VIETNAM NATIONAL UNIVERSITY OF HO CHI MINH CITY
THE INTERNATIONAL UNIVERSITY
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

AI-POWERED FLOWCHART GENERATION: CONVERTING NATURAL LANGUAGE FROM COMPLEX PROCEDURAL DOCUMENTS INTO STRUCTURED DIAGRAMS

By
Nguyen Nhat Khiem

Instructed by
Tran Thanh Tung, PhD

A thesis submitted to the School of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
Bachelor of Data Science Ho Chi Minh City, Vietnam

July 2025

AI-POWERED FLOWCHART GENERATION: CONVERTING NATURAL LANGUAGE FROM COMPLEX PROCEDURAL DOCUMENTS INTO STRUCTURED DIAGRAMS

APPROVED BY:

Assoc. Prof. Vo Thi Luu Phuong, Chair

Dr. Tran Thanh Tung, Commissary

Dr. Le Hai Duong, Secretaray

THESIS COMMITTEE

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my supervisor, Dr. Tran Thanh Tung, whose guidance has been a cornerstone of this research. He has not only provided me with invaluable knowledge in the field of artificial intelligence but has also instilled in me a spirit of curiosity and unwavering determination. His mentorship extended far beyond academics, as he thoughtfully taught me the fundamental importance of perseverance, critical thinking, and ethical responsibility in all aspects of research. Without his insightful support, constant encouragement, and profound insights, this work would not have been possible.

This research is supported by the Central Interdisciplinary Laboratory in Electronics and Information Technology (AI and Cooperation Robot), International University – VNU-HCM. I would like to extend my sincere thanks to the laboratory for providing essential computational resources that supported the experiments conducted in this study.

I would also like to extend my sincere gratitude to my dear friends, who have consistently been a wellspring of inspiration and unwavering motivation. Their thoughtful discussions, steadfast support, and genuine camaraderie have served as a reliable beacon of strength during the inevitable challenging moments encountered throughout this demanding journey.

Special thanks are due to my beloved family: my parents, who gifted me life, lovingly nurtured my dreams, and supported me unconditionally through each and every phase of my academic journey. Their unwavering belief in my abilities has been the very foundation of my resilience and ultimate success.

Furthermore, I want to express my heartfelt gratitude to my esteemed mentor from my previous company, Mr. Tran Viet Thi, whose invaluable teachings in both professional knowledge and the profound wisdom of Buddhist philosophy have guided me with a sense of peace and ease through the various hardships encountered during the completion of this thesis. His insightful wisdom and consistent encouragement have been truly instrumental in helping me maintain essential balance and clarity throughout this entire journey.

Adding to this, I would like to specifically acknowledge two predecessors, Nguyen Hoang Anh Tu and Nguyen Quang Dieu, for their invaluable support throughout my four years of university. Their guidance through various hardships was instrumental in paving the way for me to successfully complete this thesis. Their willingness to share their experiences and offer assistance was truly appreciated and made a significant positive impact on my academic progress.

Last but certainly not least, I want to express my sincere and heartfelt thanks to everyone who has contributed to this research, whether through direct collaboration, insightful feedback, or simply by offering much-needed encouragement. Your collective efforts, unwavering faith in my potential, and even the smallest yet significant acts of kindness have served as a continuous guiding light throughout this entire intricate process.

"True love is born from understanding."
– Gautama Buddha

Table of Contents

List of Tables

List of Figures

List of Algorithms

Abstract

Procedural documents are essential in domains such as technical documentation, legal compliance, and process modeling, but their complexity often makes them difficult to interpret and communicate efficiently. Traditional manual methods for extracting and visualizing procedural logic are time-consuming, error-prone, and lack scalability. While some AI-based summarization and diagramming tools exist, they frequently suffer from limited control, poor structure preservation, or reliance on cloud infrastructure—raising concerns around privacy, determinism, and usability.

This thesis introduces IUFlowGen, an AI-assisted, human-in-the-loop system for converting procedural documents into structured, interactive flowcharts. By integrating large language models (LLMs), retrieval-augmented generation (LightRAG), vector-based and graph-based indexing, and DOT-based visualization, the system automates the extraction of steps, actors, entities, and logical relationships from unstructured text. Unlike prior systems, IUFlowGen runs entirely on local infrastructure, offering enhanced privacy and eliminating token or usage restrictions. It supports interactive querying, multi-level visualization modes, and is designed for extensibility and verifier-guided refinement.

The system was experimentally validated through a user study involving participants with and without AI assistance, under documents of varying complexity. Results show that IUFlowGen significantly improves both accuracy and efficiency in flowchart generation, particularly for documents with unclear procedural structure. With a modular architecture and a strong emphasis on interpretability and privacy, IUFlowGen offers a solution for organizations seeking to extract actionable workflows from complex documentation. This research demonstrates the feasibility and advantages of integrating retrieval-enhanced LLMs and interactive visualization for procedural understanding in academic, legal, and technical domains.

Chapter 1
INTRODUCTION

1.1 Motivation

In today's rapidly evolving information landscape, professionals and learners alike are increasingly confronted with lengthy, complex documents that span various domains such as information technology, healthcare, legal systems, and physics [1]. These documents are often procedural in nature and require not only significant time to read, but also a high level of expertise to comprehend. A key source of this complexity lies in both their dense structure and challenging content. Structurally, they often involve multiple actors, numerous entities, and a wide variety of actions described in domain-specific terminology. Moreover, procedural documents frequently include intricate logical structures such as conditional branches (e.g., `if-else`), parallel operations, merges, and nested flows. Conceptually, they may describe technically demanding or abstract processes that are difficult to follow, even for trained professionals. The growing volume of such documentation—particularly in specialized technical fields—raises a critical challenge: how can users effectively navigate and understand the intricate processes they describe? This challenge highlights the need for intelligent assistants capable of aiding comprehension.

One compelling solution is the use of visualizations to augment the reading and interpretation process [2]. Among the various forms of visualization, flowcharts stand out as particularly effective. Flowcharts can transform dense procedural text into a structured sequence of visual elements—steps, decisions, inputs, and outcomes—thereby making abstract or convoluted procedures more concrete and approachable. By decomposing processes into intuitive diagrams, flowcharts not only facilitate faster comprehension but also support decision-making and collaboration. This visual representation accelerates understanding and reduces cognitive load, enabling users to grasp procedural logic at a glance rather than sifting through paragraphs of technical detail.

The rise of advanced artificial intelligence, particularly in natural language understanding (NLU), has made it increasingly feasible to automate this transformation [3]. Modern large language models (LLMs) possess strong capabilities in parsing unstructured text, identifying semantic relationships, and reasoning over procedural instructions [4]. Complementing this are visualization tools such as Graphviz and DOT language, which provide a programmatic interface for rendering flowcharts [5]. Additionally, the emergence of the prompt engineer role reflects a new paradigm in interacting with LLMs—crafting precise instructions to guide AI systems in generating outputs that align with user intent [6]. This convergence of AI understanding and structured visualization technology presents an opportunity to automate the translation of complex documents into meaningful visual diagrams.

However, one notable limitation in this emerging field is the lack of established metrics to objectively evaluate the effectiveness of AI systems in transforming complex procedural documents into accurate flowcharts. While qualitative user feedback and expert reviews are commonly used, they fall short in standardizing assessments of structural correctness, completeness, and interpretability. The absence of benchmarking frameworks poses a signif-

icant challenge in validating and comparing such systems. As a result, there is a compelling need to define quantitative metrics that can measure the accuracy and quality of flowchart generation—ensuring not only reproducibility but also facilitating meaningful evaluations of AI-driven procedural understanding tools.

In light of these needs and technological advancements, this thesis presents the design and implementation of an AI-powered system that automatically generates flowcharts from complex procedural documents. The system combines natural language understanding with structured diagram generation to enhance procedural comprehension. Its effectiveness is evaluated using metrics for accuracy and completeness through a study.

1.2 Problem Statement

This thesis proposes IUFlowGen, a local, AI-assisted system that converts complex procedural documents—often involving multiple actors, conditional logic, and domain-specific language—into structured flowcharts. By integrating large language models, retrieval-augmented generation, and DOT-based visualization, the system extracts procedural logic from unstructured text. It aims to reduce manual effort, improve clarity, while this thesis also proposes evaluation metrics to assess the accuracy and completeness of AI-generated flowcharts.

1.3 Scope and Assumption

This thesis defines the scope and boundaries of the proposed system through a set of clearly stated assumptions and infrastructure considerations. These guide both the system's design and its intended use cases, ensuring feasibility, clarity, and contextual relevance.

1. **Document Validity and Structure:** The system assumes that all input documents are valid and logically structured in terms of procedural flow. It does not validate the correctness, ethics, or real-world applicability of the content. The system focuses solely on extracting and visualizing procedural logic from the text as provided.

2. **Language and Format Constraints:** Input documents must be written in English and provided in a machine-readable format, such as digitally encoded PDF files with selectable text. The system does not support scanned images, image-based PDFs, or handwritten notes due to the absence of integrated optical character recognition (OCR) capabilities [7].

3. **Content Neutrality:** The system is content-neutral and does not perform fact-checking, semantic validation, or appropriateness assessment. It treats the text as a source of procedural information, irrespective of its truthfulness or correctness.

4. **Use of AI Technologies:** The system leverages pre-existing artificial intelligence tools, especially large language models (LLMs), as black-box components for natural language understanding and generation. It does not require modification or detailed analysis of these models, focusing instead on practical integration.

5. **Human-in-the-Loop Validation:** To ensure accuracy and contextual reliability, the system adopts a Human-in-the-Loop (HITL) approach [8]. AI-generated flowcharts

serve as draft suggestions, subject to review, correction, and approval by end users or domain experts before finalization.

6. **Deployment Environment:** The system is designed for execution in a local, controlled environment. It assumes that required dependencies are properly set up and that the system can be ported to compatible machines. However, it does not support multi-platform deployment, web-based distribution, or commercial packaging.

7. **Trusted Users and Operators:** It is assumed that both document providers and system operators are trustworthy and have the technical competency to use the system correctly. This includes the responsibility for preparing suitable input documents and managing the environment in which the system runs.

These assumptions and scope limitations are critical to understanding the system's operational context. They provide a focused framework for evaluating the system's performance in procedural flowchart generation, emphasizing usability, interpretability, and data privacy.

1.4 Objectives

This thesis presents a comprehensive solution for transforming procedural documents into interactive flowcharts using artificial intelligence. The following key contributions have been made throughout the development and evaluation of the system:

1. **Development of an Automated Flowchart Generation System:** The core objective of this thesis is the development of an AI-powered system capable of automatically processing procedural documents and generating corresponding interactive flowcharts. The system operates end-to-end by identifying structured steps, decisions, and relationships from unstructured text and then visualizing them in diagrammatic form. This transformation enhances users' comprehension by providing a clear, visual overview of complex processes.

2. **Design of a Modular Processing Pipeline Integrating Advanced AI Techniques:** A modular processing pipeline has been designed and implemented, integrating advanced natural language understanding (NLU), prompt engineering, vector-based representations, and retrieval-augmented generation (RAG) [9]. These components operate together to extract procedural semantics from input text. The system treats the underlying language model as a black-box component, focusing instead on coordinating these technologies into a reliable and interpretable workflow.

3. **Construction of a Full-Stack Application for End-User Interaction:** A complete full-stack application has been developed to enable seamless interaction with the system. This includes a user-friendly front-end for document upload and flowchart interaction, a back-end for AI communication and orchestration, and dynamic rendering components. This architecture ensures usability, responsiveness, and extensibility for real-world use.

4. **Support for Multi-Domain Procedural Documents:** The system is designed to process procedural texts from multiple domains, such as agriculture, education, IT, cybersecurity, and manufacturing. Rather than relying on domain-specific ontologies, it leverages the generic structure of procedural language, allowing it to generalize effectively across contexts and applications.

5. **Implementation of a Feedback-Based Benchmarking Methodology:** A user-centric evaluation framework has been implemented to assess the system's output quality. This involves collecting qualitative feedback from domain experts and users, focusing on structural accuracy, clarity, and relevance. In the absence of standard metrics for flowchart evaluation, this feedback-driven benchmark serves as a practical and insightful validation method.

Together, these contributions establish a foundation for AI-assisted procedural understanding and offer a scalable framework for future research in flowchart generation and intelligent document analysis.

1.5 Solution

To address the problem of converting complex procedural documents into comprehensible visual flowcharts, this thesis proposes an AI-driven system named **IUFlowGen**. At its core, IUFlowGen employs a modular pipeline that decomposes the overall task into smaller, manageable subtasks—each delegated to specialized components of the system. It strategically combines the strengths of large language models (LLMs) with auxiliary tools and techniques to ensure output accuracy, stability, and usability.

The backbone of IUFlowGen is a large language model (LLM), which interprets the semantic structure of procedural texts and identifies the relevant steps, decisions, and logical relationships. However, due to the inherently non-deterministic and sometimes unstable nature of LLM outputs, relying solely on a single generative pass can lead to inconsistent or incoherent results [10]. To overcome this, the system adopts a controlled prompting strategy combined with additional layers of validation and refinement.

A key component of the solution is the integration of retrieval-augmented generation (RAG) [9], specifically implemented using LightRAG [11]. This module enhances the AI's contextual understanding by injecting relevant internal document references into the prompt prior to generation. LightRAG achieves this by first transforming the input document into graph-based and vector-based representations, allowing it to retrieve semantically and structurally relevant snippets during query time. By narrowing the model's focus to high-precision contextual segments derived from both the graph and vector indices, the system significantly improves its ability to follow the document's procedural logic and produce coherent outputs.

To further increase reliability and enforce structural consistency, IUFlowGen applies regular expressions (regex) as a post-processing mechanism. Regex patterns are used to extract structured components—such as node names, relationships, and decision logic—from the raw outputs of the language model. This enables validation and transformation of the AI-generated content into well-formed `DOT` code, a graph description language used for rendering flowcharts with tools such as Graphviz.

By orchestrating AI generation, context retrieval, pattern-based filtering, and graph construction, IUFlowGen successfully transforms unstructured procedural text into clean, structured flowchart diagrams. This approach not only leverages the LLM's natural language comprehension capabilities but also compensates for its limitations by ensuring structural integrity and visual clarity in the final output.

Ultimately, IUFlowGen guides the AI toward fulfilling clearly defined objectives, ensuring that each generated flowchart adheres to the procedural structure of the source text while remaining interpretable, interactive, and useful to end users.

Chapter 2
RELATED WORK

This chapter explores recent advancements in AI-driven procedural text processing, with a particular emphasis on natural language understanding, retrieval-augmented generation, and visual representation techniques. It critically examines how current systems interpret and structure procedural knowledge, highlighting their capabilities and inherent limitations. In doing so, it identifies key research gaps—particularly the lack of end-to-end, interactive systems that preserve both semantic meaning and procedural logic. Alongside this review, the chapter introduces foundational concepts such as language models, embedding spaces, and graph-based reasoning, which are essential for understanding the performance boundaries of existing solutions and for contextualizing the design of the proposed human-in-the-loop visualization system.

2.1 Related Works

Recent advancements in artificial intelligence have significantly expanded the potential for procedural document processing, particularly through the use of natural language understanding, retrieval-augmented generation, and automated visualization techniques. Research in this field can be categorized into three main groups: (1) AI systems for procedure interpretion without generating them from text, (2) AI-based summarization without visual output, and (3) applied tools for procedural understanding and visualization. While each group shows meaningful progress, current solutions continue to exhibit notable limitations, especially in terms of interactivity, structural accuracy, and privacy-preserving execution.

AI Systems for Flowhart Interpretation without Generation

In *Parsing and Understanding Flowchart Using Generative AI*, Abdul Arbaz, Heng Fan, and others present *GenFlowchart* [12], a framework aimed at improving the interpretation of existing flowcharts through the use of generative AI. The system incorporates segmentation models and optical character recognition (OCR) [7] to extract flowchart components, such as shapes and connectors, from images. It then applies language modeling to infer procedural relationships. While effective at deconstructing existing visuals, *GenFlowchart* does not support the generation of flowcharts from unstructured procedural text. As such, its functionality is confined to flowchart parsing rather than creation—limiting its use in automation workflows where the source input is purely textual.

In *Extracting Procedural Knowledge from Technical Documents*, Shivali Agarwal et al. [13] propose a system that focuses on procedural step extraction from technical manuals. Their approach involves identifying and classifying operational instructions using rule-based and machine learning techniques. While this system is effective at isolating meaningful instructions and presenting them in sequence, it lacks a visual output layer. The extracted steps are returned as plain text, with no built-in support for flowchart generation or diagrammatic interpretation. Consequently, users must still manually convert the output into a visual form, making the tool less suited for real-time or large-scale procedural visualization tasks.

AI-Based Summarization without Visual Output

In *Summarizing Long Regulatory Documents with a Multi-Step Pipeline*, Brian Lester, Kenton Lee, and Dan Roth [14] introduce a multi-phase framework designed to tackle the challenges of summarizing extensive regulatory documents. The system segments lengthy texts into smaller sections, applies extractive summarization to each, and subsequently generates a cohesive abstractive summary. This method is effective in capturing key procedural information and enhancing document digestibility for readers. However, the system is limited in scope—it focuses exclusively on textual summarization and does not support the generation of structured visual outputs such as flowcharts. As a result, while it succeeds in condensing complex documents, it fails to map procedural steps and relationships into visual workflows, which are essential for clarity and automation in domains dealing with process-heavy documentation.

In *Long Document Summarization with Workflows and Gemini Models* [15], Randy Spruyt explores how transformer-based models can condense long procedural documents into workflow-style summaries. The proposed system segments the input document into logical sections and produces concise summaries that capture the intent of each block. This helps reduce reading time and cognitive load, especially in enterprise or regulatory settings. Despite its ability to extract key content, the system lacks visual rendering capabilities and does not attempt to generate structured diagrams. This absence of visualization limits its usefulness in tasks that require understanding of flow, branching logic, or procedural dependencies—critical features for systems meant to support decision-making or automation.

AI Tools for Procedural Understanding and Visualization

ChatGPT 4o [16], developed by OpenAI, has been widely used to interpret procedural content and generate structured outputs in natural language. The model is capable of identifying steps, dependencies, and actors within textual descriptions, making it suitable for general-purpose reasoning tasks. However, *ChatGPT* lacks native support for graphical rendering. When integrated with tools such as Graphviz or Mermaid, it can generate basic flowcharts, but these are often static, imprecise, or visually cluttered. Moreover, since *ChatGPT* typically runs on cloud infrastructure, it poses privacy risks in environments dealing with confidential documents. As such, its use in enterprise or regulated settings remains constrained by security and output fidelity concerns.

Eraser Flowchart Maker [17], reviewed in several productivity and AI forums, is a commercial tool that allows users to input procedural text and receive an automatically generated flowchart. The platform emphasizes visual clarity and ease of use. It employs NLP to identify key actions and conditions, and maps them into a flowchart layout. However, users frequently report that the tool oversimplifies complex logic and occasionally omits critical steps. Additionally, its reliance on token-limited free tiers and premium subscriptions restricts its accessibility. The lack of customization for node structure or condition modeling also limits its applicability in more technical or regulated domains.

NoteGPT [18], developed as an AI-driven flowchart assistant, enables users to convert meeting notes and outlines into simple diagrams. It supports casual task planning and is optimized for ease of use in collaborative settings. The system parses action items and their sequencing, rendering them into a basic visual form. While useful for informal use, *NoteGPT* is limited by its lack of depth in logic interpretation. It struggles with nested conditions, loops, and parallel branches—features common in technical procedures. Its dependency on cloud APIs raises similar privacy concerns, making it less suitable for secure or offline environments.

Collectively, these works illuminate the strengths and limitations of AI in procedural understanding, highlighting the need for an integrated, end-to-end system capable of semantic parsing, structural modeling, interactive visualization, and privacy-preserving local execution—thereby laying the groundwork for innovation in human-in-the-loop flowchart generation.

2.2 Research Gap

Based on the observations and thorough literature review, artificial intelligence has demonstrated substantial progress in natural language understanding and procedural text interpretation. However, several limitations and research gaps hinder its complete applicability for structured visual outputs, particularly in flowchart generation from unstructured documents. Traditional Retrieval-Augmented Generation (RAG) models, while effective for summarization, often fail to preserve the procedural logic, order, and dependencies required in diagrammatic representations. Moreover, the non-deterministic nature of large language models results in inconsistent outputs, where identical inputs may produce varying results—compromising accuracy and structural fidelity.

These limitations become critical in practical deployments where interpretability and repeatability are essential, such as regulatory workflows or system documentation. In addition, existing tools—such as ChatGPT-4o, NoteGPT, and Eraser Flowchart Maker—exhibit notable shortcomings. ChatGPT, for instance, lacks a native visualization engine and often generates static or cartoonish diagrams when integrated with external tools like Graphviz or Mermaid. These outputs are typically uninteractive and occasionally include hallucinated nodes or false procedural paths that do not exist in the source content. Eraser Flowchart Maker requires users to understand the document beforehand and manually specify what to include in the chart—limiting its ability to automate flowcharting across the entire document and reducing usability for non-experts. NoteGPT, while simple and accessible, suffers from token limitations (around 3000 tokens), lacks interactive refinement capabilities, and does not allow users to download or export the generated flowcharts. Furthermore, all three tools are cloud-based or subscription-restricted, raising concerns over data privacy, offline accessibility, and long-term sustainability.

There is also a notable research gap in the lack of systems that integrate structured preprocessing, controlled prompt engineering, and graph-based reasoning within a modular, human-in-the-loop framework. While LLMs are widely used for generation, their potential is often underutilized without mechanisms to guide, constrain, and refine their output. This thesis aims to develop a reproducible and locally deployable solution that leverages semantic retrieval, rule-based filtering, and DOT-based rendering to deliver interpretable and privacy-preserving procedural flowcharts—bridging the gap between unstructured text and structured visual understanding.

2.3 Theoretical Background

2.3.1 Large Language Models (LLMs) and Natural Language Understanding (NLU)

Definition and Characteristics

Large Language Models (LLMs) [4] are a class of deep learning models trained on vast amounts of textual data to perform complex natural language processing tasks. Built on the transformer architecture, these models—such as ChatGPT-40 [16], LLaMA [19], and Qwen [20] —use self-attention mechanisms to capture relationships between words across varying contexts, enabling them to generate coherent, context-aware language. At their core, LLMs perform natural language understanding (NLU) [3], which involves interpreting the structure, semantics, and intent of human language to carry out tasks such as summarization, question answering, classification, and reasoning.

In this thesis, LLMs are used to process unstructured procedural documents by identifying steps, extracting decisions, and revealing relationships between actions and conditions. Their ability to understand language allows them to simulate human-like comprehension and reorganize information into more structured representations. However, a critical limitation of LLMs is that they are constrained by the training data they were exposed to prior to a fixed cutoff date. Any knowledge or terminology introduced after this cutoff—or any domain-specific knowledge not present in the training corpus—may not be accurately understood or may be entirely missing from the model's output. This restricts the model's ability to reason about the most recent developments or proprietary systems unless supplemented by external information.

Applications and Limitations

Beyond academic research, LLMs have seen widespread adoption in real-world applications. They power AI chatbots for customer support, assist in legal and medical document analysis, automate software code generation, summarize long-form reports in enterprise settings, and provide creative content generation in journalism, marketing, and education. These diverse use cases highlight the versatility of LLMs, but also reinforce the importance of adapting them responsibly—especially in structured tasks like procedural flowchart generation where output consistency and logical correctness are paramount.

