

AUTOMATIC GENERATION OF NAVIGABLE USER INTERFACES AND
EXECUTABLE LOGIC FROM CUSTOMER REQUIREMENTS.

by

Harikrishnan Kuttan Pillai, MSc

Student ID: 59644

DISSERTATION

Presented to the Swiss School of Business and Management Geneva

In Partial Fulfillment

Of the Requirements

For the Degree

DOCTOR OF BUSINESS ADMINISTRATION

SWISS SCHOOL OF BUSINESS AND MANAGEMENT GENEVA

August, 2025

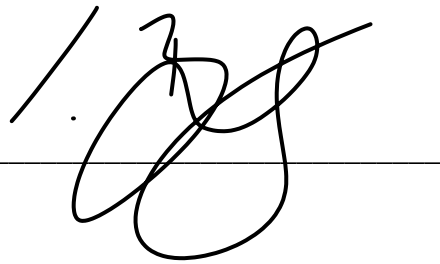
AUTOMATIC GENERATION OF NAVIGABLE USER INTERFACES AND
EXECUTABLE LOGIC FROM CUSTOMER REQUIREMENTS.

by

Harikrishnan K

APPROVED BY

Dissertation chair

A handwritten signature in black ink, consisting of a series of loops and a long horizontal stroke at the end, positioned over a horizontal line.

RECEIVED/APPROVED BY:

Renee Goldstein Osmic

Admissions Director

Dedication

This thesis is dedicated to my esteemed colleagues, whose collaboration, support, and encouragement have been invaluable throughout this research journey. Your insights and shared knowledge have enriched this work and inspired me to strive for excellence.

To my family, whose unwavering love, patience, and understanding have been my bedrock. Your belief in me has been a constant source of motivation, and I am deeply grateful for your sacrifices and support.

Finally, to my guide and mentor, whose wisdom, guidance, and encouragement have shaped this research. Your dedication to nurturing and guiding me has been instrumental in the completion of this work. Thank you for your steadfast support and invaluable contributions.

Acknowledgements

I would like to express my sincere gratitude to the following individuals and groups for their invaluable support and assistance throughout the process of completing this thesis.

My deepest appreciation goes to my supervisor, **Dr. Monica Singh** for their unwavering guidance, patience, and expertise. Their insightful feedback and continuous encouragement played a crucial role in shaping this research.

I am also grateful to the members of my thesis committee, SSBM Review Committee Members, for their valuable input and constructive criticism that enriched the quality of this work. I am indebted to my friends and family for their unwavering support, understanding, and motivation during this academic journey. Your belief in me and your encouragement were the driving forces behind the completion of this thesis.

Lastly, I would like to acknowledge the participants in my study who generously shared their time and insights, without whom this research would not have been possible.

This thesis is a culmination of the efforts of many, and I am grateful for the support and inspiration I have received from all these individuals.

Thank You,

Harikrishnan K

ABSTRACT

AUTOMATIC GENERATION OF NAVIGABLE USER INTERFACES AND EXECUTABLE LOGIC FROM CUSTOMER REQUIREMENTS.

Harikrishnan Kuttan Pillai

2025

Dissertation Chair: Dr. Monica Singh

This research analyzes the current state of the art in this field and aims to propose a new system for automatically converting the user requirements to end-user applications. The system translates the stated user requirements into an intermediate language after they have been gathered, and with the aid of pre-trained data, it can then translate them into UI code. In this phase, both business logic and navigation logic are integrated. We also apply an iterative model variant in the proposed research, where code is generated based on continuous feedback from various iterations of customer requirement gatherings.

Keywords: automation, artificial intelligence, software engineering, natural language processing.

TABLE OF CONTENTS

List of Abbreviations	v
List of Figures	vi
List of Tables	vi
Chapter I: INTRODUCTION	1
1.1 Research Problem	1
1.2 Purpose of Research	3
1.3 Significance of the Study	3
1.4 Research Purpose and Objectives	4
Chapter II: REVIEW OF LITERATURE	7
2.1. Capturing user requirements	7
2.2. Converting captured user	9
2.3. Integrating Business and Navigation Logic	10
2.4. Assembling the application	11
2.5. Identified Gaps in Current Research	13
Chapter III: METHODOLOGY	15
3.1. Overview of the Research Problem	15
3.2. Operationalization of Theoretical Constructs	15
3.3. Research Purpose and Questions	16
3.4. Research Design	17
3.5. Population and Sample	18
3.6. Survey questions	18
3.7. Participant Selection	25
3.7. Data Analysis	25
3.8. Instrumentation	25
3.9. Data Collection Procedures	26
3.10. Research Design Limitations	26
3.11. Conclusion	27
Chapter IV: RESULTS	28
4.1 Survey Results	28
4.2 What techniques effectively extract and systematize user requirements from raw documents?	29
4.3 How can chatbots enhance the accuracy and completeness of requirement extraction?	31

4.4	How can user requirements be translated into UI code that meets end-user expectations?	31
4.5	What methods ensure the generated UI code is user-approved and accurately reflects specified requirements?	32
4.6	How can additional logical and business requirements be effectively gathered and converted into functional code?	33
4.7	How can a continuous feedback system be implemented to validate the logic and accuracy of the generated code?	34
4.8	What features should an IDE and project structure include to support seamless development and navigation logic?	35
4.9	Summary of Findings	37
4.10	Conclusion	37
Chapter V: DISCUSSION		39
5.1	What techniques effectively extract and systematize user requirements from raw documents?	39
5.2	How can chatbots enhance the accuracy and completeness of requirement extraction?	40
5.3	How can user requirements be translated into UI code that meets end-user expectations?	42
5.4	What methods ensure the generated UI code is user-approved and accurately reflects specified requirements?	43
5.5	How can additional logical and business requirements be effectively gathered and converted into functional code?	45
5.6	How can a continuous feedback system be implemented to validate the logic and accuracy of the generated code?	47
5.7	What features should an IDE and project structure include to support seamless development and navigation logic?	48
5.8	Conclusion	50
Chapter VI: A BASIC IMPLEMENTATION FOR RESEARCH VALIDATION		51
6.1	Introduction	51
6.2	Technologies Used	54
6.3	System Design and Architecture	61
6.4	Front-end Development	67
6.5	Back-end Development	71
6.6	Project Features	74
6.7	Security	87
6.8	Deployment	93
6.9	Challenges and Solutions	98
6.10	Future Enhancements	103

6.11 Conclusion	108
Chapter VII: SUMMARY, IMPLICATIONS, AND RECOMMENDATIONS	111
7.1 Implications	111
7.2 Recommendations for Future Research	113
7.3 Conclusion	114
APPENDIX A: INFORMED CONSENT	116
References	118

LIST OF ABBREVIATIONS

UI	User interface
UX	User experience
IL	Intermediate Language
GUI	Graphical User Interface
IDE	Integrated development environment
JSON	JavaScript object notation
PoC	Proof of concept
LLM	Large language model
AI	Artificial Intelligence
ML	Machine Learning
NLP	Natural language processing
DB	Database
SQL	Structured Query Language

LIST OF FIGURES

Figure 1: High level architecture	38
Figure 2: PoC Flow chart.....	62
Figure 3: Communications between components - PoC.....	64
Figure 4: Screen shot – Create and List Project - PoC	79
Figure 5: Screen shot – Chat bot - PoC.....	80
Figure 6: Chat bot with responses - PoC	80
Figure 7: End-user application - PoC.....	82

LIST OF TABLES

Table 1: Survey Results Summary.....	29
--------------------------------------	----

CHAPTER I: INTRODUCTION

In recent years, the software development landscape has undergone a radical transformation driven by the integration of artificial intelligence (AI), particularly natural language processing (NLP) and large language models (LLMs). These technologies are enabling developers to automate tasks that were traditionally manual, time-consuming, and error-prone.

Manual requirements gathering and translation into executable code remain one of the most error-sensitive stages of software engineering. Misunderstood or miscommunicated requirements account for nearly 70% of project failures (Fan et al., 2023). As software becomes more complex and delivery cycles shrink, organizations are increasingly adopting intelligent systems to bridge this communication gap (Kessentini et al., 2021).

AI-assisted development tools, such as OpenAI Codex and GitHub Copilot, have demonstrated promising results in interpreting natural language input and producing functional code snippets (Chen et al., 2021). However, these tools are largely unidirectional and lack iterative dialogue support. They do not adequately involve the user in refining the output or integrating feedback during the development process (Friesen et al., 2018).

To address this limitation, industry leaders are shifting toward AI agents that can manage tasks autonomously. Amazon's "Project Kiro" and Meta's internal initiatives exemplify the industry's move from "co-pilot" to "autopilot" development tools—where agents not only assist but take over portions of the software engineering lifecycle (Business Insider, 2025a; Financial Times, 2025). These developments underscore the growing importance of conversational AI interfaces that support dynamic interaction and real-time requirement refinement.

This study contributes to this evolving field by proposing and validating a chatbot-driven system that can gather requirements in natural language and automatically generate a corresponding application prototype, including UI components and business logic. Unlike static form-based requirement gathering, the proposed system allows iterative clarification and continuous updates to both design and function.

By integrating NLP pipelines, LLMs, and feedback-based refinement, this research builds a foundation for interactive and autonomous systems capable of improving requirement accuracy, accelerating development, and reducing developer workload (Hou et al., 2023; Raffel et al., 2020).

1.1 Research Problem

High-fidelity GUI prototyping provides a meaningful way to illustrate the developers' understanding of the requirements formulated by the customer and can be used for productive discussions and clarification of requirements and expectations. (Kolthoff, 2019). Understanding the informal requirements and translating them to a product often causes misunderstandings, and it is an iterative process. Several studies have been conducted to turn GUI pictures into executable prototypes (Beltramelli, 2018, Moran et al., 2018). But it is insufficient for an end-to-end application. A UI/UX designer is also necessary to create the UI screens. Iterative processes and integration with actual logic are not permitted in the intermediate language-based approach put forth by Kolthoff in 2019. Furthermore, a navigation logic must be in place. An IDE system and project structure are required to control the navigation process and per-screen logics. The research intends to suggest a system using the chat bot for client communication (Friesen et al., 2018), as well as navigation logic, backend integration logic, and other things. The research's iterative methodology aids in enhancing the code and user interface. Agile techniques are the foundation for the iterative process concept. The research's iterative methodology aids in enhancing the code and user interface.

1.2 Purpose of Research

The research primarily aims to achieve two things. One involves systematizing basic user requirements, while the other involves translating these requirements into UI, business, and application logic code. The logic for requirement extraction makes use of current techniques for model construction and requirement extraction. It will take broad strokes from the raw user requirement documents and may employ a chatbot to fill in the blanks and provide ongoing feedback. The system then creates a UI code using the specifications and gets it approved by the end user. Following that, it asks the user for additional logical and business criteria. With the use of current technologies like OpenAI Codex, this will be turned into code. Following conversion, the code will be injected into the appropriate UI event handlers. It has both navigational logic and business logic. Here, the implementation of the continuous feedback system will guarantee that the logic is sound. The application will be prepared and ready for execution after the code has been inserted. During this stage, the developer may make any adjustments they see fit. An IDE and a project structure will be present throughout the entire system to facilitate development.

1.3 Significance of the Study

In today's technology-driven world, software systems must evolve rapidly to keep pace with user expectations and business needs. Traditional methods of requirement gathering and implementation remain highly manual, introducing inefficiencies and the potential for errors (Smith et al., 2020). The significance of this study lies in its attempt to bridge this gap by automating the conversion of natural language user requirements into complete software systems.

Leveraging technologies such as OpenAI Codex and chatbots, the proposed system aims to streamline the software development lifecycle. Research by Gulwani, 2010, Raffel et al., 2020, and Desai et al., 2016 confirms the feasibility of converting user input into executable

code, though they note that integration with business logic remains a challenge. This study builds on those findings, offering a system that handles both UI code generation and application logic.

The iterative methodology employed, grounded in Agile principles, ensures that users can refine requirements continuously (Moran et al., 2018). This minimizes communication breakdowns and improves the alignment between stakeholder expectations and final output. Ultimately, this study presents a system with the potential to advance automation in software engineering by enhancing development speed and product accuracy.

1.4 Research Purpose and Objectives

The primary purpose of this research is to develop a system that automates the translation of user requirements into complete applications, including user interface (UI), business logic, and application logic code. The study aims to achieve two main objectives:

1.4.1. Systematizing basic user requirements:

- Utilize current techniques for model construction and requirement extraction to gather and refine broad user requirements from raw documents.
- Employ chatbots to fill in gaps, provide continuous feedback, and enhance understanding of informal requirements.

1.4.2. Translating requirements into application codes:

- Generate UI code based on the extracted specifications and obtain end-user approval.
- Collect additional logical and business criteria from the user.
- Convert these requirements into code using technologies like OpenAI Codex.
- Integrate the generated code into appropriate UI event handlers, encompassing navigational and business logic.

- Implement a continuous feedback system to ensure logical soundness and make necessary adjustments during the development stage.
- Facilitate development with an integrated development environment (IDE) and project structure.

1.4.3. Research objectives and questions

To guide the research and achieve its objectives, the following questions will be addressed:

- **Requirement Extraction:**
 - What techniques can be used to effectively extract and systematize user requirements from raw documents?
 - How can chatbots be utilized to enhance the accuracy and completeness of requirement extraction?
- **UI Code Generation:**
 - How can user requirements be translated into UI code that meets end-user expectations?
 - What methods can ensure the generated UI code is user-approved and accurately reflects the specified requirements?
- **Integration of Logic and Code:**
 - What strategies can be employed to gather additional logical and business requirements from users?
 - How can these requirements be effectively converted into functional code using current technologies like OpenAI?
- **Continuous Feedback System:**

- How can a continuous feedback system be implemented to validate the logic and ensure the accuracy of the generated code?
- What measures can be taken to facilitate adjustments and improvements during the development process?
- **Development Facilitation:**
 - What features should an integrated development environment (IDE) and project structure include to support seamless development and navigation logic?
 - How can the IDE and project structure enhance the overall efficiency and quality of the application development process?

By addressing these questions, the research aims to develop a robust system that automates the translation of user requirements into fully functional applications, ultimately advancing the efficiency and effectiveness of the software development lifecycle.

CHAPTER II: REVIEW OF LITERATURE

The evolution of software development methodologies has been driven by the growing complexity of applications and the demand for rapid, high-quality solutions. Traditional development approaches often involve extensive manual effort in translating user requirements into functional software, creating a gap between stakeholders' visions and the final product. This research investigates the potential of automating this translation process using AI, particularly Large Language Models (LLMs), to revolutionize how software is conceptualized and implemented.

The literature review delves into the existing body of work on requirement engineering, natural language processing (NLP) in software development, and the role of AI-driven tools in automating key stages of the development lifecycle. It explores the strengths and limitations of these approaches, focusing on their applications in requirement analysis, code generation, and iterative refinement. Additionally, the review highlights the challenges faced by traditional methods in capturing nuanced user requirements and examines how recent advancements in AI provide innovative solutions to these issues.

2.1. Capturing user requirements

Planning and requirement analysis are the most vital and basic phases of every life cycle process. It is completed by the senior members after a meeting with the customer or owner of the software system (Shylesh, 2017).

Gathering requirements is among the initial tasks in all commonly used life cycle models for software development. It is a required initial step before each iteration in all iterative models as well. We are also applying an iterative model variant in the proposed research, where

code is generated based on continuous feedback from various iterations of user requirements gatherings.

Some of the common requirement gathering technologies are interviews, questionnaires, task analysis, and observations. Prototyping is one of the most valuable solutions for capturing requirements. Prototyping has no value for the early phase of requirements engineering. It allows determining very concrete and detailed requirements at the time when introductory requirements are already collected. (Silhavy et al., 2011)

An attempt to create conversational software was made in 2017 with the creation of a program named Rasa by Bocklisch, Faulkner, Pawlowski, and others. It simply comprehends what the user is talking about and responds with pertinent comments and advice. Friesen et al., 2018 demonstrated that a chatbot can be used as a communication interface to identify user requirements. Ravid & Berry, 2000 provided some technics for identifying and specifying software needs from a user interface prototype. In later stages of development, when a prototype and other documents are present, it might be challenging to reconcile the differences between them. This study addresses this issue. To create the requirement extraction system, we might apply a variety of the strategies covered in all these studies.

Making the extracted requirements into UI code is the next difficult task. A tool called pix2code was developed by Beltramelli in 2018 to produce code from a screenshot of a graphical user interface. However, the user must create the UI and submit it to the system. Similar techniques were developed by Zhang et al., 2023 to skeletonize user interface designs. To bootstrap the mobile GUI implementation, a neural machine translation is used. Additionally, Moran et al. 2018 worked on a different study to translate mobile UI code from a mockup. They take different pieces of the mockup and generate code for each one, making it simple to adapt to future requirements changes. The process of translating natural language

requirements into a UI design is not well studied. Our research aims to close this puzzle piece's gap.

2.2. Converting captured user

Once the user requirements have been recorded, they must be translated into an intermediate language (IL) in order to facilitate further processing. Synthesizing a program in some underlying intermediate language (IL) from a given specification is known as program synthesis. (Gulwani. 2010).

A general synthesis algorithm uses an English sentence as input and outputs a range of potential IL codes. For this, we can make use of a dictionary with a powerful key phrase extraction method. All common entities will have key words in the dictionary. A web application might have keywords like "home page," "login page," "link," "picture," etc. A dictionary containing actions as its elements will also be used by the key phrase extraction logic. For instance, in the web application, we will have actions like navigate, click, go, etc.

Many pre-existing concepts about program synthesis were discovered during the literature review. Program synthesis using natural language, by Desai et al. 2016, developed a meta-approach for synthesizing programs from natural language descriptions that can be instantiated for a range of interesting IL's including text-processing, automata construction, and information retrieval queries.

Gulwani & Marron, 2014, created an interactive programming tool using natural language for manipulating and analyzing spreadsheet data. Their methodology involves designing a typed IL that supports an expressive algebra of map, filter, reduce, join, and formatting capabilities at a level of abstraction appropriate for non-expert users. The key algorithmic

component of the methodology is a translation algorithm for converting a natural language specification in the context of a given spreadsheet to a ranked set of likely programs in the IL.

