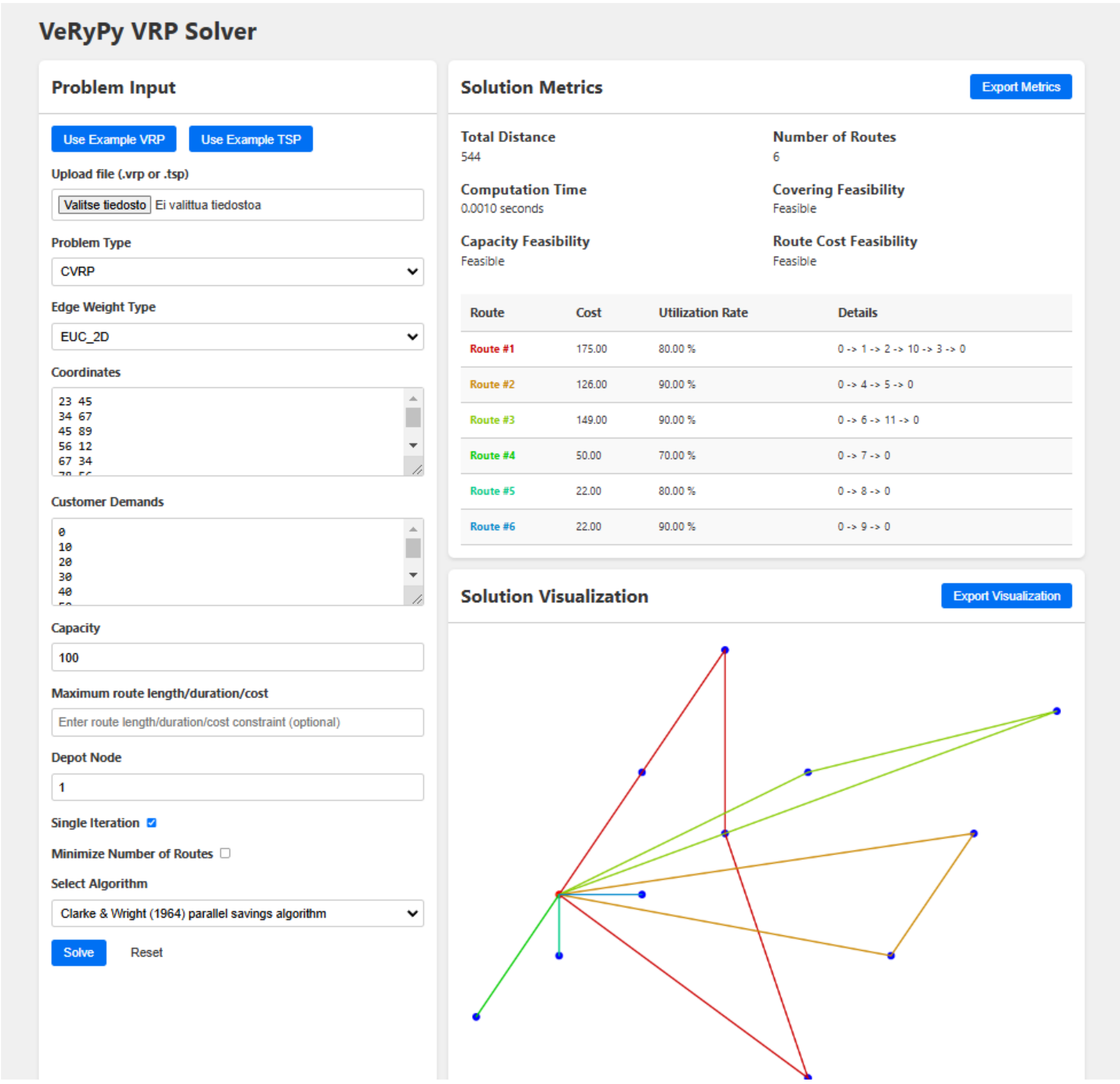


Figure 5.45. *VeRyPy GUI after iteration 5.*

5.5.3 Analysis

One of the key architectural decisions in this iteration was to always generate a temporary `.vrp` file based on the input parameters, regardless of whether the user uploaded a file or entered data manually. While this approach ensured that all modifications were consistently applied before solving the problem, it also introduced potential inefficiencies. For example, if the user had not changed any parameters after uploading a `.vrp` file, regenerating it became redundant. However, generating a temporary `.vrp` file could prove beneficial in future iterations, particularly if there is a need to export the inputted problems as `.vrp` files.

A recurring challenge was Copilot's tendency to revert previous changes, particularly in larger files like `scripts.js`. As the lines of code increased with each iteration, Copilot sometimes seemed to lose track of prior modifications, resulting in unintentional resets of certain UI elements, such as the route name color and the algorithm dropdown logic. This behavior highlights the observation that Copilot's internal context tracking may degrade as the file size grows or changes occur too frequently, making it difficult for Copilot to maintain a stable understanding of the evolving codebase.

Another issue emerged when Copilot modified the frontend logic for handling coordinate and customer demand indexes without adjusting the corresponding server logic. This type of issue also occurred in earlier iterations. It caused errors when solving VRPs, as the server still expected the old data structure. Ideally, Copilot would detect dependencies between modified components and maintain consistency across the application. However, this likely exceeds the context or response length limit. As a result, Copilot's inability to manage these dependencies increased the cognitive load, as debugging and modifications still relied heavily on manual oversight and validation.

Copilot's tendency to modify unrelated parts of the VeRyPy GUI complicated development. A notable example was the `drawSolution` function, which initially worked as expected but was unexpectedly altered by Copilot. While AI-generated modifications sometimes improved code efficiency, they also introduced breaking changes. When attempting to revert the function using Copilot, the generated version did not fully restore its previous logic. Ultimately, the most reliable solution was to manually copy and paste the original function from the VeRyPy GUI repository.

Given Copilot's tendency to occasionally introduce unexpected errors, I often found myself debugging code manually rather than relying on Copilot's debugging assistance. While Copilot could be useful in diagnosing specific issues when provided with error logs, its lack of contextual awareness across the entire GUI made it risky to use for direct debugging. Fixes in one area often led to unintended modifications elsewhere, highlighting the limitations of relying on Copilot for comprehensive troubleshooting.

Another significant limitation encountered during this iteration was the response length limit, which occurred when generating modifications for larger files (`server.py` and `scripts.js`). These files contained a total of 700 to 800 lines of code. Based on token calculations [97], 100 lines of Python code correspond to approximately 1000 tokens, while 100 lines of JavaScript is roughly 700 tokens. Although the GUI source files should fit well within the GPT-4o model's context window, the primary constraint regarding the response length limit issue appears to be the model's output limit, which is a maximum of 16,384 output tokens.

To mitigate issues related to the response length limit, several strategies can be employed. One approach is to separate stable event listeners and utility functions into dedicated files, reducing the likelihood that Copilot will regenerate unnecessary code. For example, each event listener function could be placed in its own file, and these files could then be consolidated into a single import. Another option is to explicitly specify which function or event listener needs modification in the prompt. While this sometimes prevents Copilot from generating full files, it is not entirely reliable, as Copilot often generates the entire file regardless of the prompt's specification. The most effective method appears to be using Copilot's 'Modify using Copilot' feature to highlight and edit only specific sections of code. This results in faster output generation and avoids unnecessary file-wide changes, but it sacrifices access to context outside the highlighted section, and modifications are not saved in the Copilot chat logs. Additionally, this approach requires the developer to have a strong understanding of the codebase in order to select the appropriate section for modification.