As powerful as LLMs are, their probabilistic nature introduces unpredictability in output, and their finite context window limits their ability to reason over large documents. Consequently, in this thesis, LLMs are not treated as end-to-end solvers but rather as components within a controlled pipeline—where retrieval, prompt design, and structural post-processing help ensure accurate and reliable flowchart construction.

2.3.2 Generative AI

Definition and Characteristics

Generative AI [21] refers to the class of artificial intelligence techniques that produce new content—such as text, images, or code—based on learned patterns from training data. Large Language Models fall within this category and are capable of generating coherent text by predicting the most likely next token given a context. This property is what allows them to simulate reasoning and produce step-by-step descriptions of a process.

Challenges and Integration in This Thesis

However, generative AI is inherently non-deterministic [10]: the same input may yield slightly different outputs upon repeated execution due to stochastic sampling methods. While such variability can be useful in creative domains, it introduces challenges in applications requiring structured, repeatable output. In procedural flowchart generation, this randomness may cause the AI to omit steps, infer incorrect relationships, or alter the structure with each pass.

Moreover, generative models do not inherently understand logic—they mimic it through pattern prediction. To compensate, this thesis uses a controlled generation pipeline in which generative models are paired with retrieval, filtering, and post-processing techniques to ensure the integrity and stability of the final output.

2.3.3 Graphviz and DOT Language

Concepts and Usage

Graphviz [5] is an open-source graph visualization tool designed to render structured information into readable and aesthetically optimized diagrams. It operates on input written in the DOT language, a domain-specific textual format used to describe graphs in terms of nodes, edges, and their various attributes. The DOT language allows users to define labels, shapes, directions, edge styles, clustering, and layout preferences. It supports both directed and undirected graphs and can represent complex relationships such as decision trees, hierarchical flows, and system architectures.

A defining strength of Graphviz lies in its layout optimization algorithms, which intelligently minimize edge crossings, prevent node overlaps, and maintain a coherent visual hierarchy. These features make it highly effective for visualizing flowcharts, dependency graphs, and process models. In the context of this thesis, Graphviz is used as a chart plotting engine—the procedural logic extracted from documents is translated into DOT code, which is then rendered by Graphviz into a polished flowchart. This translation from language to structure provides an essential bridge between AI comprehension and human-readable visualization.

Applications and Relevance to this Thesis

Graphviz has been adopted widely in both academic and industrial contexts due to its powerful capabilities and flexibility. It is often used for software architecture diagrams, compiler dependency trees, knowledge representation, and workflow modeling. A practical example close to this research is its use within educational settings—such as by the IT faculty at International University— as Fig. 2.1 to visualize course structures and curriculum pathways. By mapping prerequisite relationships and module sequences into a visual graph, Graphviz helps students and educators better understand program progression and dependencies.

In this thesis, Graphviz plays a critical role in the final step of the flowchart generation pipeline. Once the AI system has identified and structured procedural content, it outputs DOT code that encodes the visual logic of the process. Graphviz then renders this code into a clean, readable diagram, complete with step names, decision branches, and sub-process groupings. Graphviz is chosen for several key reasons:

- **Layout optimization:** Its layout engines—such as `dot`, `neato`, and `fdp`—automatically reduce edge crossings and node overlaps, ensuring a tidy and interpretable result.

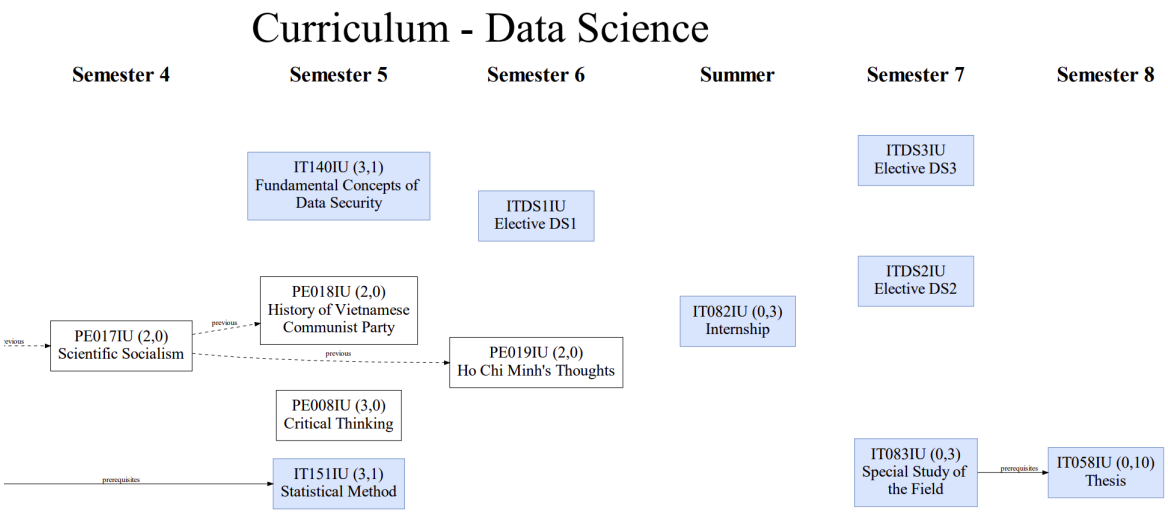

Figure 2.1: A section of Data Science Curriculum at International University.

- **Styling and customization:** Graphviz supports extensive customization, including styling for nodes and edges, subgraph nesting, and rank-based layout. This flexibility allows the system to visually distinguish between different types of entities such as actions, decisions, and actors.
- **Portability and integration:** It offers strong portability with support for various output formats such as SVG, PNG, and interactive HTML. This enables seamless integration with modern web-based tools and user interfaces.

Together, these features make Graphviz an indispensable visualization component in the proposed AI-driven system.

2.3.4 Prompt Engineering

Concepts and Techniques

Prompt engineering is the process of carefully designing and structuring input prompts to guide large language models (LLMs) in generating accurate, relevant, and task-specific outputs. Since LLMs are sensitive to how instructions are phrased and how context is framed, the quality of their responses often hinges on the clarity, completeness, and consistency of the input prompt. Prompt engineering transforms LLMs from generic language tools into task-aligned agents capable of simulating reasoning, extracting structured information, or following procedural logic.

In this thesis, prompt engineering is used to direct the LLM in identifying steps, conditions, and relationships embedded in procedural text. Two key prompting strategies are employed. The first is few-shot prompting, introduced by Brown et al. (2020) in their landmark paper "Language Models are Few-Shot Learners" [6], where the model is provided with a few example input-output pairs to guide its behavior on new tasks. This helps the model understand the expected format, structure, and granularity of the desired output without requiring fine-tuning. The second is chain-of-thought prompting, proposed by Wei et al. (2022) in "Chain of Thought Prompting Elicits Reasoning in Large Language Models" [22], which instructs the model to explicitly reason step-by-step before providing a final answer. This method improves the logical coherence of outputs, especially in

tasks involving multi-step reasoning or conditional logic. Both prompting techniques are adapted dynamically based on the nature of the input text and the current task phase in the flowchart generation pipeline.

Role in This Thesis

Prompt engineering plays a foundational role in the pipeline developed for this thesis, as it enables the controlled interaction between the AI system and the procedural content it is meant to interpret. By designing prompts that explicitly request structured outputs—such as numbered steps, actor-action pairs, or cause-effect relationships—the system can reduce the ambiguity and unpredictability often associated with generative AI. This is especially important in flowchart generation, where each node and edge must represent a specific, traceable component of the original document.

Additionally, prompt templates in this system are augmented with contextual information retrieved, making them context-aware and document-specific. Instead of asking the model to reason from scratch, the prompt includes semantically or structurally related fragments from the original document, which anchor the model's response in real content. This hybrid approach increases the reliability of the AI-generated output, minimizes hallucinations, and helps preserve the fidelity of the original procedural structure.

By combining well-crafted prompts with retrieval-based augmentation and consistent formatting constraints, this system leverages prompt engineering not just as an input mechanism, but as a control strategy to ensure that the outputs of the LLM are interpretable, logically sound, and ready for conversion into flowchart structures.

2.3.5 Embedding Models

Concept and Functionality

Embedding models [23] are a class of neural models designed to convert natural language into dense, fixed-size vector representations that capture semantic meaning. Unlike traditional bag-of-words or one-hot encoding methods, which are sparse and lack contextual awareness, embeddings encode the relative meaning of text elements into a continuous vector space. This allows similar meanings to be located close to one another, facilitating tasks such as semantic search, clustering, and retrieval. In essence, embedding models bridge the gap between symbolic language and numerical computation by transforming words, sentences, or documents into machine-readable formats that preserve contextual information.

Techniques and Mathematical Foundation

Modern embedding models, such as Sentence-BERT [24], extend traditional BERT-based [25] architectures by introducing a Siamese or triplet network structure, enabling the comparison of sentence-level embeddings. Given a sentence or document x, the embedding model maps it to a vector representation $\mathbf{v} \in \mathbb{R}^d$, where d is the embedding dimension (e.g., 384, 768, or 1024). This can be formally expressed as:

$$f(x) = \mathbf{v}, \quad \text{where } f : \text{text} \to \mathbb{R}^d$$

To compare the similarity between two texts x_1 and x_2, one can compute the cosine similarity between their respective vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ using the formula:

$$\text{cosine\_sim}(\mathbf{v}_1, \mathbf{v}_2) = \frac{\mathbf{v}_1 \cdot \mathbf{v}_2}{\|\mathbf{v}_1\| \, \|\mathbf{v}_2\|}$$

This similarity score ranges from -1 (opposite meanings) to 1 (identical meanings), and is commonly used in retrieval systems to rank the relevance of stored documents given a query vector, as illustrated in Fig. 2.2

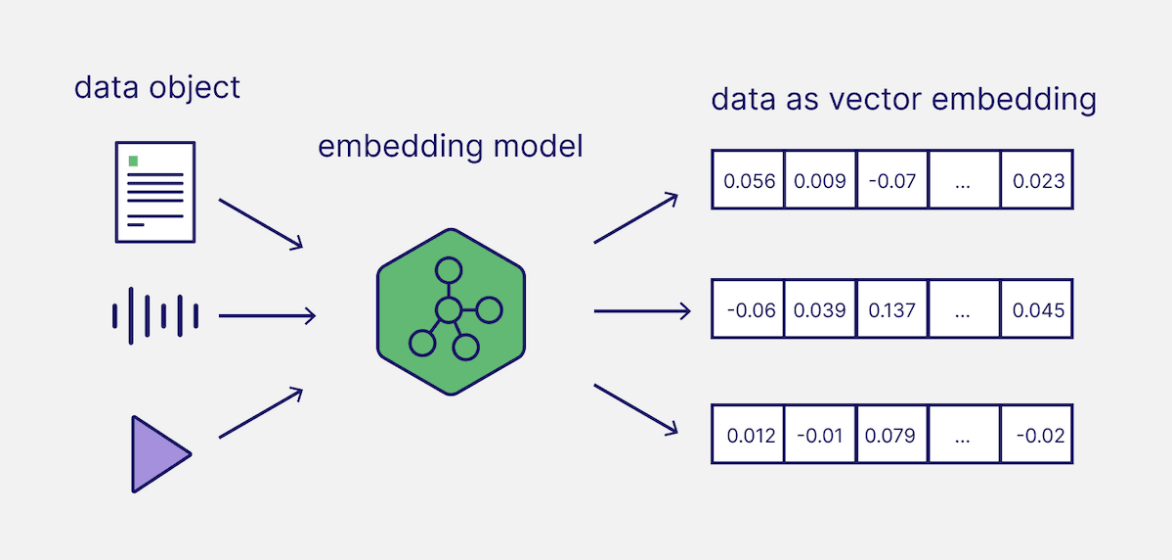

Figure 2.2: Illustration of a typical embedding model.

Embedding vectors are typically generated from the final hidden layer of a transformer encoder and pooled via mean or max aggregation over token embeddings. The result is a fixed-size vector that encapsulates both local and global contextual meaning, making it ideal for downstream tasks such as nearest-neighbor search in vector databases.

Applications in This Thesis

In this thesis, embedding models are used to encode both document fragments and query instructions into vectors for efficient semantic retrieval. When a procedural document is uploaded, it is segmented into smaller chunks—such as paragraphs or instructions—which are each transformed into embeddings and stored in a vector database. During the flowchart generation process, these vectors are queried using cosine similarity to identify the most semantically relevant content to include as prompt context for the large language model.

This approach addresses two key challenges. First, it allows the system to operate effectively on long documents by narrowing the attention of the LLM to only the most relevant parts. Second, it enhances the precision of flowchart generation by grounding the model's output in text that is semantically aligned with the current procedural focus.

By integrating embedding models with vector-based retrieval, the system gains the ability to perform semantic enrichment, content filtering, and localized context injection—ultimately improving the quality and accuracy of the AI-generated flowcharts.

2.3.6 Vector Databases and Graph Databases

Concepts and Roles

To support efficient retrieval and contextual reasoning, this thesis employs both vector databases and graph databases as complementary components in the information access

layer of the system. These two database paradigms differ fundamentally in structure and query logic, but together they provide the semantic and logical backbone required to support meaningful flowchart generation.

A vector database is designed to store and query high-dimensional embeddings—dense numerical vectors that encode the semantic meaning of textual content. Vector databases enable approximate nearest neighbor (ANN) search [26], which is essential when working with large collections of document fragments. Given a user query or intermediate representation of a procedural step, the system computes an embedding and retrieves the most semantically similar content using similarity metrics. Popular vector database frameworks include FAISS [27], which is widely adopted for its performance and scalability in large-scale retrieval systems.

In contrast, a graph database represents data as nodes (entities) and edges (relationships), allowing explicit modeling of structure, hierarchy, and logic. This makes graph databases well-suited for procedural workflows, where steps follow ordered sequences or branch based on decision conditions. A graph model allows the system to traverse dependencies and ensure procedural correctness, for example by identifying which steps precede or follow a decision node. Well-known graph database platforms include Neo4j [28] and systems adhering to property graph or RDF models [29].

Mathematical Formalization

Formally, the embedding model maps a textual input x (e.g., a sentence or paragraph) into a fixed-size vector $\mathbf{v} \in \mathbb{R}^d$, where d is the dimensionality of the embedding space. Let $\mathbf{q}$ be the query vector, and $\mathbf{v}_i$ be a document fragment in the database. The cosine similarity between the query and document vectors is computed as:

$$\text{cosine\_sim}(\mathbf{q}, \mathbf{v}_i) = \frac{\mathbf{q} \cdot \mathbf{v}_i}{\|\mathbf{q}\|\|\mathbf{v}_i\|}$$

The system ranks all vectors $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$ by similarity to $\mathbf{q}$ and returns the top-k results. This allows the AI to focus on semantically relevant content when generating or refining a flowchart.

For the graph database, procedural logic is modeled as a directed graph $G = (V, E)$, where V is a set of nodes representing steps, decisions, or entities, and $E \subseteq V \times V$ defines directed edges between them. Given a current node $v \in V$, the set of immediate successors is defined as:

$$\text{Next}(v) = \{u \in V \mid (v, u) \in E\}$$

Traversal algorithms such as breadth-first search (BFS) or depth-first search (DFS) can be applied to explore sub-processes or resolve all downstream steps. These graph operations are used to maintain and enforce logical structure in the flowchart output, ensuring that all transitions and branches conform to the original procedural intent.

Applications and Illustration

In this thesis, the integration of vector and graph databases supports a hybrid retrieval model. The vector database enables flexible semantic search, allowing the system to locate document segments that are contextually aligned with the current task. Meanwhile, the graph database enforces structural logic by maintaining explicit representations of procedural flow and relationships. This dual-retrieval mechanism enhances both the content relevance and logical validity of the resulting AI-generated flowcharts.

The vector database reduces the search space for relevant content, ensuring that the language model works only with high-signal data. The graph database, in turn, validates relationships, orders transitions, and supports decision-tree branching in the visual output. Together, they create a robust knowledge base that supports both natural language reasoning and procedural integrity in flowchart generation.

2.3.7 Baseline Retrieval-Augmented Generation (RAG)

Concept and Publication

Retrieval-Augmented Generation (RAG) [9] is a method introduced by Lewis et al. (2020) in their paper *"Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks"*. RAG enhances large language models by combining two components: a retriever and a generator. The retriever, usually implemented as a vector search engine, identifies relevant passages from a knowledge base, while the generator, typically a language model like BART [30] or GPT, produces answers or outputs based on the retrieved passages.

How It Works

RAG begins by embedding the user's query using a dense encoder and performing a similarity search over a vector store. The top-k matching passages are appended to the input prompt fed into the language model. The model then uses this expanded context to generate a more accurate, informed, and grounded output. RAG has proven highly effective in tasks such as question answering, summarization, and open-domain reasoning.

Application in This Thesis

In this thesis, baseline RAG is employed to enrich prompts with relevant procedural context retrieved from segmented document embeddings. This augmentation enables the language model to better understand the specific task at hand by focusing its reasoning on semantically relevant content, rather than processing the entire document indiscriminately.

Limitations

Despite its strengths, baseline RAG exhibits several key limitations:

- It retrieves based purely on vector similarity, ignoring the logical or structural relationships between document segments.
- It may introduce redundant or conflicting information if retrieved passages are semantically close but procedurally unrelated.
- It does not differentiate between the semantic value and the procedural function of a passage, which is critical in flowchart generation tasks.

2.3.8 Lightweight Retrieval-Augmented Generation (LightRAG)

Concept and Motivation

LightRAG is a recent retrieval-augmented generation framework introduced by Guo et al. (2024) in *"LightRAG: Simple and Fast Retrieval-Augmented Generation"* [11]. The

method is designed to maintain the benefits of retrieval-augmented generation while significantly simplifying the architecture and reducing memory and latency overhead. Unlike traditional RAG systems, which typically rely on heavy-weight retrievers or complex multi-stage indexing, LightRAG emphasizes a lightweight design optimized for speed, efficiency, and ease of deployment.

Key Techniques and Innovation

The core idea of LightRAG is to streamline the retrieval process without sacrificing relevance. It adopts a simpler memory system and eliminates the need for multiple retriever-generator iterations. In this thesis, the LightRAG framework is adapted and extended to handle procedural content by introducing a dual retrieval mechanism, including:

- **Semantic retrieval**, using dense vector similarity to locate contextually relevant document fragments.
- **Structural retrieval**, leveraging graph traversal techniques to extract steps that are logically connected within a procedural workflow.

Together, these retrieval paths ensure that the information passed to the language model is both semantically rich and procedurally coherent.

Application in This Thesis

In this work, LightRAG [11] functions as a lightweight but highly effective context augmentation module. When generating flowchart nodes, it retrieves document segments that match the current step's meaning and logical position in the process. These segments are obtained by converting the input document into both vector and graph data structures—where the graph explicitly models entities, objects, and their relationships. By maintaining this structural representation, LightRAG ensures that the contextual snippets preserve the procedural roles and dependencies inherent in the original text. Retrieved results are then ranked, filtered, and composed into an input window that guides the large language model, enabling it to produce outputs that faithfully reflect the intent and structure of the original document without relying on large or complex retrieval infrastructures.

Advantages over Baseline RAG

Compared to baseline RAG in Table 2.1, LightRAG is faster and easier to scale, while also supporting task-specific modifications like dual retrieval. Its modular design allows for greater control over retrieval, enabling the integration of graph-based logic essential for flowchart generation. Furthermore, LightRAG's simplicity improves transparency and interpretability—critical in use cases where structural precision is required. In this thesis, its adoption enables an AI system that is both procedurally aware and computationally efficient.

2.3.9 Human-in-the-Loop

Concept and Rationale

The human-in-the-loop (HITL) [8] paradigm refers to a collaborative model in artificial intelligence where human users intervene in the decision-making or output validation process of an AI system. This approach is particularly valuable in domains where automated

Table 2.1: Comparison between Baseline RAG and LightRAG.

Feature	**Baseline RAG**	**LightRAG**
First Introduced By	Lewis et al., 2020	Guo et al., 2024
Retrieval Type	Vector similarity (embedding only)	Dual: vector similarity + graph traversal
Context Precision	Moderate (topical relevance)	High (topical + structural relevance)
Structural Awareness	None	Yes (graph-aware retrieval)
Processing Pipeline	Coupled (retrieve → generate)	Decoupled (retrieve → filter → generate)
Noise Handling	Limited filtering	Heuristic filtering and ranking
Use Case Fit	Open-domain QA, summarization	Procedural reasoning, flowchart generation
Computational Overhead	Lower	Slightly higher, but optimized for locality

reasoning must be audited for correctness, clarity, or contextual accuracy. While large language models (LLMs) are capable of generating semantically rich and coherent responses, they can also produce hallucinated or structurally inconsistent content—especially when interpreting complex, ambiguous procedural documents, as illustrated in Fig. 2.3.

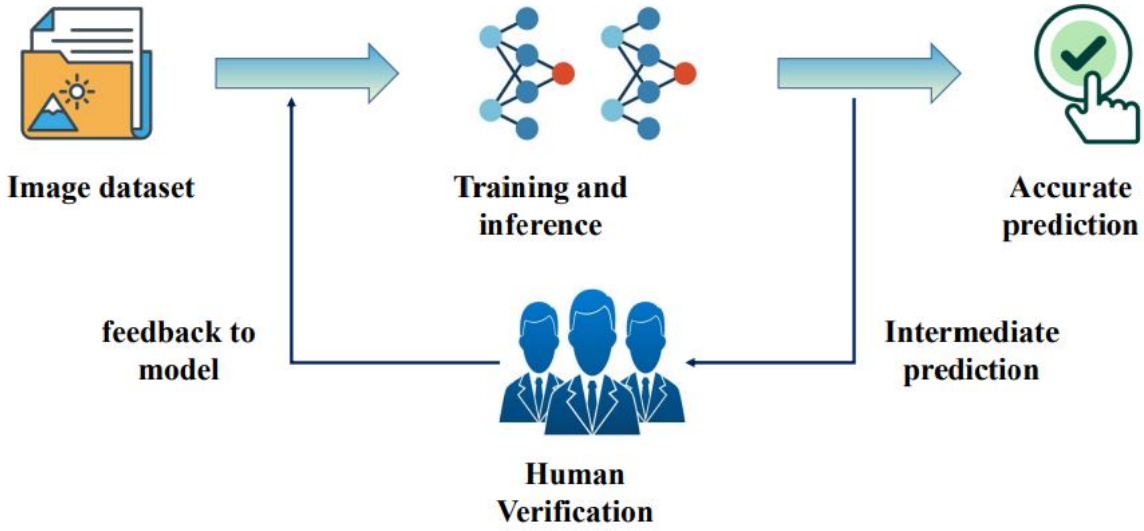

Figure 2.3: Human in The Loop Flow.

In the context of procedural flowchart generation, human verification ensures that every element of the AI-generated flowchart reflects the original intent and meaning of the source document. The verifier—typically a domain expert—reviews the generated flowchart, identifies inaccuracies, and provides corrections or refinements. This interactive loop enables the system to combine the efficiency of automation with the precision of expert oversight, reducing the risk of critical errors and enhancing trust in the final output.

Role in This Thesis

In this thesis, the human-in-the-loop mechanism is implemented as a core component of the **IUFlowGen** system. After the procedural document is processed and a flowchart is generated by the AI pipeline, the result is rendered interactively in the frontend UI. Both document providers and expert verifiers are able to zoom, pan, and explore the generated flowchart. Users may issue clarification queries through a built-in query system, while verifiers assess whether the procedural logic, node definitions, and transitions accurately represent the source material.