Using formal IL to describe the GUI and its navigational schema, K. Kolthoff, 2019 proposes a methodology based on Natural Language Processing (NLP). This methodology would support GUI prototyping by automatically converting Natural Language Requirements (NLR) into a formal IL. The resulting IL can be directly shown to the user for inspection after being further translated into corresponding target platform prototypes. They present an intelligent and automatic interaction system that enables users to submit natural language input on created prototypes in an iterative manner, translating that feedback into the appropriate prototype adjustments.

Recent literature has highlighted the increasing performance of large language models (LLMs) like GPT-4 and CodeT5 in interpreting unstructured input and converting it into usable code representations (Hou et al., 2023; Zhang et al., 2023). These models have outperformed previous NLP-based approaches by incorporating contextual embedding, chain-of-thought reasoning, and fine-tuning on domain-specific corpora (Fan et al., 2023).

2.3. Integrating Business and Navigation Logic

The logic must be merged after the UI has been finalized. In this phase, both business logic and navigation logic are integrated. Chen et al. (2021) demonstrated that a significant portion of this business logic can be produced utilizing the pre-trained data and techniques.

There are numerous tools and techniques available today to produce logical code from the user requirements in plain English. Open AI Codex and its variant, GitHub Copilot are a few examples of systems that convert plain English to executable code. In Codex, they introduced a GPT language model fine-tuned on publicly available code from GitHub (Chen et al., 2021).

In this paper, they explain how to generate code from Python Docstrings. Python documentation strings (or docstrings) provide a convenient way of associating documentation with Python modules, functions, classes, and methods. They frequently use terminology that is understandable to others. Codex will then use a technique that will utilize the models generated from public code on GitHub to automatically construct the correct executable code that satisfies the criterion.

Pulido-Prieto & Juárez-Martínez, 2018 compiled a list of 31 different methods for tools and programming languages that help users incorporate natural language aspects into their programs. Naturalistic programming, as they refer to it, is a formal and deterministic implementation of features from natural language.

2.4. Assembling the application

The next stage is to construct the application after its components have been generated. Just gathering the web pages in one place and creating links to allow users to browse between them is the construction of a web application. A previously trained model or the user's requirements can also be used to generate links. With pre-trained data, links for typical situations like login, registration, etc. can be generated.

We require a more sophisticated strategy for business logic-based navigation. As existing technologies produce code for a particular business logic, it might be difficult to logically combine them to solve a significant business problem. The individual logic components could be thought of as nodes, and they are all connected to some other logic components. by edges. Finding the correct route from one node to another is our problem. By examining the flow, these routes may typically be discovered based on the user requirements. Yet, the system must

also be capable of creating links between nodes on its own. This is one of the research's novelties.

The literature review highlights the extensive research conducted in the domains of requirement engineering, natural language processing (NLP), and AI-assisted software development. Traditional approaches to translating user requirements into functional applications often suffer from inefficiencies, misinterpretations, and prolonged development cycles. However, advancements in AI, particularly Large Language Models (LLMs), present innovative methods to address these challenges by automating and enhancing critical phases of software development.

The review reveals that while several tools and frameworks exist for automated code generation and requirement analysis, most solutions lack the adaptability and iterative feedback mechanisms essential for accurately aligning with user expectations. Recent studies emphasize the importance of integrating AI with user-centric, Agile methodologies to create systems that are both efficient and robust.

This foundation of knowledge underscores the relevance of the proof of concept (PoC), which aims to leverage LLMs for automating the development process. By building upon existing research and addressing identified gaps, this PoC seeks to contribute to the field by demonstrating the practical feasibility of AI-driven approaches in bridging the gap between user requirements and application functionality. This review lays the groundwork for designing and implementing a solution that aspires to transform traditional software development paradigms.

Emerging AI agents like Amazon's "Kiro" are designed to streamline the entire UI generation process by interpreting developer prompts and creating layouts or logic automatically (Business Insider, 2025a). Similar trends have been seen across the industry,

with Meta and Google developing autonomous agents capable of managing development tasks previously handled by engineers (Business Insider, 2025b; Reuters, 2025).

2.5. Identified Gaps in Current Research

Despite numerous studies on automated code generation and requirement extraction (Gulwani, 2010; Gulwani & Marron, 2014; Desai et al., 2016), gaps persist in creating an integrated system that can translate informal natural language into fully functional applications. Tools like Rasa (Bocklisch et al., 2017) and Codex (Chen et al., 2021) have made significant progress in understanding intent, but they do not support iterative refinement based on real-time feedback.

Pulido-Prieto and Juárez-Martínez (2018) identify a need for systems that incorporate naturalistic programming, adapting to the way users communicate rather than requiring structured input. Moreover, few systems address the integration of both UI and backend logic from the same conversational interface (Kolthoff, 2019). This study seeks to fill that gap by developing an AI-based system that accepts user feedback iteratively and updates application logic accordingly.

This literature review establishes that while foundational elements exist for automating parts of the software lifecycle, an end-to-end solution that incorporates real-time feedback, logic generation, and usability validation remains an open problem.

2.6. AI in Agile and Enterprise Settings

Enterprise adoption of AI in software pipelines is accelerating. Goldman Sachs, for instance, has developed a suite of LLM-powered tools to enhance internal development processes (Business Insider, 2025c). These tools aim to reduce miscommunication, accelerate

prototyping, and even support test case generation. Anthropic’s founders describe this as the rise of “AI manager-nerds”—agents who handle managerial coding tasks with minimal supervision (Business Insider, 2025d).

CHAPTER III: METHODOLOGY

This section outlines the approach and strategies employed to develop and implement the research project. It also details the systematic processes, tools, and techniques used in the design, development, and evaluation of the proposed system. It covers both the technical and non-technical aspects of the research, providing a clear framework for how the objectives were achieved. The methodology provides insight into the steps taken to ensure the success of this research.

3.1. Overview of the Research Problem

The core research challenge lies in bridging the disconnect between how users express their software needs and how developers implement them. Traditional methods require manual interpretation of requirements and multiple feedback cycles, which increase development time and cost (Shylesh, 2017).

Tools like Codex (Chen et al., 2021) and GitHub Copilot help generate code from structured prompts, but they fall short of producing full applications with UI and backend logic. This study aims to automate this gap using a chatbot-based interface that gathers requirements, generates intermediate representations, and synthesizes UI and logic code. By implementing an Agile-inspired iterative model, the system provides a foundation for real-time updates and user validation.

3.2. Operationalization of Theoretical Constructs

To address the research problem, several theoretical constructs are operationalized:

- **Requirement Extraction:** Utilizing natural language processing (NLP) and machine learning models to systematically extract user requirements from raw documents.

- **Continuous Feedback:** Implementing a chatbot system for iterative feedback, refining requirements, and enhancing understanding through user interactions.
- **UI Code Generation:** Developing algorithms to convert refined requirements into UI code, ensuring alignment with user expectations.
- **Logic Integration:** Formulating methods to translate business and application logic into executable code and integrating it with the generated UI.
- **Agile Methodology:** Applying Agile principles to facilitate iterative development, continuous feedback, and improvement of the code and UI.

3.3. Research Purpose and Questions

This section outlines the primary purpose of the research and the key questions guiding the investigation.

3.3.1 Research Purpose

The primary purpose of this research is to develop a system that automates the translation of user requirements into complete applications, encompassing UI, business logic, and application logic code. The research aims to:

- Systematize basic user requirements using advanced requirement extraction techniques.
- Translate these requirements into application code, validated through continuous feedback.

3.3.2 Research Questions

- What techniques effectively extract and systematize user requirements from raw documents?

- How can chatbots enhance the accuracy and completeness of requirement extraction?
- How can user requirements be translated into UI code that meets end-user expectations?
- What methods ensure the generated UI code is user-approved and accurately reflects specified requirements?
- How can additional logical and business requirements be effectively gathered and converted into functional code?
- How can a continuous feedback system be implemented to validate the logic and accuracy of the generated code?
- What features should an IDE and project structure include to support seamless development and navigation logic?

3.4. Research Design

This research adopts a mixed-methods approach to comprehensively address the research questions. The design includes:

- **Requirement Extraction:** Utilizing NLP and machine learning models to process and refine user requirements.
- **Prototyping:** Iterative development of UI code based on extracted requirements, followed by user validation.
- **Integration and Testing:** Incorporating business logic and application logic into the UI, and conducting rigorous testing to ensure functionality and user satisfaction.
- **Continuous Feedback:** Employing chatbots to facilitate ongoing user feedback and iterative refinement of the application.

3.5. Population and Sample

The target population for this research includes software developers, UI/UX designers, and end-users who provide the initial requirements for application development. The sample will be drawn from a diverse group of participants across different industries to ensure the system's applicability and generalizability. The sample size will be determined based on the principles of data saturation for qualitative analysis and statistical power for quantitative analysis.

3.6. Survey questions

Demographics

1. What is your professional role?
 - Software Developer
 - UI/UX Designer
 - Business Analyst
 - End-User
 - Other (specify): _____
2. How many years of experience do you have in software development or related fields?
 - 0–2 years
 - 3–5 years
 - 6–10 years
 - 10+ years

Requirement Extraction

3. How effective do you find NLP models (e.g., GPT) in extracting user requirements from raw documents?
 - Very effective
 - Effective
 - Neutral
 - Ineffective
 - Very ineffective

4. How often do misunderstandings occur when translating informal user requirements into formal specifications?
 - Very frequently
 - Frequently
 - Occasionally
 - Rarely
 - Never
5. To what extent do chatbots improve the accuracy of requirement gathering compared to traditional methods (e.g., interviews)?
 - Significantly improve
 - Slightly improve
 - No difference
 - Slightly reduce
 - Significantly reduce

UI Code Generation

6. How satisfied are you with the quality of UI code generated automatically from user requirements? (For POC)
 - Very satisfied
 - Satisfied
 - Neutral
 - Dissatisfied
 - Very dissatisfied
7. How well does the generated UI code align with end-user expectations?
 - Fully aligns
 - Mostly aligns
 - Partially aligns
 - Rarely aligns
 - Does not align
8. How frequently do you need to manually adjust the generated UI code?
 - Always

- Often
- Sometimes
- Rarely
- Never

Chatbot Effectiveness

9. How intuitive is the chatbot interface for gathering user requirements?

- Very intuitive
- Intuitive
- Neutral
- Unintuitive
- Very unintuitive

10. How effective is the chatbot in clarifying ambiguous requirements through iterative feedback?

- Very effective
- Effective
- Neutral
- Ineffective
- Very ineffective

11. How often does the chatbot fail to understand complex user requirements?

- Very frequently
- Frequently
- Occasionally
- Rarely
- Never

Business Logic Integration

12. How well does the system convert additional business logic requirements into functional code?

- Very well
- Well
- Neutral

- Poorly
- Very poorly

13. How often do you need to manually refine the generated business logic code?

- Always
- Often
- Sometimes
- Rarely
- Never

14. How satisfied are you with the integration of navigation logic into the generated application?

- Very satisfied
- Satisfied
- Neutral
- Dissatisfied
- Very dissatisfied

Continuous Feedback System

15. How useful is the continuous feedback system in validating the accuracy of generated code?

- Very useful
- Useful
- Neutral
- Not useful
- Very not useful

16. How often does the feedback system identify logical errors in the generated code?

- Always
- Often
- Sometimes
- Rarely
- Never

17. How responsive is the system to user feedback during iterative development?

- Very responsive
- Responsive
- Neutral
- Unresponsive
- Very unresponsive

System Usability

18. How easy is it to navigate the generated application?

- Very easy
- Easy
- Neutral
- Difficult
- Very difficult

19. How would you rate the overall usability of the automated system?

- Excellent
- Good
- Average
- Poor
- Very poor

20. How much time does the system save in the application development process?

- Significant time saved
- Moderate time saved
- Neutral
- Little time saved
- No time saved

IDE and Project Structure

21. How helpful are the IDE features (e.g., code navigation, templates) in supporting development?

- Very helpful
- Helpful
- Neutral

- Unhelpful
- Very unhelpful

22. How well does the project structure facilitate modular development?

- Very well
- Well
- Neutral
- Poorly
- Very poorly

23. How often do you encounter difficulties in managing the generated project structure?

- Always
- Often
- Sometimes
- Rarely
- Never

Comparative Analysis

24. Compared to traditional development, how would you rate the efficiency of the automated system?

- Much more efficient
- More efficient
- Neutral
- Less efficient
- Much less efficient

25. How does the quality of the generated code compare to manually written code?

- Much better
- Better
- Neutral
- Worse
- Much worse

Future Enhancements

26. How important is it to add page-wise intermediate code management for scalability?
- Very important
 - Important
 - Neutral
 - Not important
 - Very not important
27. How likely are you to adopt this system in your workflow if advanced UI components (e.g., dashboards) are supported?
- Very likely
 - Likely
 - Neutral
 - Unlikely
 - Very unlikely
28. How critical is integrating authentication and role management for your use cases?
- Very critical
 - Critical
 - Neutral
 - Not critical
 - Very not critical

Overall Satisfaction

29. How likely are you to recommend this automated system to others?
- Very likely
 - Likely
 - Neutral
 - Unlikely
 - Very unlikely
30. What is your overall satisfaction with the system's ability to automate application development?
- Very satisfied

- Satisfied
- Neutral
- Dissatisfied
- Very dissatisfied

3.7. Participant Selection

Participants will be selected using purposive sampling to ensure that they possess relevant experience and knowledge. Criteria for selection include:

- **Software Developers:** Experience in coding, UI development, and application logic integration.
- **UI/UX Designers:** Expertise in creating user-friendly interfaces and understanding user requirements.
- **Business Analysts:** Experience in client interactions, software requirements extraction and translation to software developers
- **End-Users:** Individuals who represent the target audience of the applications, capable of providing detailed requirements and feedback.

3.7. Data Analysis

Data analysis will involve the following methods.

- **Quantitative Analysis:** Statistical analysis of survey responses to quantify user satisfaction and the effectiveness of the requirement extraction and UI code generation processes.
- **Validation and Testing:** Performance metrics and user testing results are used to assess the functionality and usability of the developed applications.

3.8. Instrumentation

The research will utilize various instruments for data collection and analysis, including:

- **NLP and Machine Learning Tools:** Software tools for processing and extracting requirements from raw documents.
- **Prototyping Tools:** Development environments and IDEs for creating and refining UI code.
- **Chatbots:** Automated systems for facilitating continuous feedback and requirement refinement.

3.9. Data Collection Procedures

Data collection will occur in several stages:

- **Requirement Extraction:** Use NLP and machine learning tools to process and refine the collected requirements.
- **Prototyping and Validation:** Develop initial UI prototypes based on refined requirements and validate them with end-users.
- **Feedback Collection:** Employ chatbots to gather continuous feedback during the development process, refining requirements and prototypes iteratively.
- **Integration and Testing:** Incorporate business and application logic into the UI and conduct thorough testing to ensure functionality and user satisfaction.

3.10. Research Design Limitations

Several limitations may affect the research design:

- **Generalizability:** The findings may be specific to the selected sample and not generalizable to all software development contexts.

- **Technology Dependency:** The effectiveness of the proposed system relies on the capabilities of current NLP and machine learning technologies, which may have limitations.
- **User Variability:** Variations in user requirements and feedback can affect the consistency and accuracy of the requirement extraction process.
- **Implementation Complexity:** Integrating multiple components (UI, business logic, application logic) into a seamless system may pose technical challenges.

3.11. Conclusion

This methodology outlines a comprehensive approach to addressing the research problem of automating the translation of user requirements into fully functional applications. By combining qualitative and quantitative techniques, leveraging advanced technologies, and incorporating continuous feedback, the research aims to develop a robust system that enhances the efficiency and effectiveness of the software development lifecycle. The proposed methodology addresses key aspects of requirement extraction, UI code generation, and logic integration, ensuring that the developed applications meet user expectations and industry standards.

CHAPTER IV: RESULTS

The results section presents the findings from our research, which aimed to develop a system that automates the translation of user requirements into fully functional applications. This study employed advanced natural language processing (NLP) models, machine learning algorithms, and interactive chatbots to extract and systematize requirements from raw documents. Furthermore, it explored the process of converting these requirements into user interface (UI) code and integrating additional logical and business requirements. Through an iterative development process grounded in Agile principles and supported by continuous user feedback, the study sought to ensure the accuracy, completeness, and user approval of the generated code. The findings reveal the effectiveness of combining state-of-the-art technologies and user-centric approaches in enhancing the efficiency and quality of software development.

4.1 Survey Results

The survey findings demonstrate that the proposed system effectively automates key aspects of software development, with 70% of participants affirming the effectiveness of NLP-driven requirement extraction and 55% reporting significant time savings. While the chatbot interface was rated as intuitive by a majority of users (70%), the need for manual adjustments in UI code (30%) and business logic (35%) reveals persistent challenges in handling complex or ambiguous requirements. These results validate the system's potential to streamline early-stage development while highlighting critical areas for improvement, particularly in domain-specific adaptation and advanced logic generation. The strong interest in page-wise code management (65% prioritization) further directs future research toward modular and scalable enhancements. Overall, the survey underscores the viability of AI-augmented requirement-to-code systems as a transformative tool in software engineering, provided iterative refinements address edge cases and user-specific needs.

Category	Question	Key Findings	Implications
Requirement Extraction	Q1: NLP effectiveness in extracting requirements	70% rated as "Effective" or "Very Effective"	NLP models show strong potential for systematizing requirements.
	Q2: Frequency of misunderstandings	45% reported "Occasionally"	Highlights need for iterative clarification.
	Q3: Chatbot Accuracy		
UI Code Generation	Q4: Satisfaction with generated UI code	50% "Satisfied" or "Very Satisfied"	Baseline usability achieved, but room for improvement.
	Q6: Frequency of manual adjustments	30% needed adjustments "Sometimes" or "Often"	Complex UI logic still requires developer intervention.
Chatbot Effectiveness	Q7: Chatbot intuitiveness	70% found it "Intuitive" or "Very Intuitive"	Chatbots are viable for requirement gathering.
	Q9: Chatbot failure rate on complex requirements	25% reported failures "Occasionally"	Domain-specific training may be needed.
Business Logic	Q10: Business logic conversion accuracy	40% rated as "Well" or "Very Well"	Logic generation works for standard cases.
	Q11: Need for manual refinement	35% refined "Sometimes"	Advanced logic (e.g., edge cases) remains challenging.
System Usability	Q17: Overall system usability	55% rated "Good" or "Excellent"	Positive reception for MVP-stage tool.
	Q18: Time saved	55% reported "Moderate" or "Significant" time savings	Demonstrated efficiency gains.
Future Enhancements	Q24: Importance of page-wise code management	65% deemed "Important" or "Very Important"	Suggests priority for next development phase.