5.5.4 Conclusion

Iteration 5 enhanced the usability of the Problem Input section of the VeRyPy GUI. The indexes for the 'Customer Demands' and 'Coordinates' input fields are now managed automatically in the code, rather than within the input elements. A new utility function was also created in the backend to generate a temporary `.vrp` file, ensuring that changes made in the input fields, specifically those related to customer locations and demands, are reflected in the VRP used for the calculation. Additionally, support for reading `.tsp` formatted files was added to the GUI, along with further validation for the input fields.

Overall, this iteration reinforced key insights into AI-assisted development. Copilot's difficulty in maintaining an accurate context and handling larger modifications across multiple components, such as the `scripts.js` and `server.py` files, heightened the need for manual debugging and verification. Response length limits were also a challenge, often requiring prompts to be rewritten to avoid unnecessary AI-generated changes.

At the end of this action research, the generated lines of code in the VeRyPy-GUI repository were calculated using cloc [93], which can be seen in Figure 5.46 below. It is important to note that most of the code lines was sourced from the GitHub Copilot chat logs, which were exported as JSON files after each iteration. If these JSON chat logs are excluded, the total number of code lines is 14 129. Therefore, a total of **1105** lines of code for the VeRyPy GUI was generated using GitHub Copilot and OpenAI's GPT-4o model in this action research.

```
     102 text files.
      91 unique files.
      25 files ignored.

github.com/AlDanial/cloc v 2.04  T=0.85 s (107.3 files/s, 115664.8 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
JSON                             4              0              0          76572
Python                          80           3119           4037          12998
JavaScript                       1             67             24            610
CSS                              1             35             12            193
Markdown                         2            100             37            179
HTML                             1              4              1            141
DOS Batch                        1              0              0              5
Text                             1              0              0              3
-------------------------------------------------------------------------------
SUM:                            91           3325           4111          90701
-------------------------------------------------------------------------------
```

Figure 5.46. *Breakdown of VeRyPy lines of code at the end of this action research.*

If the GitHub Copilot chat logs are excluded, a total of 4 different source files were produced during the VeRyPy GUI generation process. Information about the VeRyPy GUI source files generated in this action research is presented in Table 5.2 below.

Filename	**Description**	**Lines of Code**
`server.py`	Backend logic of the VeRyPy GUI	161
`index.html`	Layout and elements of the VeRyPy GUI	141
`scripts.js`	Interactive functionality of the VeRyPy GUI	610
`styles.css`	Styling of the VeRyPy GUI	193

Table 5.2. *Source files of the VeRyPy GUI generated in this action research.*

6. FINDINGS

One of the strengths of leveraging LLMs in VeRyPy GUI development was their ability to generate low-level code, such as functions for parsing data with specific syntax or creating simple utility functions. However, when working with codebases where code is split across multiple files and directories, the accuracy of code generation decreased. A significant amount of time was spent debugging the generated code, manually copying and pasting error messages to GitHub Copilot, and writing prompts to fix these issues.

Although Copilot was aware of the source code files in the codebase, it did not have access to the entire development environment, including the Google Chrome web browser the researcher used for testing the GUI. As a result, Copilot had no understanding of what was rendered in the GUI or how different buttons and inputs behaved. Consequently, the researcher had to describe the GUI's behavior in prompts to achieve more accurate results, which in turn added significant time to the overall development process.

Initially, generating features for the VeRyPy GUI was quick, but the process slowed as the lines of code increased. Applying the generated code to the source files sometimes took a significant amount of time and was done after every code generation. While manually copying and pasting the generated code directly into the source files was faster, it was not recommended, as it obscured which lines had changed, making troubleshooting more difficult. A considerable amount of time was spent waiting for Copilot to generate code and apply it to the source files.

Copilot excelled at generating clear and easy-to-understand labels, placeholders, as well as function and variable names. This improved code readability and saved the researcher time by eliminating the need to manually come up with descriptive names. Clear labels and placeholders also improved the overall usability of the GUI, making it easier for users to understand what type of data should be entered into each input field.

On some occasions, Copilot generated code that was not relevant or specified by the researcher in the prompt. Another recurring issue was that Copilot reverted code changes that the researcher had made manually between prompts. Additionally, newly generated code sometimes overwrote previously generated code. This issue persisted even when the researcher added comments to specific lines instructing Copilot not to modify them. It took some time for Copilot to update the context, which helped reduce some of the reverting. However, this issue increased the cognitive load on the researcher, who had to carefully review each change and assess whether it could break parts of the GUI.

An interesting finding was that Copilot retained context best after some time had passed since the last code generation or modification. For example, continuing the GUI development after a break often yielded good results in terms of capturing the current context. However, when Copilot was used frequently within a short time period between prompts, it sometimes lost context from the last few minutes or even longer. This is likely because Copilot does not update its context quickly enough when changes occur too frequently. It seems that if Copilot is unable to access the current context, it may revert back to an older context, which would explain this behavior. Regardless, this made the GUI development process feel like taking two steps forward and one step back, as every once in a while Copilot attempted to revert unnecessary changes while simultaneously implementing and improving the GUI features.

Although Copilot was helpful for troubleshooting and debugging, it was sometimes faster and easier to make edits and resolve issues manually. Explaining problems to Copilot always carried the risk of ambiguous prompts, which could lead it in the wrong direction. Toward the end of the GUI development, relying on Copilot became less practical, as it often risked breaking parts of the GUI, reverting to an outdated context, or reaching the context window or the response length limit. Unit tests could have helped mitigate this by verifying that earlier features still worked after adding new code.

The prompts written by the researcher were concise but also short and little time was spent refining them. This could be one reason why some unwanted results and issues occasionally appeared in the AI-generated code. Spending more time writing detailed prompts might have prevented some of these issues. On the other hand, AI is still prone to errors, and writing detailed prompts can take considerable time, which can add up over the course of development.

Being familiar with best practices in software development, having technical knowledge and a clear vision for how certain features should be implemented proved to be highly beneficial. Relying solely on AI often led to questionable design choices and generated code that was difficult to maintain. For example, during the early stages of the GUI development, Copilot suggested mixing different technologies, such as combining Flask with an HTTP server in the backend, even though the backend had already been implemented using an HTTP server. This would have resulted in a poor design choice and unnecessary complexity, and the researcher had to address the situation and guide Copilot back on track. However, these kinds of changes are likely to go unnoticed by less technical developers, especially if they do not carefully review the AI-generated code.

7. DISCUSSION

The contributions of this research are twofold. First, the VeRyPy GUI was developed as the main artifact of the action research conducted in this thesis. This also includes documentation and analysis of the GUI development process, as well as a discussion of the findings in relation to the related work. The second contribution is a proposal for a more autonomous UI code generation process using LLMs, which emerged from the findings and experiences gathered during the action research. This proposal is described later in this chapter.

Evaluation of the VeRyPy GUI and Future Improvements

To evaluate the VeRyPy GUI produced during the action research iterations, it can be compared to other scientific UIs used to solve VRPs, which were presented earlier in the background chapter on page 16. One of the key differences between the VeRyPy GUI and the other scientific UIs is the ability to select between different heuristic algorithms implemented in VeRyPy. Users can test different algorithms and settings to solve VRPs and see the solution metrics and visualization in the GUI. The VeRyPy GUI also provides a way to export the solution metrics and visualization for further research which was not possible in the other UIs.