The expert verifier, often a subject matter specialist, may choose to accept the flowchart as-is, suggest modifications, or manually revise segments that do not meet the expected

procedural standard. Although editing capabilities are currently work-in-progress, the design of the system anticipates an extension where users can directly add, remove, or restructure nodes and edges through the interface—transforming the system from a read-only assistant into a co-creative design tool.

The HITL process is also integral to the experimental evaluation conducted in this thesis. In particular, a group of participants with AI assistance received system-generated flowcharts which they could refine under time constraints, while a control group created flowcharts from scratch. The comparison of performance across these groups underscores the practical benefit of combining AI automation with expert validation.

By embedding human feedback into the procedural modeling pipeline, IUFlowGen ensures not only technical robustness but also semantic alignment, enabling the system to scale across diverse procedural domains while retaining human interpretability and oversight.

Chapter 3
METHODOLOGY

The methodology chapter outlines a structured approach to addressing the research gap in procedural document visualization through artificial intelligence. Beginning with a requirement analysis, this chapter identifies key challenges in interpreting unstructured procedural texts and converting them into structured visual formats. A hybrid AI-driven system is then introduced, incorporating natural language understanding, LightRAG-based dual retrieval, graph-based modeling, and DOT-language visualization. Each phase of the pipeline—spanning text preprocessing, entity and relationship extraction, contextual retrieval, graph construction, and flowchart rendering—is presented in detail. Additionally, prompt engineering is employed to guide the behavior of the language model, enhancing consistency, reasoning quality, and control. This methodology formalizes the design of an interactive system capable of producing semantically meaningful flowcharts, while maintaining scalability, structural fidelity, and user data privacy.

3.1 Requirement Analysis

The requirement analysis defines the key roles, components, and processes involved in transforming procedural documents into structured flowcharts. It identifies the system's stakeholders, outlines their expectations, and clarifies the flow of interaction between the AI model and the user. As a human-in-the-loop system, it ensures that automated outputs are verified and refined by users to maintain accuracy and relevance. This analysis ensures that the system delivers meaningful, editable visualizations while meeting usability, consistency, and scalability requirements.

3.1.1 Use Case Description

The proposed AI-assisted flowchart generation system involves two main actors:

- **Users (Document Providers and Reviewers)**: Users are the primary actors interacting with the system. They are responsible for submitting unstructured procedural documents, which may originate from technical, legal, or administrative domains. These users may include engineers, analysts, researchers, or policy writers who aim to visualize complex workflows for better communication or operational clarity.

  Beyond providing input, users also explore the resulting AI-generated flowcharts through interactive features such as zooming, panning, and querying specific nodes or edges to gain procedural insight.

- **AI Engine**: The AI engine is the central computational component that processes the uploaded documents. It performs natural language understanding (NLU), procedural segmentation, information retrieval, and flowchart rendering. This module converts complex textual instructions into structured, interpretable diagrams, enabling faster comprehension and easier communication of procedural logic.

Each actor plays a critical role in the system's functionality. The user initiates the process by uploading a document and completes it by validating and refining the resulting flowchart. The AI engine acts as the intermediary, transforming unstructured content into a visual format suitable for analysis, communication, or decision-making. Table 3.1 outlines the key functions of each actor within the system.

Table 3.1: Description of the use cases and their actors.

Actor	Use Case Name	Use Case Description
User (Document Provider & Viewer)	Submit Document	The user uploads unstructured procedural text to the system for automated analysis and visualization.
	Explore Flowchart	The user interacts with the resulting flowchart using tools such as zooming, panning, and querying individual nodes or edges to understand procedural structure.
AI Engine	Generate Flowchart Suggestion	The AI engine processes the uploaded document using natural language understanding, information retrieval, and procedural reasoning to generate a structured, editable flowchart draft.

3.1.2 Use Case Diagram

The provided diagram provides a comprehensive overview of the system's functional requirements, elucidating the tasks it supports and the respective roles of users and stakeholders. It is a collaborative communication tool that clarifies the understanding of user roles and interactions with the system's functionalities. This enables the definition of system boundaries and ensures alignment with user expectations throughout the development process, as illustrated in Fig. 3.1.

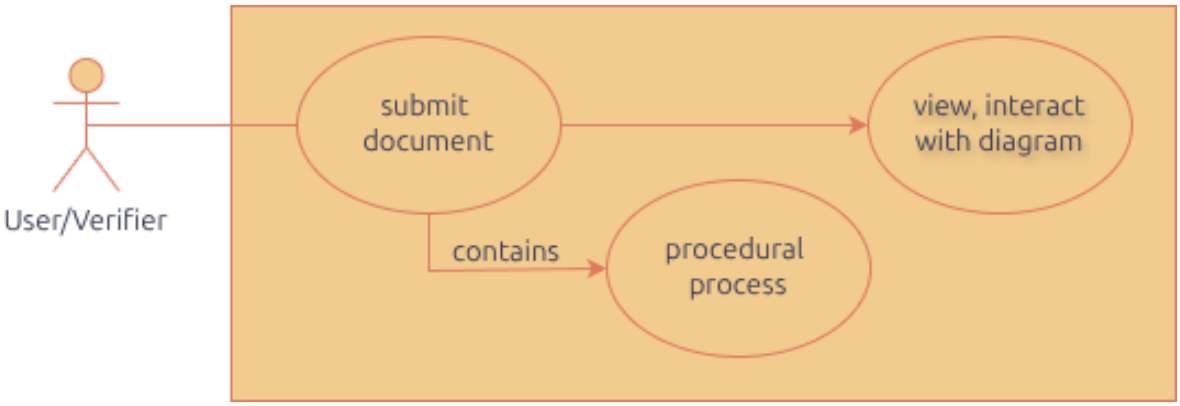

Figure 3.1: IUFlowGen use case diagram.

3.1.3 System Overview

The proposed AI-assisted flowchart generation system, IUFlowGen, operates through a human-in-the-loop architecture that combines automated document processing with expert validation. Figure 3.2 illustrates the complete workflow, which involves three key actors: the document provider (user), the AI system, and the verifier (expert reviewer).

The process begins when the user uploads a procedural document to the system. Although the document is submitted document (in PDF fomrat), it typically contains implicitly structured procedural content. The AI system takes in this document, interprets it

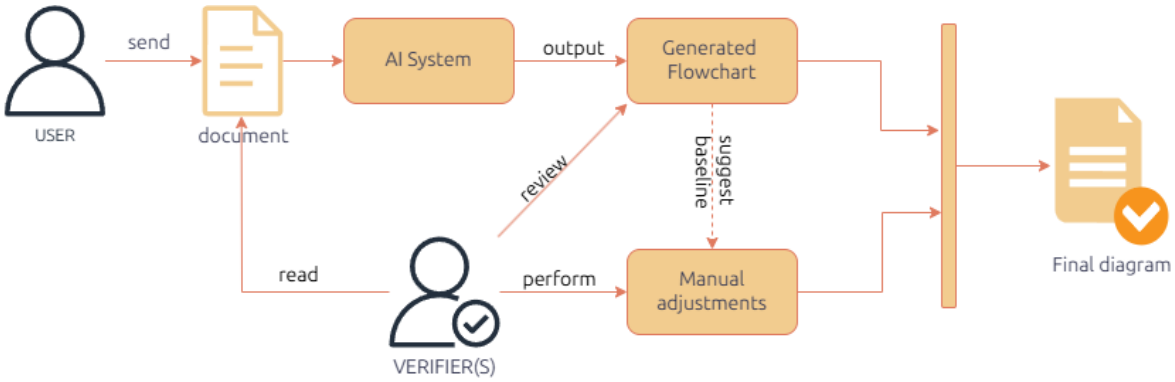

Figure 3.2: IUFlowGen system overview.

using large language models, retrieval-augmented generation, and graph construction techniques, and produces a preliminary flowchart representing the detected steps, entities, and interdependencies.

This auto-generated flowchart is then forwarded to the verifier for expert review. The verifier reads the original document alongside the suggested flowchart and evaluates its accuracy and structural logic. If any inconsistencies or omissions are found, the verifier performs manual adjustments—modifying the layout, nodes, or connections to better reflect the source material. These adjustments are guided by the flowchart generated by the AI, which serves as a suggested baseline for refinement.

After validation and refinement, the adjusted diagram is finalized and confirmed. This output represents the verified procedural logic of the original document and can be exported or deployed as the official flowchart. By supporting both automatic generation and manual correction, IUFlowGen ensures that the output is not only efficient but also trustworthy, interpretable, and suitable for complex procedural documentation.

3.2 Functional Decomposition of the System

Formally, the system's core functionalities can be broken down into two fundamental tasks: drawing the flowchart and querying for clarification. These tasks are implemented through specialized modules that collaborate to construct the final procedural visualization and enable interactive exploration. The syntax for these operations is provided below:

- **Initialize(AIEngine, D)**:
  Takes as input the AI engine (which includes the large language model and the retrieval-augmented generation modules) and the procedural document `D` in text format. Prepares the system by embedding the document into vector space, constructing graph indices if necessary, and setting up the retrieval and generation pipeline for subsequent operations.

- **GenerateFlowchart(Sys, D)**:
  Uses the initialized system `Sys` and the procedural document `D` to extract procedural steps, decisions, and logical relationships. The outputs are assembled into a structured flowchart consisting of nodes and directed edges, visually representing the procedural logic.

- **QueryElement(q)**:
  Takes as input a user query `q`, representing a clarification request or an exploration

command. The system internally matches the query against the document fragments and procedural structures it has processed, and returns a contextual explanation, a relevant excerpt from the document, or inferred logic associated with the query.

These tasks are orchestrated to ensure that the system accurately reconstructs procedural flows from textual input while maintaining usability and interpretability. The flowchart generation task captures the structural essence of the procedural document, while the querying task enables dynamic, user-driven interaction for enhanced understanding and validation.

3.3 IUFlowGen Algorithms

3.3.1 Problem Modeling

The central objective of IUFlowGen is to generate an accurate and interpretable flowchart from a complex procedural document. This process involves identifying procedural steps, extracting their internal structure, and mapping their relationships into a coherent, directed graph. The proposed solution builds on the foundation of Natural Language Understanding (NLU), leveraging Large Language Models (LLMs) to interpret text and using Retrieval-Augmented Generation (RAG) techniques to enrich and control the generation process. Through prompt engineering and context injection, the system guides the AI to produce structured outputs aligned with user intent.

However, this approach introduces a significant set of challenges. Large Language Models (LLMs), while powerful in parsing and generating language, are inherently non-deterministic and difficult to control. Even with a well-structured prompt, an LLM may produce rational-sounding but logically inconsistent outputs—frequently omitting critical procedural elements, altering step dependencies, or hallucinating nonexistent actions. This issue is magnified when handling long procedural documents that involve complex structures such as conditional branches, merges, parallel execution, and domain-specific terminology. As the input length increases, LLMs struggle to retain global context, resulting in fragmented understanding and procedural breakdowns.

To address these issues, this thesis proposes a structured five-step computational method designed to systematically extract, enrich, and organize procedural knowledge from unstructured text. This method is grounded in principles from Business Process Modeling (BPM)—an established framework used in enterprise systems for analyzing, modeling, and automating workflows. BPM emphasizes three key dimensions: the decomposition of tasks into atomic steps, the identification of involved roles and entities, and the mapping of process flow and decision points. By aligning with this structured modeling philosophy, IUFlowGen ensures both interpretability and correctness in flowchart generation. The details of this five-step approach are presented in the following section.

3.3.2 Formalizing for Diagram Building

To implement the aforementioned modeling framework, IUFlowGen decomposes the flowchart generation task into five sequential, verifiable steps. Each step builds upon the outputs of the previous one, gradually transforming unstructured procedural content into a formal graph representation. This modular design ensures greater control over LLM behavior and facilitates the validation of intermediate outputs, reducing the risk of hallucinations or omissions.

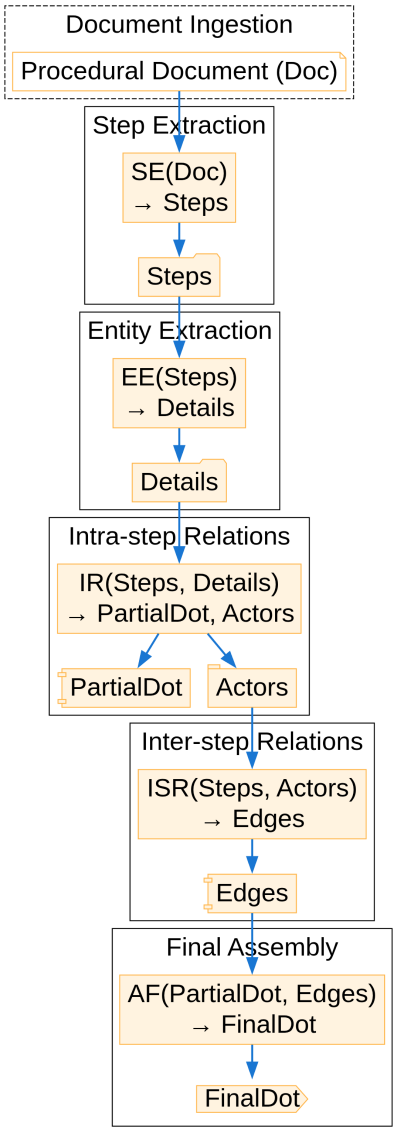

Figure 3.3: Flowchart generation pipeline.

The IUFlowGen system consists of the following five core functions:

- `Steps` ← `SE(Doc)`:
  The first step focuses on identifying the main procedural steps from the input document `Doc`. It uses LightRAG—a lightweight retrieval-augmented generation framework—to retrieve semantically relevant chunks of text and prompt the LLM to summarize or extract the core actions. Each detected step is passed through a controlled generation phase that outputs corresponding DOT code snippets, each representing a subgraph. This lays the foundation for the flowchart structure by isolating the atomic procedural units.

- `Details` ← `EE(Steps)`:
  The second step enriches each identified step by extracting associated entities, actors, actions, and attributes. This enrichment is guided by prompt engineering and LightRAG-based context injection. The output is a dictionary-like structure, where each step acts as a key and its value contains all relevant semantic details extracted from the document. This layer of enrichment transforms abstract steps into con-

crete actions with clearly defined participants and contextual attributes, mimicking the BPM emphasis on role-task mapping.

- `PartialDot`, `Actors` ← `IR(Steps, Details)`:
  The third function generates intra-step relationships. Given the enriched list of steps and their semantic contents, the system uses prompt-driven reasoning to infer internal connections within each step—for example, linking an actor to an action or identifying dependencies among sub-tasks. Each step is rendered as a subgraph in DOT language, capturing the internal procedural dynamics. The outputs at this stage include `PartialDot`, a list of formatted DOT subgraphs, and `Actors`, a global list of all agents involved across steps. This stage ensures that each step is internally consistent and visually well-formed before they are linked together.

- `Edges` ← `ISR(Steps, Actors)`**: (Core Logic Step)**
  The fourth step is the centerpiece of the entire pipeline and the most critical for ensuring overall procedural coherence. While previous steps build up the individual subgraphs, this stage connects them into a single logical structure. It performs inter-step relationship identification, leveraging the full spectrum of available information: vector-based semantic similarity (via embedding models), graph traversal data (from earlier intra-step mappings), and text-based cues retrieved through LightRAG. The system uses these inputs to determine the temporal and logical sequence of steps—for example, which steps follow from others, which represent conditional branches, and which occur in parallel. It then generates DOT edges that define how each subgraph is connected to others, incorporating attributes like `ltail` and `lhead` to preserve procedural alignment. This step is essential because it transforms a collection of isolated tasks into a structured and navigable process, reflecting core BPM concepts such as dependency modeling and transition mapping. Without this step, the resulting flowchart would remain a disjointed collection of fragments with no interpretive value.

- `FinalDot` ← `FA(PartialDot, Edges)`:
  The final step assembles the outputs from previous stages into a complete and syntactically valid DOT file. It combines the intra-step subgraphs (`PartialDot`) with the inter-step transitions (`Edges`) and adds necessary global syntax elements like the `digraph` declaration, cluster definitions, and styling attributes (e.g., `compound=true`). The output is a fully-rendered DOT code file that can be directly processed by Graphviz or similar visualization tools. This final stage ensures that the structure remains coherent, interpretable, and presentation-ready—supporting further refinement or integration into user-facing interfaces.

To reinforce the practicality of the proposed pipeline, Table 3.2 summarizes how each step contributes to the overall objective of accurate and interpretable flowchart generation from complex procedural documents.

Table 3.2: Justification of IUFlowGen Pipeline Design.

Pipeline Step	**Justification**
`SE(Doc)` — Step Extraction	Identifies core actions from unstructured text, providing a structured starting point for flowchart generation.
`EE(Steps)` — Entity Enrichment	Captures relevant actors, entities, and roles, adding semantic context to each step.
`IR(Steps, Details)` — Intra-Step Relationships	Organizes internal logic of each step, enabling modular subgraph generation.
`ISR(Steps, Actors)` — Inter-Step Relationships	Establishes accurate flow between steps using structured and semantic clues; ensures logical consistency.
`FA(PartialDot, Edges)` — Final Assembly	Combines all parts into a complete and renderable diagram, ready for visualization and user interaction.

In conclusion, this five-step pipeline addresses the major limitations of LLMs in procedural modeling by combining AI-driven text interpretation with rule-based post-processing and structured flowchart construction. Rooted in Business Process Modeling principles, it ensures modularity, control, and reliability—enabling high-fidelity, interpretable flowchart generation from complex textual inputs.

3.3.3 Formalizing the Query Function

Additionally, beyond flowchart generation, the IUFlowGen system supports an interactive querying feature that allows users to explore and clarify the procedural content represented in the flowchart. This functionality is essential for enabling human-in-the-loop collaboration, where users and experts can validate, interpret, or expand upon specific elements of the visualized process.

The querying capability is defined by the following function:

- `Answer` $\leftarrow$ `Query(q)`:
  A system function that takes as input a user query q, typically phrased as a natural language question related to a specific procedural step, actor, or transition. The system first semantically embeds the query and performs a similarity search over the original procedural document fragments and enriched context stored during the flowchart construction phase.

  Using a combination of retrieval-augmented generation and context-aware prompting, the system synthesizes a response that provides clarification, elaboration, or justification for the queried element. This may include textual explanations, extracted source passages, or logical reasoning derived from the AI engine.

  The output `Answer` is a human-readable explanation aligned with the content and structure of the original document, supporting end-user understanding and expert validation.

This query module operates independently of the flowchart rendering engine, but draws upon the same knowledge base and indexed content extracted during the flowchart con-

struction process. It ensures that users can interactively explore the visualized workflow, resolve ambiguities, and reinforce the interpretability of AI-generated outputs.

Chapter 4
PROTOTYPING

This chapter presents the implementation of the IUFlowGen prototype, grounded in the methodology described in the preceding chapter. Building a functional AI-powered flowchart generation and query system requires the integration of multiple technologies across natural language processing, retrieval-based augmentation, and graph visualization.

To realize these capabilities, a modular system architecture is developed—designed to support both flowchart generation and human-in-the-loop query interaction. Each component in the stack, including the language model interface, retrieval pipeline, DOT code assembly logic, and visualization layer, plays a distinct role within the overall workflow. A clear understanding of how these components operate and interact offers insight into the system's robustness, extensibility, and alignment with the research objectives.

4.1 Strategies Purpose Choices

The rationale behind selecting specific technologies and design strategies in building the IUFlowGen system is to ensure a reliable, extensible, and interactive framework for procedural flowchart generation and querying. Each core component—retrieval augmentation, large language model orchestration, DOT-based visualization, and front-end integration—has been chosen to address the system's dual objectives: structured automation and human-in-the-loop usability.

IUFlowGen leverages modern AI tooling such as Python-based pipelines, Nano Vectordatabase for efficient vector search, Graphviz for rendering graph-based structures, and Streamlit to deliver a lightweight, interactive web-based interface. These technologies were selected for their maturity, ecosystem support, and seamless compatibility across modular AI workflows. In addition, retrieval-augmented generation strategies (specifically LightRAG) are adopted to maximize contextual relevance and minimize hallucination in large language model outputs.

These implementation choices reflect IUFlowGen's mission to create a privacy-conscious, locally deployable, and domain-agnostic system. The architecture ensures clarity, accuracy, and responsiveness in both automated flowchart generation and user-driven clarification. Together, these strategies form a cohesive framework capable of scaling with future enhancements while maintaining fidelity to the procedural integrity of input documents.

4.2 Front End Tools

4.2.1 Streamlit

Streamlit is a modern Python-based web framework that simplifies the development of interactive applications, particularly in the fields of data science and machine learning. It allows developers to build intuitive front-end interfaces using pure Python without requiring advanced web development skills. For IUFlowGen, Streamlit serves as the main front-end engine, supporting real-time input submission, output rendering, and interaction with AI-generated flowcharts.

Designed for rapid prototyping and deployment, Streamlit streamlines the connection between backend logic and user interaction. Its automatic UI generation, seamless widget integration, and support for live updates make it ideal for human-in-the-loop systems where user feedback is essential. In contrast to traditional full-stack web frameworks, Streamlit reduces overhead by focusing on simplicity, maintainability, and speed.

Within the IUFlowGen system, Streamlit allows the front end to remain lightweight and responsive, while still supporting complex integration with language models, retrieval engines, and graph visualization modules. It enables a unified user interface that handles both procedural document submission and interactive flowchart review. The use of Streamlit ensures that IUFlowGen remains locally deployable, easily extendable, and immediately usable by both technical and non-technical users.

4.2.2 D3.js

D3.js (Data-Driven Documents) is a powerful JavaScript library designed to produce dynamic, interactive, and highly customizable visualizations in web browsers using HTML, SVG, and CSS. In IUFlowGen, D3.js is embedded directly within the Streamlit interface to enhance the interactivity of the generated procedural flowcharts. By enabling features such

as zooming, panning, tooltip display, and interactive node traversal, D3.js transforms static DOT-generated visualizations into an exploratory canvas. Users can focus on individual procedural elements, follow conditional branches, and inspect actors or entities involved in each sub-process. This significantly improves the usability of the system, particularly in expert review workflows where interpretability is key.

Unlike pre-packaged visualization solutions, D3.js offers fine-grained control over every element of the rendered graph, enabling IUFlowGen to adapt the layout, styling, and interactivity to match procedural semantics. Its integration allows for dynamic updates without page reloads, and it ensures that flowcharts are not just readable—but explorable. D3.js thus plays a crucial role in bridging the gap between raw procedural content and meaningful human insight, directly contributing to the system's transparency and accessibility.

4.3 Back End Tools

4.3.1 Python

Python serves as the foundational programming language for the IUFlowGen system. Its versatility, readability, and extensive library ecosystem make it an ideal choice for implementing complex AI workflows that involve natural language processing, graph construction, data handling, and web interface development. Every core component of IUFlowGen—from embedding pipelines to visualization modules—is written in Python.

The language's strong integration with machine learning frameworks (such as Langchain and Yolo), graph libraries (like NetworkX), and web tools (e.g., Streamlit) facilitates rapid development and iteration. Python's dynamic typing and scripting capabilities also allow developers to quickly prototype and test different document parsing strategies, prompt engineering techniques, and retrieval mechanisms.