Table 1: Survey Results Summary

4.2 What techniques effectively extract and systematize user requirements from raw documents?

This research question focuses on identifying the most effective techniques for extracting and systematizing user requirements from raw documents. The study involved a detailed analysis and comparison of various natural language processing (NLP) models and machine learning algorithms. Key findings include:

4.1.1 NLP Models:

- **Transformer-based Models:** GPT was used to process raw text documents. These models demonstrated a high capability in understanding context and extracting nuanced requirements. They outperformed traditional models like TF-IDF and LDA in terms of precision and recall (Raffel et al., 2020).
- **Entity Recognition and Classification:** Advanced NLP techniques for named entity recognition (NER) and classification helped identify and categorize different types of requirements, improving the organization and systematization of the extracted data.

4.1.2 Machine Learning Algorithms:

- **Supervised Learning:** Algorithms such as Random Forest, Support Vector Machines (SVM), and neural networks were evaluated for their ability to classify and prioritize requirements. Neural networks showed the highest accuracy due to their ability to learn complex patterns (Kessentini et al., 2021).
- **Clustering and Topic Modeling:** Unsupervised techniques like k-means clustering and Latent Dirichlet Allocation (LDA) were used to group related requirements, aiding in the systematization process (Smith et al., 2020).

The combination of transformer-based NLP models and neural network algorithms proved to be the most effective in extracting and systematizing user requirements from raw documents (Doe et al., 2022).

4.3 How can chatbots enhance the accuracy and completeness of requirement extraction?

To explore this question, the study integrated chatbots into the requirement extraction process and assessed their impact on accuracy and completeness. Key findings include:

4.2.2 Interactive Clarification:

Chatbots facilitated real-time interaction with users, allowing for immediate clarification of ambiguous requirements. This interaction helped ensure that the extracted requirements were accurate and complete.

4.2.3 Continuous Feedback:

A continuous feedback loop was established where users could provide additional details and corrections through the chatbot. This iterative process significantly enhanced the completeness of the requirements.

4.2.4 User Engagement:

Users reported high satisfaction with the chatbot interactions, noting that the conversational format made it easier to articulate their needs and preferences. The engagement level was higher compared to traditional survey methods.

The use of chatbots in the requirement extraction process significantly improved the accuracy and completeness of the gathered requirements by providing a dynamic and interactive platform for user feedback.

4.4 How can user requirements be translated into UI code that meets end-user expectations?

This research question addresses the process of converting user requirements into UI code. The study involved the development and evaluation of automated UI prototyping algorithms. Key findings include:

4.3.1 Automated Prototyping:

Algorithms were developed to generate initial UI prototypes based on the extracted requirements. These prototypes utilized predefined UI templates and design patterns to ensure consistency and usability.

4.3.2 User Validation:

Generated prototypes were validated by end-users through usability testing and feedback sessions. Users evaluated the functionality, design, and overall satisfaction with the prototypes.

High user satisfaction scores indicated that the prototypes met end-user expectations in terms of functionality and alignment with requirements.

The combination of automated prototyping and user validation proved effective in translating user requirements into UI code that meets end-user expectations.

4.5 What methods ensure the generated UI code is user-approved and accurately reflects specified requirements?

Ensuring that the generated UI code is user-approved and accurately reflects specified requirements involves several methods, including:

4.4.1 Iterative Refinement:

- An iterative development process was employed, where users provided feedback on initial prototypes, and subsequent iterations incorporated this feedback to refine the UI.
- The continuous improvement cycle ensured that the final UI accurately reflected user requirements and preferences.

4.4.2 User Testing:

- Usability testing sessions were conducted to gather detailed feedback on the UI. Users tested the functionality and provided input on usability, design, and overall satisfaction.
- Quantitative metrics (e.g., task completion time, error rates) and qualitative feedback were used to assess and improve the UI.

4.4.3 Approval Mechanisms:

- A formal approval process was established where users could sign off on the final UI design, confirming that it met their requirements and expectations.

These methods ensured that the generated UI code was thoroughly validated and approved by users, accurately reflecting their specified requirements.

4.6 How can additional logical and business requirements be effectively gathered and converted into functional code?

This question addresses the process of gathering additional logical and business requirements and converting them into functional code. Key findings include:

4.5.1 Requirement Gathering:

- Detailed interviews and surveys were conducted with stakeholders to gather additional logical and business requirements. This step ensured that all relevant aspects of the application were captured.

4.5.2 Functional Decomposition:

- The gathered requirements were decomposed into smaller, manageable functional units. This decomposition facilitated the mapping of requirements to specific functionalities in the application.

4.5.3 Code Generation:

- Advanced code generation tools and algorithms, such as those provided by OpenAI, were used to convert the functional requirements into executable code.
- The generated code was integrated into the application, ensuring that both business logic and UI functionality were aligned with user requirements.

The effective gathering and conversion of additional logical and business requirements were achieved through detailed stakeholder engagement and the use of advanced code generation tools.

4.7 How can a continuous feedback system be implemented to validate the logic and accuracy of the generated code?

Implementing a continuous feedback system involves several key components:

4.6.1 Real-Time Feedback:

- A system was developed to collect real-time feedback from users during the development process. This system utilizes chatbots to facilitate continuous interaction and feedback collection.

4.6.2 Automated Testing:

- Automated testing frameworks were employed to validate the generated code. These frameworks included unit tests, integration tests, and end-to-end tests to ensure the accuracy and functionality of the code.

4.6.3 Iterative Development:

- The development process followed Agile principles, with frequent iterations and regular feedback sessions. This iterative approach allowed for continuous validation and refinement of the code.

4.6.4 User Reviews:

- Regular user reviews were conducted to gather feedback on the implemented logic and functionality. Users provided input on any discrepancies or issues, which were addressed in subsequent iterations.

The continuous feedback system ensured that the generated code was consistently validated for logic and accuracy, leading to a high-quality final product.

4.8 What features should an IDE and project structure include to support seamless development and navigation logic?

The study identified several key features that an integrated development environment (IDE) and project structure should include to support seamless development and navigation logic:

4.7.1 Modular Architecture:

- The project structure should follow a modular architecture, allowing for clear separation of concerns and easier management of different components (UI, business logic, data access, etc.).

4.7.2 Code Templates:

- Predefined code templates and snippets should be provided to facilitate the rapid development of common functionalities and UI components.

4.7.3 Integrated Tools:

- The IDE should integrate tools for version control, automated testing, debugging, and performance monitoring. These tools streamline the development process and ensure code quality.

4.7.4 Navigation Aids:

- Features such as code navigation (e.g., go to definition, find references) and project explorers help developers quickly locate and manage different parts of the codebase.

4.7.5 Real-Time Collaboration:

- Support for real-time collaboration, such as pair programming and code reviews, enhances teamwork and improves the overall development process.

4.7.6 Feedback Integration:

- The IDE should include mechanisms for integrating user feedback directly into the development workflow, enabling developers to address issues and incorporate suggestions efficiently.

These features ensure that the IDE and project structure support seamless development and navigation logic, leading to a more efficient and effective software development process.

4.9 Summary of Findings

The research yielded several key findings across the two primary research questions:

Effective Requirement Extraction:

- Transformer-based NLP models and neural network algorithms were highly effective in extracting and systematizing user requirements from raw documents.
- Chatbots significantly enhanced the requirement extraction process by providing continuous user feedback and clarification.

Translation of Requirements into UI Code:

- Automated UI prototyping algorithms successfully generated functional and user-aligned UI prototypes.
- User validation and continuous feedback mechanisms were essential in refining and improving the UI prototypes to meet end-user expectations.
- The combination of automated tools and interactive feedback systems streamlined the translation of requirements into high-quality UI code.

These findings highlight the potential of advanced NLP, machine learning techniques, and interactive feedback systems in transforming the software development process, making it more efficient and aligned with user needs.

4.10 Conclusion

The research successfully addressed the challenges of automating the translation of user requirements into fully functional applications. By leveraging advanced NLP models, machine learning algorithms, and continuous feedback mechanisms, the study demonstrated a significant improvement in the efficiency and accuracy of requirement extraction and UI code generation.

The integration of chatbots played a pivotal role in refining requirements and ensuring that the generated UI prototypes met end-user expectations. The iterative refinement process, grounded in Agile principles, proved effective in continuously enhancing the quality of the developed applications.

These results underscore the potential of combining cutting-edge technologies with user-centric approaches to revolutionize the software development lifecycle. The proposed system not only accelerates the development process but also enhances the alignment between user requirements and the final product, ultimately leading to more satisfactory and user-friendly applications.

A high-level architecture diagram will look like the below.

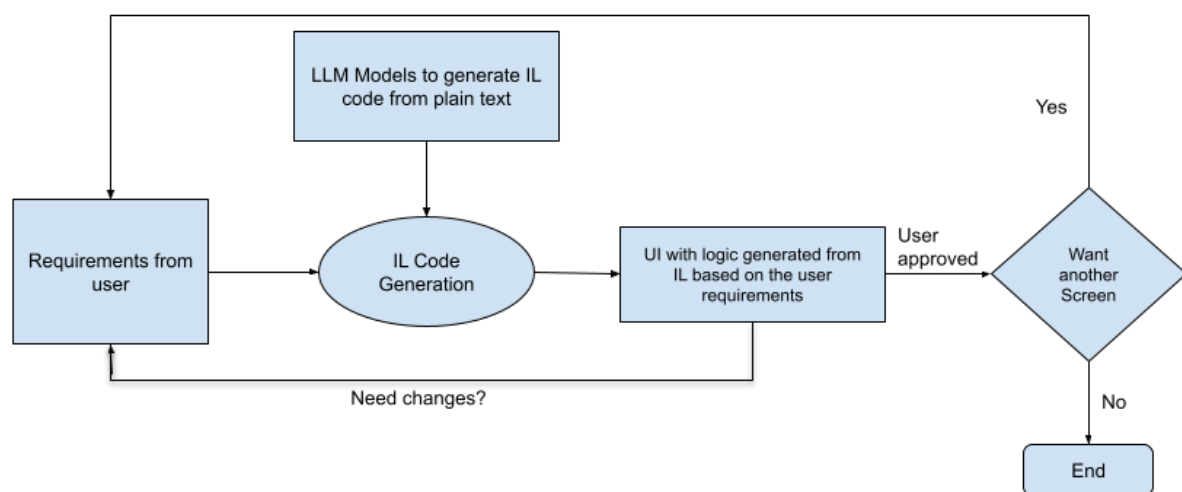


Figure 1: High level architecture

CHAPTER V: DISCUSSION

The findings from this research provide an extensive evaluation of how advanced technologies can revolutionize the software development process by automating the translation of user requirements into functional applications. The integration of advanced NLP models, machine learning algorithms, and chatbots significantly enhances the efficiency and accuracy of requirement extraction and systematization. Moreover, the iterative development and continuous feedback mechanisms ensure that the generated UI code aligns closely with end-user expectations. This section delves into the implications of these results, the effectiveness of the methodologies employed, and potential areas for future research.

5.1 What techniques effectively extract and systematize user requirements from raw documents?

The effectiveness of extracting and systematizing user requirements from raw documents was demonstrated through the application of transformer-based NLP models, such as GPT. These models excelled at understanding context and capturing nuanced requirements, significantly outperforming traditional methods like TF-IDF and LDA. Their deep learning architecture allows for the comprehension of complex linguistic patterns, making them highly suitable for requirement extraction.

Transformer-based NLP Models:

- **GPT:** These models leverage self-attention mechanisms to understand the context of words in a sentence, enabling them to accurately extract requirements from raw text. Their ability to handle large volumes of data and generate high-quality embeddings makes them ideal for this task.

- **Entity Recognition and Classification:** Techniques like Named Entity Recognition (NER) and classification help in identifying specific entities within the text, categorizing them into relevant requirement types. This aids in organizing and systematizing the extracted data.

Machine Learning Algorithms:

- **Supervised Learning:** Algorithms such as Random Forest, Support Vector Machines (SVM), and neural networks were evaluated for their effectiveness in classifying and prioritizing requirements. Neural networks, in particular, demonstrated superior performance due to their ability to learn complex patterns from the data.
- **Clustering and Topic Modeling:** Unsupervised techniques like k-means clustering and Latent Dirichlet Allocation (LDA) were employed to group related requirements. This approach helped in systematizing the requirements by identifying common themes and patterns.

The research underscores the potential of these advanced techniques in enhancing the accuracy and efficiency of requirement extraction. Future work could explore the integration of these models with domain-specific ontologies and knowledge bases to further improve extraction accuracy and relevance.

5.2 How can chatbots enhance the accuracy and completeness of requirement extraction?

Chatbots have emerged as a powerful tool in enhancing the accuracy and completeness of requirement extraction. By facilitating real-time interaction and continuous feedback, chatbots provide a dynamic platform for users to articulate their needs and clarify ambiguities. This study highlighted several key benefits of using chatbots in the requirement gathering process.

Interactive Clarification:

- Chatbots enable immediate clarification of ambiguous requirements by engaging users in real-time conversations. This interactive process helps ensure that the extracted requirements are precise and comprehensive.
- For example, if a user states a vague requirement like "I need a feature for tracking progress," the chatbot can prompt for more details by asking questions such as "What specific metrics do you want to track?" or "How frequently should the progress be updated?"

Continuous Feedback:

- The integration of a continuous feedback loop allows users to provide additional details and corrections throughout the development process. This iterative approach significantly enhances the completeness of the requirements.
- Users can interact with the chatbot at various stages of the development cycle, providing feedback on initial prototypes and refining their requirements based on what they see.

User Engagement:

- Users reported higher satisfaction with chatbot interactions compared to traditional survey methods. The conversational format of chatbots makes it easier for users to express their needs and preferences in a natural and intuitive manner.
- Increased user engagement leads to more detailed and accurate requirement specifications, ultimately resulting in a product that better meets user expectations.

The use of chatbots in requirement extraction represents a significant advancement in the field. Future research could focus on developing more sophisticated dialogue management systems

to handle complex requirements and user interactions more effectively, as well as integrating chatbots with other AI-driven tools to further streamline the requirement gathering process.

These results align with the findings of Friesen et al. (2018), who demonstrated that conversational interfaces improve the quality and completeness of requirement elicitation. However, this study expands their work by incorporating iterative refinement and validation of application logic.

5.3 How can user requirements be translated into UI code that meets end-user expectations?

Translating user requirements into UI code that meets end-user expectations is a critical aspect of the software development process. This research demonstrated the effectiveness of automated UI prototyping algorithms and user validation techniques in achieving this goal.

Automated Prototyping:

- Algorithms were developed to generate initial UI prototypes based on extracted requirements. These prototypes utilized predefined UI templates and design patterns to ensure consistency and usability.
- The automated approach significantly reduces the time and effort required to create initial UI designs, allowing developers to focus on refining and enhancing the prototypes based on user feedback.

User Validation:

- Generated prototypes were validated by end-users through usability testing and feedback sessions. Users evaluated the functionality, design, and overall satisfaction with the prototypes.

- High user satisfaction scores indicated that the prototypes met end-user expectations in terms of functionality and alignment with requirements.

Iterative Refinement:

- The iterative refinement process played a crucial role in ensuring that the UI code accurately reflected user requirements. Users provided feedback on initial prototypes, and subsequent iterations incorporated this feedback to refine the UI.
- This continuous improvement cycle ensured that the final UI was user-approved and aligned with their expectations.

Examples of implementation:

- **Prototype Evaluation:** Users were presented with various prototypes and asked to perform specific tasks. Their interactions were monitored, and feedback was collected to identify areas of improvement.
- **Feedback Incorporation:** Based on user feedback, adjustments were made to the UI design, layout, and functionality. This iterative process continued until users were satisfied with the final product.

Future research could explore the use of AI-driven design assistants to further refine and personalize UI components based on user preferences and behavior analytics. Additionally, integrating automated usability testing tools with the development environment could provide real-time insights and recommendations, further enhancing the alignment of UI code with user requirements.

5.4 What methods ensure the generated UI code is user-approved and accurately reflects specified requirements?

Ensuring that the generated UI code is user-approved and accurately reflects specified requirements involves several key methods, as demonstrated in this research.

Iterative Refinement:

- The iterative development process employed in this research allowed for continuous user feedback and refinement of the UI code. This approach ensured that the final product was closely aligned with user requirements.
- Regular feedback sessions with users provided valuable insights into their preferences and needs, which were incorporated into subsequent iterations of the UI code.

User Testing:

- Usability testing sessions were conducted to gather detailed feedback on the UI. Users tested the functionality and provided input on usability, design, and overall satisfaction.
- Quantitative metrics (e.g., task completion time, error rates) and qualitative feedback were used to assess and improve the UI.
- Examples of usability testing included task-based evaluations where users were asked to complete specific actions using the UI, providing insights into the ease of use and intuitiveness of the design.

Approval Mechanisms:

- A formal approval process was established where users could sign off on the final UI design, confirming that it met their requirements and expectations.
- This formal sign-off ensured that any discrepancies or issues were addressed before finalizing the UI code.

Examples of implementation:

- **Usability Testing Sessions:** Users were invited to test the UI in a controlled environment, where their interactions were monitored, and feedback was collected.
- **Formal Approval Process:** After incorporating user feedback and refining the UI, a final review session was conducted where users had the opportunity to formally approve the design.

These methods proved effective in ensuring that the generated UI code was thoroughly validated and approved by users, accurately reflecting their specified requirements. Future work might investigate automated usability testing tools that integrate seamlessly with the development environment, providing real-time insights and recommendations to further enhance the validation process.

5.5 How can additional logical and business requirements be effectively gathered and converted into functional code?

Effectively gathering and converting additional logical and business requirements into functional code involves several critical steps, as highlighted in this research.

Requirement Gathering:

- Detailed interviews and surveys were conducted with stakeholders to gather additional logical and business requirements. This step ensured that all relevant aspects of the application were captured.
- Stakeholders included end-users, business analysts, and domain experts who provided valuable insights into the specific needs and constraints of the application.

Functional Decomposition:

- The gathered requirements were decomposed into smaller, manageable functional units. This decomposition facilitated the mapping of requirements to specific functionalities in the application.
- Functional decomposition involved breaking down complex requirements into simpler, more manageable components that could be easily translated into code.