In order to remain within the scope of this thesis, some potential improvements to the VeRyPy GUI were excluded. For instance, it would be useful to have a real-world map in the GUI where the user could select or input locations using addresses and street names, and VeRyPy would calculate routes visiting each location while following the actual streets and roads. Because this feature would be relatively complex and large, it would be listed as an epic, which could then be divided into smaller user stories. Furthermore, this feature would be especially useful for users working in logistics, such as planning optimal routes for delivering goods to customers.

Some options could also be applied to the vehicles used in calculations related to the real-world map feature. For instance, vehicles could be marked as a car, bike, or a person walking, and these options would be respected while calculating the solution. Additionally, there are some algorithm-specific parameters, so adding input fields for these could be a good improvement.

A feature for playing back the visualization in stages could also be valuable. This could help users gain a better understanding of how the routes were calculated and how the algorithms work in detail. This feature could be especially useful for users focused on developing algorithms.

Finally, the usability of the input fields could be improved in the future. For instance, the coordinates and customer demands could be added using separate input fields instead of editing them in the textarea elements. Also, some info elements could be added to guide the user throughout the GUI as well as descriptions and information of the algorithms.

Factors Influencing the Results

There are several factors that influenced the results of this action research. First, the researcher's knowledge and skills had a direct effect on how AI was used to generate code for the VeRyPy GUI. For example, writing prompts, reviewing the generated code, testing the GUI, and how the encountered bugs and issues were described in prompts affected the direction of the code generations.

Another factor affecting the results was the selected AI tools and LLMs. There are numerous LLMs available from different companies, and comparing them closely would have been outside the scope of this research. There is ongoing debate about which model delivers the most accurate results, but it is difficult to compare models unless they are used with the same prompts in the same context and conditions. Also, the versions of the AI tools used in this action research affected the experiences, as some of the recurring issues, such as reverting changes, losing context, and occasionally slow code generation speed, may have been fixed in other versions.

Additionally, some AI copilots have been marketed and promoted more heavily than others, making it even more difficult to objectively evaluate the models. For instance, Cursor is currently the fastest growing software as a service (SaaS) company to go from $1M to $100M annual recurring revenue in 12 months [98, 99], and it seems to be rapidly gaining mind-share as one of the the leading AI-powered code editors. It will be interesting to see which AI copilot and LLM ultimately becomes the most favored among developers, though there will likely always be some variation in preferences.

It is also important to acknowledge that since the researcher was both using AI tools to generate the VeRyPy GUI and documenting the workflow, some aspects of the process may not have been fully captured in the documentation presented in the results chapter. Some passing thoughts, observations and certain interactions with AI such as additional prompts, quick manual fixes or debugging steps may not be explicitly recorded. This happened especially in the early iterations, but the documentation detail increased as the researcher became more familiar with the research process. However, all chat conversations between the researcher and GitHub Copilot are documented and available in the VeRyPy-GUI repository [94] under the path *verypy/gui/copilot_chat_logs* [100]. These logs provide a comprehensive record of the prompts used as well as the AI-generated responses throughout the VeRyPy GUI development.

Connection to Related Work

As Pandey et al. [56] mentioned in their study, the biggest issue with using GitHub Copilot lies in generating code for unique business logic and distributing code to multiple files. Although most of the unique business logic had already been implemented in VeRyPy (such as functions for different heuristic algorithms), AI struggled a little in dividing functionality into multiple files. It was up to the researcher in this AR to guide Copilot to split the code into multiple files. Also, sometimes AI-generated code included import statements of incorrect directories even when the researcher provided them as context.

Bajcetic, Draskovic, and Bojic [63] found in their research that AI-generated code provided a good starting point, and it covered most of the defined scenarios. This was also confirmed in this action research and while the AI-generated code contained some bugs and issues, it worked for the most part. Using AI was most effective in the beginning, for example, the VeRyPy GUI design generated using Vercel V0 AI tool fulfilled most of the requirements, and it did not require that much iteration or fine-tuning. As Bajcetic, Draskovic, and Bojic [63] also mentioned in their paper, a more robust process is needed to better utilize AI, especially for larger and more complicated systems.

Bilgram and Laarmann [57] and Mastropaolo et al. [58] mentioned the importance of defining tasks, descriptions, and requirements clearly. This was also something the researcher in this AR found while using AI tools to generate code. Ambiguous requirements and prompts produced unwanted results such as code which did not work quite as expected. Having clear requirements and specifications is always important but it is worth noting that AI does not eliminate the need for clear requirements.

Barke, James, and Polikarpova [62] defined two interaction modes that developers use when working with AI copilots: acceleration and exploration. While exploration mode was used by the researcher in this AR in the beginning of the action research to generate the initial iteration plan and user stories, acceleration mode was used extensively throughout the GUI code generation process. The researcher always had an idea of what to implement next based on the iteration plan and observations made during the iterations. Barke, James, and Polikarpova [62] also mentioned that developers tend to discard AI-generated code quickly while in acceleration mode and this was also something the researcher experienced. One reason for discarding code was that the researcher could usually quickly determine whether the generated code would cause issues or if it was of suboptimal quality.

Liang, Yang, and Myers [60] mentioned that one of the biggest usability issues in AI-powered code generation was that developers are likely to give up on using generated code if it does not perform the desired action. This was also reflected in the researcher's experience in this AR when they repeatedly discarded AI-generated code and in some cases gave up on using Copilot to generate code for simple tasks. Also, the researcher preferred to debug code manually as Copilot was sometimes slow to generate responses and it tends to suggest more generic solutions. Debugging more complex applications with multiple source files was challenging for Copilot due to the limited context window and API rate limits.

Proposal for Autonomous AI-powered UI Code Generation Process

Based on the results and findings of this action research, leveraging AI and LLMs in UI code development is not yet a fully automated process as it still requires considerable manual instruction, validation, and correction. Furthermore, while it is possible to construct a working software directly within AI-powered platforms, it is challenging to integrate code generation with an existing codebase without relying on manual copy and paste. Although AI can be applied at various stages of software development, its usage remains fragmented, resulting in an unorganized and inefficient process. This chapter explores a potential approach to integrating AI into software development, specifically focusing on a more autonomous UI code generation process.

First, it is important to highlight those aspects of AI-powered software development which require the most manual work. Based on the results and findings of this action research, these manual aspects are writing prompts, describing the desired functionalities, reviewing changes, and testing the system after applying changes. Automating these aspects and establishing a cycle for implementing new features incrementally could, in theory, lead to a more autonomous process for utilizing AI in software development.

Building on the findings of this action research, Figure 7.1 presents a proposal for a more autonomous approach to AI-powered UI code generation. This process is structured into two iterative cycles: the specification cycle and the implementation cycle. The process begins in the specification cycle by collecting the system's specification and requirements from the user, which are then converted into user stories. These user stories are then transformed into features, organized, and added to a Product Backlog, following agile project management practices. These user stories and features are presented to the user, where they can accept or reject them. The specification cycle will continue until the user accepts or confirms the user stories and features for the system. In other words, this validation aspect requires some manual work where the user provides the requirements and reviews the user stories and features generated by AI. Once the user accepts the user stories and features, the process enters the implementation cycle.