Python's ecosystem includes efficient tools for embedding generation, semantic similarity computation, graph traversal, and regular expression parsing. These tools are central to IUFlowGen's procedural document analysis pipeline. Additionally, Python's compatibility with remote execution tools like Paramiko and visualization libraries like Graphviz ensures that the entire system remains unified under a single language runtime.

For IUFlowGen, Python not only acts as the orchestration layer but also as the experimentation environment, enabling reproducibility, extensibility, and long-term maintainability. Its popularity in research and industry ensures ongoing support and an evolving ecosystem of compatible tools.

4.3.2 LightRAG

LightRAG (Lightweight Retrieval-Augmented Generation) is a streamlined adaptation of the traditional RAG (Retrieval-Augmented Generation) architecture designed to reduce latency, simplify system complexity, and increase retrieval control. Originally introduced by Guo et al. (2024), LightRAG focuses on accelerating inference while preserving the core benefits of context retrieval and knowledge grounding. In IUFlowGen, LightRAG plays a foundational role in supplying large language models with task-relevant, semantically rich, and structurally coherent content extracted from procedural documents.

Unlike conventional RAG frameworks that rely solely on embedding similarity and perform multiple cross-attention steps over retrieved passages, LightRAG implements a dual-retrieval strategy. It combines vector-based semantic search (via NanoVector) with structure-aware graph traversal (via NetworkX) to identify both topically relevant and procedurally linked document fragments. This dual mechanism improves both retrieval accuracy and the logical consistency of the LLM output, especially in tasks involving multi-step reasoning and flowchart construction.

The lightweight nature of LightRAG enables integration into local environments without dependency on cloud APIs or large retriever-generator pipelines. Its decoupled architecture separates retrieval from generation, giving IUFlowGen better control over what content is passed into the model prompt and reducing the chance of hallucinations or context drift. Additionally, LightRAG supports context filtering, prompt compression, and fallback behavior—features essential for working within LLMs' context window constraints.

In IUFlowGen, LightRAG is used across multiple stages: step extraction, actor identification, clarification queries, and inter-step relation discovery. Each invocation dynamically selects the most relevant slices of procedural content to ground the language model's reasoning process. This controlled augmentation leads to more stable, reproducible, and interpretable outputs compared to end-to-end free-form LLM prompting.

The system's integration of LightRAG aligns with its overall goals of privacy preservation, local deployment, and task-specific reliability. By limiting retrieval to a curated local vector index and leveraging structural heuristics, LightRAG ensures that IUFlowGen delivers semantically grounded and structurally aligned flowchart representations of procedural documents.

4.3.3 NanoVector

NanoVector is a lightweight, in-memory vector database developed to support efficient local embedding retrieval for semantic search applications. Unlike heavier alternatives such as FAISS or Milvus, NanoVector is designed for small to medium-scale use cases where simplicity, transparency, and local control are prioritized. In IUFlowGen, NanoVector is used to index and retrieve semantically relevant segments from procedural documents using embedding vectors generated from sentence encoders.

NanoVector's primary role is to serve as the backend for the LightRAG retrieval system. Once a procedural document is embedded into high-dimensional vectors, NanoVector stores these representations and allows for top-k nearest neighbor retrieval using cosine similarity. This process is invoked during both the flowchart generation and query clarification stages, where it is essential to provide the language model with the most relevant context within its token window constraints.

The system supports vector insertion, batch indexing, and filtered retrieval operations, and it is tightly integrated into IUFlowGen's Python stack. Since NanoVector operates fully in RAM and avoids complex disk-backed storage mechanisms, it offers exceptionally fast lookup speeds for smaller corpora, making it ideal for single-user or research-focused workflows.

Its simplicity and transparency also enhance reproducibility and debugging. Developers can inspect the vector store, override embeddings, and tune similarity thresholds without dealing with opaque optimizations. Furthermore, NanoVector's ease of integration with Python's NumPy and scikit-learn libraries makes it a flexible choice for rapid experimentation.

Overall, NanoVector provides a fast, minimalistic, and self-contained retrieval solution for IUFlowGen, enabling responsive and privacy-conscious document augmentation that does not depend on external APIs or cloud-based infrastructure.

4.3.4 NetworkX

NetworkX is a Python library used for the creation, manipulation, and analysis of complex network graphs. In IUFlowGen, it plays a critical role in representing and reasoning over

procedural structures, especially when modeling dependencies between actors, actions, and decisions. While the final visualization is handled by Graphviz, NetworkX is responsible for the underlying logic and traversal of the graph structure.

After entities and actors are extracted from document segments, NetworkX is used to build directed graphs (`DiGraphs`) that encode relationships within and between procedural steps. Nodes represent actions or entities, and edges define conditional or sequential transitions. This internal graph serves as an intermediate representation before generating the final DOT code for visualization.

IUFlowGen uses NetworkX for several algorithmic tasks: identifying isolated or redundant nodes, verifying cyclic dependencies, and computing reachable subgraphs. These operations help ensure that the flowchart accurately reflects the logic of the input document. Additionally, NetworkX provides a way to label, color, and cluster nodes based on actor roles, which enhances readability in the final rendered diagram.

The integration of NetworkX into IUFlowGen allows for flexible experimentation with layout logic, custom edge weighting, and graph analytics—all of which contribute to the structural fidelity of the generated flowcharts. Its pure Python implementation and compatibility with NumPy make it an ideal tool for embedding procedural reasoning into the system.

4.3.5 Ollama

Ollama is a containerized LLM inference engine that simplifies the deployment and execution of open-source language models on local hardware. It provides a user-friendly command-line interface and runtime environment for managing models in the GGUF format, enabling consistent and efficient execution without relying on cloud APIs. In IUFlowGen, Ollama acts as the core serving backend for models like Phi-4 and DeepSeek-R1, handling all inference requests related to procedural parsing, reasoning, and generation.

A key benefit of Ollama is its support for quantized models—particularly q8 formats—where model weights are compressed to 8-bit integers. This compression drastically reduces memory usage and inference time while retaining the accuracy needed for structured reasoning tasks. As a result, large models with 14 billion parameters can be executed on mid-range consumer hardware with acceptable performance and minimal latency. This makes Ollama a practical and accessible solution for student-led research and academic prototyping.

Ollama is tightly integrated into the IUFlowGen pipeline via LangChain and Python subprocess calls. It exposes a local API endpoint that accepts inference requests with custom prompts and context augmentation. This design enables IUFlowGen to dynamically route different tasks—such as step extraction, entity relation inference, and query clarification—to the same model instance without duplicating overhead.

Moreover, Ollama supports model switching and hot reloading, allowing users to test different LLMs interchangeably during development. This flexibility is leveraged in IUFlowGen to compare outputs between Phi-4 and DeepSeek-R1 and to dynamically adapt the inference backend based on system resource constraints or user preference.

Ollama's lightweight infrastructure and privacy-friendly architecture align perfectly with IUFlowGen's design goals of local control, modularity, and interpretability. It acts as the inference backbone of the system, enabling advanced LLM capabilities to be embedded into procedural visualization workflows without introducing infrastructure complexity or external data leakage risks.

4.3.6 Local LLMs: Phi-4 and DeepSeek-R1

IUFlowGen incorporates two state-of-the-art local large language models: **Phi-4** by Microsoft Research and **DeepSeek-R1** by DeepSeek AI. Both are 14-billion-parameter dense transformer models designed to handle tasks involving natural language understanding, procedural reasoning, and instruction following.

Phi-4 is trained on a mixture of high-quality web and synthetic data with a strong emphasis on STEM and code-related reasoning. It demonstrates competitive performance across multiple academic benchmarks while maintaining inference efficiency. **DeepSeek-R1**, derived from the Qwen family, is fine-tuned for long-context multi-step reasoning and shows state-of-the-art results in math, logic, and open-domain evaluations.

To balance performance and computational efficiency, both models are deployed in **q8 quantized format**, enabling high-accuracy reasoning on testing hardware. Within IUFlowGen, they are used interchangeably across the pipeline—supporting procedural step extraction, inter-step relation discovery, and user query clarification.

Their local deployment avoids reliance on commercial APIs, ensuring privacy, reproducibility, and offline usability—all of which are central to IUFlowGen's design philosophy.

4.3.7 LangChain

LangChain is a high-level orchestration framework designed to build language model-based applications using composable modules. In IUFlowGen, LangChain acts as the controller layer that coordinates various steps of the language model pipeline—such as prompt templating, context injection, tool invocation, and output parsing. Its modular architecture allows seamless integration with retrieval systems, custom prompts, and external tools, making it an essential component in building the procedural reasoning workflow.

LangChain is used to define multiple chains in IUFlowGen, including chains for step extraction, entity recognition, inter-step relation mapping, and DOT code synthesis. Each chain involves structured prompts, dynamically injected context (retrieved via LightRAG and NanoVector), and optional post-processing functions. The framework supports both synchronous and asynchronous execution, enabling batch processing of document segments and real-time querying by users.

A significant feature of LangChain is its support for advanced prompting strategies, such as few-shot prompting and chain-of-thought reasoning. IUFlowGen leverages this to improve the consistency and interpretability of the model's outputs, particularly in subgraph construction and procedural clarification.

LangChain also provides extensive observability features, such as step-wise logging, token counting, and memory buffering. These features are used during development to refine prompt engineering strategies and improve system transparency. With its extensive ecosystem and compatibility with tools like Ollama and OpenAI-compatible APIs, LangChain future-proofs the IUFlowGen system by enabling integration with more advanced agents and multi-modal workflows if needed.

4.3.8 Paramiko

Paramiko is a Python library that provides an implementation of the SSHv2 protocol, allowing secure remote connections to servers for command execution and file transfer. In IUFlowGen, Paramiko is used to offload certain computationally expensive or isolated tasks—such as remote rendering with Graphviz or running GPU-accelerated inference—to trusted machines over SSH.

By using Paramiko, IUFlowGen maintains a lightweight local footprint while leveraging the processing power of remote systems when needed. The system initiates a secure connection, sends necessary files or command instructions, and retrieves the resulting outputs (e.g., rendered flowchart images or embedded document fragments). This separation of computation and control enables more flexible resource management in environments where compute resources are distributed.

Paramiko's reliability, support for key-based authentication, and compatibility with Linux-based systems make it a robust solution for secure remote orchestration. It also allows the system to be extended into cluster or multi-user setups in the future, where document parsing or model inference could be parallelized across multiple nodes.

In IUFlowGen, Paramiko is tightly integrated with the document processing pipeline and can be invoked conditionally depending on the user's hardware constraints or deployment preferences. This remote execution capability reinforces the system's modularity and enables it to scale beyond single-device environments.

4.3.9 Regular Expressions

Regular expressions (regex) are used throughout the IUFlowGen system for text preprocessing, output validation, and syntax correction. As LLMs often produce outputs with non-deterministic structure or subtle formatting inconsistencies, regex provides a deterministic and fast mechanism to ensure that generated text (e.g., DOT code, actor names, subgraph labels) conforms to expected patterns.

Regex is used at multiple stages of the pipeline. During document ingestion, it helps identify and segment procedural patterns such as numbered steps, conditionals, and decision points. After AI-based generation, regex filters are applied to clean malformed strings, extract node/edge definitions, and remove hallucinated or invalid syntax elements.

In the DOT code generation module, regex plays a critical role in ensuring that the AI-generated code adheres to Graphviz syntax. It detects and removes duplicated node declarations, normalizes spacing and bracket usage, and highlights mismatched delimiters. Regex also supports label standardization, which improves visual consistency across subgraphs.

Because of its speed, flexibility, and language-agnostic nature, regex remains a powerful post-processing tool in IUFlowGen. It acts as the final quality assurance layer before visualization, helping convert flexible AI-generated structures into strict, renderable graph formats.

4.4 System Architecture

The system architecture of **IUFlowGen** adopts a hybrid local-server design to facilitate modular, AI-assisted flowchart generation from procedural documents. It comprises two tightly coupled execution environments: the **Local Machine**, which manages user interaction and document upload, and the **Server Component**, which performs the core AI processing and reasoning tasks. The system operates under a human-in-the-loop paradigm, ensuring both automated transformation and expert-level validation, as illustrated in Fig.4.1.

The **Local Machine** serves as the entry and interaction point for the user. Through a Streamlit-based frontend embedded with D3.js, users upload procedural documents, explore generated flowcharts, and issue clarification queries. Upon upload, the document is converted to plain text and encapsulated in a working directory, which is transferred to the server. The local machine also handles two-way communication: sending queries to

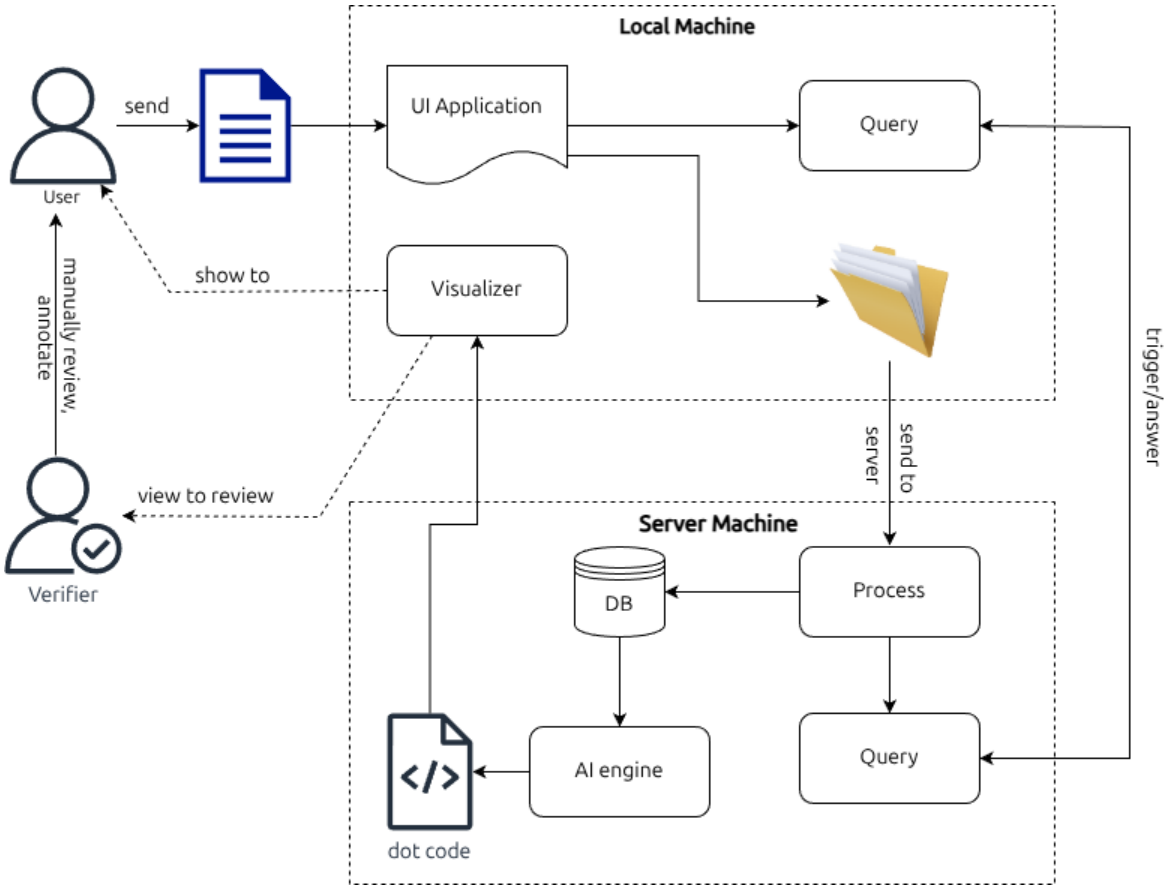

Figure 4.1: IUFlowGen System Architecture.

the server and receiving responses, and receiving the final DOT code for beautification and rendering. The resulting flowchart is then displayed to both the user and an expert verifier for review.

The **Server Component** is responsible for the core computational workflow. Upon receiving the working directory, it processes the procedural document by embedding it into vector representations (using NanoVector) and building graph structures (via NetworkX). This structured representation forms a local semantic-structural database, enabling the AI engine to perform accurate reasoning. The AI engine—integrating Ollama-hosted models such as Phi-4 and DeepSeek-R1—leverages this knowledge base to extract procedural steps, recognize actors and entities, and synthesize both intra- and inter-step relationships into DOT code. The final flowchart representation is transmitted back to the local environment for visualization.

A dedicated expert verifier interacts with the visualized output through the same UI. Their role is to manually review, validate, and, if necessary, annotate or refine the generated flowchart to ensure alignment with the document's original procedural intent. This layer of human oversight mitigates potential semantic drift or structural misinterpretation introduced by AI inference.

By distributing responsibilities between local interactivity and remote reasoning, IUFlowGen preserves user privacy, supports modular scaling, and guarantees both efficiency and traceability. This architecture is illustrated in Fig.4.1.

4.5 Environment Setup

The implementation of IUFlowGen is deployed across a hybrid architecture consisting of a local machine for user interaction and rendering, and a remote server for computationally intensive AI processing. This separation ensures efficient utilization of resources while maintaining an interactive user experience. The local machine is primarily responsible for collecting procedural documents, forwarding them to the server, handling user queries, and rendering flowcharts using Streamlit and D3.js. In contrast, the server hosts the AI engine and performs all major computational tasks including document preprocessing, semantic retrieval, procedural reasoning, and DOT code generation.

Table 4.1: IUFlowGen Deployment Environment.

Component	**Local Machine**	**Server Machine**
CPU	AMD Ryzen 5 5600H (6 cores)	AMD EPYC 7742 (64 cores)
GPU	NVIDIA RTX 3050 Laptop (4 GB VRAM)	Dual NVIDIA A100 SXM4 (160 GB total)
RAM	20 GB DDR4	128 GB ECC DDR4
Operating System	Ubuntu 24.04.2 LTS	Ubuntu 24.04.1 LTS
Python Version	Python 3.12	Python 3.12
Frontend Stack	Streamlit, embedded D3.js	–
Backend Stack	Paramiko (for communication)	LangChain, LightRag, Ollama
LLMs Used	–	Phi-4, DeepSeek-R1 (Q8 quantized) via Ollama
AI Task Distribution	Query input handling and flowchart rendering	Document processing, query handling, flowchart generation

The local system runs on Ubuntu 24.04.2 and is equipped with an AMD Ryzen 5 5600H processor (6 cores), 20GB of RAM, and an NVIDIA RTX 3050 GPU (4GB VRAM). These specifications are sufficient to manage lightweight tasks such as user interaction and frontend rendering. However, they are not suited for handling large language model inference or high-volume vector processing.

To meet the computational demands of flowchart generation and AI inference, IUFlowGen offloads these operations to a high-performance server environment. The server is equipped with a 64-core AMD EPYC 7742 CPU, 128GB of ECC RAM, and dual NVIDIA A100 SXM4 GPUs with 80GB of VRAM each (160GB total). This setup supports large-scale semantic retrieval, graph modeling, and the execution of quantized local LLMs such as Phi-4 and DeepSeek-R1. Most AI operations, including LightRAG-based retrieval and DOT generation, are performed entirely on the server and returned to the local machine for visualization.

A summary of the hardware and software configurations for both environments is presented in Table 4.1.

Chapter 5
IMPLEMENTATION

This chapter presents the implementation of the IUFlowGen system, focusing on how its core components interact to transform procedural documents into structured, interactive flowcharts. The system is designed around a human-in-the-loop architecture, combining local and remote processing to balance usability and performance. Key modules include document preprocessing, LightRAG-based retrieval, LLM inference, graph construction, and query resolution. Together, these components form a robust pipeline capable of extracting semantic structure and procedural logic from unstructured text. The implementation prioritizes modularity, efficiency, and privacy, allowing the entire pipeline to run across user and server environments without external cloud dependencies. The complete source code for IUFlowGen is available at `https://github.com/Khim3/IUFlowGen`.

5.1 Local Implementation

5.1.1 File Submission

The IUFlowGen system begins execution on the local machine, where the user interacts through a browser-based frontend built with Streamlit and D3.js. This frontend provides a platform for uploading documents, initiating queries, and rendering flowcharts returned from the server. One of the key functions performed locally is converting the uploaded procedural document into a text file and packaging it into a working directory. This directory is then transferred to the server to initiate the processing pipeline.

The `File Submission` stage, described in Algorithm 1, handles this preprocessing. It is responsible for receiving the uploaded document, extracting its text content, generating a unique session folder, and storing the converted file along with its metadata. This ensures that the input data is organized and ready for remote computation.

Algorithm 1 `File Submission`

Require: `uploadedFile` is a PDF uploaded by the user; `viewDoc` is a boolean flag.
Ensure: Creates text file and working folder from uploaded document and sends to server.

```
fileName ← ExtractName(uploadedFile)
if viewDoc = True then
    DisplayPDF(uploadedFile)
end if
tempPdfPath ← CreateTempFile(uploadedFile)
outputFolder ← CreateFolder(fileName)
textFile ← PdfToText(fileName, tempPdfPath, outputFolder)
if GenerateButtonClicked() then
    SendFolderToRemote(fileName)
end if
return (textFile, outputFolder)
```

This process forms the initial stage of the IUFlowGen pipeline and is responsible for handling the document ingestion and remote dispatching workflow on the local machine. It

begins by capturing the user's uploaded file through the Streamlit interface and extracting its base name to initialize a consistent folder structure. If the user opts to preview the document, it is rendered inline using a PDF viewer component, enhancing transparency and traceability.

Once previewed, the PDF file is temporarily stored in the local filesystem, after which a dedicated output folder is created to house the extracted resources. The system then invokes a PDF-to-text conversion module that processes the uploaded document and stores a plaintext version inside the output folder. This folder acts as the working directory for downstream modules.

Upon user interaction—specifically, clicking the "Process Document" button—the entire working folder is securely transferred to a remote server for advanced AI-based processing. This ensures the input is systematically prepared, and dispatched, enabling subsequent modules to operate with high efficiency and contextual consistency.

5.1.2 Document Processing

After the uploaded procedural document is converted into plaintext and transferred to the remote server, the next step in the IUFlowGen workflow is to process this text to prepare it for AI reasoning as described in Algorithm 2. This stage involves executing a command remotely from the local machine to invoke a Python-based backend routine responsible for extracting vector embeddings and graph-based representations. These structured formats are essential for the AI engine to perform semantic retrieval, flowchart generation, and query resolution. The process stores both vectorized representations and graph structures into a working directory on the remote machine, forming the foundational data structures for subsequent modules.

Algorithm 2 `Text Processing Invocation`

Require: `WORKING_DIR` is the path to the directory transferred to the server.
Ensure: Executes vectorization and graph extraction processes on the server.

```
command ← ConstructCommand(WORKING_DIR)
(stdin, stdout, stderr) ← sshClient.ExecCommand(command)
output ← stdout.read().decode()
return output
```

This command initiates server-side processing by constructing a secure SSH command string that invokes the text processing module on the remote machine. The script, defined in `text_processing.py`, leverages advanced preprocessing techniques to convert the procedural text into vector embeddings using embedding models and constructs procedural graphs with the help of `NetworkX`. These artifacts are then saved in the working directory associated with the document.

The output of this step includes debug logs and processing results, which are returned to the local machine through SSH output streams. These results confirm whether the document was successfully parsed and stored in a format suitable for flowchart synthesis and semantic querying. By encapsulating this logic into a remote call, `IUFlowGen` ensures scalability, modularity, and clear separation between user interaction and AI computation layers.