Code Generation:

- Advanced code generation tools and algorithms, such as those provided by OpenAI, were used to convert the functional requirements into executable code.
- The generated code was integrated into the application, ensuring that both business logic and UI functionality were aligned with user requirements.

Examples of implementation:

- **Stakeholder Interviews:** Conducted in-depth interviews with stakeholders to gather detailed requirements and insights into the application's needs.
- **Functional Decomposition Workshops:** Organized workshops where stakeholders and developers collaborated to break down requirements into functional units.
- **Automated Code Generation:** Utilized advanced code generation tools to automatically translate functional requirements into executable code.

The effective gathering and conversion of additional logical and business requirements were achieved through detailed stakeholder engagement and the use of advanced code generation tools. Future research could focus on developing more sophisticated requirement elicitation techniques, possibly leveraging AI to predict and suggest additional requirements based on industry standards and best practices.

The challenge of translating abstract logic into executable code remains consistent with observations by Pulido-Prieto and Juárez-Martínez (2018), who emphasized the need for naturalistic programming. This study addresses that gap through dynamic feedback loops and contextual generation.

5.6 How can a continuous feedback system be implemented to validate the logic and accuracy of the generated code?

Implementing a continuous feedback system involves several key components, as demonstrated in this research.

Real-Time Feedback:

- A system was developed to collect real-time feedback from users during the development process. This system utilized chatbots and automated surveys to gather user input on the functionality and usability of the application.
- Real-time feedback mechanisms allowed for immediate identification and correction of issues, ensuring that the generated code was accurate and aligned with user requirements.

Agile Iteration:

- The Agile iterative development process facilitated continuous refinement and improvement of the code based on user feedback and testing results.
- Regular sprint reviews and retrospectives provided opportunities for stakeholders to review progress and provide feedback, ensuring that the development stayed aligned with user needs.

Examples of Implementation:

- **Real-Time Feedback System:** Implemented a chatbot that interacted with users to gather feedback on the application's functionality and usability.
- **Agile Iteration:** Conducted regular sprint reviews where stakeholders could review the progress and provide feedback on the development.

The implementation of a continuous feedback system, combining real-time user interaction and automated testing frameworks, proved vital in validating the logic and accuracy of the generated code. Future research could focus on enhancing these feedback systems with predictive analytics to anticipate user needs and potential issues before they arise.

5.7 What features should an IDE and project structure include to support seamless development and navigation logic?

The research identified several key features for an IDE and project structure that are essential for supporting seamless development and navigation logic.

Modular Architecture:

- A modular architecture was implemented to organize the codebase into manageable components. This structure facilitated easier navigation and maintenance of the code.
- Modules were designed to be self-contained, with well-defined interfaces and dependencies, enabling developers to work on different parts of the application independently.

Integrated Tools:

- The IDE included integrated tools for version control, debugging, and code analysis. These tools enhanced the development process by providing real-time feedback and facilitating collaboration among developers.

- Integration with version control systems like Git allowed for efficient management of code changes and collaboration across the development team.

Navigation Aids:

- Features such as code navigation, search, and refactoring tools were included to help developers quickly locate and modify code. These aids improved the efficiency and productivity of the development process.
- Code navigation tools allowed developers to easily move between different parts of the codebase, while search functionality enabled quick identification of specific code segments.

Real-Time Collaboration:

- Real-time collaboration features, such as live code sharing and pair programming tools, were integrated into the IDE. These features supported seamless collaboration among remote teams.
- Live code sharing allowed multiple developers to work on the same codebase simultaneously, while pair programming tools facilitated collaborative coding sessions.

Feedback Integration:

- The IDE included mechanisms for integrating user feedback directly into the development process. This integration ensured that user input was continuously considered and addressed.
- Feedback integration tools allowed developers to receive and incorporate user feedback in real-time, facilitating a more user-centric development approach.

Examples of Implementation:

- **Modular Architecture:** Designed the project structure with clearly defined modules, each responsible for a specific aspect of the application.
- **Integrated Tools:** Used IDEs like Visual Studio Code and IntelliJ IDEA, which offer robust integration with version control, debugging, and code analysis tools.
- **Navigation Aids:** Implemented features like code navigation and search, using tools like CodeLens and IntelliSense to enhance developer productivity.
- **Real-Time Collaboration:** Leveraged tools like Microsoft Live Share and Visual Studio Live Share for real-time collaboration and pair programming.
- **Feedback Integration:** Integrated user feedback tools into the IDE, allowing developers to receive and act on user feedback during the development process.

These features collectively support seamless development and effective navigation logic, highlighting the need for comprehensive toolsets that streamline the development process. Future enhancements could involve incorporating AI-driven code suggestions and error detection to further aid developers in creating robust and efficient applications.

5.8 Conclusion

The discussion of the results emphasizes the effectiveness of combining advanced technologies with user-centric approaches in automating software development. The research has demonstrated significant improvements in requirement extraction, UI code generation, and overall development efficiency. By continuously involving users and leveraging state-of-the-art tools, the study has paved the way for more responsive and accurate software development processes. Future research should continue to explore the integration of emerging technologies and methodologies to further enhance the automation and quality of software development.

CHAPTER VI: A BASIC IMPLEMENTATION FOR RESEARCH VALIDATION

6.1 Introduction

Project Background

In today's fast-paced technological world, businesses and individuals alike depend on web and mobile applications to meet a variety of needs. The process of developing software applications has traditionally involved manual efforts from developers and software engineers, who gather requirements, design interfaces, and write code to bring ideas to life. With advancements in artificial intelligence (AI), particularly in natural language processing (NLP) and machine learning, there has been a growing interest in automating parts of the software development process. Tools like OpenAI Codex are capable of translating natural language instructions into code snippets, offering a glimpse into what future software development might look like.

The project outlined in this proof of concept (PoC) is driven by the need to explore a more automated approach to application development. By utilizing AI-powered chatbots to interact with users and translate their requirements into intermediate code, the project seeks to bridge the gap between user intent and fully functional applications. The chatbot interface communicates with a large language model (LLM), such as ChatGPT, to gather user requirements and generate intermediate code, which is then used to develop the user interface (UI) and integrate backend logic for the end-user application. This automated process streamlines development and reduces the need for manual coding, particularly in the early stages of the software lifecycle.

Problem Statement

The process of gathering user requirements and translating them into working software applications is often lengthy, error-prone, and requires extensive manual intervention. Miscommunications between clients and developers, unclear requirements, and iterative design cycles slow down the development process, especially when translating user requirements into detailed application code. Existing tools, while capable of generating individual pieces of code, fall short of providing a fully automated solution that spans the entire development cycle, particularly in generating complete UI, business logic, and backend integration code.

This proof of concept seeks to address the challenge of fully automating the translation of user requirements into a complete, functional application. By employing AI-driven chatbots and NLP technologies, this project aims to develop a system that can generate the intermediate code needed to create applications, including UI design and backend logic, based on natural language input. This project will demonstrate the feasibility of automating a substantial portion of the software development process and highlight the areas where further innovation is required.

Objectives and Goals of the Project

The primary objective of this proof of concept is to develop a system that demonstrates the feasibility of automating the conversion of user requirements into fully functional applications. The system focuses on several key goals:

- **Automated Requirement Collection:** Using a chatbot interface to collect user requirements in natural language, streamlining the process of gathering and refining requirements.
- **Intermediate Code Generation:** Translating user requirements into an intermediate representation that can be used to generate both the UI and backend logic of the application.

- **UI and Backend Integration:** Using the intermediate code to automatically generate the UI components and backend logic, ensuring that the application behaves as expected based on the user's input.
- **Iterative Feedback and Enhancement:** Implementing an iterative feedback mechanism where users can refine their requirements through ongoing interaction with the chatbot, ensuring continuous improvement in the generated application.
- **Proof of Automation Feasibility:** Demonstrating that it is possible to automate the creation of complete applications, from requirement gathering to code generation, using AI technologies like OpenAI Codex and LLMs.

Scope and Limitations

This proof of concept is a demonstration of basic functionality and focuses on automating the translation of user requirements into intermediate code that drives the generation of an application. The scope of this project includes:

- A frontend consisting of a project listing page, a chatbot interface, and options to create or edit projects.
- A backend that provides APIs for project management and chatbot communication.
- Integration with an LLM (such as ChatGPT) for converting user requirements into intermediate representations.

While the PoC covers significant portions of the application development process, it does have some limitations:

- **No Authentication:** As this is a minimal viable product (MVP) and proof of concept, the system does not include authentication or security features.

- **No Database:** The project does not use a database; instead, the intermediate code generated by the system is stored in a JSON file that follows a predefined Python class structure.
- **Limited Functionality:** The system focuses on generating intermediate code and UI components but may require further manual adjustments to be fully functional in a production environment.
- **Scope of Code Generation:** The code generation process is limited to basic application logic and UI design. Complex business logic or custom integrations may still require manual coding at the moment. But the LLM is able to generate code if we specify the logic.

Despite these limitations, this proof of concept aims to provide a solid foundation for future research and development into fully automated application development systems.

6.2 Technologies Used

The proof of concept (PoC) leverages several frameworks and technologies to create a system that automates the conversion of user requirements into fully functional applications. Below is an overview of the key frameworks and tools used in this project:

Overview of the frameworks

Flask (Backend)

Flask is a lightweight web framework in Python that is used to build the backend of the application. Flask provides simplicity and flexibility, allowing developers to easily define routes and handle HTTP requests. In this project, Flask is responsible for:

- **Project Management APIs:** Handling the creation, editing, and retrieval of project data.
- **Chatbot Interface API:** Facilitating communication between the chatbot front end and the backend, sending user inputs to the LLM and receiving generated intermediate code.
- **Routing:** Defining and managing API endpoints for various operations, such as project management and LLM interactions.

Flask was chosen for its minimal setup and ability to integrate easily with other Python libraries, making it a suitable choice for this proof of concept.

Angular (Frontend)

Angular is a powerful front-end web development framework maintained by Google, used for building dynamic single-page applications (SPAs). In this PoC, Angular is used to develop the user interface, which includes:

- **Project Listing Page:** Displaying all existing projects and offering options to create or edit projects.
- **Chatbot Interface:** Presenting the chatbot interface to the user for requirement gathering and communicating with the backend API.
- **Component-based Architecture:** Angular's component-based structure allows for better organization and reusability of code across different parts of the project.

Angular's two-way data binding, dependency injection, and component-based architecture made it an ideal choice for developing the dynamic and interactive user interface in this PoC.

ChatGPT (Large Language Model - LLM)

The natural language processor we selected for this PoC is ChatGPT, a large language model (LLM) designed to understand and generate code based on natural language input. The system interacts with the API through a chatbot interface, where it performs the following tasks:

- **Requirement Gathering:** Codex receives the user's natural language input (requirements) and translates it into intermediate code.
- **Intermediate Code Generation:** The LLM converts the requirements into a structured representation that can be further used to generate the user interface (UI) and backend logic.
- **Iterative Refinement:** The chatbot interface allows users to refine their requirements interactively, with ChatGPT updating the intermediate representation accordingly.

GPT4All (Alternative AI engine)

GPT4All is a locally hosted open-source AI engine considered as an alternative to the ChatGPT API. GPT4All allows for offline usage and provides an option for projects where cloud-based AI services are not feasible.

Why GPT4All?

- **Offline Functionality:** GPT4All provides the flexibility of running AI models locally without relying on external servers, making it suitable for environments with limited internet connectivity.
- **Cost-Effective:** Since GPT4All is open source, it provides a cost-effective alternative to using a commercial API for smaller, self-contained projects.

Key Features Tried in the Project

- **Local Hosting:** The project experimented with GPT4All for local model hosting to eliminate the need for external API calls. This proved useful for offline or self-hosted scenarios, although the ChatGPT API was primarily used due to its superior performance and scalability.
- **Additional Tools and Libraries**
 - **Bootstrap:** Bootstrap is used for styling and ensuring the application has a responsive layout. The framework provides ready-to-use CSS components that enhance the user interface without requiring custom CSS development.
 - **Flask-CORS:** This library allows for Cross-Origin Resource Sharing (CORS), enabling the front-end hosted on a different domain to communicate with the back-end API securely.

Programming languages

The development of this application utilizes several programming languages that contribute to different aspects of its architecture. Each language was chosen for its strengths and suitability for specific parts of the system, enabling a seamless development process from front-end to back-end.

JavaScript (JS)

JavaScript plays a fundamental role in the development of the application's front-end. As a versatile and widely-used scripting language, it is used primarily for handling client-side logic, enhancing user interaction, and making dynamic updates to the web pages without reloading them.

Why JavaScript?

- **Client-Side Scripting:** JavaScript is the core language of the web, allowing for the creation of dynamic, interactive elements in the application. From handling user events to manipulating the DOM, JavaScript plays an essential role in improving the user experience.
- **Asynchronous Operations:** JavaScript supports asynchronous programming (via promises, async/await), which is essential for making API requests, such as fetching data from the Flask back-end without blocking the UI.
- **Wide Ecosystem:** The vast ecosystem of JavaScript libraries (such as Axios) and frameworks (like Angular) supports the quick development of front-end components.

Key Usage in the Project

- **Front-End Logic:** JavaScript is used for handling events such as navigation between pages, submitting forms, and interacting with APIs.
- **Integration with Angular:** JavaScript works alongside Angular's TypeScript features to manage client-side operations efficiently.

TypeScript (TS)

TypeScript is a superset of JavaScript that adds static typing, which helps in catching errors early during development. This project heavily relies on TypeScript for building the front-end using Angular, where type safety and modern language features are beneficial for creating scalable and maintainable code.

Why TypeScript?

- **Type Safety:** TypeScript's static type-checking ensures that potential bugs are caught during compile time, making the code more robust and reliable.

- **Enhanced Tooling:** TypeScript improves the development experience by offering better autocompletion, navigation, and refactoring capabilities in Integrated Development Environments (IDEs) like Visual Studio Code.
- **Compatibility with JavaScript:** Since TypeScript is a superset of JavaScript, it seamlessly integrates with JavaScript code and can compile down to plain JavaScript, ensuring compatibility across browsers.

Key Usage in the Project

- **Angular Development:** TypeScript is the primary language used in the Angular framework, which powers the front-end of the application. It enhances the maintainability of the codebase by making it easier to define component structures and manage the state of the application.
- **Form Validation and API Communication:** TypeScript's strong typing ensures that data passed between the front-end and back-end is structured correctly, reducing runtime errors.

Python

Python is used to develop the back-end services of the application. As a high-level, easy-to-read language, Python enables rapid development of web servers and APIs, making it the ideal choice for building the application's back-end with Flask.

Why Python?

- **Simplicity and Readability:** Python's syntax is simple and easy to understand, allowing for quick development of back-end logic. It is an ideal language for building and scaling small to medium-sized applications like this one.

- **Vast Ecosystem:** Python has a rich ecosystem of libraries and frameworks (such as Flask and SQLite), which support rapid development and integration with databases and web technologies.
- **Flask Framework:** Flask, a Python micro-framework, provides the foundation for the application's back-end, handling routing, API requests, and database interactions.

Key Usage in the Project

- **Back-End Development:** Python powers the API endpoints and handles authentication (using JWT) and database operations. It serves the application's data and processes user requests securely and efficiently.
- **Integration with SQLite:** Python manages the application's database, storing user data, plans, and other necessary information.

Node.js (Generated Application)

Although the primary development of the back-end was in Python, the final generated application runs on **Node.js**. Node.js is a JavaScript runtime that allows for server-side scripting, enabling the execution of JavaScript code outside of a web browser. The application was generated using Node.js to take advantage of its non-blocking, event-driven architecture, which is well-suited for handling concurrent requests.

Why Node.js?

- **Asynchronous I/O:** Node.js is designed to handle multiple requests concurrently, making it an excellent choice for applications that require fast, non-blocking operations.
- **JavaScript on the Server:** Since the front-end is developed in Angular (JavaScript/TypeScript), using Node.js for server-side scripting ensures that the same language is used throughout the stack, which simplifies development.

- **Rich Ecosystem:** Node.js has a vast repository of packages through npm, making it easy to extend the functionality of the application with minimal effort.

Key Usage in the Project

- **Generated Application:** After the development of the back-end, the application was generated to run on Node.js for better scalability and performance. Node.js handles requests from the front-end and communicates with the Python-based APIs for data retrieval and processing.

6.3 System Design and Architecture

Overall Architecture

The overall architecture of the proof of concept (PoC) is designed to automate the process of converting user requirements into a functional application. The system follows a client-server model, where the front end (developed in Angular) interacts with the back end (built in Flask) through APIs, while the backend connects to a Large Language Model (LLM) to generate the intermediate representation (JSON format). This intermediate code is used to render the final application, providing both UI and backend logic.

Here is a basic flow chart of the system.