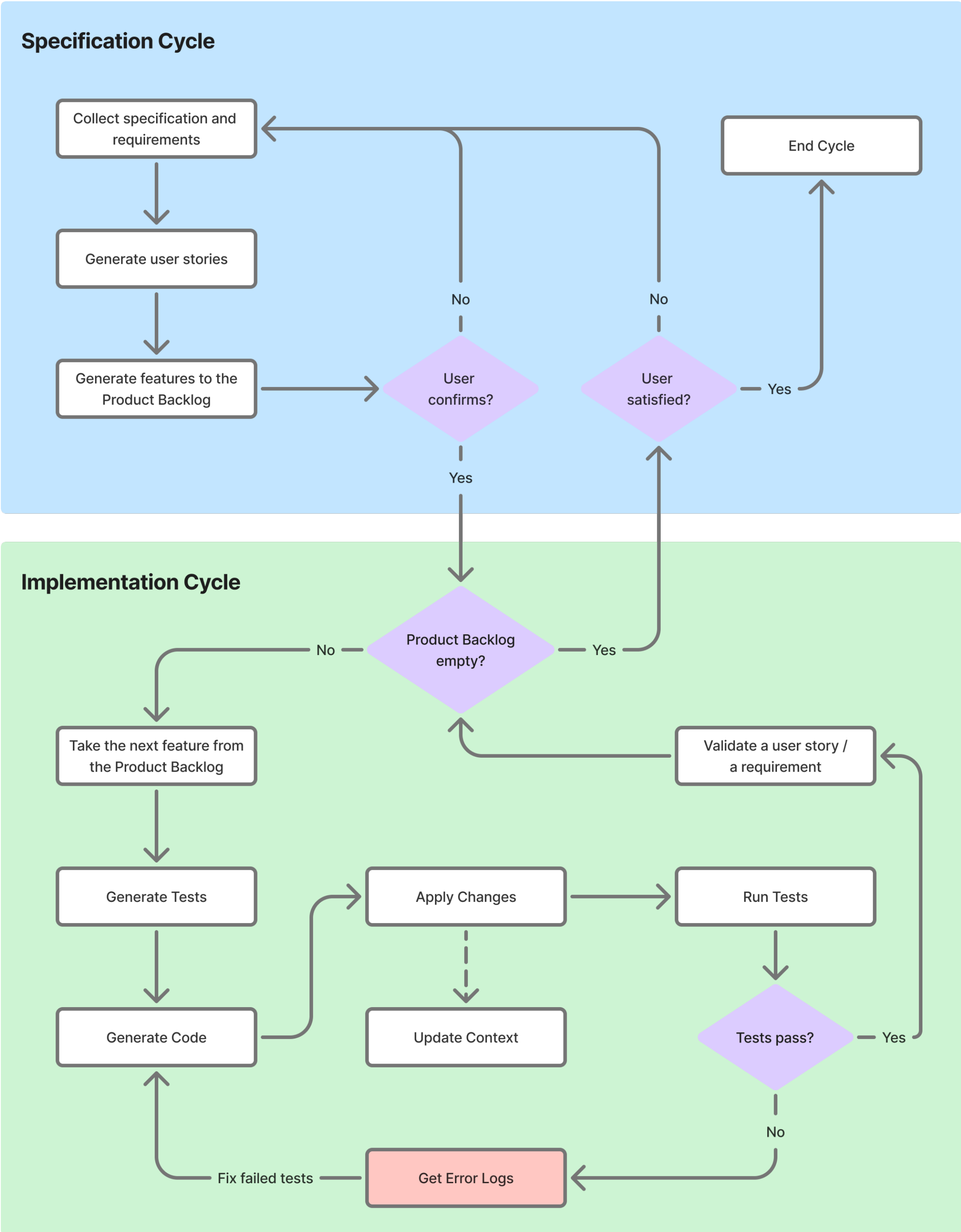


Figure 7.1. *Proposal for an autonomous AI-powered UI code generation process.*

The implementation cycle begins by checking if the Product Backlog contains any features generated earlier in the specification cycle. If there are features to be implemented, a feature is selected from the backlog for implementation. Before generating any code, tests are generated against the selected feature, which will be used to validate the generated code autonomously. The context would be updated and accessed in the background whenever new changes or code are applied to the system. After new code is applied, tests are run to ensure it does not break any previously working features and to verify that the tests associated with the currently implemented feature will pass. If any tests fail, the error logs are used as input for generating code that addresses the issues to eventually pass all test. When the tests pass, the user story or requirement corresponding to the implemented feature will be validated or marked as completed. Using LLMs to validate features linked to user stories was also discussed by Kolthoff et al. [65] in their paper.

The last step of the implementation cycle is to check if the Product Backlog is empty. If there are no more features left to be implemented, the process will exit the implementation cycle and return to the specification cycle to check if the user is satisfied with the implemented features. If the user is not satisfied and has improvement ideas or the requirements have changed, the specification cycle will continue so that the user can write prompts for refining, modifying, or adding features and requirements. The whole process will end when the user is satisfied with the software.

The core idea of the implementation cycle is to use practices of test-driven development (TDD). In TDD, the emphasis is on writing tests for a feature first and then implementing code until these tests are passed. Using LLMs to generate both code and tests was also studied by Fakhoury et al. [64] in their paper, yielding great results, such as improved code generation accuracy, less cognitive load on the developer, and using tests to ensure that the code does what the developer intended. Integrating TDD with code generation would be important to eliminate a significant portion of the manual testing and validation typically done by the developer against AI-generated code.

One of the key limitations of using LLMs in code generation currently is their inability to manage large contexts and occasionally losing some parts of the context. To address this issue, some kind of module would be needed to keep track of the context in the background. This context module could, for example, construct a UML diagram based on the working directory and source files, which is provided to the necessary modules automatically alongside every prompt. Rasheed et al. [13] also envisioned the creation of UML diagrams as a step in the AI-powered code generation process in their recent paper. The biggest limitation of this kind of context management is the limited context windows of LLMs. However, given that the context windows of LLMs have been increasing rapidly, it is likely that this issue may be resolved as models evolve.

Some risks and drawbacks can be identified in this process. The biggest risk is that AI could hallucinate, generate inaccurate, impossible, or irrelevant tests, and potentially enter endless loops while attempting to modify the code until it passes the tests. However, some kind of automatic validation and testing should be included to create more autonomous and robust processes. Additionally, the context management module functions properly only if the LLM's context window is large enough to track the context of the entire codebase, even as changes occur frequently.

Specialized AI agents could be used in the different modules of this process. For example, one AI agent could generate user stories based on the specification and requirements provided by the user, while another agent would take over to generate features based on these user stories. Additionally, one agent could be responsible for generating tests, while another agent keeps track of the current context and provides it to the other agents in the background.

The idea of using AI agents and dividing the software development process into modules was also discussed by Rasheed et al. [13] and Liu et al. [30] in their papers. Although designing systems that revolve around AI agents has recently gained a lot of attention, their application in software engineering has been studied as far back as the late 1970s [101]. Furthermore, in 2000, Jennings [102] proposed the following hypothesis: *As well as being suitable for designing and building complex systems, the agent-oriented approach will succeed as a mainstream software engineering paradigm*. With the recent rapid developments in AI copilots and LLMs, we are now closer than ever to designing and implementing fully autonomous AI agents as part of software engineering processes.

8. CONCLUSION

In this thesis, AI tools were studied and used to evaluate their suitability in software engineering for generating user interface (UI) code. A graphical user interface (GUI) was generated for VeRyPy, a Python library, as part of the action research component of this thesis to gain insights into the practical use and current limitations of AI tools focused on UI development. Various features for the VeRyPy GUI were implemented using GitHub Copilot and OpenAI's GPT-4o model over five iterations, while following the requirements, user stories, and iteration plan for the GUI. The workflow and experience were documented in detail in the results chapter, and the generated code, as well as the Copilot chat logs, are available in the VeRyPy-GUI repository [94]. The VeRyPy GUI was implemented as a small-scale web application, with the backend generated in Python and the frontend in HTML, CSS, and JavaScript.