5.1.3 DOT Code Generation

Once the vector and graph data are prepared and stored in the working directory after Algorithm 2, the next operation in the IUFlowGen system is to generate the raw DOT code representing the flowchart. This task is executed remotely via a command that invokes a Python-based pipeline on the server, illustrated in Algorithm 3. The pipeline script, named visualizer.py, leverages the vector database, graph structure, and AI engine to construct the procedural flowchart logic as raw DOT code. The result is streamed back to the local machine, where it can be further beautified and rendered into an interactive chart.

Algorithm 3 DOT Code Generation

Require: WORKING_DIR is the active working directory on the server.
Ensure: Generates raw DOT code from preprocessed vector and graph data.

```
command ← ConstructCommand(WORKING_DIR)
(stdin, stdout, stderr) ← sshClient.ExecCommand(command)
output ← stdout.read().decode()
return output
```

This function initiates the AI flowchart generation pipeline on the remote server by executing the visualizer.py script through SSH. The pipeline consumes the vectorized document and procedural graphs stored in the working directory, applies LightRAG for retrieval, and invokes the local LLM via Ollama to reason over the steps, entities, and relations. The output of this reasoning is a raw DOT code string, representing the logical structure of the procedural workflow.

The raw DOT code is then transmitted back to the local machine for further refinement and rendering. This modular split between computation (server) and interaction (local) ensures system scalability, data locality, and performance, allowing users to operate interactively with the flowchart while maintaining AI-heavy processing remotely.

5.1.4 Flowchart Rendering and Beautification

Once the AI engine generates the raw DOT code from the server (Algorithm 11), the IUFlowGen system transitions to the final stage of flowchart rendering on the local machine. This step involves saving the streamed DOT code, cleaning escaped characters, applying aesthetic styles, and ultimately rendering the diagram using D3.js embedded within Streamlit. This process enhances both the visual clarity and the usability of the generated flowchart for end users and verifiers as follows Algorithm 4.

Algorithm 4 Render and Beautify Flowchart

Require: output_folder is the working directory from previous steps.

```
raw_code ← RunChain(output_folder)
Write raw_code to raw_code.txt
Read raw_code from raw_code.txt
dot_code ← CleanDotCode(raw_code)
if chart_mode = ''detailed'' then
    full_code ← BeautifyDotCode(dot_code)
    Replace rankdir with user-selected option
    Render full_code to Streamlit UI
else
    dot ← ConvertClustersToNodes(dot_code)
    dot ← BeautifyDotCode(dot)
    Render dot to Streamlit UI
end if
```

This rendering pipeline begins by invoking the `RunChain()` function, which executes the AI flowchart generation script on the server and streams the resulting DOT code back to the local machine. The raw code is saved locally for reference and debugging, and is then passed through a cleaning function to remove extraneous escape sequences (such as improperly escaped quotation marks).

Next, the cleaned DOT code is transformed using two formatting functions: `BeautifyDotCode()` and `ConvertClustersToNodes()`. The former applies consistent color schemes, typography, and layout adjustments based on user preferences. The latter simplifies the diagram by converting cluster-based subgraphs into high-level nodes, making the flowchart easier to navigate.

IUFlowGen provides two rendering options to accommodate different user needs:

- **Detailed View**: This mode renders the entire DOT code in its original structure, displaying all procedural steps, actors, and intra/inter-step relationships in full fidelity.
- **Overview Mode**: This simplified mode collapses each procedural subgraph into a single labeled node. Only the main steps and their interconnections are shown, enabling users to understand the high-level workflow without being overwhelmed by detail.

Finally, the processed DOT code is rendered interactively in Streamlit using D3.js and the Graphviz WebAssembly backend. This allows end users to zoom, pan, and explore the flowchart in real time. The visualization accommodates both the original procedural structure and any expert modifications made during verification, reinforcing clarity and usability throughout the workflow.

5.1.5 Interactive Query Interface

Once the procedural flowchart is fully generated and rendered, IUFlowGen provides an optional but powerful feature for user clarification: a live query interface. This feature (Algorithm 5) allows users to ask natural language questions regarding the structure, logic, or entities within the generated flowchart. It is especially valuable when users encounter ambiguous steps, seek to validate transitions, or need elaboration on specific procedural elements.

The query interface operates within the Streamlit front-end using an interactive chat format. User queries are routed to the remote AI engine, which performs context-aware reasoning using LightRAG over the previously embedded document representations (Algorithm 2 , 6). The server then returns an answer that is streamed back and displayed to the user in real time.

Algorithm 5 `Clarification Query Interaction`

Require: `output_folder` contains the document's working directory.

```
if messages not in session then
    Initialize chat with system greeting
end if
for all message in chat history do
    Render message to UI
end for
user_input ← WaitForChatInput()
if user_input exists then
    Append user_input to chat
    response ← ExecuteQuery(user_input, output_folder)
    Stream response back to UI with typing effect
    Append response to chat
end if
```

This function begins by initializing a conversational environment. If no prior messages exist, a system prompt invites the user to seek clarification. The user then inputs their query using Streamlit's chat input component. Once submitted, the query is forwarded to the remote backend via SSH, where the LightRAG pipeline retrieves relevant document fragments, combines them with the user prompt, and passes the enriched context to the LLM.

The AI model then synthesizes a coherent, grounded response which is streamed back to the user. IUFlowGen mimics a dynamic typing experience to increase interactivity and engagement. All chat exchanges are stored in the session state to ensure conversational continuity.

This query capability is particularly helpful in professional or academic scenarios where procedural accuracy is critical. It bridges the gap between AI-generated output and human understanding by enabling real-time feedback, validation, and on-demand explanations—ensuring transparency, interpretability, and user trust in the system.

5.2 Server Implementation

The server-side of IUFlowGen is responsible for executing all core AI functionalities, including document preprocessing, vector embedding, graph construction, and flowchart synthesis. While the local machine handles user interaction and visualization, the server performs the heavy computation by responding to remote commands sent over SSH.

Each function—from embedding text and building graphs to generating DOT code and answering queries—is encapsulated in Python scripts triggered by the frontend. This separation ensures efficient use of GPU resources and maintains a responsive user interface. The subsections that follow describe each of these backend operations in detail, highlighting how the server collaborates with the local system to complete the IUFlowGen pipeline.

5.2.1 Text Processing

The first core operation executed on the server is text processing (Algorithm 6), which prepares the procedural document for AI-driven reasoning and visualization. This operation is remotely triggered by the local machine after the plaintext version of the document has been transferred. The function uses the LightRAG framework and leverages the Phi-4 large language model for procedural analysis, as well as the Nomic embedding model for semantic vectorization. The output of this routine consists of indexed vector representations and graph structures derived from the input document. These outputs are saved to the working directory and serve as the foundational knowledge base for both the chart generation and query resolution functions.

Algorithm 6 `Text Process`

Require: `working_dir` is the path to the document working directory.

```
document_path ← GetTxtFileFromDirectory(working_dir)
Initialize RAG ← LightRAG
with LLM model: Phi-4 via Ollama
with embedding: Nomic-Embed (dimension = 768)
if document_path ≠ ∅ then
    document_text ← ReadFile(document_path)
    RAG.insert(document_text)
else
    return Error: No text file found
end if
return Processed vectors and graph saved in working_dir
```

This routine is invoked via a CLI call in the format `python3 text_processing.py "<working_dir>"`. Upon execution, the function initializes a LightRAG pipeline with key components: the Phi-4 LLM, which performs high-context document understanding and breakdown, and an embedding model (Nomic-Embed) responsible for mapping document segments into a vector space. These vectors and their corresponding text spans are indexed and stored for fast retrieval during downstream operations.

The algorithm first locates the document within the specified working directory, then reads and inserts the content into the LightRAG engine. This enables both structured semantic search and efficient step-by-step reasoning for flowchart construction. By decoupling this process from the local interface and executing it in a dedicated server environment, the system ensures high performance, memory scalability, and centralized data preparation across multiple user sessions.

5.2.2 Flowchart Generation Engine

The `visualizer.py` module serves as the core AI engine responsible for generating the DOT code that forms the basis of the final flowchart. After the initial document has been processed into vector and graph representations, this module orchestrates multiple stages of AI-assisted reasoning and procedural reconstruction. It combines retrieval-augmented generation, prompt engineering, structural analysis, and graph composition techniques to incrementally synthesize a complete and coherent representation of the procedural logic embedded in the document.

Specifically, the engine executes a sequence of sub-tasks, including step extraction, entity and actor recognition, intra-step and inter-step relation inference, and final DOT

code assembly. Each of these tasks is designed to maintain logical consistency, semantic accuracy, and syntactic correctness in the output. The result is a machine-generated but human-verifiable flowchart that faithfully represents the intent and structure of the input document.

Step Identification

The first operation in the `visualizer.py` module is dedicated to identifying the main procedural steps embedded in the input document. This is accomplished using a combination of LightRAG's document-wide semantic querying and few-shot prompt engineering. The objective is to extract structurally significant headers or named steps from the procedural text and convert them into initial DOT code subgraphs, each representing a core segment of the workflow.

The process begins by scanning the document to list the major steps using a high-level query across the full content. This list is then passed into a few-shot prompt, designed to instruct the large language model (LLM) to transform the sequence into valid DOT code syntax. Specifically, each extracted step is formatted as a `subgraph cluster_i` with a corresponding step label. This forms the structural backbone of the procedural flowchart.

Algorithm 7 `Step Identification`

Require: `WORKING_DIR` is the directory containing the input document.
Ensure: `dotCodeSteps` is the DOT code fragment representing extracted steps.

```
rag ← InitLightRAG(WORKING_DIR)
docText ← ReadTextFileFrom(WORKING_DIR)
rag.insert(docText)
stepList ← rag.query(\Scan and list all steps.")
prompt ← FewShotPrompt(stepList)
dotCodeSteps ← LLM.invoke(prompt)
stepNames ← ExtractStepNamesFrom(dotCodeSteps)
return (stepNames, dotCodeSteps)
```

This step uses the Phi-4 and DeepSeek-R1 models interchangeably, both running in quantized 8-bit format to ensure efficiency. LightRAG supports a global retrieval mode for capturing semantic-level structures across the document, which is ideal for parsing high-level procedural sequences. By enforcing a strict output format via few-shot examples, the system ensures syntactically valid DOT code is produced, with each cluster labeled to reflect a procedural phase.

The final output includes the list of step names, a numbered representation for human readability, and a snippet of DOT code that lays the groundwork for constructing detailed intra-step relationships in later stages (Algorithm 8, 9, 10).

Entity and Context Extraction

Once the procedural steps are identified (Algorithm 7), IUFlowGen enriches each step by extracting its internal semantics, including the associated actors, actions, entities, and relevant contextual details. This process transforms the surface-level outline of the procedure into a detailed semantic map that serves as the backbone for graph construction and reasoning in later phases.

To accomplish this, each step is used to formulate a targeted query, which is submitted to the AI engine powered by the LightRAG retrieval pipeline. Using globally retrieved

context and prompt engineering techniques, the model infers structured attributes that describe what occurs within each procedural step. The output is stored in a dictionary data structure where each key corresponds to a step name, and its value contains structured semantic elements extracted from the source text.

Algorithm 8 `Entity and Detail Extraction`

Require: `stepNames` is the list of procedural step names.
Ensure: `dotCodeStepDetails` is a list of enhanced subgraph DOT fragments.

```
stepDetails ← {}
for all step ∈ stepNames do
    query ← "For the step 'step', extract Actor, Action, Entities, and Relevant Info"
    response ← rag.query(query)
    stepDetails[step] ← response
end for
fewShotPrompt ← ConstructPrompt(stepDetails)
dotCodeStepDetails ← [ ]
for all step ∈ stepNames do
    inputPrompt ← fewShotPrompt(step, stepDetails[step])
    dot ← LLM.invoke(inputPrompt)
    CleanedDot ← RegexClean(dot)
    dotCodeStepDetails.append(CleanedDot)
end for
return (stepDetails, dotCodeStepDetails)
```

Each query issued to the AI engine is designed to extract four core attributes:

- **Actor(s)**: The agent(s) performing the action, explicitly or implicitly mentioned.
- **Action**: The main verb or procedural directive within the step.
- **Entities**: Objects, inputs, or items that are manipulated or referenced.
- **Relevant Info**: Supporting context such as conditions, parameters, or constraints.

The resulting dictionary—referred to as `stepDetails`—encapsulates a comprehensive semantic interpretation of the procedural document. This enriched structure not only facilitates downstream graph construction and LLM prompting but also improves the interpretability and traceability of the generated flowchart. Rather than relying on raw text, subsequent modules (Algorithm 9) can now reason over well-structured, human-readable information aligned with each procedural phase.

Intra-Step Relationship Construction

Building upon the extracted semantic details of each procedural step, this phase in IU-FlowGen focuses on identifying intra-step relationships—connections between actors and entities—where actions serve as edge labels. This relationship modeling enhances the expressiveness of the flowchart and enables precise procedural tracing.

Each step's description, obtained from the semantic dictionary, is used to formulate a prompt that guides the AI engine in inferring and formatting these internal relationships. The goal is to extract directional links in the form of "`Actor → Entity`" accompanied by an action label (e.g., ``` ``Label: Perform'' ```).

Algorithm 9 `Intra-Step Relationship Extraction and Normalization`

Require: `stepDetailsDict` contains semantic attributes for each step.
Require: `stepNames` is the list of extracted step titles.
Ensure: Returns `dotCodeList` and `actorDict`.

```
dotCodeList ← [ ]                          ▷ List to collect normalized subgraph strings
actorDict ← {}                             ▷ Map cluster ID to actor nodes
for all (stepName, stepDetail) ∈ zip(stepNames, stepDetailsDict.values()) do
    query ← FormatRelationshipPrompt(stepName, stepDetail)
    rawDotCode ← rag.query(query)
    normalizedDot ← NormalizeClusterNodeLabels(rawDotCode)
    dotCodeList.append(normalizedDot)
    clusterID ← ExtractClusterID(normalizedDot)
    actors ← ExtractActors(normalizedDot)
    actorDict[clusterID] ← actors
end for
return dotCodeList, actorDict
```

This process generates a list of intra-step DOT code fragments, where each fragment contains:

- Nodes for actors and entities, mapped to unique identifiers (e.g., `actor_1_1`, `entity_1_1`).
- Directed edges from actors to entities, labeled with their associated actions.

To ensure clarity and control during later stages, a normalization function is applied to the DOT code. This standardizes cluster and node naming conventions across steps, making downstream merging and editing more manageable. Additionally, the algorithm extracts a dictionary of actor node references by cluster, which will be used in identifying inter-step transitions (Algorithm 10).

The outcome of this stage includes:

- `dotCodeList` – A seamless string of intra-step flowchart subgraphs.
- `actorDict` – A mapping of each subgraph cluster to its internal actors.

This relational modeling stage ensures that IUFlowGen captures not only the procedural structure but also the internal dynamics of each step, laying the foundation for accurate visualization and interactivity.

Inter-Step Relationship Construction

After establishing intra-step semantics and structural mappings (Algorithm 9, 8, 7), IUFlowGen proceeds to infer logical relationships between procedural steps. These inter-step transitions capture the overall document flow—dependencies, conditions, and sequences—by identifying directed edges between previously defined step clusters.

This process uses a mixed-reasoning query mode in LightRAG, allowing the engine to synthesize relations from document semantics, flow logic, and vector similarity. The retrieved edge list is then validated and beautified to conform to DOT syntax using a language model-based validation routine.

Algorithm 10 Inter-Step Relationship Identification and Normalization

Require: numOfSteps is a numbered list of step titles.
Require: step1DOT is the initial DOT subgraph definitions from Step 1.
Ensure: Returns a validated DOT edge block and node-level conversions.

```
queryPrompt ← FormatEdgeQuery(numOfSteps)
rawEdges ← rag.query(queryPrompt, mode="mix")
fewShotPrompt ← BuildPromptWithExamples(rawEdges, step1DOT)
dotEdges ← llm.invoke(fewShotPrompt)
validatedEdges ← ValidateDOTCode(dotEdges)
clusterActorMap ← Dictionary mapping clusters to actors
nodeLevelEdges ← ReplaceClustersWithActors(validatedEdges, clusterActorMap)
return nodeLevelEdges
```

The LightRAG query response contains structural transitions such as:

```
From: 3 To: 5 Label: "requires service discovery completion"
```

This raw list is translated into:

```
cluster_3 -> cluster_5 [label="requires service discovery completion"];
```

Validation and Beautification. Since raw AI outputs may contain syntactic inconsistencies, a dedicated validation module reformats the transitions into a clean digraph format. This ensures compliance with Graphviz rendering requirements, including appropriate label encapsulation, indentation, and syntactic correctness.

Cluster-to-Node Rewriting. To visualize inter-cluster relations, IUFlowGen rewrites cluster-based edges using anchor nodes (actors) within each subgraph. This is achieved using a cluster-to-actor mapping dictionary generated in Step 3. The following format is used:

```
actor_3_1 -> actor_5_1 [label="requires service discovery completion",
ltail=cluster_3, lhead=cluster_5];
```

This conversion ensures proper semantic linking between procedural stages and enables interactive rendering in downstream visualizations. The final output of this step is a clean, DOT-compliant string representing transitions between procedural phases, enriched with contextual labels and anchored between subgraph nodes.

Graph Assembly and Transfer

Once all intra-step subgraphs and inter-step transition edges have been generated, IUFlowGen proceeds to assemble the final DOT graph specification. This operation unifies both components into a syntactically valid and renderable digraph structure.

Algorithm 11 Graph Assembly and File Transfer

Require: dotCodeStep3 contains intra-step DOT subgraphs.
Require: interStepRelations is the validated DOT edge list between clusters.
Ensure: Returns a full DOT graph string and saves it to persistent storage.

```
combinedDOT ← combine(dotCodeStep3, interStepRelations)
SaveToFile("combined_graph.txt", combinedDOT, append=True)
TransferToLocal("combined_graph.txt")
return combinedDOT
```

The function `combine_dot_code` formats the final output using the following structure:

```
digraph G {
    compound=true;
    rankdir=TB;

    <intra-step subgraphs>

    <inter-step edges>
}
```

This completed graph specification is written to a persistent text file `combined_graph.txt`. The system appends a delimiter before each new graph block to separate multiple generation attempts, providing a transparent and versioned history of outputs. This mechanism serves three key purposes:

- **Fail-safe**: Enables manual inspection and recovery in the event of downstream rendering issues or malformed graph components.
- **Model Selection**: Facilitates comparison across multiple model runs or prompt variations to choose the most accurate or visually optimal output.
- **Auditability**: Preserves full traceability of generated artifacts throughout the AI workflow, supporting reproducibility and debugging.

The finalized DOT code is then transferred to the local machine, where it is beautified, styled, and rendered in the interactive frontend using D3.js. This completes the server-side pipeline and hands off control to the user-facing visualization and query interface.

Chapter 6
RESULTS AND EVALUATION

This chapter provides a comprehensive overview of the outcomes derived from the implementation and testing of the IUFlowGen system. It begins by detailing the functional components that were successfully realized, including the document preprocessing pipeline, AI-assisted procedural understanding, flowchart synthesis, and interactive visualization through the browser-based interface. Each of these components was rigorously integrated and validated to ensure they function cohesively in a modular and extensible architecture.

Following the implementation results, this chapter also presents the experimental setup designed to evaluate the system's effectiveness and usability. The evaluation process involves task-based testing with real users and domain experts, where participants interact with the system to generate flowcharts and query procedural documents. Their feedback is used to assess the system's accuracy, completeness, and responsiveness, as well as the quality of the AI-generated outputs. Together, the results and the evaluation insights offer a foundation for understanding the system's practical performance and identifying opportunities for refinement in future work.

6.1 System Prototyping

The IUFlowGen system prototype was developed to demonstrate the core functionalities of AI-assisted procedural flowchart generation within an interactive and user-friendly environment. The system architecture supports two primary modes of operation: document transformation and semantic querying, each designed to accommodate both casual users and expert verifiers.

On the local machine, users interact with a frontend application that facilitates document upload, real-time flowchart visualization, and natural language querying. The interface emphasizes simplicity and accessibility, allowing users to convert procedural documents into flowcharts with minimal technical effort. Interactive features such as zoom, pan, and view toggles (overview vs. detailed) were implemented using embedded D3.js, offering flexibility in navigating complex workflows.

Once a document is submitted, it is preprocessed locally and transferred to a remote server where the AI engine performs text analysis, graph construction, and flowchart generation. The resulting DOT code is sent back to the local frontend for rendering and inspection. Users can ask questions about the visualized workflow using a chat-based interface, which streams AI-generated responses to clarify procedural logic.

For verification purposes, the system supports a human-in-the-loop model. Expert verifiers can review the generated flowcharts via the same frontend, validate the procedural integrity, and provide feedback. This collaborative interaction ensures both automation and quality control, confirming the system's practicality in real-world use cases. The prototype thus demonstrates a seamless integration of user interaction, AI processing, and expert validation in a cohesive flowchart synthesis pipeline.

6.1.1 Flowchart Visualization UI

Figure 6.1 illustrates the main user interface of the IUFlowGen system, which renders a procedural flowchart based on the AI-generated DOT code. The interface is built using Streamlit and enhanced with embedded D3.js, allowing users to interactively explore each element of the flowchart. This visualization aids in understanding complex procedural documents by translating them into clear and navigable workflows.

The interface includes a toggle to switch between two visualization modes: *Detailed* and *Overview*. In **Detailed** mode , indicated as **(2)** in Fig 6.1, the flowchart reveals all internal nodes, including actors, actions, entities, and contextual information extracted from the original document. This level of granularity enables verifiers to examine the semantic breakdown of each step. In contrast, the **Overview** indicated as **(1)** in Fig 6.1 mode abstracts the flow into inter-step relationships only, helping users grasp the overall procedural logic at a glance.

These dual view modes are critical in supporting both fine-grained validation and high-level analysis. The interactive capabilities, such as zooming and panning, ensure that users can focus on specific segments of interest while maintaining awareness of the broader document structure.

Figure 6.1: Main flowchart visualization UI.

6.1.2 Overview and Detailed View Modes

Figure 6.2 presents IUFlowGen's dual-mode visualization system that supports both **overview** and **detailed** rendering of procedural flowcharts. These two modes cater to users with different levels of analysis needs, providing flexibility in interpreting complex workflows.

The **overview mode**, labeled as **(1)** in Fig 6.2 , abstracts each procedural step as a single labeled node, omitting intra-step details such as actors or specific actions. This is particularly helpful when users seek a high-level understanding of the document's logical flow without being overwhelmed by granular information.

In contrast, the **detailed mode**, labeled as **(2)** in Fig 6.2 , reveals the full internal structure of each step, displaying nodes for actors, entities, actions, and contextual relations. This mode is suited for experts or verifiers who need to inspect the procedural logic at a fine-grained level for validation, correction, or refinement.

By enabling toggling between the two modes, IUFlowGen ensures both accessibility and analytical depth in the visualization experience, serving both general users and domain experts effectively.