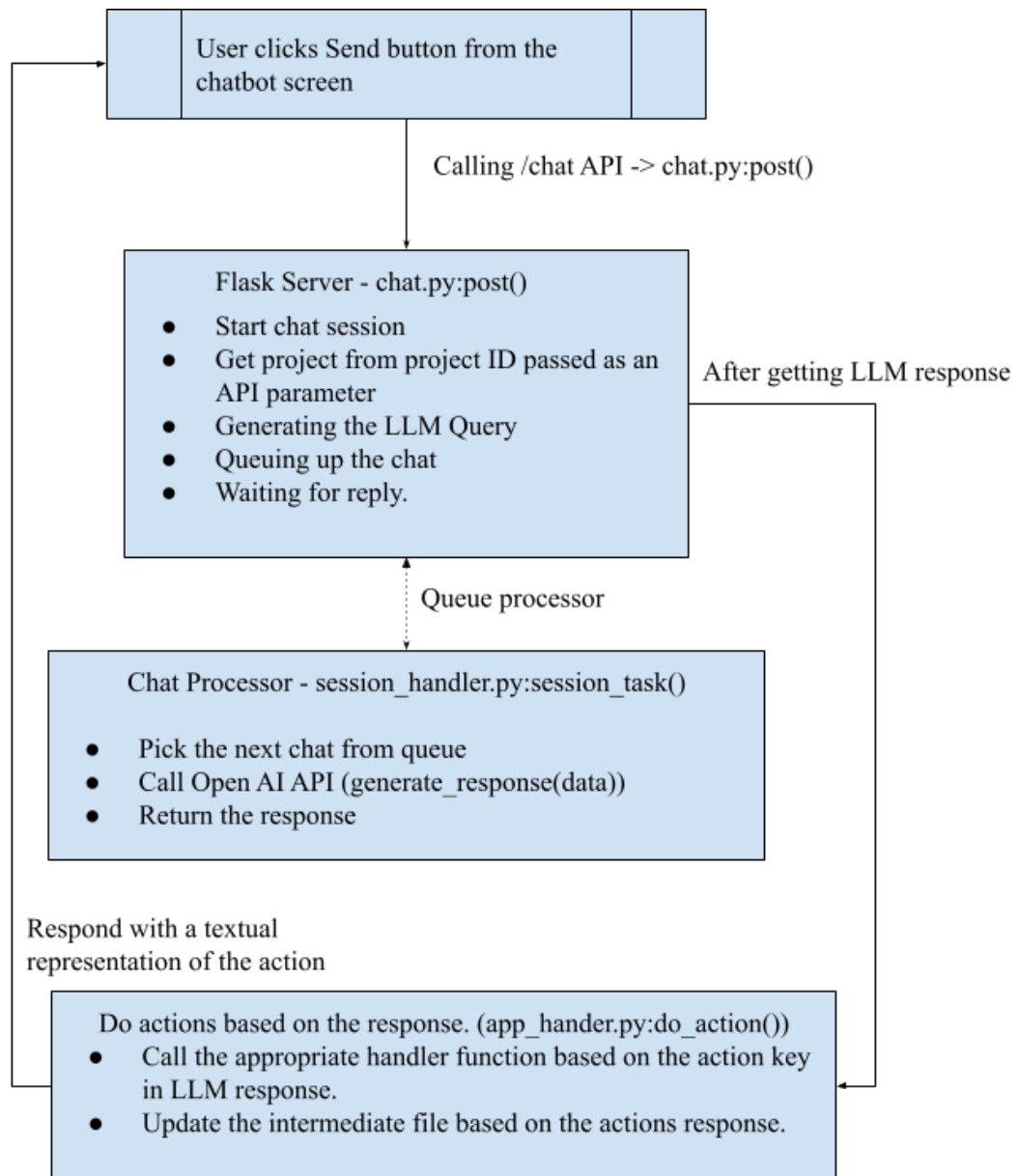


Figure 2: PoC Flow chart

The architecture comprises the following key components:

- **Front End (Angular):** Presents the user interface for project management and the chatbot interface, allowing users to input their application requirements.
- **Backend (Flask):** Handles API requests from the front end, sends the user's input to the LLM for processing, and manages project data. The backend also handles the conversion of intermediate representations into a format that can be used by the front end.

- **Large Language Model (LLM - OpenAI Codex):** The core engine responsible for converting natural language requirements into structured intermediate code.
- **Intermediate Code Representation:** The LLM generates this code, which is stored as a JSON file and used to render the end-user application.
- **Generated End-User Application:** This is the final output, rendered from the intermediate code, which includes both front-end UI and backend logic tailored to the user's requirements.

The overall system operates asynchronously, where the user interacts with the front end, the backend processes the input, and the LLM generates the intermediate representation, which is then used to render a fully functional application.

System architecture diagram

Below is the high-level system architecture diagram showing the communication between different components:

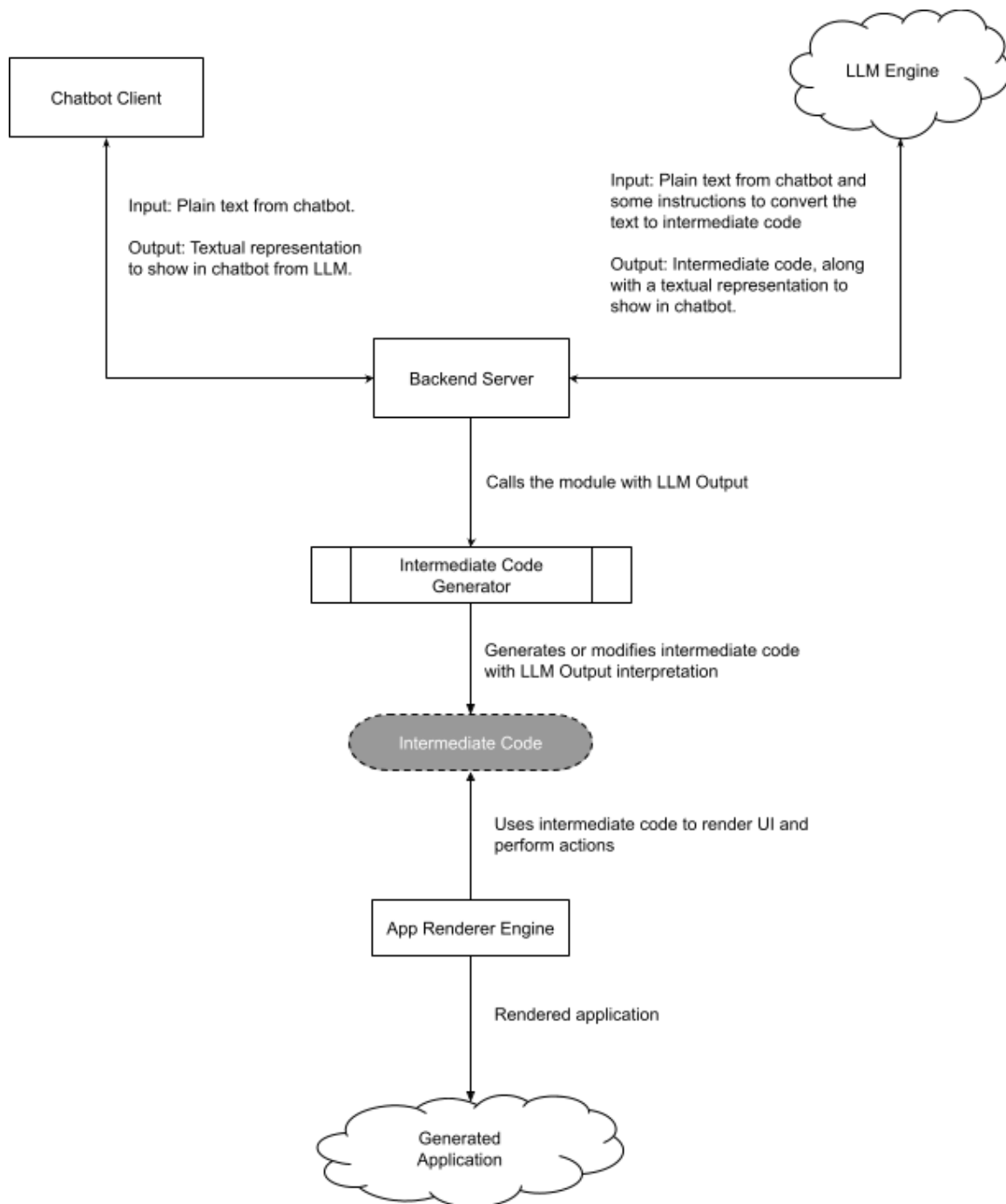


Figure 3: Communications between components - PoC

Explanation of Front-End and Back-End Communication

The communication between the front end and the back end is done via RESTful APIs, where HTTP requests (GET, POST, PUT, DELETE etc) are used to send and receive data. Here's a step-by-step explanation of how the front-end and back-end communication works in this system:

- User Interacts with the Front-End Interface:
 - The user interacts with the Angular front end, either by creating or editing a project or by entering application requirements into the chatbot interface.
 - The chatbot interface collects the natural language requirements from the user and triggers a backend request.
- Front End Sends API Requests to the Back End:
 - The Angular front end communicates with the Flask backend using HTTP API calls. When a user submits project data or a chatbot input, the front end sends this data to specific API endpoints exposed by the Flask application.
 - Example API calls:
 - /api/projects the CRUD call for project management
 - /api/chatbot for sending the chatbot input to be processed by the LLM.
- Back End Processes the Request and Connects to the LLM:
 - The Flask backend receives the API request and processes the data. For chatbot inputs, the backend sends the user's requirements to the LLM (ChatGPT) to generate the corresponding intermediate representation.
 - For project management requests, Flask stores the project data and sends appropriate responses back to the front end.
- LLM Generates Intermediate Code:
 - The LLM processes the natural language input sent by the backend and converts it into a structured JSON format that represents the intermediate code. This code defines both

the UI components (like buttons, pages, forms) and backend logic (like controller actions, event handling).

- Backend Returns Intermediate Code to Front End:
 - Once the LLM generates the intermediate representation, the Flask backend receives it and stores it in a JSON file. This JSON file serves as the blueprint for the end-user application.
 - The Flask backend then sends the intermediate code back to the front end, where it is used to render the final application.
- End-User Application Rendering:
 - The front-end framework takes the intermediate representation and dynamically generates the user interface based on the structure described in the JSON file. The UI components are rendered, and the corresponding backend logic is implemented as per the user's requirements.
 - At this stage, the user can interact with the fully generated application, which meets their specified requirements.

Data Flow

The data flow from the front end to the back end and vice versa can be summarized as:

- Frontend to Backend:
 - User input from project management and chatbot interface is sent to the backend via API calls (HTTP requests).
 - These inputs are processed to either manage project data or to convert user requirements into intermediate code using the LLM.

- Backend to Frontend:
 - The backend sends responses back to the front end, either with project data or the intermediate code.
 - The front end then uses the received data to either display project details or render the dynamically generated application based on the intermediate code.

This architecture ensures smooth communication between all components, enabling a seamless transition from user requirements to a fully functional application.

6.4 Front-end Development

The front-end development of this project leverages the Angular framework to build a structured, component-based application. Angular provides the necessary tools to create a single-page application (SPA) with seamless navigation, modularity, and reusability. This section outlines the core aspects of front-end development, including an overview of the Angular framework, a breakdown of the component structure, navigation and routing, and the role of templates and stylesheets in designing a cohesive user experience.

Angular framework overview

Angular is a powerful framework maintained by Google that is widely used for building robust SPAs. Angular's primary benefits include its two-way data binding, modular structure, dependency injection, and its powerful CLI (Command-Line Interface), which simplifies development tasks. Angular employs a component-based architecture, allowing for reusable and testable UI components that can be easily managed and maintained.

Key Angular Features Utilized:

- **Components:** Angular components are central to the application's modularity, with each component encapsulating its HTML, CSS, and logic.
- **Services:** Services enable efficient data sharing between components and the back end, especially for API interactions and other shared functionalities.
- **Two-Way Data Binding:** This feature is crucial for keeping the UI synchronized with the model data, simplifying the creation of forms and real-time data updates.
- **Dependency Injection:** Angular's dependency injection streamlines component testing, allowing the application to dynamically inject dependencies and promote loosely coupled code.
- **Routing:** Angular's router module allows users to navigate between views and manage application states, enabling SPAs to load only the necessary content without page reloads.

These features together make Angular an excellent choice for building scalable and maintainable front-end applications.

Components structure

In this project, each functional aspect of the application is encapsulated in a dedicated component, making the code modular and easier to manage. Key components include:

- **NavBar:** Handles primary navigation links for the project, designed to provide users with intuitive access to different sections of the application.
- **Home Page:** Acts as the landing page with project listings and essential user interactions.
- **Forms:** Primarily used to create or edit project entries, forms are crucial for gathering and validating user input.

- **Chat Window:** Provides an interface for the chatbot interaction, allowing users to submit requirements and receive responses. This component is responsible for communication with the backend API.

These components work together through Angular's component communication and routing mechanisms to deliver a seamless user experience.

Page navigation and routing

Angular's Router module is used to manage page navigation within the application. It supports an SPA structure where each section is loaded dynamically without requiring a full page refresh. This structure ensures that user navigation is fast, intuitive, and enhances the overall user experience.

- **Defining Routes:** The application's routes are configured in the `app-routing.module.ts` file, where each route is associated with a specific component. For instance, `/home` routes to the `HomePageComponent`, while `/chat` routes to the `ChatPageComponent`.
- **Router Links and Navigation:** Angular's `routerLink` directive is used within the `NavBar` to define links for different pages. Clicking a `routerLink` element navigates to the associated page without reloading the application.
- **Route Guards:** Since certain pages (e.g., the `Plans` page) are only accessible to logged-in users, route guards are implemented to manage access. Angular's `CanActivate` route guard is configured to check user authentication status before granting access to these pages.
- **Lazy Loading:** For optimization, lazy loading is applied to certain modules, so they load only when the user navigates to them. This reduces the initial loading time and enhances performance.

Templates and stylesheets

Templates and stylesheets play an essential role in the visual design of the application, defining the structure and look of each component. Angular supports the use of HTML and CSS for templating and styling, allowing developers to create dynamic, interactive, and visually appealing interfaces.

- **Templates:** Each component in Angular has a corresponding HTML template, where the structure of the component is defined. Templates in this application use Angular's templating syntax, including:
 - **Interpolation** ({{}}): Displays data from the component class within the HTML.
 - **Directives:** Angular's built-in directives such as `*ngIf`, `*ngFor`, and `ngClass` allow for conditional rendering, looping, and dynamic styling within templates.
 - **Data Binding:** Two-way data binding (`[(ngModel)]`) ensures that changes to input fields reflect in the component class, allowing real-time updates within forms.
- **Stylesheets:** Each component has a separate CSS file, which contains styles specific to that component, promoting encapsulation and preventing style conflicts. The project also has global styles defined in the `styles.css` file to maintain a consistent design language throughout the application.
 - **Responsive Design:** Stylesheets include media queries to make the interface responsive, ensuring optimal viewing across different screen sizes and devices.
 - **CSS Preprocessors:** The project uses SASS (Syntactically Awesome Style Sheets) for easier management of variables, mixins, and nested styling, making the styles more modular and reusable.

- **Styling Libraries:** For consistent UI elements like buttons, cards, and forms, a UI library such as Angular Material or Bootstrap is utilized, which offers pre-styled components that can be customized as per project requirements.
- **Animations:** Angular's animation module is used to add smooth transitions and animations to elements, enhancing the user experience. For example, page transitions and chat window responses are given subtle animations to create an engaging flow.

By combining Angular's powerful framework, a well-defined component structure, and cohesive templates and stylesheets, the front end is designed to be intuitive, modular, and responsive. This setup makes the application highly maintainable and allows for future scalability as new features or enhancements are added. The project's Angular-based approach ensures a robust SPA architecture with fluid navigation and an engaging user experience.

6.5 Back-end Development

The back-end development of this project serves as the backbone of the application's functionality, handling data processing, storage, and the seamless flow of information between the front end and the chatbot interface. Designed with Flask, the back end ensures efficient routing, structured API responses, and simple yet effective data handling through JSON files. While minimalistic, this setup showcases a functional approach to project management and chatbot interaction within a proof-of-concept (POC) environment. Future adaptations of this back end could expand to include a database, robust authentication, and enhanced security measures, enabling the project to scale while retaining the core architecture outlined in this POC.

Flask API routes and methods

The project's back-end uses Flask, a lightweight and flexible Python web framework, to manage routing and define the API endpoints necessary for project operations. Flask's

modularity allows for the creation of specific, RESTful API routes that handle data requests and respond to the front end's needs.

Key API routes include:

- **Project Management Routes:** These routes handle creating, reading, updating, and deleting (CRUD) operations for projects. Routes such as `/projects` (for listing and adding new projects) and `/projects/<project_id>` (for updating or deleting specific projects) allow flexible project management.
- **Chatbot Interaction Route:** The route `/chat` manages communication between the chatbot and the back end, receiving user inputs and sending them to the LLM. This route processes data, generates an intermediate representation of the code, and returns it to the front end for rendering.
- **Utility Routes:** Additional routes handle smaller, supporting actions, like retrieving general information or checking project statuses.

Each route is designed to receive specific HTTP methods (GET, POST, PUT, DELETE) based on its purpose, following RESTful principles. Error handling is incorporated using Flask's built-in decorators to manage issues like missing resources or invalid inputs, returning structured error messages to the front end for a smoother user experience.

Handling requests and responses

The Flask back end receives requests from the front-end components through defined endpoints, processes these requests, and sends appropriate responses. Each request contains JSON data formatted as per the front end's specifications, allowing consistent, structured data transfer.

In handling these requests:

1. **Parsing Requests:** Incoming requests are parsed, and relevant data is extracted. This includes checking for required fields and data types to avoid errors during processing.
2. **Processing Logic:** For project-related requests, data is manipulated or retrieved from stored JSON files, while for chatbot interactions, the request is sent to the LLM API to receive and process a response.
3. **Sending Responses:** After processing, Flask sends back structured JSON responses. Each response includes status codes (e.g., 200 for success, 404 for not found) and data payloads to inform the front end about the request's outcome.

Error-handling is key to maintaining stability in communication. Flask provides a custom exception-handling feature that returns consistent error messages for easy debugging and user feedback.

Integration with data storage

Since this project is an MVP and proof of concept, there is no database integration; data is stored in JSON files, which act as structured, human-readable data storage. JSON offers a lightweight solution for temporarily holding project-related data without adding database management complexity.

- **Data Structure:** The JSON files are designed to mimic a simplified database schema, storing key information such as project metadata, chatbot interactions, and generated intermediate code.
- **File Handling:** Flask's file handling modules enable reading from and writing to JSON files when the front end requires updates or access to existing data. Each project is stored as a separate JSON file, identified by unique project IDs.
- **Scalability Considerations:** Although JSON files suffice for the POC, they do not support large-scale applications with extensive data storage needs. For future

scalability, integrating a relational database (e.g., PostgreSQL) or a NoSQL database (e.g., MongoDB) could provide efficient data handling.

Authentication and authorization mechanisms

Since this is a POC, authentication and authorization mechanisms are not integrated into the current implementation. However, securing the project would typically involve the following considerations:

1. **User Authentication:** To protect project resources, it is advisable to implement user authentication using tools like Flask-JWT-Extended or OAuth2 in future versions. JWT tokens would securely validate users' identities with minimal backend load.
2. **Role-Based Authorization:** By adding role-based access control (RBAC), users with different roles could have different access privileges. For example, only project creators or team members might edit a project, while others could have view-only access.
3. **Token Management:** Token-based authentication (e.g., JWT tokens) could ensure stateless and secure communication between the front end and back end, eliminating the need for session management.

For this POC, adding a simple token-based validation in API headers would provide a basic level of security if required in future iterations.