AI tools, such as GitHub Copilot and Vercel V0, were most efficient in the early stages of the VeRyPy GUI development. However, as the codebase grew larger, integrating new code with existing code became increasingly slow, and new code generations would often hit the response length limit of GitHub Copilot and OpenAI's GPT-4o model. Furthermore, the context was sometimes not updated fast enough, resulting in generated code that would revert previously working features and introduce unnecessary modifications. This led to considerable time spent troubleshooting, manually copying and pasting errors, and writing prompts to GitHub Copilot. Due to the limited context window of LLMs, manual intervention and careful review of each change were necessary, as the AI tools often overlooked earlier prompts and requirements or introduced modifications that could break existing features.

Although using LLMs and AI tools significantly accelerated the GUI development process, they did not remove the need for continuous iteration, testing, and validation. In addition to these aspects, requirements engineering continues to be an essential part of software development. If the system requirements or prompts are unclear, LLMs are likely to produce unwanted results, much like developers would. These aspects seem to remain fundamental in software development regardless of whether AI is utilized or not.

While the context limitations of LLMs became noticeable as more features were added to the VeRyPy GUI, recent data suggests that the context windows of LLMs are expanding fast. It is expected that, in the near future, even larger codebases will fit within these context windows, which could greatly accelerate AI-powered software development. However, human involvement remains essential, along with knowledge of best practices in software engineering, to fully leverage AI's potential in software development.

The findings of this research sparked the idea for a more autonomous approach to generating UI code using LLMs, which was presented in Figure 7.1 on page 101. This process would follow the principles of test-driven development, where AI is used to generate user stories and features based on the requirements provided by the user. Corresponding tests would then be generated for these features before generating the implementation code. These tests would be executed whenever new code is applied to the codebase with the context being updated in the background. Once all tests pass, the user story or requirement corresponding to the implemented feature is marked as completed. This cycle would repeat until all features have been implemented.

I hope this thesis will be valuable to software engineers and researchers interested in the role of AI in software development, specifically in UI code generation. It is exciting to see how AI tools and LLMs will evolve and be integrated into software engineering processes in the near future.

REFERENCES

[1] Abdulaziz Aldoseri, Khalifa N. Al-Khalifa, and Abdel Magid Hamouda. "AI-Powered Innovation in Digital Transformation: Key Pillars and Industry Impact". In: *Sustainability* 16.5 (2024). ISSN: 2071-1050. DOI: 10.3390/su16051790. URL: https://www.mdpi.com/2071-1050/16/5/1790.

[2] David Mhlanga. "Industry 4.0 in Finance: The Impact of Artificial Intelligence (AI) on Digital Financial Inclusion". In: *International Journal of Financial Studies* 8.3 (2020), p. 45.

[3] Li Yang et al. "Artificial Intelligence and Robotics Technology in the Hospitality Industry: Current Applications and Future Trends". In: *Digital Transformation in Business and Society*. Springer International Publishing, 2019, pp. 211–228. ISBN: 9783030082765.

[4] Sook Fern Yeo et al. "Investigating the impact of AI-powered technologies on Instagrammers' purchase decisions in digitalization era–A study of the fashion and apparel industry". In: *Technological Forecasting and Social Change* 177 (2022), p. 121551.

[5] Shakked Noy and Whitney Zhang. "Experimental evidence on the productivity effects of generative artificial intelligence". In: *Science (American Association for the Advancement of Science)* 381.6654 (2023), pp. 187–192. ISSN: 0036-8075.

[6] Klint Finley. *How developers spend the time they save thanks to AI coding tools*. Published by GitHub Blog. Nov. 14, 2024. URL: https://github.blog/ai-and-ml/generative-ai/how-developers-spend-the-time-they-save-thanks-to-ai-coding-tools (visited on 02/23/2025).

[7] Thomas Kwa et al. *Measuring AI Ability to Complete Long Tasks*. 2025. arXiv: 2503.14499 [cs.AI]. URL: https://arxiv.org/abs/2503.14499.

[8] Jussi Rasku. *VeRyPy*. Dec. 2, 2023. URL: https://github.com/yorak/VeRyPy (visited on 10/28/2024).

[9] GPT-Lab. *GPT-Lab*. 2025. URL: https://gpt-lab.eu/ (visited on 01/04/2025).

[10] Ken Schwaber. *Agile Project Management with Scrum*. 1st edition. Safari tech books online. Microsoft Press, 2009. ISBN: 9780735636002.

[11] Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit*. Agile Software Development Series. Addison-Wesley Professional, 2003. ISBN: 9780321150783.

[12] Ian Sommerville. *Software engineering*. Tenth edition. Always learning. Pearson, 2016. ISBN: 978-1-292-09614-8.

[13] Zeeshan Rasheed et al. *TimeLess: A Vision for the Next Generation of Software Development*. 2024. arXiv: 2411.08507 [cs.SE]. URL: https://arxiv.org/abs/2411.08507.

[14] Mike Cohn. *User Stories Applied: For Agile Software Development*. Addison-Wesley Professional, 2004.

[15] Ahmed Seffah and Eduard Metzker. "The obstacles and myths of usability and software engineering". In: *Commun. ACM* 47 (2004), pp. 71–76. DOI: 10.1145/1035134.1035136.

[16] Behnam Faghih, Mohammad Azadehfar, and Seraj Katebi. *User Interface Design for E-Learning Software*. 2014. arXiv: 1401.6365 [cs.CY]. URL: https://arxiv.org/abs/1401.6365.

[17] GeeksforGeeks. *Qualities of Good User Interface Design*. Aug. 22, 2022. URL: https://www.geeksforgeeks.org/qualities-of-good-user-interface-design/ (visited on 03/13/2025).

[18] Tutorials Point. *Software User Interface Design*. 2025. URL: https://www.tutorialspoint.com/software_engineering/software_user_interface_design.htm (visited on 03/13/2025).

[19] Jakob Nielsen. "Iterative user-interface design". In: *Computer* 26.11 (1993), pp. 32–41.

[20] Brad A. Myers and Mary Beth Rosson. "Survey on user interface programming". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '92. Association for Computing Machinery, 1992, pp. 195–202. ISBN: 0897915135. DOI: 10.1145/142750.142789.

[21] Gerald L. Lohse and Peter Spiller. "Internet retail store design: How the user interface influences traffic and sales". In: *Journal of Computer-Mediated Communication* 5.2 (1999), JCMC522.

[22] MAT Pratama and AT Cahyadi. "Effect of User Interface and User Experience on Application Sales". In: *IOP Conference Series: Materials Science and Engineering*. Vol. 879. 1. IOP Publishing. 2020, p. 012133.

[23] Microsoft. *Visual Studio IntelliCode*. 2024. URL: https://visualstudio.microsoft.com/services/intellicode/ (visited on 11/12/2024).

[24] JetBrains. *PyCharm*. 2024. URL: https://www.jetbrains.com/pycharm/features/ai/ (visited on 11/12/2024).

[25] Cursor. *Cursor*. 2024. URL: https://www.cursor.com/ (visited on 11/12/2024).

[26] Figma. *Figma*. 2024. URL: https://www.figma.com/ai/ (visited on 11/12/2024).