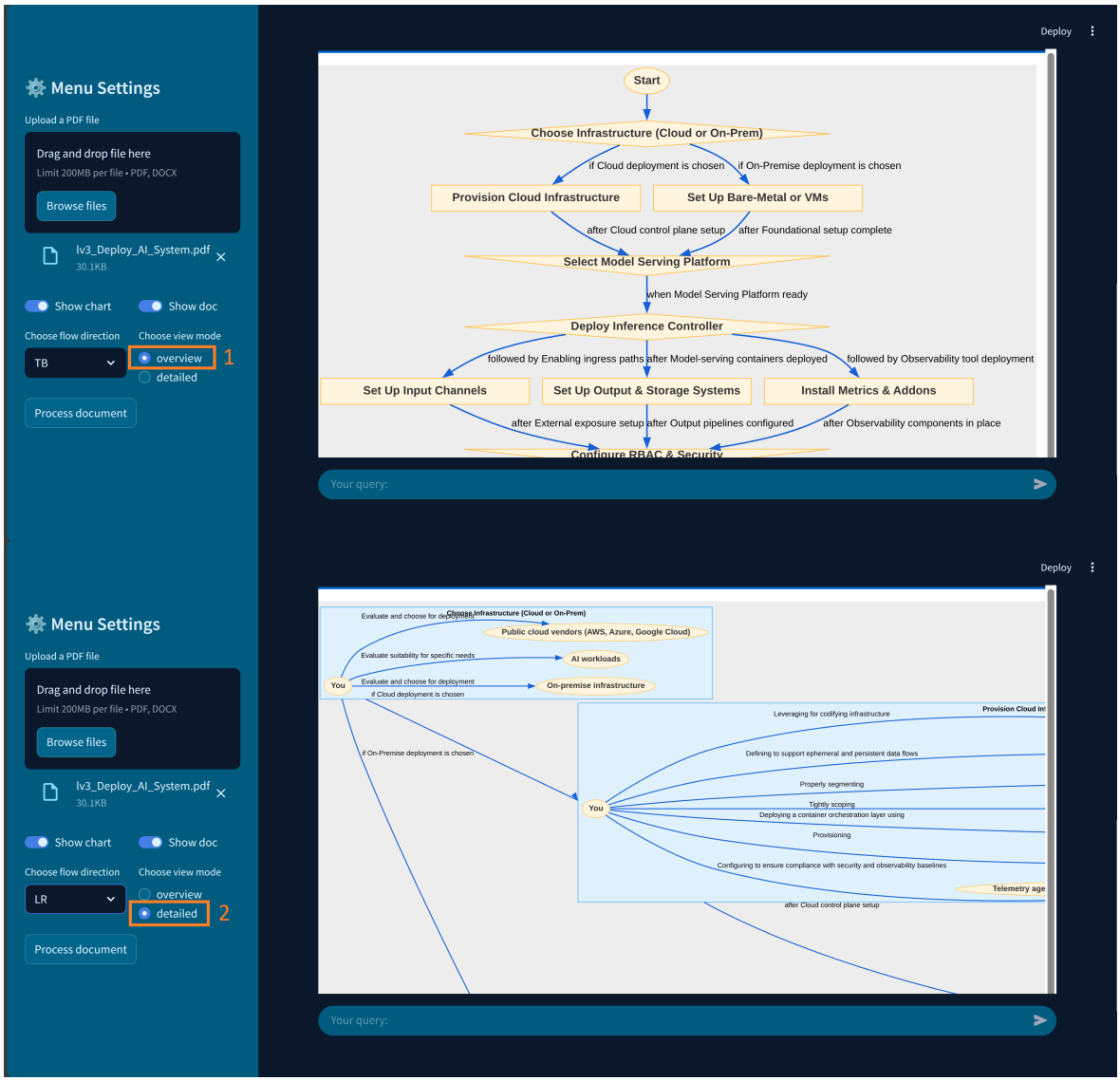

Figure 6.2: Overview and Detailed view mode toggle in IUFlowGen.

6.1.3 Layout Direction Toggle (LR and TB)

Figure 6.3 illustrates the layout direction toggle functionality within the IUFlowGen interface. This feature provides users with the flexibility to switch between two distinct orientations for flowchart rendering: **Left-to-Right (LR)** and **Top-to-Bottom (TB)**.

The **LR** layout, labeled as **(1)**, arranges procedural steps horizontally, which is particularly useful for workflows that emphasize sequential flow or timeline-based progression. On the other hand, the **TB**, labeled as **(2)**, stacks steps vertically, making it easier to interpret hierarchies or branching paths often found in complex multi-layered documents.

This directional toggle improves the usability of the system by accommodating different reading preferences and use cases. It also enhances readability, especially when dealing with large procedural structures that require careful examination of dependencies and transitions.

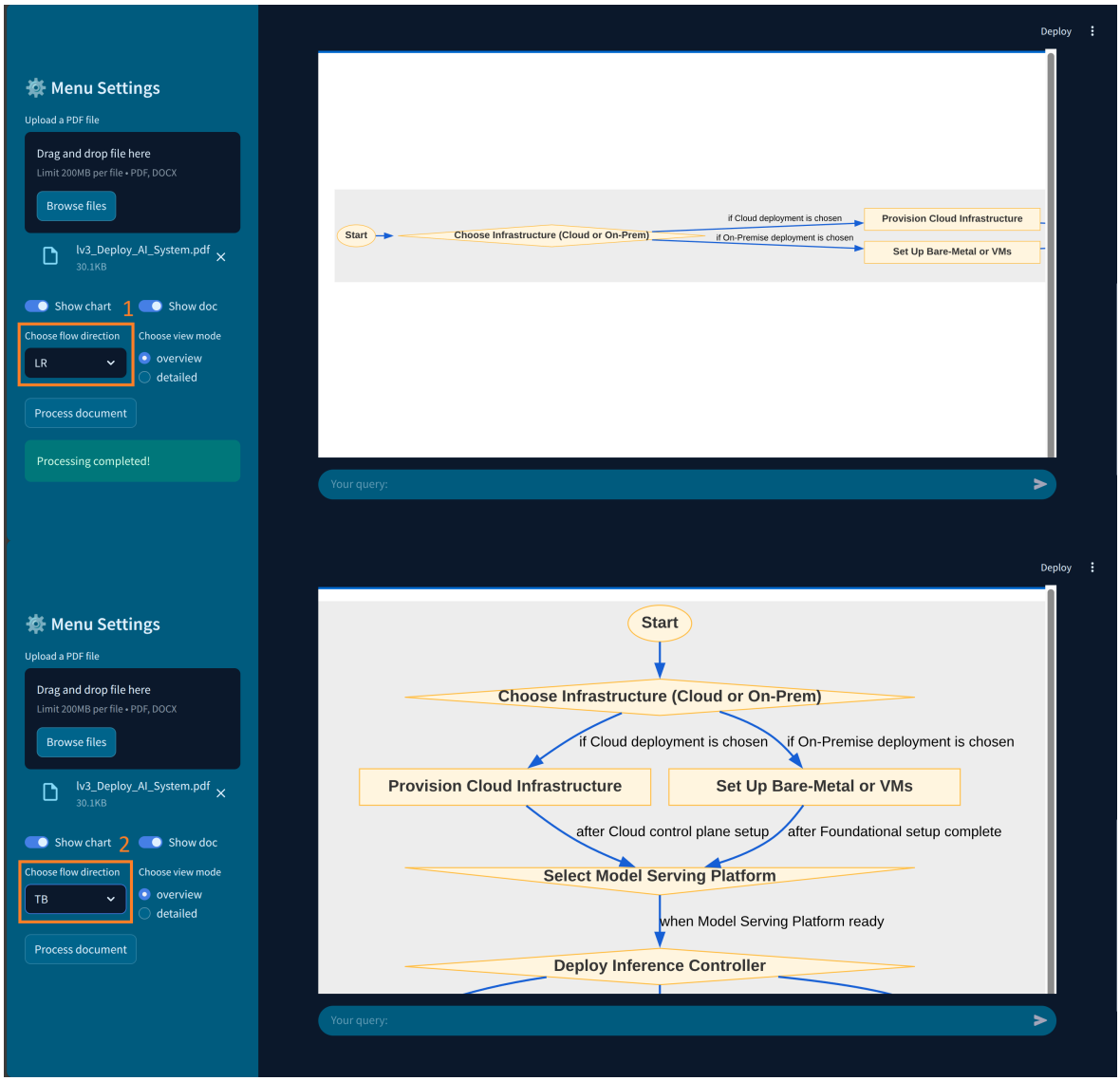

Figure 6.3: Layout direction toggle between Left-to-Right and Top-to-Bottom.

6.1.4 Query Function UI

Figure 6.4 illustrates the query function embedded within the IUFlowGen interface, enabling users to clarify any uncertainties related to the generated flowchart. Once the flowchart is fully plotted, users may engage in a natural language conversation through a dedicated chat interface to query specific elements, relationships, or steps within the procedural structure.

This feature is particularly useful when users encounter ambiguous sections in the flowchart or require additional context regarding procedural logic. The query function leverages the same backend AI engine used in flowchart generation, ensuring consistent understanding and accurate responses. Users can ask about the role of an actor, the rationale behind a specific transition, or the meaning of conditional branches.

By integrating this human-in-the-loop clarification capability, IUFlowGen reinforces its goal of providing interpretable and interactive visualizations of complex documents, further improving user confidence in the automated reasoning process.

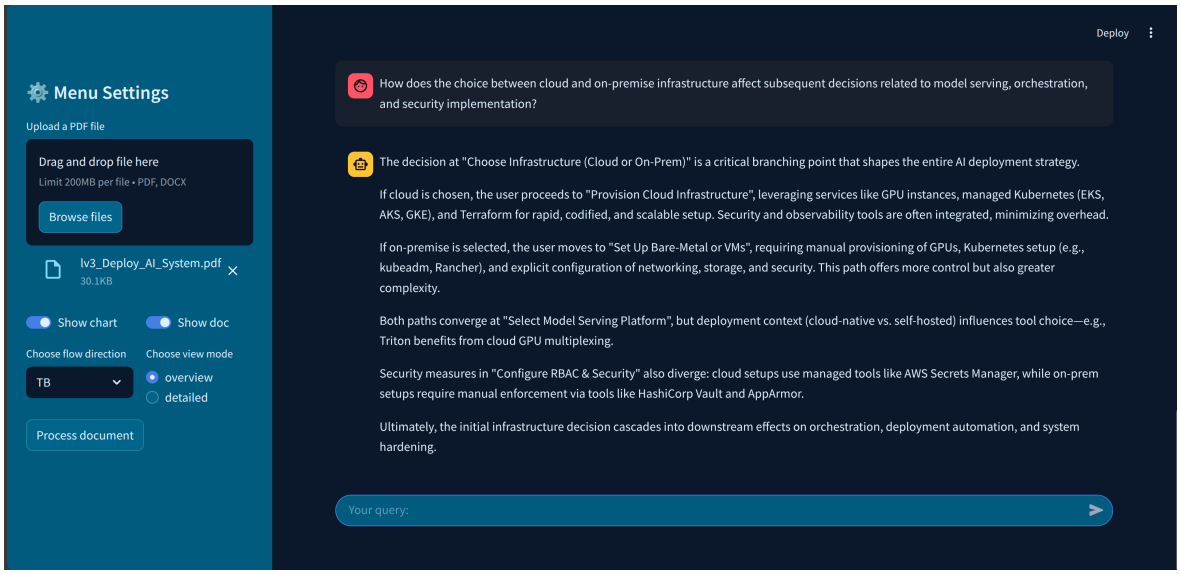

Figure 6.4: Query function interface for flowchart clarification.

6.1.5 Submitted Document Viewer

Figure 6.5 showcases the built-in document viewer within IUFlowGen, which allows users to access and review the originally submitted procedural document alongside the generated flowchart. This viewer supports PDF rendering directly in the user interface, ensuring a seamless and synchronized experience during visual analysis.

As highlighted by the **orange rectangle** in Figure 6.5, the integration of this viewer serves two essential purposes: it allows users to verify the accuracy of the AI-generated flowchart against the original textual source, and it provides verifiers with direct access to the raw document for auditing or annotation purposes. This dual-view mechanism enhances trust, interpretability, and validation during system use.

By facilitating a side-by-side comparison, the system promotes transparency in automated document processing and ensures that procedural logic extracted from the source remains verifiable and traceable throughout the visualization pipeline.

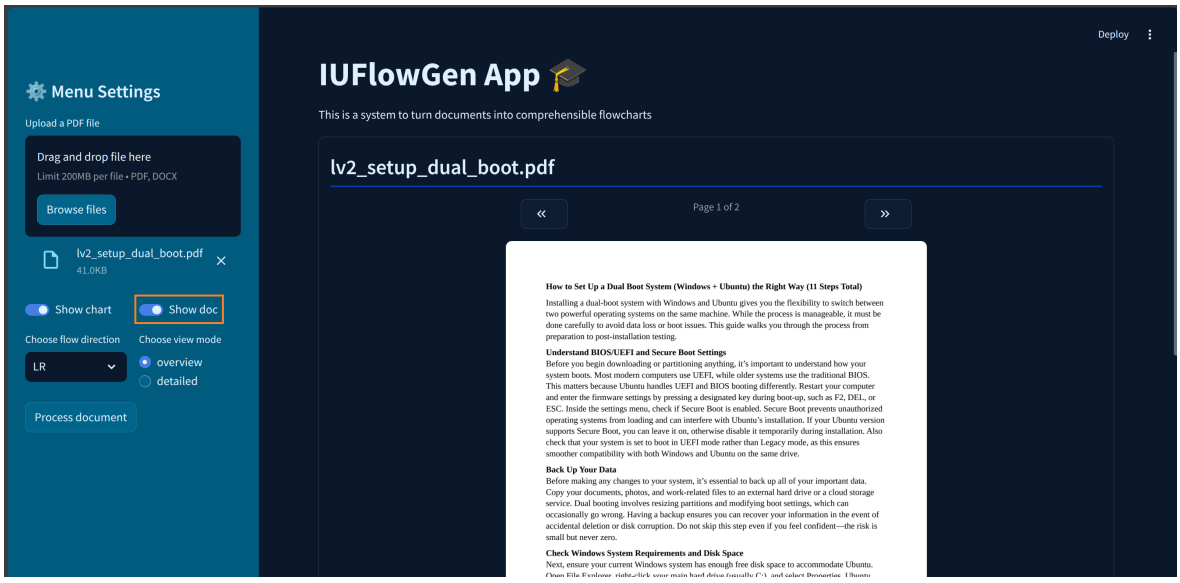

Figure 6.5: Submitted document display alongside flowchart output.

6.1.6 Interactive Zoom and Pan

Figure 6.6 demonstrates the zoom and pan functionality integrated into IUFlowGen's flowchart viewer. This feature enhances interactivity by enabling users to fluidly explore large and complex diagrams. The figure presents two distinct states: the default full-view state labeled as **(Before)** in Figure 6.6, and the zoomed-in, panned view focused on a particular subgraph, labeled as **(After)** in Figure 6.6.

The **zoom** function allows users to magnify specific areas of the flowchart, which is particularly useful for examining intricate actor-entity relationships or decision nodes within dense procedural logic. The **pan** capability complements this by allowing users to move across different sections of the diagram without losing visual continuity or needing to reset the interface.

Together, these features support precise validation and intuitive exploration, especially in complex flowcharts where tracking hierarchical flows and transitions is essential. This level of interactivity enhances usability for both general users and expert verifiers.

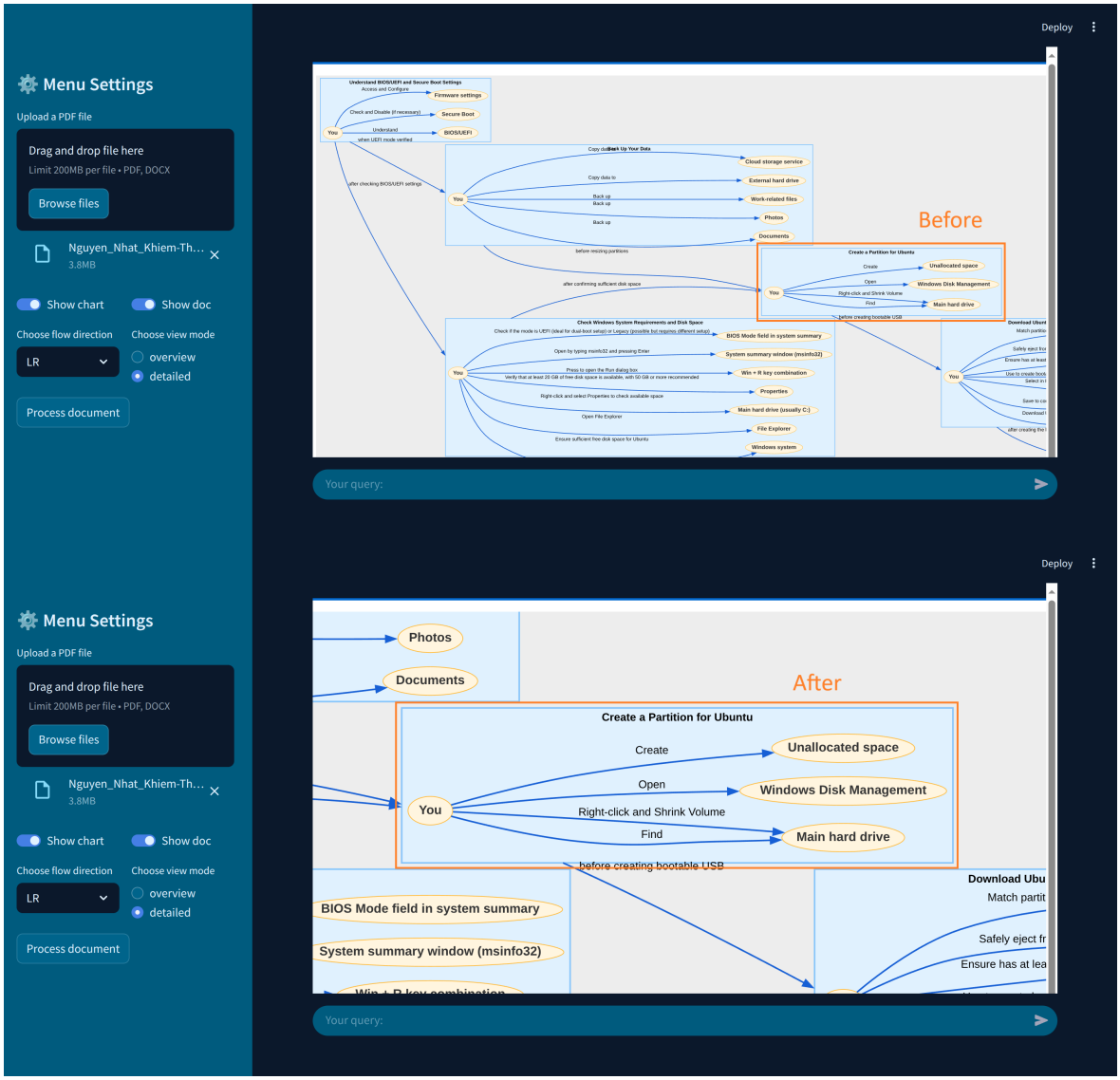

Figure 6.6: Interactive zoom and pan capabilities for flowchart navigation.

6.2 Experiment Design and Setup

To evaluate the effectiveness of IUFlowGen, a user study was designed to measure how much the system supports users in improving both the speed and correctness of procedural flowchart creation. In the absence of standardized metrics for evaluating AI-generated flowcharts, this thesis also proposes a structured rubric that separately measures correctness in terms of **accuracy** (correct logical flow and step ordering) and **completeness** (coverage of entities, conditions, and dependencies). The study setup emphasizes the importance of benchmarking AI assistance against human-only performance across various levels of document complexity.

To this end, a four-tier complexity scale was developed, ranging from straightforward synthetic procedures to real-world documentation. This gradation ensures that system performance can be observed under increasingly challenging conditions, thereby offering a comprehensive assessment of its practical value.

Document Complexity Levels:

- **Level 1 (Easy – Synthetic):** Documents in this category are short and simple, covering topics such as making tea or cleaning a room. The number of steps and involved entities is minimal, with each step clearly numbered and separated. The logic is strictly linear, with no conditional or parallel structures, making it suitable for users with no technical background.

- **Level 2 (Moderate – Synthetic):** These documents introduce moderate complexity, such as setting up an application or connecting a router. Steps are no longer explicitly numbered, and branching logic (e.g., simple if-else paths) may be included. Although more entities are involved, the structure remains accessible to general users familiar with basic procedures.

- **Level 3 (Complex – Synthetic):** This level simulates technical or legal processes like user onboarding or service agreements. The documents are longer and include implicit procedural logic with embedded dependencies, conditionals, and merging flows. Steps are not marked, and the text may include multiple actors or overlapping roles. This level is more demanding and requires users to interpret structure and intent through deeper semantic reasoning.

- **Level 4 (Advanced – Real-World):** The document at this level is a partial excerpt from the academic paper *"IU-SmartCert: A Blockchain-Based System for Academic Credentials with Selective Disclosure"* [31], co-authored by Dr. Tran Thanh Tung and Dr. Le Hai Duong. It represents a true real-world scenario with several complicating factors:

  - The document is long and dense, requiring extended attention span and multi-pass reading.
  - It involves many actors and entities (e.g., Issuer, Holder, Verifier, Blockchain Node) whose roles evolve across procedural phases.
  - Numerous cross-references such as citations [1], [2], etc., interrupt the logical flow and may confuse AI context windows or human readers unfamiliar with academic referencing.

  - Embedded diagrams and figures split procedural segments, introducing a visual-textual discontinuity that complicates sequential understanding.
  - Procedural boundaries are implicit, with no explicit markers, requiring advanced reasoning to determine where one step ends and another begins.

  This document serves as the most rigorous test case for evaluating how well IUFlowGen assists users in processing and visualizing complex domain-specific workflows.

Participant Groups: The study involved **10 participants** from the IT faculty, divided evenly into two groups:

- **Group 1 – Without AI Assistance:** Participants read each procedural document independently and manually created a flowchart based on their understanding. No tool-generated suggestions were provided.
- **Group 2 – With AI Assistance:** Participants were shown the flowcharts generated by IUFlowGen and were allowed to:
  - Accept the AI-generated chart as-is,
  - Make minor corrections or refinements to the chart, or
  - Redraw the chart entirely using the AI output as a reference.

Task Procedure: Each participant completed four tasks corresponding to the four document complexity levels, presented in increasing order. Time constraints were enforced to simulate real-world productivity scenarios:

- **Levels 1 and 2:** 10 minutes per task
- **Levels 3 and 4:** 15 minutes per task

Prior to beginning, all participants received equal training on flowchart notation and the web-based interface. Group 2 participants accessed AI-generated outputs on a separate screen and interacted with them based on the options outlined above.

Evaluation and Scoring: All submitted flowcharts were reviewed by an expert evaluator — Dr. Tran Thanh Tung (Vice Dean of IT at International University) — using a structured rubric composed of five clearly defined criteria. Instead of assigning a single numeric score, each flowchart was assessed on a checklist basis to evaluate the accuracy and completeness of procedural representation:

- **Step Coverage:** Are all major procedural steps present and captured?
- **Logical Flow:** Are the steps arranged in a coherent and valid sequence, with correct use of order, branches, and merges?
- **Relation Accuracy:** Are the procedural relationships (e.g., transitions, dependencies, conditions) between steps captured correctly?
- **Entity Representation:** Are all relevant actors, roles, and entities correctly represented and appropriately linked to steps?

- **Structural Clarity:** Is the flowchart visually and structurally readable, with well-formed connections and minimal ambiguity?