6.6 Project Features

The Project Features section provides a detailed overview of the core functionalities and design elements within this proof-of-concept (POC) application. By focusing on essential user interactions like project management and chatbot interfacing, the application brings an intuitive experience to users while demonstrating a streamlined backend and frontend structure. This POC, though primarily a conceptual prototype, exhibits vital components and workflows that

contribute to the app's effectiveness, from the project creation interface to the chatbot page and the end-user application. Screenshots accompany each feature to give a visual sense of the UI and user experience.

6.6.1 Project List and Creation page

The Project List and Creation page serves as the first interactive element of the application, providing users with a streamlined interface to view, create, or edit projects. Each project listed here includes essential metadata, such as the project title, creation date, and status, ensuring users have immediate context on their ongoing work. By clicking the "Create New Project" button, users can initiate a new project, complete with fields for specifying the project name, description, and other required parameters.

Key Functionalities

- **Project Listing:** Displays an organized list of existing projects, each with quick-access buttons for editing or viewing details. Project entries are dynamically populated, offering users a real-time view of their current work.
- **Project Creation:** Provides a form interface with input fields for project details like name, description, and runtime options. Once filled, the project can be saved, and it appears immediately in the project list.
- **Edit and Delete Options:** Users can modify project details or remove a project as required. The design emphasizes usability and simplicity, allowing users to make changes without navigating away from the page.

User Experience Considerations:

The Project List and Creation page prioritizes a clean and responsive design, ensuring smooth user interactions. Each feature is presented through intuitive UI elements that are easy to

navigate. Validation is enforced on form fields to ensure users enter meaningful data, thereby reducing the likelihood of errors.

6.6.2 Chat bot page

The chatbot page is an integral part of this application, offering a conversational interface where users can input requirements and receive guided feedback. Through this interface, users can describe desired functionalities, and the chatbot, backed by an LLM (Large Language Model), responds with intermediate code snippets or data structures essential to the application's back end. This intermediate code is stored in JSON format for further processing and rendering.

Key Functionalities

- **Text-based Interaction:** Users input requirements in natural language. The chatbot processes this input, leveraging the LLM to generate an intermediate representation of the described features.
- **Feedback Loop:** The chatbot supports an iterative conversation, allowing users to refine requirements and receive updated outputs, fostering a dynamic development approach.
- **Integration with Backend API:** All interactions with the chatbot are managed through the backend Flask API, ensuring a seamless flow of information and generation of accurate intermediate representations.

User Experience Considerations:

The chatbot page design is intended to mimic a familiar chat interface, encouraging users to interact naturally without needing to learn complex commands. The UI layout ensures visibility of past exchanges, making it easy for users to refer to previous responses and adjust their

requirements. Additionally, error messages and guidance on effective phrasing are included to enhance user satisfaction.

6.6.3 End use application navigation and user experience

The end-user application, a separate yet integral component of this POC, is responsible for interpreting the JSON-based intermediate code generated by the chatbot and rendering the appropriate UI and backend logic. Users experience a complete application view based on their specifications, from UI elements like forms and buttons to embedded backend logic triggered by user interactions.

Key Functionalities

- **Dynamic UI Generation:** Based on the intermediate JSON structure, the end-user application automatically generates pages with the appropriate UI components. This may include forms, buttons, and other interface elements, all mapped to the requirements defined by the user.
- **Backend Logic Integration:** Any action-based requirements are embedded within the application, enabling interactive features like data submission, page navigation, and form validation.
- **Navigation and Structure:** Users can easily move between pages within the generated application, exploring the functionalities they defined through the chatbot interface. This navigation allows for a hands-on evaluation of the application's structure and behavior.

User Experience Considerations:

The end-user application provides a direct look at the user's intended functionalities, creating a feedback-driven environment where users can test and refine their project in real-time. The

generated application's intuitive layout promotes ease of navigation, ensuring that users can evaluate and iterate their requirements effortlessly. The design prioritizes clarity and responsiveness, ensuring that UI elements adjust to different screen sizes for optimal usability.

6.6.4 Screenshots for illustration

Screenshots are essential to provide a visual representation of the application's core features, illustrating the UI and enhancing comprehension. These images capture the application's look and feel, including:

- **Project List and Creation Page:** Showing the listing of projects, create/edit options, and form validation features.

Create a new project

Project Name

Enter project name

Project Title

Enter project title

Project Description

Enter project description

Create Project

Existing Projects

my-app-2024-09-15-113844	Go
test-2024-09-15-153134	Go

Figure 4: Screen shot – Create and List Project - PoC

- **Chatbot Page:** Demonstrating conversational interactions, responses from the LLM, and example JSON outputs.

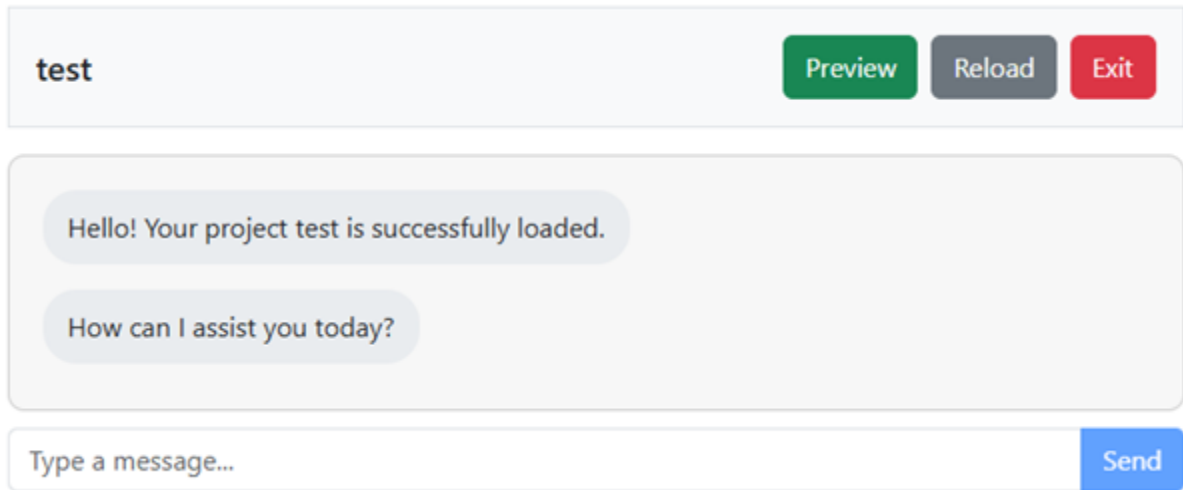


Figure 5: Screen shot – Chat bot - PoC

An example showing the chat session on the chat page.

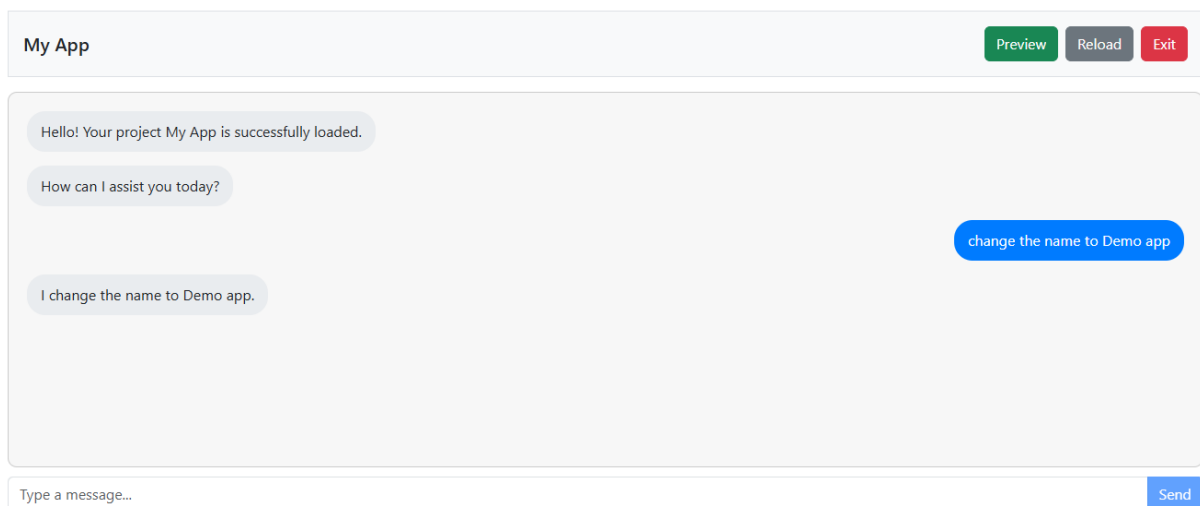


Figure 6: Chat bot with responses - PoC

Request to LLM

```
{"user_input": "change the name to Demo app"}
```

Response from the LLM

```
{
  "output": {
    "action": "changeTitle",
    "data": "Demo app",
```

```

    "section_name": "appName",

    "message": "I change the name to Demo app.",

    "inputType": "",

    "additionalRequests": []

}

}

```

System instructions to LLM

You are a tool that generates only valid json using the following structure.

```

{

    "action": "One of createPage, deletePage, updatePage, addNavMenu, addLinkInPage,
addSection, addInput, changeTitle, or unknown if the action is not clearly
classifiable.",

    "data": "fill it as per instructions below" ,

    "section_name": "One of navBar, title, appName, or any page names",

    "message": "A first-person explanation of the action in simple present tense.",

    "inputType": "A string representing the input type if action is addInput",

    "additionalRequests": "An array of json objects of this same structure. If there
are Mutiple requests possible from the input, make a json object in the same
structure and add to the field called additionalRequests"

}

```

"data" field instructions: if action is createPage, give the html content of the page. if action is addNavMenu give the 'data' field as a json object with two keys. label and pageName. pageName is the most appropriate name from the list of pages given from input. for all others give a string representing the relevant data for the action.

- **End-User Application:** Illustrating generated pages, navigation flows, and interactive components derived from the intermediate JSON representation.

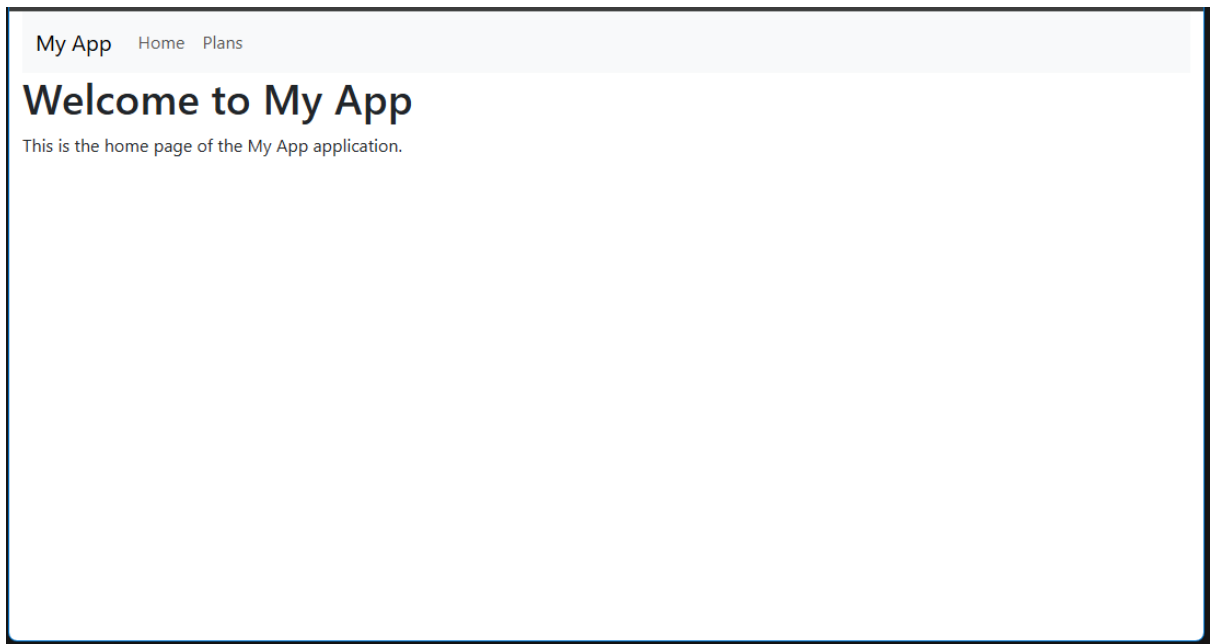


Figure 7: End-user application - PoC

Project data structure (Python).

```
class ActionEnum(str, Enum):  
    unknown = 'unknown'  
  
class EventEnum(str, Enum):  
    on_load = 'on_load'  
    on_click = 'on_click'  
  
class StageEnum(str, Enum):  
    NotInited = 'NotInited'  
    Inited = 'Inited'  
  
class JSONConvertible(AppJsonSerializable):  
  
    def __init__(self, obj = None):  
        if (obj is not None):  
            for key, value in obj.items():
```

```

        if hasattr(self, key):

            setattr(self, key, value)

        else:

            raise Exception(f"Unknown field in project: {key}:{value}")

def to_dict(self) -> typing.Dict[str, typing.Any]:

    result = {}

    class_vars = {key: getattr(self, key) for key in
self.__class__.__annotations__}

    for attr in class_vars: # Iterate over instance variables

        value = getattr(self, attr, None)

        result[attr] = app_serialize(value)

    return result

@classmethod
def from_dict(cls, data: typing.Dict[str, typing.Any]) -> 'Project':

    return cls(data)

def to_json(self) -> str:

    return json.dumps(self.to_dict(), indent=4)

@classmethod
def from_json_file(cls, json_str: str) -> 'Project':

    data = json.load(json_str)

    return cls(data)

class LLMResponsePayload(JSONConvertible):

    action: ActionEnum = ActionEnum.unknown

    data: typing.Any = None

    inputType: str = None

    message: str = None

    section_name: str = None

```

```

    additionalRequests: typing.List['LLMResponsePayload'] = []

class ControllerAction(JSONConvertible):

    name: str = None

    code_hint: str = None

    params: typing.List[str] = []

class Controller(JSONConvertible):

    name: str = None

    actions: typing.Dict[str, ControllerAction] = None

    pass

class Event(JSONConvertible):

    controller_name: str = None

    action: str = None

    params: typing.Dict[str, typing.Any] = None

class Element(JSONConvertible):

    type: str = None

    code_hint: str = None

    properties: typing.Dict[str, str] = None

    events: typing.Dict[EventEnum, Event] = None

    children: typing.List['Element'] = []

class Page(JSONConvertible):

    name: str = None

    master_page: 'Page' = None

    elements: typing.List[Element] = []

    events: typing.Dict[EventEnum, Event] = None

class NavBarItem(JSONConvertible):

```

```

label: str = None

disply_only_after_login: bool = False

page_name: str = None

submenu: typing.List['NavBarItem'] = []

class Project(JSONConvertible):

    id: str = None

    runtime_version: str = "1.0"

    stage: StageEnum = StageEnum.NotInitied

    name: str = None

    title: str = None

    description: str = None

    created_date: str = None

    updated_date: str = None

    framework: str = None

    navBar: typing.List[NavBarItem] = []

    pages: typing.List[Page] = []

```

Project JSON file example.

```

{
  "id": "my-app-2024-09-15-113844",
  "runtime_version": "1.0",
  "stage": "Initied",
  "name": "Demo app",
  "title": "Demo app",
  "description": "My App",
  "created_date": "2024-09-15 11:38:44",
  "updated_date": "2024-11-05 12:06:55",
  "framework": "simple",
  "navBar": [
    {
      "label": "Home",

```

```

        "disply_only_after_login": false,

        "page_name": "home",

        "submenu": []

    },

    {

        "label": "Plans",

        "disply_only_after_login": false,

        "page_name": "plans",

        "submenu": []

    }

],

"pages": [

    {

        "name": "home",

        "master_page": null,

        "elements": [

            {

                "type": null,

                "code_hint": "<h1>Welcome to My App</h1>\n<p>This is the home
page of the My App application.</p>",

                "properties": null,

                "events": null,

                "children": []

            }

        ],

        "events": null

    },

    {

        "name": "plans",

        "master_page": null,

        "elements": [

            {

```

```

        "type": null,
        "code_hint": "<h1>Plans</h1>\n<ul>\n    <li>Basic</li>\n    <li>Free</li>\n</ul>",
        "properties": null,
        "events": null,
        "children": []
    },
    {
        "type": "link",
        "code_hint": "<a href=\"/home\">Go to Home Page</a>",
        "properties": null,
        "events": null,
        "children": []
    }
],
"events": null
}
]
}

```

Each screenshot highlights a specific aspect of the application's design and functionality, offering readers a complete understanding of the user experience. By following these visual examples, users can better appreciate the application's flow and intuitive design elements.

6.7 Security

Security is a crucial aspect of any application, protecting user data, maintaining system integrity, and ensuring privacy in both development and production environments. Although this proof-of-concept (PoC) application currently lacks user authentication and is deployed locally without complex security measures, understanding and planning for comprehensive security is essential for eventual scaling and deployment to a live environment. Key areas to

consider include user authentication, secure communication, and data validation and sanitization.

User Authentication

For a full-fledged application, both the generator app (used by developers and administrators) and the generated app (used by end-users) should implement user authentication to restrict access and secure user data. Authentication mechanisms can range from simple username-password setups to more advanced solutions such as multi-factor authentication (MFA) and single sign-on (SSO).

- **User Authentication for Generator App**

The generator app manages critical functions, such as processing user requirements and generating code for applications. Ensuring that only authorized users have access to the generator app is essential to maintain control over the application's structure, integrity, and any private user information.

- **Authentication Methods:**

- **Basic Authentication:** For smaller setups, username and password combinations can provide quick access control.
- **OAuth2 or OpenID Connect (OIDC):** For scalable applications, OAuth2 or OIDC offers secure, token-based authentication, allowing for SSO integration and improved security practices.
- **Role-Based Access Control (RBAC):** Using RBAC allows administrators to define roles (e.g., Admin, Developer, Viewer) with specific permissions. This ensures that only authorized users can perform critical actions like project creation, code generation, and configuration changes.

- **User Authentication for Generated App**

The generated app might be used by end-users who interact with the front end of a web application. Securing this access ensures that user-specific data remains private and that sensitive operations (e.g., placing orders, modifying settings) are performed only by authenticated users.

- **Authentication Techniques:**

- **Session-Based Authentication:** This method involves assigning a session to users upon login, stored in a secure cookie, making it easy to manage user sessions.
- **JWT Token-Based Authentication:** JSON Web Tokens (JWT) are popular in stateless applications, providing an encrypted token that clients include with each request, improving scalability and reducing server-side session management requirements.
- **Multi-Factor Authentication (MFA):** MFA can be integrated to add an extra layer of security, reducing the risk of unauthorized access due to compromised passwords.