[27] Quanjun Zhang et al. *A Survey on Large Language Models for Software Engineering*. 2024. arXiv: 2312.15223 [cs.SE]. URL: https://arxiv.org/abs/2312.15223.

[28] Xinyi Hou et al. "Large Language Models for Software Engineering: A Systematic Literature Review". In: *ACM Transactions on Software Engineering and Methodology* 33.8 (2024). ISSN: 1049-331X. DOI: 10.1145/3695988.

[29] Angela Fan et al. “Large Language Models for Software Engineering: Survey and Open Problems”. In: *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. 2023, pp. 31–53. DOI: 10.1109/ICSE-FoSE59343.2023.00008.

[30] Junwei Liu et al. *Large Language Model-Based Agents for Software Engineering: A Survey*. 2024. arXiv: 2409.02977 [cs.SE]. URL: https://arxiv.org/abs/2409.02977.

[31] Perplexity. *Context Window Limitations of LLMs*. July 15, 2024. URL: https://www.perplexity.ai/page/context-window-limitations-of-FKpx7M_ITz2rKXLFG1kNiQ (visited on 12/21/2024).

[32] Dave Bergmann. *What is a context window?* Published by IBM. Nov. 7, 2024. URL: https://www.ibm.com/think/topics/context-window (visited on 12/21/2024).

[33] GitHub. *Prompt engineering for Copilot Chat*. GitHub Copilot documentation. 2024. URL: https://docs.github.com/en/copilot/using-github-copilot/prompt-engineering-for-github-copilot (visited on 12/11/2024).

[34] Google AI. *Long context*. Gemini API documentation. Apr. 24, 2025. URL: https://ai.google.dev/gemini-api/docs/long-context (visited on 05/02/2025).

[35] Anthropic. *Claude 3.7 Sonnet*. Anthropic. 2025. URL: https://www.anthropic.com/claude/sonnet (visited on 04/29/2025).

[36] OpenAI. *o3-mini*. OpenAI o3-mini documentation. OpenAI. 2025. URL: https://platform.openai.com/docs/models/o3-mini (visited on 04/29/2025).

[37] OpenAI. *What are tokens and how to count them?* 2025. URL: https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them (visited on 03/03/2025).

[38] Alec Radford et al. *Improving Language Understanding by Generative Pre-Training*. 2018.

[39] Alec Radford et al. “Language Models are Unsupervised Multitask Learners”. In: *OpenAI blog* 1.8 (2019), p. 9.

[40] Tom Brown et al. “Language Models are Few-Shot Learners”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[41] Cheng-Ping Hsieh et al. *RULER: What’s the Real Context Size of Your Long-Context Language Models?* 2024. arXiv: 2404.06654 [cs.CL]. URL: https://arxiv.org/abs/2404.06654 (visited on 03/26/2025).

[42] Valentina Alto. *Building LLM Powered Applications: Create Intelligent Apps and Agents with Large Language Models*. First edition. Expert insight. Packt Publishing Ltd., 2024. ISBN: 1-83546-263-4.

[43] DeepSeek-AI et al. *DeepSeek-V3 Technical Report*. 2025. arXiv: 2412.19437 [cs.CL]. URL: https://arxiv.org/abs/2412.19437 (visited on 03/18/2025).

[44] Chuan Li. *OpenAI's GPT-3 Language Model: A Technical Overview*. Published by Lambda. Lambda. June 3, 2020. URL: https://lambda.ai/blog/demystifying-gpt-3 (visited on 04/29/2025).

[45] Iliya Valchanov. *How Much Did It Cost to Train GPT-4? Let's Break It Down*. Published by Team-GPT. July 12, 2024. URL: https://team-gpt.com/blog/how-much-did-it-cost-to-train-gpt-4/ (visited on 03/18/2025).

[46] Will Knight. *OpenAI's CEO Says the Age of Giant AI Models Is Already Over*. Published by WIRED. Apr. 17, 2023. URL: https://www.wired.com/story/openai-ceo-sam-altman-the-age-of-giant-ai-models-is-already-over/ (visited on 03/18/2025).

[47] Laura Funderburk. *automate-tech-post*. June 7, 2023. URL: https://github.com/lfunderburk/automate-tech-post (visited on 04/24/2025).

[48] Iqra Rai. *6 ChatGPT Plugins to Improve Your Digital Marketing*. Published by SocialBu. 2024. URL: https://socialbu.com/blog/chatgpt-plugins (visited on 04/29/2025).

[49] Compose AI. *Compose AI*. 2025. URL: https://www.compose.ai/ (visited on 04/29/2025).

[50] Meghashyam Thiruveedula. *Revolutionizing Invoice Processing: Leveraging LLMs for Effortless Extraction*. Published on Medium. Feb. 29, 2024. URL: https://meghashyamthiruveedula.medium.com/revolutionizing-invoice-processing-leveraging-llms-for-effortless-extraction-6b8a0ba48897 (visited on 04/24/2025).

[51] SAP. *What is an AI Copilot?* July 2, 2024. URL: https://www.sap.com/resources/what-is-ai-copilot (visited on 04/29/2025).

[52] Kylie Robison. *Inside the launch — and future — of ChatGPT*. Published by The Verge. Dec. 12, 2024. URL: https://www.theverge.com/2024/12/12/24318650/chatgpt-openai-history-two-year-anniversary (visited on 12/16/2024).

[53] GitHub. *GitHub Copilot Extension for Visual Studio Code*. Available on the Visual Studio Code Marketplace. Microsoft. May 1, 2025. URL: https://marketplace.visualstudio.com/items?itemName=GitHub.copilot (visited on 05/02/2025).

[54] GitHub. *GitHub Copilot*. 2024. URL: https://github.com/features/copilot (visited on 11/19/2024).

[55] James Prather et al. "“It's Weird That it Knows What I Want”: Usability and Interactions with Copilot for Novice Programmers". In: *ACM Transactions on Computer-Human Interaction* 31.1 (2023), pp. 1–31.

[56] Ruchika Pandey et al. *Transforming Software Development: Evaluating the Efficiency and Challenges of GitHub Copilot in Real-World Projects*. 2024. arXiv: 2406.17910 [cs.SE]. URL: https://arxiv.org/abs/2406.17910.

[57] Volker Bilgram and Felix Laarmann. "Accelerating Innovation with Generative AI: AI-augmented Digital Prototyping and Innovation Methods". In: *IEEE engineering management review* 51.2 (2023), pp. 1–5. ISSN: 0360-8581.

[58] Antonio Mastropaolo et al. *On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot*. 2023. arXiv: 2302.00438 [cs.SE]. URL: https://arxiv.org/abs/2302.00438.

[59] Fang Liu et al. *Exploring and Evaluating Hallucinations in LLM-Powered Code Generation*. 2024. arXiv: 2404.00971 [cs.SE]. URL: https://arxiv.org/abs/2404.00971 (visited on 03/18/2025).

[60] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. *A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges*. 2023. arXiv: 2303.17125 [cs.SE]. URL: https://arxiv.org/abs/2303.17125 (visited on 03/19/2025).

[61] Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. *In-IDE Code Generation from Natural Language: Promise and Challenges*. 2021. arXiv: 2101.11149 [cs.SE]. URL: https://arxiv.org/abs/2101.11149 (visited on 03/19/2025).