Each criterion was marked as either satisfied (✓) or not satisfied (✗). The total number of positive marks per participant was used to compare outcomes across document complexity levels and group conditions. This structured rubric allows targeted analysis of flowchart quality and highlights specific areas where AI assistance provides measurable benefits.

Expected Outcomes: It is hypothesized that the AI-assisted group will demonstrate progressively greater advantages as document complexity increases, particularly in satisfying the five evaluation criteria:

- **Levels 1 and 2:** Both groups are expected to meet nearly all criteria, especially Step Coverage and Logical Flow, due to the simplicity and explicit structure of the documents. Minor differences may occur in Entity Representation, but overall performance should be comparable.
- **Level 3:** The AI-assisted group is expected to outperform the manual group, particularly in Relation Accuracy and Structural Clarity, where procedural dependencies become harder to infer manually. The manual group may overlook or misorder steps, and fail to represent all entities correctly.
- **Level 4:** This level functions as a qualitative stress test rather than a benchmarked evaluation. Given the document's real-world complexity, with implicit logic, unmarked step boundaries, multiple actors, cross-references, and embedded figures, AI assistance is expected to help users better satisfy key criteria such as Step Coverage and Logical Flow. Particular attention will be paid to whether participants can correctly represent actor-driven relationships and resolve ambiguous procedural sequences.

6.3 Experiment Results

To evaluate the impact of AI assistance on flowchart construction, the experimental results were collected from both participant groups (with and without AI assistance). Each participant completed tasks across four levels of procedural document complexity, with time and correctness recorded for every level. The flowcharts from Levels 1 and 2 were assessed based on predefined reference diagrams, while Levels 3 and 4 were manually evaluated and scored by the verifier.

6.3.1 Results for Level 1 and 2 (Synthetic Documents)

Table 6.1 presents the correctness percentage and time taken for participants to complete flowcharts for Level 1 and Level 2 documents. Both groups worked on the same document set, with a strict time limit of 10 minutes per level. These levels were intended to test basic procedural understanding and interpretation accuracy in simpler scenarios.

Each criterion mentoioned above was marked as either satisfied (✓) or not satisfied (✗). The total number of positive marks per participant was used to compare outcomes across document complexity levels and group conditions. This structured rubric allows targeted analysis of flowchart quality and highlights specific areas where AI assistance provides measurable benefits.

As expected, participants from both groups performed well. However, the AI-assisted group consistently completed tasks faster and achieved slightly higher correctness due to the reference chart support that reduced decision-making overhead.

Table 6.1: Participant performance on Level 1 and 2 documents.

Group	Student	Level	Time	Step	Flow	Relation	Entity	Clarity	Total (✓)
No AI	Student 1	L1	10m	✗	✓	✗	✗	✓	2
		L2	10m	✗	✓	✗	✓	✓	3
	Student 2	L1	5m58s	✓	✓	✗	✓	✓	4
		L2	7m36s	✓	✓	✗	✓	✓	4
	Student 3	L1	10m	✓	✓	✗	✓	✓	4
		L2	10m	✗	✓	✗	✓	✗	2
	Student 4	L1	7m10s	✓	✓	✓	✓	✓	5
		L2	10m	✗	✓	✗	✓	✓	3
	Student 5	L1	5m40s	✓	✗	✗	✓	✗	2
		L2	9m17s	✗	✓	✗	✓	✗	2
AI Assisted	Student 1	L1	6m30s	✓	✓	✓	✓	✓	5
		L2	6m	✓	✓	✗	✓	✓	4
	Student 2	L1	5m44s	✓	✓	✓	✓	✓	5
		L2	7m6s	✓	✓	✗	✓	✓	4
	Student 3	L1	4m2s	✓	✓	✓	✓	✓	5
		L2	6m41s	✓	✓	✗	✓	✓	4
	Student 4	L1	5m26s	✓	✓	✓	✓	✓	5
		L2	6m19s	✓	✓	✗	✓	✓	4
	Student 5	L1	4m20s	✓	✓	✓	✓	✓	5
		L2	4m42s	✓	✓	✗	✓	✓	4

6.3.2 Results for Level 3 and 4 (Complex and Real-World Documents)

Table 6.2 summarizes the performance of both groups on Level 3 (complex synthetic) and Level 4 (real-world) documents. These documents featured longer and less structured content, often involving multiple actors, ambiguous step boundaries, and domain-specific logic. Compared to Levels 1 and 2, they required significantly more reasoning and abstraction.

Evaluation was conducted using the same five-point rubric applied to previous levels: Step Coverage, Logical Flow, Relation Accuracy, Entity Representation, and Structural Clarity. Each criterion was scored as either satisfied (✓) or not satisfied (✗) by the expert verifier.

The results show that participants in the AI-assisted group consistently achieved higher rubric satisfaction rates, particularly in Logical Flow and Relation Accuracy—two of the most challenging aspects in long procedural texts.

Table 6.2: Participant performance on Level 3 and 4 documents.

Group	Student	Level	Time	Step	Flow	Relation	Entity	Clarity	Total (✓)
No AI	Student 1	L3	15m	✗	✗	✗	✓	✓	2
		L4	15m	✗	✗	✗	✗	✓	1
	Student 2	L3	7m36s	✓	✓	✗	✓	✗	3
		L4	13m11s	✓	✗	✓	✗	✓	3
	Student 3	L3	15m	✓	✓	✗	✗	✓	3
		L4	13m	✓	✗	✓	✓	✓	4
	Student 4	L3	10m11s	✗	✓	✗	✗	✓	2
		L4	15m	✗	✗	✗	✗	✓	1
	Student 5	L3	9m17s	✓	✓	✗	✓	✓	4
		L4	15m	✓	✗	✗	✗	✓	2
AI Assisted	Student 1	L3	11m42s	✓	✓	✗	✓	✓	4
		L4	15m	✓	✓	✓	✓	✓	5
	Student 2	L3	11m55s	✓	✓	✗	✓	✓	4
		L4	15m	✗	✓	✗	✓	✓	3
	Student 3	L3	4m20s	✓	✓	✗	✓	✓	4
		L4	15m	✓	✓	✓	✓	✗	4
	Student 4	L3	7m45s	✓	✓	✗	✓	✓	4
		L4	12m42s	✓	✓	✓	✗	✗	3
	Student 5	L3	4m50s	✓	✓	✗	✓	✓	4
		L4	13m47s	✓	✓	✗	✓	✗	3

6.3.3 Observations on the Collected Results

The results reveal consistent performance differences between the *No AI* and *AI-Assisted* groups across all levels of document complexity. In Levels 1 and 2, although most participants in the No AI group were able to submit flowcharts, their average rubric scores were noticeably lower—3.4 and 2.8 out of 5, respectively—as shown in Table 6.3. Errors frequently involved missed entities, incomplete relation mapping, or confusion in step sequencing. Several submissions also failed to meet the time limit or included structurally ambiguous transitions.

This disparity became more pronounced at Levels 3 and 4, where the complexity of procedural logic increased significantly. The average rubric pass rate for the No AI group dropped to 2.8 for Level 3 and 2.2 for Level 4, with some participants scoring as low as 1 out of 5, as illustrated in Table 6.3. These results highlight the challenge of manually inferring step transitions, actor relationships, and conditional structures in longer or domain-specific documents. Submissions at these levels often exhibited missing or unordered steps, broken logic, or omitted actor-context relationships.

In contrast, the AI-Assisted group maintained consistently high rubric scores, averaging 5.0, 4.0, 4.0, and 3.6 from Level 1 to Level 4, respectively. Notably, all AI-assisted participants submitted within the allotted time, and their outputs were structurally complete across all tasks. While minor corrections were needed—especially in Level 4, where the document included implicit logic and embedded references—most flowcharts were usable with only superficial adjustments. Participants consistently reported that the AI-generated outputs served as effective first drafts, particularly in identifying core steps and clarifying procedural flow.

6.3.4 Summary of Performance Across Groups

Table 6.3 summarizes the average time and rubric-based correctness of each group across all levels. From this overview, it is evident that participants with LLM assistance consistently

outperformed their non-assisted counterparts, especially at higher complexity levels. The system proved particularly valuable in handling unstructured, multi-step documents where manual parsing is challenging and time-consuming.

Table 6.3: Performance summary by group across document complexity levels.

Level	Group	Avg. Time	Avg. Rubric Passes (/5)
Level 1 (Easy)	No AI	7m 45s	3.4
	AI Assisted	5m 12s	5.0
Level 2 (Moderate)	No AI	9m 22s	2.8
	AI Assisted	6m 09s	4.0
Level 3 (Complex)	No AI	11m 24s	2.8
	AI Assisted	8m 34s	4.0
Level 4 (Advanced)	No AI	14m 14s	2.2
	AI Assisted	14m 18s	3.6

To better illustrate these trends, Figure 6.7 visualizes the comparison across levels, showing how the AI-assisted group achieved higher rubric pass rates while also completing tasks in less time.

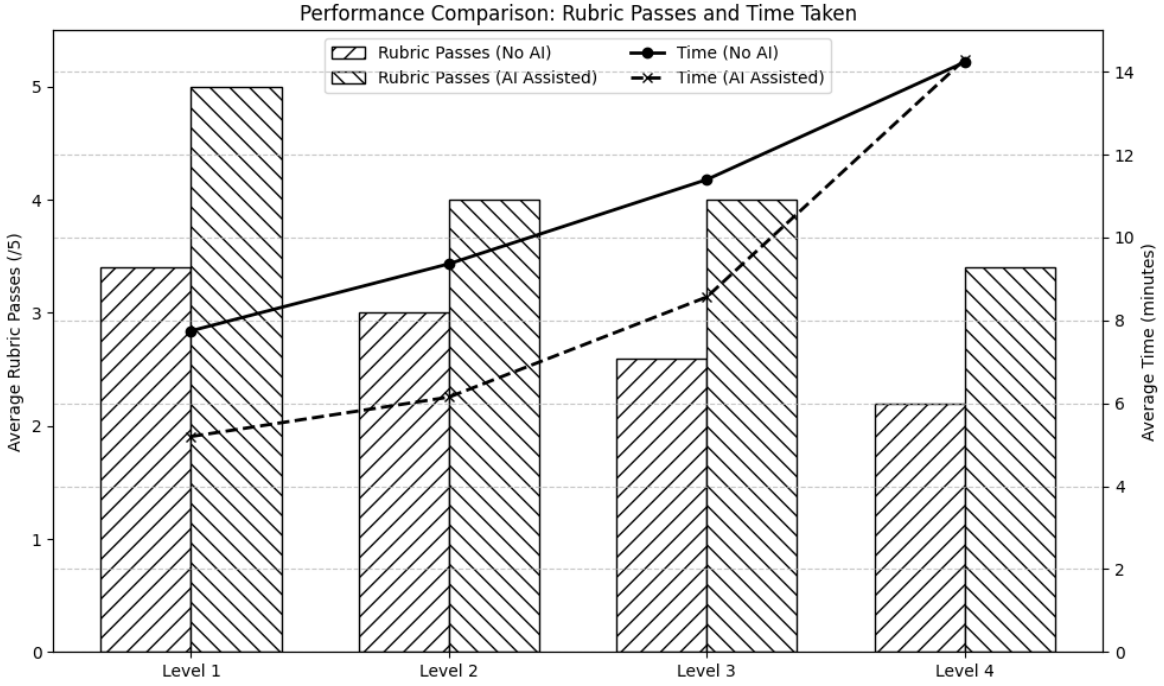

Figure 6.7: Comparison of average rubric passes and time across document complexity levels.

Overall, the results support the idea that LLM-powered flowchart suggestions notably improve user performance in both efficiency and output quality, especially when processing complex or real-world procedural documents.

The experiment confirmed the IUFlowGen system's ability to operate securely and efficiently in a locally distributed human-in-the-loop environment. The results validated the implementation of core components such as data retrieval, prompt-driven flowchart generation, and interactive visualization. Moreover, the system demonstrated its robustness when handling documents of varying complexity and its capability to assist users in understanding

and modeling procedural logic. These findings affirm the system's readiness for real-world deployment scenarios where procedural transparency and interpretability are critical.

Chapter 7
DISCUSSION

This chapter discusses the implications of the system's design, implementation, and evaluation results. It highlights key analytical insights drawn from experimental data, outlines the system's strengths and limitations, and compares IUFlowGen's approach to existing tools and research in procedural text visualization.

7.1 System Analysis

The IUFlowGen system was developed to enhance the extraction and visualization of procedural knowledge from unstructured documents. Based on the experimental outcomes and system behavior during deployment, this section presents an analytical perspective on the system's core functionality and behavior across different use cases.

Interpretability and Transparency

IUFlowGen addresses the interpretability gap in procedural document comprehension by offering structured, flowchart-based representations. These visual outputs allow users to quickly grasp high-level workflows, dependencies, and control flows without reading lengthy narratives. The inclusion of both overview and detailed visualization modes enables adaptable granularity for novice and expert users alike.

Interactive Human-in-the-Loop Model

A defining feature of the system is its human-in-the-loop workflow. Rather than fully automating the document understanding process, IUFlowGen allows users to inspect, query, and refine AI-generated flowcharts. This approach ensures that while the AI accelerates the initial parsing and layout construction, final validation remains in human control—improving trust, correctness, and interpretability.

Adaptability to Document Complexity

Through experimentation, IUFlowGen has demonstrated its robustness across documents of increasing complexity. From simple cooking instructions to technical documentation and academic manuscripts, the system showed scalable performance. Although accuracy declined slightly on high-complexity inputs, the AI-generated suggestions still provided valuable structural cues for participants, significantly improving performance compared to unaided attempts.

Performance in Resource-Constrained Environments

The system was designed to operate in locally-hosted settings, avoiding dependence on commercial APIs or cloud LLM providers. This ensures privacy, reproducibility, and accessibility in academic or enterprise environments where external data transfer is restricted. Despite

using quantized local models, IUFlowGen maintained a balance between performance and latency suitable for interactive use.

7.2 Strengths

The IUFlowGen system presents notable advancements in document understanding and visualization by combining large language models (LLMs), retrieval-augmented generation (RAG), and interactive human-in-the-loop interfaces. One of its key strengths is its ability to deconstruct complex procedural documents into interpretable flowcharts without requiring extensive user intervention. This design significantly reduces the time and effort typically required for manual flowchart construction, particularly in lengthy or unstructured technical documents.

Another major strength of IUFlowGen lies in its dual-mode visualization—**overview** and **detailed**—allowing users to switch between high-level process summaries and fine-grained procedural views. This flexibility accommodates both lay users and expert verifiers, enabling a deeper understanding of multi-actor systems and nested logic in procedural texts.

The use of locally hosted LLMs, including Phi-4 and DeepSeek-R1, further enhances IUFlowGen's practicality. By eliminating reliance on cloud-based APIs, the system safeguards user privacy, maintains reproducibility, and ensures operability in restricted or offline environments. Despite the use of quantized models, the AI engine is capable of generating coherent and structurally consistent flowcharts, demonstrating an effective balance between model efficiency and reasoning quality.

Finally, IUFlowGen's modular pipeline and graph-based intermediate representations allow extensibility. Components such as semantic vector indexing, DOT code post-processing, and query-based clarification can be independently upgraded or replaced, facilitating integration into broader AI-assisted documentation tools or domain-specific NLP applications.

7.3 Limitations

Despite its strengths, the IUFlowGen system has several limitations that affect its efficiency and scalability in certain contexts. A key challenge lies in the resource intensity and latency of the document processing pipeline. Specifically, the phase involving entity extraction and semantic vector embedding requires substantial computational power, particularly when handling lengthy or information-dense documents. Since both graph construction and embedding must occur concurrently, the process becomes time-consuming and memory-heavy, especially on machines without high-performance GPUs.

Another limitation concerns the system's reliance on prompt-based LLM inference, which is inherently non-deterministic. Although IUFlowGen incorporates regular expressions and structured prompt patterns to regulate the output format, occasional inconsistencies or hallucinated relations may still occur—especially in complex procedural texts. These inconsistencies necessitate human verification, reducing full automation potential.

The current system architecture is also limited in terms of concurrency. IUFlowGen is designed to process one document request at a time and cannot handle multiple simultaneous user requests. This restricts its applicability in multi-user environments or high-throughput batch scenarios, making it more suitable for single-session use cases such as academic research or individual verification tasks.

Finally, the usability of the system partially depends on the clarity of the original document. For highly unstructured or domain-specific inputs with ambiguous procedural logic,

the AI engine may struggle to infer accurate step boundaries or relationships, requiring substantial manual correction from the user or verifier. While this limitation is mitigated by the human-in-the-loop model, it still poses a barrier to full autonomy and speed in certain workflows.

These limitations highlight areas for future improvement, such as optimizing the document preprocessing pipeline, enabling multi-user support, and fine-tuning LLMs for procedural reasoning with greater stability.

7.4 Comparison

IUFlowGen introduces a privacy-preserving and interactive solution for generating flowcharts from procedural text. Unlike most existing tools, it runs entirely on local hardware, supports document-level reasoning through RAG-based prompting, and integrates an interactive flowchart viewer with future support for direct output editing. Its primary goal is to combine transparency, structure preservation, and interpretability for complex real-world documents.

When compared to general-purpose LLMs such as **ChatGPT-4o**, IUFlowGen offers a more focused approach. While ChatGPT is capable of identifying procedural logic and even drafting Graphviz syntax, it lacks a built-in visualization system. Users must manually render outputs using external tools. Additionally, as a cloud-hosted service, ChatGPT poses privacy and usage risks in regulated environments, even though its responses are typically fast and free from token constraints.

NoteGPT, on the other hand, enables casual users to convert meeting notes and action items into flowcharts through a streamlined cloud interface. However, it lacks deep logic modeling capabilities, does not include a query mechanism, and offers limited free usage. It is best suited for lightweight personal planning and informal workflow mapping.

Eraser Flowchart Maker provides an improved UI and allows direct editing of nodes, making it user-friendly. Nevertheless, it lacks the ability to reason over procedural structure and cannot support semantic queries. Like NoteGPT, it also operates in a cloud environment with tiered feature access.

GenFlowchart, an academic framework, focuses on the interpretation of visual diagrams rather than generating flowcharts from textual input. It excels at parsing static flowcharts using OCR and segmentation, but is not designed to process raw procedural documents or assist users in interpreting multi-step instructions.

In contrast, IUFlowGen was built specifically to address these gaps. It supports converting complex documents into structured flowcharts, offers an interactive querying feature, includes a visual output module, and guarantees data confidentiality through local execution. While latency remains a challenge during document embedding and parsing, IUFlowGen avoids rate-limiting or paywalls, and it is being extended to allow direct flowchart edits—bridging the gap between automation and manual refinement.

Table 7.1 compares IUFlowGen to these alternatives across key feature dimensions.

Table 7.1: Comparison of IUFlowGen and Related Tools (✓: Yes, ✗: No, WIP: Work in Progress).

Feature	IUFlowGen	ChatGPT-4o	NoteGPT	Eraser	GenFlowchart
Flowchart from Text	✓	✓	✓	✓	✗
Interactive Query System	✓	✓	✗	✗	✗
Visual Output (Built-in)	✓	✗	✓	✓	✓
Local Execution / Privacy	✓	✗	✗	✗	✓
Direct Output Editing	WIP	✓(prompt)	✗	✓	✗
Low Latency	✗	✓	✓	✓	✓
Unlimited Use	✓	✓	✗	✗	✓

Overall, IUFlowGen stands out for its ability to handle end-to-end procedural reasoning and visualization locally, with user-guided refinement and semantic transparency. Its hybrid approach makes it particularly suited for secure, technical, or educational environments where privacy, control, and structure matter most.

7.5 Future Work

The IUFlowGen system holds significant potential for future development and practical deployment. One promising direction lies in optimizing the system's performance, particularly in the document processing pipeline. Currently, embedding large documents and constructing semantic graphs in real time leads to considerable latency and resource consumption. Future versions of the system could benefit from task-specific lightweight embedding models and asynchronous processing techniques to accelerate flowchart generation while reducing memory and compute overhead.

Another critical area for improvement is the user interface and user experience (UI/UX). While the current prototype provides essential interactivity, a more robust front-end design with intuitive controls and real-time feedback would enhance usability. For instance, integrating adjustable layout settings, interactive flowchart previews, and guided prompts could support a smoother user experience for both novice users and expert verifiers. Packaging the system as a web-based application or mobile-ready platform would also facilitate broader accessibility and deployment in education, legal, and enterprise settings.

A particularly impactful enhancement would be enabling direct flowchart editing within the system. At present, users and verifiers must manually modify the output diagram or redraw their output if corrections are required. Future iterations of IUFlowGen should support in-system modifications—such as adding, deleting, or repositioning nodes and edges—through drag-and-drop functionality or context-aware suggestions. This would reduce the cognitive load on users, improve verification workflows, and allow the system to evolve into a fully collaborative visual modeling platform.

These proposed enhancements aim to transform IUFlowGen from a procedural prototype into a scalable, accessible, and professional-grade tool for structured document understanding and visualization. Such improvements would not only extend the system's functionality but also support its adoption in real-world decision-making and information design contexts.

Chapter 8
CONCLUSION

The IUFlowGen system represents a significant advancement in AI-assisted procedural understanding and document visualization. By integrating large language models (LLMs), retrieval-augmented generation (RAG), and graph-based rendering techniques, IUFlowGen transforms complex textual instructions into structured, interpretable flowcharts. Its modular, locally executed design emphasizes transparency, reproducibility, and user control—making it suitable for domains where data privacy and clarity are paramount.

A core achievement of IUFlowGen is the implementation of a multi-stage pipeline that decomposes procedural text into steps, entities, and relationships, and reconstructs them to form coherent diagrams. This pipeline, supported by prompt engineering, augmented data retrieval, and regular expression post-processing, ensures that outputs are logically faithful to the source material. The system is further distinguished by its dual-mode visualization—offering both overview and detailed charts—and an integrated query mechanism for user clarification.

IUFlowGen also includes a full-stack application that supports document upload, flowchart rendering, and interactive exploration—all without reliance on cloud-based services. This enables deployment in secure or offline environments and makes the system a privacy-conscious alternative to commercial AI tools. In user studies, participants assisted by IUFlowGen consistently outperformed unaided users in both time and accuracy, especially on higher-complexity documents—validating its impact on real-world productivity and comprehension.

Most importantly, the system has successfully fulfilled its core objectives: translating long and complex procedural documents into usable flowcharts in a way that is both efficient and understandable. The implications of this are broad—IUFlowGen can be extended to assist users in navigating legal regulations, academic research papers, technical standards, and policy documents. In each of these domains, where stepwise logic and detailed interpretation are essential, the system can serve as an assistive bridge between unstructured text and human understanding.

Despite these contributions, IUFlowGen still faces limitations. The system requires considerable time and compute resources during the document processing stage due to simultaneous vector embedding and entity extraction. Additionally, it currently supports only one active document session at a time and lacks built-in editing capabilities for modifying flowcharts directly within the interface. These limitations highlight opportunities for future enhancement, including session concurrency, interface interactivity, and performance optimization.

In summary, IUFlowGen provides a robust and extensible foundation for AI-guided flowchart generation. It offers an effective, privacy-preserving, and user-friendly framework for translating procedural documents into actionable visual knowledge—supporting domains such as education, legal documentation, technical writing, and systems analysis.

References

[1] L. T. Phong, "Quy trinh, thuc trang xay dung nghi dinh o viet nam." https://moj.gov.vn/qt/tintuc/Pages/nghien-cuu-trao-doi.aspx?ItemID=2548, June 2021.