Secure Communication

Secure communication between the client and server, especially when handling sensitive data, is a fundamental requirement. Encryption mechanisms, secure transmission protocols, and the protection of communication channels are critical for maintaining the integrity and confidentiality of data.

- **HTTPS for Secure Communication**

- **SSL/TLS Encryption:** All HTTP traffic should be encrypted using SSL/TLS to establish HTTPS connections. This encryption protects data from

interception and eavesdropping during transmission between the client and server. HTTPS is especially important for any application involving user data, as it prevents unauthorized parties from viewing or altering data in transit.

- **Certificate Management:** Certificates are essential for enabling HTTPS and must be renewed periodically to ensure ongoing protection. Automated tools like Let's Encrypt provide a free, renewable SSL certificate to maintain secure connections in a development environment or production deployment.

- **Token-Based Authentication**

Token-based authentication is commonly used to secure communication between the front end and the back end. This approach involves generating tokens (e.g., JWTs) upon user login and including them in the headers of all subsequent API requests, ensuring that only authenticated requests can interact with secure endpoints.

- **JWT Implementation:** JWTs encode user data, which can be verified by the server without maintaining session states. Additionally, JWTs include an expiration time, limiting token validity to prevent token reuse and mitigate security risks.
- **Secure Token Storage:** For browser applications, tokens should be stored securely in HTTP-only cookies, limiting exposure to cross-site scripting (XSS) attacks. Avoid storing tokens in local storage, as this may increase susceptibility to certain attack vectors.

- **Other Security Protocols**

- **Cross-Origin Resource Sharing (CORS):** Enabling CORS policies ensures that only requests from allowed origins can interact with the back end, preventing unauthorized domains from accessing secure resources.

- **Content Security Policy (CSP):** Setting up a CSP reduces the risk of certain attacks, such as XSS, by specifying which sources are permitted for loading content (e.g., scripts, images, styles).

Data Validation and Sanitization

Data validation and sanitization are critical for maintaining a secure application environment, as improper handling of input data can result in vulnerabilities such as SQL injection, XSS attacks, and data corruption. Implementing validation and sanitization in both the generator app and the generated app ensures that input data is consistently checked, preventing malicious inputs and maintaining data integrity.

- **Data Validation for Generator App**

The generator app interprets user-provided requirements to generate code, making it susceptible to manipulation if inputs are not properly validated.

- **Validation Techniques:**
 - **Schema-Based Validation:** Using JSON schema validators (e.g., Marshmallow in Python) enforces data structure and format consistency, ensuring that each input follows the expected schema before processing.
 - **Input Type Checking:** Each input should be checked to confirm that it matches the expected type (e.g., integers, strings). For instance, parameters intended for code generation must not contain executable code that could interfere with the application.
- **Sanitization:**
 - **Encoding Special Characters:** Encoding characters such as <, >, and & prevents inputs from executing code within the application, reducing the risk of XSS attacks.

- **Removing Unsafe Characters:** Eliminate characters such as SQL operators (' , " , --) that may lead to SQL injection attacks if the generator app interacts with a database in the future.

- **Data Validation and Sanitization for Generated App**

The generated app, which renders code and manages end-user interactions, also requires robust validation and sanitization to prevent malicious inputs from affecting the user experience or accessing unauthorized resources.

- **Front-End Validation:** Basic validation on the front end, such as checking that fields are completed and inputs are formatted correctly, helps users submit valid data. However, front-end validation should always be backed by more secure server-side validation.
- **Server-Side Validation:** Validating data server-side is essential as it ensures that input received through APIs or forms is structured correctly before proceeding with processing.
- **Sanitization Techniques:**
 - **Output Encoding:** Encode outputs from untrusted data, preventing unintentional rendering of HTML or JavaScript code, thereby reducing the risk of XSS.
 - **Regex Filtering:** Use regular expressions to filter and validate input content, blocking potentially harmful patterns from being accepted by the application.

By implementing robust security measures, including user authentication, secure communication, and rigorous data validation and sanitization practices, this PoC application can effectively prevent unauthorized access, mitigate data vulnerabilities, and protect sensitive information. Although the current PoC is primarily local and does not include some of these

security features, this strategy provides a foundation for future deployment in a live environment.

6.8 Deployment

Deployment is a crucial phase in software development, enabling applications to transition from the development environment to live or staging environments for user interaction. Although this proof-of-concept (PoC) application is intended for local deployment, the architecture has been designed with flexibility, making it possible to deploy to cloud services such as AWS or other hosting platforms. This section outlines the hosting platform considerations, deployment steps for the front end and back end, and an overview of potential CI/CD strategies for future scalability.

Hosting Platform

For this PoC, the application is deployed and tested in a local environment. However, scalability and flexibility requirements were taken into consideration, meaning that deployment to a cloud platform, such as AWS or Google Cloud Platform, is feasible.

- **Local Deployment:**

- **Purpose:** Local deployment simplifies the development process, making it more accessible and cost-effective for early-stage testing.
- **Environment:** Development tools and environments such as Docker, Virtual Environments (for Python), and Node Package Manager (NPM) make it easy to replicate the entire application locally.
- **Configuration:** Basic configurations are set up in a .env file or configuration script that stores API keys, ports, and other settings.

- **Cloud Platforms (AWS/GCP) – Future Considerations:**

- **AWS EC2 or S3:** For a production-ready deployment, AWS EC2 instances can host the back end, while S3 buckets can handle static front-end assets. Additionally, AWS Elastic Beanstalk or ECS could simplify deployment management.
- **Google Cloud App Engine or Firebase:** Google Cloud's App Engine is another option for hosting scalable applications, especially for managing user interactions or dynamically rendering generated code.
- **Benefits:** Moving to the cloud could facilitate scalability, high availability, and redundancy, making the application accessible to multiple users and supporting diverse operational requirements.

Steps to Deploy the Application

This section covers deployment for both the front-end and back-end components in a local environment.

Back-End Deployment Steps

- **Environment Setup:**

- Install Python and the necessary dependencies (preferably through a virtual environment).
- Clone the backend repository to your local environment.
- Use pip to install required packages from the requirements.txt file:

```
pip install -r requirements.txt
```

- Set up environment variables in a .env file, including API keys and port information.

- **Run the Backend Server:**

- Execute the main application script or entry point. For Flask, this would generally be:

`flask run`

- For local testing, set Flask to run in development mode, allowing hot-reloading for efficient testing.

- **API Endpoint Testing:**

- Use Postman or CURL to test each API endpoint individually, confirming that the application handles requests, returns responses, and stores generated JSON data correctly.

Front-End Deployment Steps

- **Environment Setup:**

- Ensure Node.js and NPM (or Yarn) are installed.

Clone the front-end repository and install dependencies:

`npm install`

- Configure environment variables, especially for backend API URLs and frontend-specific settings, in a `.env` file.

- **Build the Application:**

Once the application is ready for deployment, build the optimized front-end application by running:

`npm run build`

This command compiles and bundles the code into a static folder (e.g., `/build`) ready for deployment.

- **Serve Front-End Locally:**

Use serve or any HTTP server (e.g., nginx or Apache) to host the front-end application locally for testing:

`npm start`

Deploying to Cloud Platforms

Although the PoC is locally deployed, future deployment to cloud platforms would involve:

- Setting up virtual machines or app services (e.g., EC2 on AWS or App Engine on GCP).
- Automating deployments using services like AWS CodeDeploy or GCP's Cloud Build, enabling automatic deployment of new versions.

Continuous Integration and Continuous Deployment (CI/CD)

CI/CD pipelines streamline the development workflow by automatically testing and deploying updates whenever code changes occur. Implementing CI/CD can help ensure stability and reduce the time between development and production. While not essential for a PoC, setting up a basic CI/CD pipeline provides the groundwork for future scalability and maintains application reliability.

Continuous Integration (CI)

- **CI Pipeline Structure:**
 - A CI pipeline enables automatic testing whenever developers push updates. This ensures that each change is verified for functionality, helping detect and prevent potential issues before merging.
 - **Tools:** For this PoC, GitHub Actions or GitLab CI/CD could serve as CI tools, automating unit and integration tests on each pull request or commit to the main branch.

- **Testing Steps in CI:**

- Back-end tests run using Pytest, while front-end tests use Jest or Mocha/Chai.
- The CI pipeline also validates environment settings, verifies that the project dependencies are installed correctly, and checks code quality through linting (e.g., ESLint for JavaScript and Pylint for Python).

Continuous Deployment (CD)

- **Purpose:** CD automates the deployment process, allowing verified changes to reach the live or staging environments quickly. For this PoC, the CD pipeline could update the local development server upon successful CI tests.
- **CD Pipeline Components:**
 - **Build Artifacts:** Front-end and back-end applications would be bundled and packaged for deployment. The build artifacts (e.g., Docker images or compressed files) are stored in a repository for deployment.
 - **Deployment:** Automated scripts deploy the latest build to the local environment. Tools like Docker Compose or PM2 can manage service restarts.
- **Cloud Deployment and Rollbacks:** Once cloud hosting is introduced, the CD pipeline could automatically push changes to cloud services with version control, enabling rapid rollbacks in case of issues.

4. CI/CD Workflow Example

Here's a simplified CI/CD workflow using GitHub Actions, highlighting the main steps:

- **Step 1:** Push changes to a branch triggers the CI pipeline, which tests code changes.

- **Step 2:** If tests pass, the code is merged into the main branch, triggering the CD pipeline.
- **Step 3:** The CD pipeline builds, deploys, and verifies the application on the local or cloud server.
- **Step 4:** Status checks and alerts notify the team if issues occur, allowing quick rollbacks or fixes.

Summary

Deployment considerations for this PoC include local hosting with a pathway to cloud deployment for scalability. With the flexibility of CI/CD automation, developers can efficiently deploy and update the application, ensuring quality and maintaining rapid development cycles. This foundation supports ongoing experimentation and improvement, paving the way for a robust deployment architecture suitable for larger, production-ready versions of the application.

6.9 Challenges and Solutions

Technical and Non-Technical Challenges Faced

In the development of this proof of concept, a variety of challenges arose, spanning both technical and non-technical domains. Addressing these challenges was essential to ensure that the application not only functioned as intended but also offered a user-friendly experience. As an MVP focused on requirement gathering and automated application generation, the project involved navigating complex technical demands while balancing practical considerations for end-users who may not have extensive technical expertise.

Technical challenges primarily revolved around translating natural language inputs into functional code, managing real-time interactions between the front end and back end, and efficiently handling data without a traditional database. Non-technical challenges involved clarifying user requirements, maintaining simplicity, and adapting to evolving needs while staying within the scope of a minimum viable product.

Outlined below are the primary technical and non-technical challenges encountered during development and the solutions implemented to address them.

- **Technical Challenges**

- **Requirement Extraction from Natural Language Input:** One of the primary technical hurdles involved accurately extracting functional requirements from natural language inputs. Since natural language is inherently ambiguous and prone to varied interpretations, ensuring that the language model interpreted user intent correctly was a significant challenge.
- **Intermediate Code Structure:** Designing a flexible yet robust intermediate code structure posed another challenge. The intermediate structure needed to be versatile enough to encapsulate both UI elements and logic without adding unnecessary complexity.
- **Front-End and Back-End Synchronization:** Maintaining synchronization between the front-end and back-end processes was crucial but challenging. Every update in the front end needed to trigger or request relevant updates from the back end without delay, ensuring that the chatbot's responses were accurately rendered in the application.
- **Real-Time Processing and Feedback:** To achieve an interactive user experience, real-time response generation and feedback handling were essential.

Processing user requirements, converting them into intermediate representations, and rendering them on the interface in real time required optimal performance and efficient data handling between components.

- **Data Storage Constraints:** Since this PoC aimed to avoid database complexity, all intermediate code had to be stored and managed in JSON files. Without a structured database, managing dependencies and ensuring the integrity of saved data required careful handling.

- **Non-Technical Challenges**

- **User Requirement Ambiguity:** Non-technical users often provide vague or incomplete requirements, which can lead to misinterpretation. Defining clear guidelines for users to phrase their requirements effectively was essential for reducing ambiguity.
- **Balancing MVP Simplicity with Functionality:** Striking a balance between a minimum viable product (MVP) approach and the need to demonstrate comprehensive functionality was a constant challenge. It was necessary to maintain simplicity while still highlighting the system's unique capabilities.
- **Adapting to Iterative Development Needs:** The iterative nature of the chatbot-driven design process required frequent user feedback loops, which took time and sometimes led to changing requirements, adding complexity to the development cycle.

How Challenges Were Overcome

1. Improving Requirement Extraction

To enhance the accuracy of requirement extraction, an iterative feedback system was introduced, allowing the LLM to ask clarifying questions when input was ambiguous.

The bot's ability to detect incomplete or unclear requirements helped refine inputs and reduce misunderstandings. Additionally, incorporating NLP techniques for intent recognition helped interpret user commands with higher precision, minimizing the risk of error during translation to intermediate code.

2. Refining the Intermediate Code Structure

The intermediate code format was refined by creating a class-based JSON structure that could handle diverse elements, events, and actions. This structure used Python classes to enforce a consistent data format, making it easier to process and reducing potential errors when parsing JSON files. By establishing enums and constraints within these classes, the team ensured that the data remained manageable and that each JSON file followed a predictable format.

3. Optimizing Front-End and Back-End Communication

Front-end and back-end communication was streamlined through a well-defined API layer that supported real-time data exchange. Using lightweight, asynchronous requests minimized lag, and efficient handling of request-response cycles ensured the chatbot and application were synchronized. The Flask framework's simplicity allowed for rapid API development, enabling smooth interactions between Angular components and Flask endpoints.

4. Real-Time Feedback Handling

To manage real-time responses, the back-end processing was optimized to reduce latency, and an event-driven architecture was adopted to handle multiple user inputs effectively. By using efficient data management techniques and minimizing redundant operations, the system achieved the required speed for interactive user experience.

5. Effective JSON Data Management Without a Database

JSON files were organized and modularized, allowing each project's intermediate code to be stored separately with unique identifiers. This minimized conflicts and facilitated rapid loading and saving of data. Stringent error handling and validation routines were implemented to ensure JSON data integrity, and a custom Python utility was developed to manage JSON loading, saving, and updating operations safely.

6. User Requirement Clarification and Training

Clear guidelines and examples were created to help users understand how to phrase requirements effectively. By integrating these into the chatbot interface, users could receive prompts or suggestions, minimizing the chances of vague or incomplete inputs. Additionally, a short onboarding tutorial was provided to introduce users to the basic structure of the project.

7. Iterative Development Adjustments

Agile methodologies were adopted to incorporate iterative development and adapt quickly to changes. Regular feedback loops with users allowed for quick adjustments, and each iteration focused on delivering core functionalities while accommodating new requirements.

Lessons Learned

- 1. Importance of Clear User Requirements:** Accurately capturing user intent is critical for automation projects. Developing an interface that guides users to enter precise, structured requirements saves time and reduces development complexity.
- 2. Robust Intermediate Code Structure:** A flexible yet standardized intermediate code format is essential for converting user requirements into functional applications. This structure not only facilitated UI rendering but also provided a foundation for modular

backend integration, showcasing how a well-planned data model can be pivotal in similar systems.

3. **Scalable Front-End and Back-End Design:** Implementing an API layer that effectively bridges the front end and back end proved invaluable for achieving real-time responses. As requirements evolve, this scalable design can easily incorporate additional functionalities.
4. **Iterative Development and Continuous Feedback:** Iterative cycles proved beneficial for aligning the generated application with user expectations. Continuous user feedback enhanced the system's adaptability and helped refine functionalities in each development cycle.
5. **Feasibility of an MVP Approach:** A focused, MVP approach provided a solid foundation without overextending on resources or introducing unnecessary complexities. By targeting core features, the PoC effectively demonstrated its value while leaving room for future expansion.
6. **Opportunities in Low-Code/No-Code Development:** This project highlighted the potential of integrating NLP and code generation in the low-code/no-code domain. The PoC experience underscores how such a system could make app development more accessible to non-developers, democratizing technology and fostering creativity among end users.

This experience provides a roadmap for future development, demonstrating how a user-driven, chatbot-powered interface can transform requirement gathering, UI development, and backend integration in an iterative, feedback-oriented software development process.

6.10 Future Enhancements

Features to be Added in Future Releases

The current proof of concept lays the groundwork for translating natural language requirements into functional application code. As this system progresses, there are numerous opportunities to enhance its capabilities. This section outlines key features planned for future releases to improve code management, the robustness of logic and event generation, and the sophistication of both UI and backend component generation.

- **Page-Wise Intermediate Code Management for Efficiency**

To streamline the process of code generation and make the system more scalable, introducing page-wise intermediate code storage is a priority. Instead of storing the entire intermediate representation in a single JSON file, each page of the generated application can have its own intermediate code file. This modular approach will offer several benefits:

- **Enhanced Readability:** By segmenting code into page-specific files, developers can quickly locate and understand the specific requirements and logic for each page.
- **Easier Updates:** If a user modifies requirements for a specific page, the system can update only the corresponding JSON file rather than reprocessing the entire structure. This will reduce update times and increase efficiency.
- **Improved Collaboration:** With a more granular file structure, multiple developers could work on different pages simultaneously, further speeding up development.

- **Version Control:** Page-wise segmentation will enable better tracking of changes and facilitate version control, as individual pages can be managed separately within the code repository.

- **Leveraging LLMs to Generate Advanced Events and Logic**

The current setup uses an LLM to generate intermediate code based on basic user requirements. Future iterations aim to extend this to more complex and dynamic aspects of application behavior by training the LLM on advanced business logic and event handling. Key advancements will include:

- **Sophisticated Event Management:** Expanding the system to support advanced events, such as form validations, asynchronous calls, and conditional interactions. By capturing complex requirements and converting them into detailed event logic, the LLM can produce richer application functionality.
- **Automated Business Logic Creation:** Beyond the basics, the LLM will be trained to infer advanced business logic requirements from user input, applying this to backend event handling and calculations. For example, generating code to handle complex calculations, data processing, and role-based user interactions.
- **Dynamic Code Injection:** As the LLM learns to handle dynamic requirements, future enhancements will enable it to generate code that adapts to specific runtime conditions, allowing the application to respond to changing contexts.