[62] Shraddha Barke, Michael B. James, and Nadia Polikarpova. "Grounded Copilot: How Programmers Interact with Code-Generating Models". In: *Proceedings of the ACM on Programming Languages* 7.OOPSLA1 (2023). DOI: 10.1145/3586030.

[63] Lenka Bajcetic, Drazen Draskovic, and Dragan Bojic. "Implementation of Graphical User Interface using Virtual Assistants based on LLMs". In: *2023 31st Telecommunications Forum (TELFOR)*. IEEE, 2023, pp. 1–4. ISBN: 9798350303131.

[64] Sarah Fakhoury et al. "LLM-Based Test-Driven Interactive Code Generation: User Study and Empirical Evaluation". In: *IEEE transactions on software engineering* 50.9 (2024), pp. 2254–2268. ISSN: 0098-5589. (Visited on 03/19/2025).

[65] Kristian Kolthoff et al. "Interlinking User Stories and GUI Prototyping: A Semi-Automatic LLM-Based Approach". In: *2024 IEEE 32nd International Requirements Engineering Conference (RE)*. IEEE, 2024, pp. 380–388. ISBN: 9798350395112.

[66] Fangwen Mu et al. "ClarifyGPT: A Framework for Enhancing LLM-Based Code Generation via Requirements Clarification". In: *Proceedings of the ACM on software engineering* 1.FSE (2024), pp. 2332–2354. ISSN: 2994-970X.

[67] Jason Wu et al. *UICoder: Finetuning Large Language Models to Generate User Interface Code through Automated Feedback*. 2024. arXiv: 2406.07739 [cs.CL]. URL: https://arxiv.org/abs/2406.07739.

[68] Shanchao Liang et al. *WAFFLE: Multi-Modal Model for Automated Front-End Development*. 2024. arXiv: 2410.18362 [cs.SE]. URL: https://arxiv.org/abs/2410.18362.

[69] Yuxuan Wan et al. *Automatically Generating UI Code from Screenshot: A Divide-and-Conquer-Based Approach*. 2024. arXiv: 2406.16386 [cs.SE]. URL: https://arxiv.org/abs/2406.16386.

[70] Ramesh Sharda et al. *The Vehicle Routing Problem: Latest Advances and New Challenges*. 1. Aufl. Vol. 43. Operations Research/Computer Science Interfaces. Springer-Verlag, 2008. ISBN: 0387777776.

[71] Grigorios D. Konstantakopoulos, Sotiris P. Gayialis, and Evripidis P. Kechagias. "Vehicle routing problem and related algorithms for logistics distribution: a literature review and classification". In: *Operational research* 22.3 (2022), pp. 2033–2062. ISSN: 1109-2858.

[72] Toth Paolo and Vigo Daniele. *Vehicle Routing: Problems, Methods, and Applications, Second Edition*. Second edition. MOS-SIAM series on optimization. Society for Industrial and Applied Mathematics (SIAM), 2014. ISBN: 9781611973587.

[73] G. Clarke and J. W. Wright. "Scheduling of Vehicles from a Central Depot to a Number of Delivery Points". In: *Operations research* 12.4 (1964), pp. 568–581. ISSN: 0030-364X. DOI: https://doi.org/10.1287/opre.12.4.568.

[74] Stefan Schröder. *jsprit*. 2024. URL: https://jsprit.github.io/ (visited on 10/28/2024).

[75] Romain Montagné and David Torres Sanchez. *VRPy*. 2020. URL: https://vrpy.readthedocs.io/en/latest/index.html (visited on 10/28/2024).

[76] Google. *Google OR-Tools*. 2024. URL: https://developers.google.com/optimization (visited on 10/28/2024).

[77] Gurobi Optimization. *Gurobi Optimization*. 2024. URL: https://www.gurobi.com/ (visited on 10/29/2024).

[78] IBM. *IBM ILOG CPLEX Optimization Studio*. 2024. URL: https://www.ibm.com/products/ilog-cplex-optimization-studio (visited on 10/28/2024).

[79] Lan Peng and Chase Murray. *VeRoViz: A Vehicle Routing Visualization Toolkit*. 2020. URL: https://ssrn.com/abstract=3746037 (visited on 03/03/2025).

[80] Vroom. *vroom-frontend*. Mar. 19, 2021. URL: https://github.com/VROOM-Project/vroom-frontend (visited on 03/03/2025).

[81] Claes. Wohlin et al. *Experimentation in Software Engineering*. 1st ed. 2012. Springer Berlin Heidelberg, 2012. ISBN: 3-642-29044-2.

[82] Roel J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. 2014 edition. Springer Nature, 2014. ISBN: 9783662438398.

[83] Allen S. Lee and Ned Kock. "Action is an Artifact: What Action Research and Design Science Offer to Each Other". In: *Information Systems Action Research*. Integrated Series in Information Systems. Springer US, 2007, pp. 43–60. ISBN: 9780387360591.

[84] Jean McNiff and Ebscohost . *Action research : principles and practice*. 3rd ed. Routledge, 2013. ISBN: 978-1-136-28375-8.

[85] Ernest T Stringer and Alfredo Ortiz Aragón. *Action research*. Sage publications, 2020.

[86] OpenAI. *ChatGPT*. 2024. URL: https://chatgpt.com/ (visited on 11/11/2024).

[87] Vercel. *v0 by Vercel*. 2024. URL: https://v0.dev/ (visited on 12/21/2024).

[88] Galileo AI. *Galileo AI*. 2024. URL: https://www.usegalileo.ai/explore (visited on 12/08/2024).

[89] Burke Holland. *Announcing a free GitHub Copilot for VS Code*. Published by Visual Studio Code. Dec. 18, 2024. URL: https://code.visualstudio.com/blogs/2024/12/18/free-github-copilot (visited on 12/20/2024).

[90] GitHub. *Announcing 150M developers and a new free tier for GitHub Copilot in VS Code*. Published by GitHub Blog. Dec. 18, 2024. URL: https://github.blog/news-insights/product-news/github-copilot-in-vscode-free/ (visited on 12/20/2024).

[91] Jussi Rasku. "Toward Automatic Customization of Vehicle Routing Systems". PhD thesis. University of Jyväskylä, 2019. URL: http://urn.fi/URN:ISBN:978-951-39-7826-6 (visited on 03/03/2025).

[92] Amelia Wattenberger. *Visualizing a Codebase*. Published by GitHub Next. 2025. URL: https://githubnext.com/projects/repo-visualization/ (visited on 04/02/2025).

[93] Al Danial. *cloc*. Version 2.04. Jan. 31, 2025. URL: https://github.com/AlDanial/cloc (visited on 03/14/2025).

[94] Kristian Skogberg. *VeRyPy-GUI*. Feb. 27, 2025. URL: https://github.com/kristianskogberg/VeRyPy-GUI (visited on 02/27/2025).

[95] Kristian Skogberg. *VeRyPy GUI design conversation on V0 by Vercel*. 2024. URL: https://v0.dev/chat/vrp-solver-ui-design-KE6eNECon3e?b=b_3Bq9bKRdINX (visited on 12/22/2024).

[96] Pawel Drozdowski. *TSPLib.Net*. URL: https://github.com/pdrozdowski/TSPLib.Net/tree/master (visited on 02/25/2025).

[97] Zhu Liang. *Code to Tokens Conversion: A Developer's Guide*. Published by 16x Prompt. June 18, 2024. URL: https://prompt.16x.engineer/blog/code-to-tokens-conversion (visited on 03/19/2025).

[98] Chris Dunlop. *Cursor went from 1–100m ARR in 12 months: the fastest SAAS to achieve this*. Published on Medium. Feb. 8, 2025. URL: https://medium.com/strategy-decoded/cursor-went-from-1-100m-arr-in-12-months-the-fastest-saas-to-achieve-this-19d811c4f0bb (visited on 03/24/2025).

[99] Upasana Banerjee. *Anysphere's AI-code Editor Cursor Fastest to Reach $100M ARR in 12 Months*. Published by AIM Research. Mar. 13, 2025. URL: https://aimresearch.co/market-industry/anyspheres-ai-code-editor-cursor-fastest-to-reach-100m-arr-in-12-months (visited on 03/24/2025).

[100] Kristian Skogberg. *VeRyPy GUI Chat Logs with GitHub Copilot*. Feb. 27, 2025. URL: https://github.com/kristianskogberg/VeRyPy-GUI/tree/master/verypy/gui/copilot_chat_logs (visited on 03/23/2025).

[101] Hyacinth S. Nwana. "Software Agents: An Overview". In: *The Knowledge Engineering Review* 11.3 (1996), pp. 205–244. DOI: 10.1017/S026988890000789X.

[102] Nicholas R. Jennings. "On agent-based software engineering". In: *Artificial intelligence* 117.2 (2000), pp. 277–296. ISSN: 0004-3702.

APPENDIX A: FEATURES AND USER STORIES OF THE VERYPY GUI

Table A.1. *Features and user stories of the VeRyPy GUI.*

ID	Feature	User Story
1	Problem Input	As a Dispatcher, I want to input customer locations via a form, file upload (e.g. VRP or TSP), or a combination of these methods, so that I can quickly set up the problem.
2	Problem Input	As a Dispatcher, I want to set constraints (e.g. vehicle capacity and customer demands), so that realistic limitations are respected.
3	Problem Input	As a Dispatcher, I want the system to validate the input data, so that I don't encounter errors during solving.
4	Algorithm Selection	As an Algorithm Developer, I want to select from multiple heuristic algorithms implemented in VeRyPy, so that I can test and compare different VRP-solving methods.
5	Algorithm Selection	As an Algorithm Developer, I want to set parameters for the selected algorithms (e.g. use single iteration and minimize number of routes), so that I can experiment with their behavior.
6	Solution Metrics	As an Algorithm Developer, I want to see metrics for the solution (e.g. execution time, total distance, and number of routes) alongside the visualization, so that I can assess the quality of the solution.

Continued on next page...

ID	Feature	User Story
7	Solution Metrics	As an Algorithm Developer, I want to compare the solutions generated by different algorithms, so that I can evaluate their effectiveness.
8	Solution Metrics	As a Dispatcher, I want to view detailed route summaries (e.g. sequence of stops), so that I can analyze the results further.
9	Solution Visualization	As a Dispatcher, I want to view the generated routes on a canvas, so that I can visually interpret the solution.
10	Solution Visualization	As a Dispatcher, I want each route to be displayed in a distinct color, so that I can differentiate between them.
11	Export Solution	As a Dispatcher, I want to export the solution metrics and visualization in a commonly used format (e.g. JSON and PNG), so that I can share it with others or use it in other tools.

APPENDIX B: ITERATION PLAN FOR THE VERYPY GUI

B.1 Iteration 1: GUI Design

Generate a GUI design for the VeRyPy library. The GUI should follow a minimalistic and intuitive design, while including all necessary input fields for solving VRPs. Additionally, the GUI should provide placeholders for displaying solution visualization and metrics.

User Stories

Because this iteration focuses on designing the GUI for VeRyPy, no user stories will be implemented.

Definition of Done

A completed GUI design that includes all necessary input fields for VRPs, such as adding customer locations, specifying vehicle capacity, and selecting a heuristic algorithm. The design will be integrated with the existing VeRyPy codebase in the next iteration.

B.2 Iteration 2: Basic GUI and Problem Input

Establish the foundation for the VeRyPy GUI by implementing the GUI design in code and integrating it with the existing VeRyPy codebase. The GUI implementation should minimize external dependencies to maintain modularity and align with the architectural design of VeRyPy. Selecting an appropriate implementation method is essential as the next iterations will build upon this foundation. This iteration should also introduce interactive functionalities that allow users to input details about their VRP, along with basic input validation.

User Stories

User stories that are planned to be implemented in this iteration are shown in Table B.1 below.

Table B.1. User stories planned for implementation in iteration 2.

User Story ID	Feature
1	Problem Input
2	Problem Input
3	Problem Input

Definition of Done

Users can interact with the GUI to input details about their VRP (e.g., customer locations and vehicle capacity) either by uploading a `.vrp` file or by entering the data directly into the input fields. The GUI also allows users to preview information about their problem before execution. Input validation is implemented to ensure that errors are detected and addressed before running the algorithm. Support for algorithm selection and execution will be added in the next iteration.

B.3 Iteration 3: Algorithm Selection and Execution

Integrate the existing heuristic algorithms from the VeRyPy codebase into the VeRyPy GUI. The GUI should dynamically retrieve these algorithms and present them as selectable options, allowing users to choose their preferred algorithm for solving VRPs. This iteration also includes the functionality to solve VRPs using the selected algorithm.

User Stories

User stories that are planned to be implemented in this iteration are shown in Table B.2 below.

Table B.2. User stories planned for implementation in iteration 3.

User Story ID	Feature
4	Algorithm Selection
5	Algorithm Selection

Definition of Done

Users can choose from all the heuristic algorithms implemented in VeRyPy. The GUI computes a solution for the VRP based on the selected algorithm and the parameters entered by the user. The solution will be displayed in the GUI in the next iteration.

B.4 Iteration 4: Solution Visualization, Metrics, and Export Options

Display the solution, metrics, and a visualization of the solution in the VeRyPy GUI. Users should also be able to export the solution data, including calculated routes, metrics, and visualization, in commonly used file formats such as JSON and PNG.

User Stories

User stories that are planned to be implemented in this iteration are shown in Table B.3 below.

Table B.3. *User stories planned for implementation in iteration 4.*

User Story ID	Feature
6	Solution Metrics
7	Solution Metrics
8	Solution Metrics
9	Solution Visualization
10	Solution Visualization
11	Export Solution

Definition of Done

Users can view the VRP solution, including the calculated routes and key metrics (such as computation time and solution feasibility), visually in the GUI. Users can also export the solution and metrics to their machine.

B.5 Iteration 5: Improved Problem Input

Improve the problem input section of the VeRyPy GUI to enhance usability. Users should be able to enter coordinates and customer demands without needing to specify an index. The option to input the problem in `.tsp` format should also be available, along with the choice to enter coordinates as regular numerical values or in geographical format using longitude and latitude. Finally, the GUI must be thoroughly tested to ensure it functions correctly when users manually input problem details, upload a file, or use a combination of both methods.

User Stories

User stories that are planned to be implemented in this iteration are shown in Table B.4 below.

Table B.4. *User stories planned for implementation in iteration 5.*

User Story ID	Feature
1	Problem Input
2	Problem Input
3	Problem Input

Definition of Done

Users can input coordinates and customer demands without the need to manually provide indexes. They can also upload a file in either `.vrp` or `.tsp` format. Users can select either VRP (specifically the CVRP variant) or TSP as their problem type. The GUI should also solve problems whether they are entered by uploading a file, manually, or through a combination of these methods. Additionally, the GUI dynamically displays relevant input fields and hides irrelevant ones based on the selected problem type (VRP or TSP).