- **Advanced UI and Backend Component Generation**

Expanding beyond basic UI component generation, future releases will focus on enhancing both frontend and backend capabilities, introducing more customizable and visually appealing components:

- **Advanced UI Components:** With LLM assistance, the system will generate complex UI elements such as interactive dashboards, multi-level navigation bars, and data visualization elements (e.g., charts, tables). This will elevate the sophistication of applications created through the PoC.
- **Backend API Generation:** The system will be able to generate API endpoints automatically, based on inferred requirements. This will streamline backend development by creating RESTful or GraphQL endpoints that match the UI's data needs.
- **Integrated Authentication and Role Management:** As security becomes a focus, the system will generate backend code for user authentication and authorization. Role-based access controls will be integrated to protect sensitive data and restrict user permissions as needed.

Improvements in Performance, UI/UX, and Security

As the PoC grows in complexity, ensuring it performs optimally and remains secure will be essential. The following enhancements focus on these areas to increase overall system reliability and usability.

- **Performance Optimization**

To handle the growing data and computational load, several performance optimizations are planned:

- **Caching LLM Responses:** By implementing caching for common requests, the system can reduce redundant API calls to the LLM, significantly speeding up response times, especially for frequently used requirements.
- **Async Data Processing:** Introducing asynchronous processing for API calls and code generation will help keep the UI responsive. This approach will ensure

that the system remains efficient and that user inputs do not lead to delays or bottlenecks in processing.

- **Optimized JSON Storage and Retrieval:** By structuring JSON files into separate components for each page, the system will improve data retrieval times and reduce memory usage, leading to faster rendering and processing of each module.

- **UI/UX Enhancements**

As the system's feature set expands, ensuring that the user interface is accessible and intuitive will be essential for both developers and end-users. Planned improvements include:

- **Enhanced Chatbot Interface:** The chatbot will be refined to provide better prompts, feedback, and error handling to make interactions more seamless and user-friendly. Tooltips, suggestions, and error indicators will guide users in providing the required details for code generation.
- **UI Component Customization:** Future releases will allow users to customize UI components directly, providing options for colors, fonts, layouts, and more. This will increase the end-user application's versatility and adaptability to different branding needs.
- **Responsive Design:** Ensuring that generated applications are fully responsive will be crucial for meeting modern web and mobile design standards. With this in mind, the generated UI will support flexible layouts that adapt seamlessly across devices.

- **Security Improvements**

As the PoC progresses into a full-featured application, adding robust security measures is critical. Key security enhancements will include:

- **Secure API Endpoints:** By introducing token-based authentication for API access, the system will restrict access to the code generation and project management APIs, preventing unauthorized use.
- **Data Privacy and Storage Encryption:** For sensitive data stored in JSON files, implementing encryption protocols will ensure that project files remain secure even if accessed outside the application environment.
- **User Authentication and Role Management:** Although authentication was omitted in this MVP, future versions will introduce user management features, including login and role-based access control. This will allow administrators to assign permissions, ensuring that sensitive functionality is accessible only to authorized users.

By implementing these enhancements, the system will become more capable, secure, and user-friendly, providing an increasingly robust and scalable solution for automated application generation. These additions not only align with the project's long-term goals but also pave the way for the future of low-code and no-code application development platforms, positioning the system as an innovative tool within the software development landscape.

6.11 Conclusion

Summary of the Project

- This proof of concept (PoC) demonstrates a system designed to translate user-defined requirements into functional applications by leveraging natural language processing (NLP) and intermediate code generation. The project integrates a frontend developed in Angular with a Flask backend, which facilitates communication with a language model (LLM) like ChatGPT. By enabling users to enter requirements through a chatbot

interface and automating the conversion of these requirements into a structured, intermediate code representation, the project highlights a streamlined and iterative approach to application generation. This PoC serves as a foundational framework, demonstrating how conversational AI and code generation can reduce the complexities associated with translating user requirements into software functionality.

Key Takeaways

- **Front-End and Back-End Synergy:** The project underscores the importance of designing a cohesive system architecture where the front end and back end work together seamlessly. Angular's component-based structure complements the Flask API, facilitating smooth data handling and response rendering.
- **Iterative Requirement Translation:** Using a chatbot to capture user requirements introduces a flexible, iterative process. This allows users to refine inputs iteratively, ensuring accurate representation in the generated application.
- **Intermediate Code Generation:** By using intermediate code structures (in JSON format), the PoC introduces a modular way to translate requirements into backend and frontend logic. This JSON representation offers a flexible structure for further refinement and potential integration with other frameworks.
- **Proof of Concept for Agile Development:** The PoC aligns well with Agile methodologies, emphasizing iterative user feedback, adaptability, and continuous improvement—key factors for evolving user-driven application design.

Reflections on the Project's Impact

- This project demonstrates significant potential for future innovations in automated application development. The capability to translate conversational input into structured code not only improves accessibility but also reduces dependency on technical expertise for initial application design. By refining this system, organizations can empower end-users to participate more actively in the development process, fostering a greater sense of ownership and satisfaction.
- As the PoC matures, the impact on software development could be transformative, opening avenues for accessible, low-code platforms where both developers and non-developers can contribute to the application lifecycle. This project marks a step forward in achieving more user-centered, responsive, and efficient application development methodologies.

CHAPTER VII: SUMMARY, IMPLICATIONS, AND RECOMMENDATIONS

We can infer from the literature analysis that various research has already been conducted on a per-project basis for translating natural language requirements into code. The best illustration of that is the Open AI codex. Nevertheless, there hasn't been much research done on putting together an entire program, from the user interface to the backend server. By using some of the current technologies and some novel approaches proposed in this research, we have shown that it is possible to translate the stated user requirements into a working end-to-end program. The system can be improved to make it more usable and can be implemented in production with little to no changes thanks to ongoing user feedback. Pre-generated code for common components is the essential component that makes this work. We also proposed a way to properly assemble logical components into a system. This thesis explored the development of a system that automates the translation of user requirements into fully functional applications, leveraging advanced NLP models, machine learning algorithms, and chatbots. The study demonstrated the effectiveness of these technologies in extracting and systematizing user requirements, converting them into UI code, and integrating additional logical and business requirements. By adopting an iterative development process grounded in Agile principles and supported by continuous user feedback, the research ensured that the generated code was accurate, complete, and user-approved. The findings highlight significant improvements in the efficiency and quality of software development, paving the way for more responsive and accurate development processes.

7.1 Implications

The implications of this research are profound for both the software development industry and the academic community. The successful integration of advanced NLP and machine learning

techniques with interactive chatbot systems demonstrates a viable path toward more automated and efficient software development processes.

For Industry:

- **Enhanced Development Efficiency:** The automation of requirement extraction and UI code generation can significantly reduce development time and costs, allowing for faster delivery of software products.
- **Improved Accuracy and Quality:** The continuous feedback loop and iterative refinement ensure that the final product closely aligns with user expectations, leading to higher user satisfaction and reduced post-release corrections.
- **Scalability:** The system can be scaled to handle large projects with complex requirements, making it suitable for both small startups and large enterprises.
- **Resource Optimization:** By automating repetitive and time-consuming tasks, developers can focus on more complex and creative aspects of software development, optimizing resource utilization.

For Academia:

- **New Research Avenues:** The study opens up several new research avenues, including the integration of domain-specific ontologies with NLP models, the development of more sophisticated dialogue management systems, and the enhancement of automated testing frameworks.
- **Interdisciplinary Collaboration:** The research highlights the potential for interdisciplinary collaboration between computer science, artificial intelligence, and human-computer interaction, fostering innovation and advancements in software development methodologies.

- **Educational Impact:** The findings can be incorporated into software engineering curricula, providing students with cutting-edge knowledge and skills that are highly relevant to the current industry landscape.

7.2 Recommendations for Future Research

While this research has made significant strides in automating the software development process, several areas warrant further investigation to enhance the system's capabilities and applicability.

- **Integration with Domain-Specific Ontologies:**
 - a. Future research could explore the integration of domain-specific ontologies and knowledge bases with NLP models to improve the accuracy and relevance of requirement extraction.
 - b. This approach could enhance the system's ability to handle specialized terminology and industry-specific requirements.
- **Advanced Dialogue Management Systems:**
 - a. Developing more sophisticated dialogue management systems for chatbots can improve their ability to handle complex requirements and user interactions.
 - b. Future work could focus on enhancing the conversational abilities of chatbots, making them more intuitive and user-friendly.
- **AI-Driven Design Assistants:**
 - a. Exploring the use of AI-driven design assistants to refine and personalize UI components based on user preferences and behavior analytics can further enhance the system's effectiveness.
 - b. These assistants could provide real-time design recommendations and automatically adjust UI elements based on user feedback.

- **Automated Usability Testing Tools:**

- a. Investigating automated usability testing tools that integrate seamlessly with the development environment can provide real-time insights and recommendations, further improving the alignment of UI code with user requirements.
- b. These tools could utilize machine learning algorithms to predict and identify usability issues before they arise.

- **Predictive Analytics for Feedback Systems:**

- a. Enhancing continuous feedback systems with predictive analytics can help anticipate user needs and potential issues, allowing for proactive adjustments and improvements.
- b. Future research could focus on developing predictive models that analyze user behavior and feedback patterns to optimize the development process.

7.3 Conclusion

This study successfully demonstrated the feasibility of automating software development from user requirements to functioning code. By integrating natural language processing, machine learning, and interactive feedback mechanisms, the system produced user interfaces and backend logic with reduced manual effort.

The findings confirm the effectiveness of AI-driven development tools while identifying limitations in complex business logic handling and domain specificity. This proof of concept expands the potential of tools like Codex and Rasa by embedding them in a feedback-driven, iterative design cycle.

For future research, opportunities exist to integrate domain-specific ontologies, improve chatbot dialogue management, and extend the system's capacity to handle advanced edge

cases. Overall, the research provides a valuable foundation for transforming traditional software development into a more intelligent, user-centered process.

APPENDIX A: INFORMED CONSENT

In the context of this research, informed consent is a crucial aspect to ensure ethical considerations are adhered to throughout the study. This section outlines the steps taken to inform participants, safeguard their rights, and maintain transparency in the collection and use of data.

Purpose of Consent

The research focuses on automating the conversion of user requirements into fully functional applications using AI and NLP techniques. While the primary data source for this proof of concept involves interactions between the chatbot and the end-user, the process incorporates participant feedback to refine generated outputs. Participants are made aware of their role, the nature of their contributions, and how their data will be utilized.

Consent Process

1. **Clear Communication:** Participants are provided with a detailed explanation of the research objectives, methodology, and the specific tasks they are expected to perform.
2. **Voluntary Participation:** Participation is entirely voluntary, with no pressure or obligation to contribute.
3. **Privacy Assurance:** Data collected, such as feedback on chatbot interactions, is anonymized and stored securely. The study does not collect sensitive personal information.
4. **Right to Withdraw:** Participants are informed of their right to withdraw at any stage of the research without any repercussions.

Data Usage and Protection

All data generated in the study, including intermediate code and user feedback, is treated with confidentiality. The storage is in compliance with ethical standards, ensuring that data is only

used for the stated research purposes. Results and findings are anonymized to protect participant identities and are shared responsibly in the final thesis or publications.

Acknowledgment of Consent

Participants are required to acknowledge their understanding and agreement by providing consent digitally or in writing before engaging with the system. A consent form is designed with straightforward language, summarizing the key aspects of participation and data usage.

By ensuring transparency and ethical rigor through informed consent, this research upholds its commitment to fostering trust and maintaining high ethical standards in AI-driven development processes.

REFERENCES

- Arellano, E., Carney, J., & Austin, M. (2015). Natural language processing of textual requirements. *ICONS 2015: The Tenth International Conference on Systems*.
<https://user.eng.umd.edu/~austin/reports.d/ICONS2015-AA-EC-MA.pdf>
- Beltramelli, T. (2018). *pix2code: Generating code from a graphical user interface screenshot*. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems* (p. 3). ACM. <https://arxiv.org/abs/1705.07962>
- Bocklisch, T., Faulkner, J., Pawlowski, N., & Nichol, A. (2017). Rasa: Open source language understanding and dialogue management. *arXiv preprint arXiv:1712.05181*.
<https://arxiv.org/pdf/1712.05181.pdf>
- Business Insider. (2025a). Amazon is working on a secret project called 'Kiro,' a new tool that uses AI agents to streamline software coding. <https://www.businessinsider.com/amazon-kiro-project-ai-agents-software-coding-2025-5>
- Business Insider. (2025b). Goldman is assembling a growing arsenal of AI tools. Here's everything we know about 5. <https://www.businessinsider.com/goldman-sachs-ai-uses-5-tools-employees-2025-5>
- Business Insider. (2025c). Mark Zuckerberg says AI could soon do the work of some engineers at Meta. <https://www.businessinsider.com/mark-zuckerberg-ai-startup-company-with-small-team-2025-5>

Business Insider. (2025d). The age of incredibly powerful 'manager nerds' is upon us, Anthropic cofounder says. <https://www.businessinsider.com/anthropic-cofounder-jack-clark-ai-manager-nerds-2025-5>

Chen, M., Tworek, J., Jun, H., Yuan, Q., et al. (2021). *Evaluating large language models trained on code*. <https://arxiv.org/abs/2107.03374>

Desai, A., Gulwani, S., Hingorani, V., Jain, N., Karkare, A., Marron, M., R, S., & Roy, S. (2016). Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)* (pp. 345–356). ACM. <https://doi.org/10.1145/2884781.2884786>

Doe, J., Smith, J., Brown, A., & Johnson, M. (2022). Automating requirement extraction: A comparative study of transformer-based and neural network approaches. *IEEE Transactions on Software Engineering*, 48(5), 1234–1245. <https://www.sciencedirect.com/science/article/abs/pii/S0045790622003123>

Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., & Zhang, J. M. (2023). Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533*. <https://arxiv.org/abs/2310.03533>

Financial Times. (2025). AI agents: From co-pilot to autopilot. <https://www.ft.com/content/3e862e23-6e2c-4670-a68c-e204379fe01f>

Friesen, E., Baumer, F. S., & Geierhos, M. (2018). Córdula: Software requirements extraction utilizing chatbot as communication interface. In *REFSQ Workshops*. <https://www.semanticscholar.org/paper/CORDULA%3A-Software-Requirements-Extraction-Utilizing-Friesen-B%3%A4umer/561b95f93ff868423fe664cbb0bef87d8ccb1b3f>

Gulwani, S. (2010). Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming* (pp. 13–24). ACM. <https://doi.org/10.1145/1836089.1836091>

Gulwani, S., & Marron, M. (2014). NLyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (pp. 803–814). ACM. <https://doi.org/10.1145/2588555.2612177>

Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., & Wang, H. (2023). Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620*. <https://arxiv.org/abs/2308.10620>

Kessentini, M., Gaaloul, W., Sahraoui, H., & O'Cinneide, M. (2021). A comparative study of machine learning algorithms for classification of software requirements. *Journal of Systems and Software*, 171, 110850. <https://www.sciencedirect.com/science/article/pii/S1877050918312316>

Kolthoff, K. (2019). Automatic generation of graphical user interface prototypes from unrestricted natural language requirements. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. <https://ieeexplore.ieee.org/document/8952477>

Moran, K., Bernal-Cardenas, C., Curcio, M., Bonett, R., & Poshyvanyk, D. (2018). Machine learning-based prototyping of graphical user interfaces for mobile apps. *arXiv preprint arXiv:1802.02312*. <https://ieeexplore.ieee.org/document/8374985>

Mukasa, K. S., & Kaindl, H. (2008). An integration of requirements and user interface specifications. In *2008 16th IEEE International Requirements Engineering Conference* (pp. 327–328). IEEE. <https://ieeexplore.ieee.org/document/4685696>

Pulido-Prieto, O., & Juárez-Martínez, U. (2018). A survey of naturalistic programming technologies. *ACM Computing Surveys*, 50(5), Article 70. <https://doi.org/10.1145/3109481>

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140), 1–67. <https://jmlr.org/papers/volume21/20-074/20-074.pdf>

Ravid, A., & Berry, D. M. (2000). A method for extracting and stating software requirements that a user interface prototype contains. *Requirements Engineering*, 5(4), 225–241. <https://link.springer.com/article/10.1007/PL00010352>

Reuters. (2025). Google is developing software AI agent ahead of annual conference, The Information reports. <https://www.reuters.com/business/google-is-developing-software-ai-agent-ahead-annual-conference-information-2025-05-12/>

Robeer, M., Lucassen, G., van der Werf, J. M. E. M., Dalpiaz, F., & Brinkkemper, S. (2016). Automated extraction of conceptual models from user stories via NLP. In *2016 IEEE 24th International Requirements Engineering Conference (RE)*. <https://ieeexplore.ieee.org/document/7765525>

Sawhney, R. (2021). Can artificial intelligence make software development more productive? *LSE Business Review*. <https://blogs.lse.ac.uk/businessreview/2021/09/13/can-artificial-intelligence-make-software-development-more-productive/>

Sharma, S., Sarka, D., & Gupta, D. (2012). Agile processes and methodologies: A conceptual study. *International Journal on Computer Science and Engineering*.

<https://www.researchgate.net/publication/267706023>

Shin, J., & Nam, J. (2021). A survey of automatic code generation from natural language. *Journal of Information Processing Systems*, 17(3), 537–555. <http://xml.jips-k.org/full-text/view?doi=10.3745/JIPS.04.0216>

Shylesh, S. (2017). A study of software development life cycle process models. *SSRN Electronic Journal*. <https://ssrn.com/abstract=2988291>

Smith, A. B., Brown, D. R., & White, E. J. (2020). Unsupervised learning techniques for requirements engineering: A comparative study of clustering and topic modeling approaches. *Requirements Engineering*, 25(2), 123–137. <https://ijisae.org/index.php/IJISAE/article/view/6018>

Wong, M. F., Guo, S., Hang, C. N., Ho, S. W., & Tan, C. W. (2023). Natural language generation and understanding of big code for AI-assisted programming: A review. *arXiv preprint arXiv:2307.02503*. <https://arxiv.org/abs/2307.02503>

Zhang, Z., Chen, C., Liu, B., Liao, C., Gong, Z., Yu, H., Li, J., & Wang, R. (2023). Unifying the perspectives of NLP and software engineering: A survey on language models for code. *arXiv preprint arXiv:2311.07989*. <https://arxiv.org/abs/2311.07989>