[2] V. strategy, "The influence of visualization strategy on reading comprehension ability." https://www.researchgate.net/publication/338161080_THE_INFLUENCE_OF_VISUALIZATION_STRATEGY_ON_READING_COMPREHENSION_ABILITY, June 2019.

[3] X. Rong, "word2vec parameter learning explained." https://arxiv.org/abs/1411.2738, Nov 2014.

[4] Ashish Vaswani, Noam Shazeer, et al., "Attention is all you need." https://arxiv.org/abs/1706.03762, Jun 2017.

[5] Graphviz, "Graphviz - tse93." https://www.graphviz.org/documentation/TSE93.pdf, May 1993.

[6] Tom B. Brown, Benjamin Mann, et al., "Language models are few-shot learners." https://arxiv.org/abs/2005.14165, May 2020.

[7] C. Y. Baoguang Shi, Xiang Bai, "An end-to-end trainable neural network for image-based sequence recognition." https://arxiv.org/abs/1507.05717, July 2015.

[8] F. M. Zanzotto, "Viewpoint: Human-in-the-loop artificial intelligence." https://doi.org/10.1613/jair.1.11345, Feb 2019.

[9] Patrick Lewis, Ethan Perez, et al., "Retrieval-augmented generation for knowledge-intensive nlp tasks." https://arxiv.org/abs/2005.11401, May 2020.

[10] Shuyin Ouyang, Jie M. Zhang et al., "An empirical study of the non-determinism of chatgpt in code generation." https://arxiv.org/abs/2308.02828, Aug 2023.

[11] Zirui Guo, Lianghao Xia, et al., "Lightrag: Simple and fast retrieval-augmented generation." https://arxiv.org/abs/2410.05779, Oct 2024.

[12] Abdul Arbaz, Heng Fan, Junhua Ding et al., "Genflowchart: Parsing and understanding flowchart using generative ai." https://doi.org/10.1007/978-981-97-5492-2_8, July 2024.

[13] V. A. Shivali Agarwal, Shubham Atreja, "Extracting procedural knowledge from technical documents." https://arxiv.org/abs/2010.10156, Oct 2020.

[14] Mika Sie, Ruby Beek, Michiel Bots et al., "Summarizing long regulatory documents with a multi-step pipeline." https://arxiv.org/abs/2408.09777, Aug 2024.

[15] G. L. et al., "Long document summarization with workflows and gemini models." https://cloud.google.com/blog/products/ai-machine-learning/long-document-summarization-with-workflows-and-gemini-models, May 2024.

[16] O. Team, "Gpt-4o system card." https://arxiv.org/abs/2410.21276, Oct 2024.

[17] E. Team, "Ai flowchart generator." https://www.eraser.io/ai/flowchart-generator.

[18] N. Team, "Notegpt: Your all-in-one ai learning assistant." https://notegpt.io/.

[19] Hugo Touvron, Thibaut Lavril et al., "Llama: Open and efficient foundation language models." https://arxiv.org/abs/2302.13971, Feb 2023.

[20] Jinze Bai, Shuai Bai et el., "Qwen technical report." https://arxiv.org/abs/2309.16609, Sept 2023.

[21] S. S. Sengar, A. B. Hasan, S. Kumar, and F. Carroll, "Generative artificial intelligence: A systematic review and applications." https://arxiv.org/abs/2405.11029, May 2024.

[22] Jason Wei, Xuezhi Wang et al., "Chain-of-thought prompting elicits reasoning in large language models." https://arxiv.org/abs/2201.11903, Jan 2022.

[23] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space." https://arxiv.org/abs/1301.3781, Jan 2013.

[24] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks." https://arxiv.org/abs/1908.10084, Aug 2019.

[25] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding." https://arxiv.org/abs/1810.04805, Oct 2018.

[26] A. Andoni, P. Indyk, and I. Razenshteyn, "Approximate nearest neighbor search in high dimensions." https://arxiv.org/abs/1806.09823, June 2018.

[27] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus." https://arxiv.org/abs/1702.08734, Feb 2017.

[28] Neo4j, "Literature review about neo4j graph database as a feasible alternative for replacing rdbms." https://www.researchgate.net/publication/307180380, Dec 2015.

[29] O. Lassila and R. R. Swick, "Resource description framework (rdf) model and syntax specification." https://www.w3.org/TR/1999/REC-rdf-syntax-19990222/, Feb 1999.

[30] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension." https://arxiv.org/abs/1910.13461, Oct 2019.

[31] T.-T. Tran and H.-D. Le, "Iu-smartcert: A blockchain-based system for academic credentials with selective disclosure." https://www.researchgate.net/publication/356203713_IU-SmartCert_A_Blockchain-Based_System_for_Academic_Credentials_with_Selective_Disclosure, Nov 2021.

Appendix A
LISTINGS

Appendix A: Experimental Documents

Document 1: Level 1 (lv1_planting_tree.pdf)

How to Plant a Tree the Right Way (8 Steps Total)

Planting a tree isn't just about digging a hole and placing it in. When done properly, tree planting can help the tree thrive for decades. Follow these steps to give your tree the best possible start.

Step 1: Choose the Right Tree for Your Location Before planting, consider the climate, soil type, and sunlight in your area. Some trees thrive in full sun, while others prefer partial shade. Also, think about the tree's mature size—avoid planting large trees too close to buildings, power lines, or underground utilities. A little research now will prevent major issues later.

Step 2: Pick the Best Planting Spot Once you've selected the tree, find a planting spot that offers the right amount of light and space. Make sure the area drains well—avoid locations where water tends to pool after rain. Check that the soil isn't too compacted and that there's enough room for the tree's roots to spread out comfortably.

Step 3: Prepare the Tree and Tools Unpack your tree from its container or burlap wrap. If it's in a plastic pot, gently slide it out. Loosen the root ball slightly by using your hands or a small tool to tease apart circling roots. This helps the roots spread into the surrounding soil instead of continuing in tight loops. Also, gather your tools: a shovel, gardening gloves, water, mulch, and possibly stakes for support.

Step 4: Dig a Wide, Shallow Hole Dig a hole that is twice as wide as the root ball but no deeper than its height. This width gives roots space to grow outward, while the proper depth prevents the tree from sinking too low. Place the tree in the center of the hole. The top of the root ball should be level with or just slightly above the surrounding ground. Planting too deep is a common mistake that can suffocate roots.

Step 5: Position and Straighten the Tree Before filling the hole, step back and check the tree from all angles. Make sure it's standing straight —this matters more than you might think! Ask a friend to help hold the tree in place if necessary. Adjust its position so that the best-looking side of the tree faces your preferred direction, such as toward your house or garden path.

Step 6: Fill the Hole and Firm the Soil Begin filling the hole with the original soil you dug out, gently tamping it down as you go to remove air pockets. Don't press too hard—compacting the soil too much can block water and air. Water the soil halfway through the process to help it settle, then finish filling and gently firm the top layer.

Step 7: Water Deeply and Add Mulch Right after planting, give the tree a thorough watering. This helps the roots settle in and encourages contact between the roots and the soil. Then, apply a 2–3 inch layer of mulch in a circle around the base of the tree—but keep the mulch a few inches away from the trunk. This mulch helps retain moisture, regulate soil temperature, and reduce weed growth.

Step 8: Monitor and Care for the Tree For the first year, your new tree will need regular care. Water it deeply once or twice a week, especially during dry spells. Check the soil

moisture by sticking your finger a few inches into the ground. Avoid fertilizing in the first year—focus on root development instead. If necessary, stake the tree for support, but be sure to remove the stakes after one year to prevent girdling. With patience and care, your tree will grow strong and healthy for years to come.

Document 2: Level 2 (lv2_setup_dual_boot.pdf)

How to Set Up a Dual Boot System (Windows + Ubuntu) the Right Way (11 Steps Total) Installing a dual-boot system with Windows and Ubuntu gives you the flexibility to switch between two powerful operating systems on the same machine. While the process is manageable, it must be done carefully to avoid data loss or boot issues. This guide walks you through the process from preparation to post-installation testing.

Understand BIOS/UEFI and Secure Boot Settings

Before you begin downloading or partitioning anything, it's important to understand how your system boots. Most modern computers use UEFI, while older systems use the traditional BIOS. This matters because Ubuntu handles UEFI and BIOS booting differently. Restart your computer and enter the firmware settings by pressing a designated key during boot-up, such as F2, DEL, or ESC. Inside the settings menu, check if Secure Boot is enabled. Secure Boot prevents unauthorized operating systems from loading and can interfere with Ubuntu's installation. If your Ubuntu version supports Secure Boot, you can leave it on, otherwise disable it temporarily during installation. Also check that your system is set to boot in UEFI mode rather than Legacy mode, as this ensures smoother compatibility with both Windows and Ubuntu on the same drive.

Back Up Your Data

Before making any changes to your system, it's essential to back up all of your important data. Copy your documents, photos, and work-related files to an external hard drive or a cloud storage service. Dual booting involves resizing partitions and modifying boot settings, which can occasionally go wrong. Having a backup ensures you can recover your information in the event of accidental deletion or disk corruption. Do not skip this step even if you feel confident—the risk is small but never zero.

Check Windows System Requirements and Disk Space

Next, ensure your current Windows system has enough free disk space to accommodate Ubuntu. Open File Explorer, right-click your main hard drive (usually C:), and select Properties. Ubuntu requires at least 20 GB of space, but 50 GB or more is recommended for practical use. Then, press Win + R, type "msinfo32," and press Enter. This opens the system summary window. Look for the "BIOS Mode" field—if it says UEFI, your system is using modern firmware, which is ideal. If it says Legacy, the setup process will be slightly different but still possible.

Create a Partition for Ubuntu

To make space for Ubuntu, open Windows Disk Management by pressing Win + X and choosing "Disk Management." Find your main drive, right-click it, and choose "Shrink Volume." Specify the amount of space you want to shrink—usually at least 50,000 MB for 50 GB. After shrinking, you will see a new section labeled "Unallocated." This space will be used by Ubuntu during installation. Do not format or assign a drive letter to it in Windows.

Download Ubuntu and Create a Bootable USB Drive

Visit ubuntu.com and download the latest LTS (Long Term Support) version of Ubuntu. Save the ISO file to your computer. Next, download Rufus or another USB writing tool. Insert a USB drive with at least 4 GB of space and open Rufus. Select the ISO file and make sure the partition scheme matches your system (GPT for UEFI). Click Start and wait

for the process to finish. When it's done, safely eject the USB drive.

Boot from the USB Drive

Insert the bootable USB into your computer and restart it. Enter the boot menu by pressing the appropriate key (F12, F2, ESC, or DEL, depending on your system) and select your USB device. If you see two USB options, choose the one marked UEFI. The Ubuntu boot menu will load, giving you the option to try Ubuntu or install it directly. Select "Install Ubuntu" to begin the installation process.

Begin the Installation Process

The Ubuntu installer will guide you through selecting your language, keyboard layout, and whether to download updates or third-party software during installation. When asked how you want to install Ubuntu, you'll likely see the option "Install Ubuntu alongside Windows Boot Manager." Choose this option if available, as it simplifies the process. If it's not shown, choose "Something Else" to manually assign the unallocated space to Ubuntu. In the partition editor, create a root partition (mount point "/") using the unallocated space. You may also create a swap partition if desired, though Ubuntu handles swap files automatically on newer systems.

Set Up User Details and Finalize Installation

You'll be asked to create a user account, set a password, and name your computer. You can choose whether to log in automatically or require a password at each login. After entering these details, the installer will begin copying files to your system. This process can take anywhere from 10 to 30 minutes depending on your hardware. When the installation is complete, you will be prompted to remove your USB drive and reboot the system.

Test GRUB and Boot Both Systems

On reboot, the GRUB bootloader menu will appear, allowing you to choose between Ubuntu and Windows. Use the arrow keys to select the system you want to boot into. Start by testing Ubuntu to ensure the installation was successful. Check your Wi-Fi, display resolution, and software settings. Then reboot and select Windows to verify that your existing data and programs are still intact. If GRUB does not appear and your system boots directly into Windows, you may need to enter your BIOS and set Ubuntu as the default boot device.

Troubleshoot Boot Issues if Necessary

If Ubuntu boots but Windows does not appear in the GRUB menu, boot into Ubuntu and open a terminal. Run the command "sudo update-grub" and reboot. GRUB will scan for all available operating systems and should add Windows automatically. If your computer still does not show the boot menu, double-check your BIOS boot order or consider using a tool like Boot-Repair from a live Ubuntu session. Avoid making changes unless you're sure what they do, as incorrect bootloadermodifications can make both systems unbootable.

Maintain Your Dual Boot Setup

Now that both systems are installed and working, it's important to keep them updated. In Windows, continue using Windows Update as usual. In Ubuntu, open the terminal and run "sudo apt update and sudo apt upgrade" regularly. If you ever need to remove Ubuntu, you must delete its partitions and restore the Windows bootloader, which requires using a Windows recovery drive or bootable media. Likewise, if you upgrade to a new version of Windows, it might overwrite the GRUB bootloader—so always have a live Ubuntu USB handy just in case you need to repair the boot menu again.With your dual boot system complete and stable, you now have the flexibility to switch between Windows for daily use and Ubuntu for development, experimentation, or open-source tools. Remember to back up both systems periodically and stay informed on updates for both platforms.

Document 3: Level 3 (lv3_Deploy_AI_System.pdf)

Choose Infrastructure (Cloud or On-Prem)

Begin by evaluating the operational paradigm most suitable for deploying your AI workloads. Public cloud environments—offered by vendors such as AWS, Azure, or Google Cloud—provide access to elastic compute, managed Kubernetes orchestration, and integrated storage and monitoring stacks. These platforms are highly adaptable and particularly advantageous in use cases that demand scalability, rapid provisioning, and minimal infrastructure maintenance. Alternatively, if data localization laws, compliance requirements, or latency-sensitive inference demand localized control, an on-premise deployment may be necessary. In this scenario, you'll assume complete responsibility for hardware, networking, and cluster management. If cloud deployment is selected, continue with Provision Cloud Infrastructure. If an on-premise approach is chosen, proceed to Set Up Bare-Metal or VMs.

Provision Cloud Infrastructure

Within the cloud model, begin by provisioning GPU-accelerated instances capable of supporting high-throughput inference orkloads. Deploy a container orchestration layer using managed Kubernetes services like EKS, AKS, or GKE. IAM roles must be tightly scoped, virtual networks properly segmented, and storage classes defined to support both ephemeral and persistent data flows. Leverage erraform or Pulumi to codify your infrastructure so that environments remain reproducible and auditable. Configure telemetry agents, centralized logging, and secrets management integrations (e.g., AWS Secrets Manager or Azure Key Vault) to ensure compliance with security and observability baselines. Once your cloud control plane and supporting services are in place, continue with Select Model Serving Platform.

Set Up Bare-Metal or VMs

For on-premise deployment, provision physical servers or virtual machines equipped with compatible GPUs such as the NVIDIA A100 or H100. Container runtime environments like Docker or CRI-O must be installed and configured, and Kubernetes orchestration should be bootstrapped via kubeadm, Rancher, or equivalent tooling. Networking must be explicitly defined using static IP assignment, internal DNS, and possibly VLAN tagging. High-throughput local storage—typically NVMe-based—should be mounted using LVM or RAID strategies. Security measures such as firewall zoning, VPN tunneling, and device level hardening are essential. Following this foundational setup, continue with Select Model Serving Platform.

Select Model Serving Platform

Next, identify a model inference engine compatible with your deployment goals and supported model formats. NVIDIA Triton Inference Server, TensorFlow Serving, or TorchServe are among the preferred options, especially where dynamic batching, GPU multiplexing, or multi-model deployment is required. Encapsulate the model, its preprocessing pipeline, and health-check interfaces within a container. Define exposure methods via REST or gRPC depending on downstream system compatibility. Helm or Kustomize can be employed to template deployments and ensure repeatability across environments. After your model serving platform is prepared, proceed to Deploy Inference Controller.

Deploy Inference Controller

Deploy the model-serving containers into your orchestration fabric using StatefulSets or Deployments, depending on whether session affinity or persistence is needed. Set up ConfigMaps for runtime parameters and Kubernetes Secrets for sensitive credentials. Attach PersistentVolumeClaims if model files are dynamically pulled or logs are retained. Use Horizontal Pod Autoscalers (HPA) or KEDA to scale replicas based on system load or

request frequency. Apply node selectors, tolerations, and affinity rules to bind workloads to GPU-enabled nodes. Include readiness and liveness probes to support high availability. Once the inference infrastructure is deployed, branch immediately into parallel configuration of the following components: Set Up Input Channels, Set Up Output and Storage Systems, Install Metrics and Addons, and Configure RBAC and Security.

Set Up Input Channels

Enable ingress paths that feed real-time or batch inputs into your inference service. This may involve HTTP endpoints, WebSocket streams, or message queue subscribers using Kafka or RabbitMQ. Input schemas must be enforced using validation libraries, and authentication mechanisms such as OAuth2 or JWT must be implemented at entry points. Leverage API gateways or mesh ingress controllers like Istio Gateway or Envoy Proxy to manage routing, retries, and rate limiting. Ensure that trace and correlation IDs are attached to each inbound request for auditability and observability. Because these entry points are externally exposed, ensure that their configuration is tightly coupled with the RBAC and security policies defined in Configure RBAC and Security to maintain a secure interface surface.

Set Up Output and Storage Systems

Configure output pipelines to persist prediction results or forward them downstream. Structured outputs may be directed to relational databases like PostgreSQL, while unstructured blobs or batch exports can be sent to object stores like Amazon S3 or Azure Blob. Redis or Memcached can support short-lived caching of responses. Ensure encryption at rest for all output data and enable TLS across all outbound interfaces. Streaming outputs to data pipelines or dashboards may require CDC mechanisms or dedicated connectors. Integrate output metadata with data catalogs for lineage tracking and auditing. Once output storage is operational, the system is ready to accept and persist results from live inference traffic.

Install Metrics and Addons

Deploy observability tooling that enables visibility into all operational aspects of the system. Prometheus should be configured to scrape performance and system metrics, while Grafana visualizes latency and throughput. Fluent Bit or Fluentd can route logs to Elasticsearch, Loki, or a central SIEM platform. Use OpenTelemetry to instrument distributed traces and correlate service spans. Define dashboards that reflect system health, SLOs, and capacity planning trends. Set up alerts for anomalies, errors, and degradation, notifying responders via channels such as Slack, PagerDuty, or Opsgenie. Once observability components are in place and verified, continue to Validate the AI System to monitor system behavior under realistic conditions.

Configure RBAC and Security

Apply RBAC policies within Kubernetes to restrict access to APIs, secrets, and runtime logs. Enforce mTLS between services to prevent eavesdropping and service spoofing. Use ingress controllers to filter inbound IPs and limit request rates. Secrets must be managed with tools like HashiCorp Vault, Sealed Secrets, or external secrets operators. Perform static and dynamic scans of containers, and enforce AppArmor or SELinux profiles on nodes to restrict privileged operations. Where applicable, implement DLP scanners to prevent exfiltration of sensitive data through inputs or outputs. Once these security controls are fully applied, proceed to Validate the AI System to finalize the secure deployment perimeter.

Validate the AI System

Final system validation includes stress testing, behavioral verification, and resiliency analysis. Tools like Locust, Artillery, or k6 simulate user demand and reveal performance boundaries. Confirm that autoscaling logic triggers correctly and that observability data re-

flects real-time system conditions. Authentication and authorization flows must be retested under load. End-to-end trace analysis should be conducted to ensure no bottlenecks exist across the inference path. Outputs are validated against test datasets or live sampling tools. If validation exposes critical issues—such as misconfigurations, failed routing, or unmet performance thresholds—you must rollback to Deploy Inference Controller, revise deployment manifests, and re-execute the deployment iteration. Upon successful validation, the system may be marked production-ready and released under a monitored rollout policy.

Document 4: Level 4 (lv4_IU_Smartcert.pdf)

This section outlines the proposed system IU-SmartCert, a blockchain-based educational credential management system with a selective disclosure option.

Our system has 3 main groups of users: issuers, learners, and relying parties. An issuer can be broadly understood to include individuals or organizations qualified to issue a credential. A learner is a person who received the issued credential. A relying party in this paper is used to refer to a person or a company who uses and wants to check the validity of a credential.

Figure 1 shows key functions and typical interactions among users of our system. First, given a set of credentials to issue, the issuer organizes each credential into our proposed format that composes of two parts: a mandatory component and a list of optional components, then inputs the fingerprint of all credentials to IU-SmartCert. From the input fingerprints, the system constructs a Merkle tree and publishes the root node of the tree along with the issuer identity to the public Ethereum blockchain as a smart contract. After publishing, the issuer sends a digital credential and its receipt to each learner. Later when the learner needs to show her his credential to a relying party, like an employer, s/he can select the most relevant optional components of the credential to present along with the mandatory component to the employer. Finally, a relying party verifies the authenticity and the integrity of a given credential by checking the received data against the public key infrastructure and the Ethereum public blockchain. The verification process can be done independently without contacting the issuer, and even without using the IU-SmartCert system.

Formally, our blockchain-based credential management system provides functions to

R1. Define and issue credentials with a mandatory component and optional components for selective sharing,

R2. Verify and validate a credential,

R3. Select optional components of the credential to disclose,

R4. Revoke a credential issued by the system In addition, to increase user's flexibility and initiative, the system should have the following quality attributes:

R5. Ability to independently check the integrity and validity of a credential

R6. Security for credentials. In the following sections, we describe in detail important procedures in the IU-SmartCert system.

3.1 Defining a Credential Schema

In IU-SmartCert, issuers of credentials like universities and institutions can define their own schema and vocabularies of a credential for each program. A credential in IU-SmartCert composes of two parts:

– One mandatory component

– and a list of optional components

A mandatory component is one that learner must disclose to every replying party. This component stores all information needed to validate and evaluate a credential. For instance,

in a credential of an undergraduate student, the mandatory component is the diploma which contains the name of the university, the name of the learner, the issued date, and the enrollment year.

The list of optional components is a tool for issuers to define a flexible credential data model. An issuer can issue credentials with different levels of granularity, therefore allow learners to selectively disclose their credential on demand. For instance, a university could issue a transcript of a student by issuing the score of each course separately as a list of optional components. This credential schema allows the student to choose courses to disclose to a relying party. Conversely, if a university does not allow students to cherry-pick courses to disclose, the university issue the entire transcript as a single component of the credential. Moreover, in a stricter rule that requires students to disclose the transcript along with their diploma, the university can combine the diploma and the transcript into a single mandatory component in the input to the IU-SmartCert. Therefore, the structure helps to fulfill the requirement R1 of the system.

The procedure to define a schema and vocabularies in IU-SmartCert is performed via the data input of an issuer. For every credential, the issuer must provide a list of files corresponding to each component in the credential with the following naming rule

CredentialID.ComponentName.(R).ext

where

– CredentialID is the identity of a credential and all components of a credential will share the same value.

– ComponentName is the name of a component.

– .(R) is the marker for the mandatory component. An optional component's filename does not include this marker.

– .ext is the file extension.

For instance, consider a bachelor's credential composes of a diploma, a transcript and a scientific profile. The issuer could define that the diploma is a mandatory component, while the transcript and the scientific profile are optional components. In such case, the input to the IU-SmartCert of a credential with identifier ITIU01 is 3 pdf files named as following

IUIT01.diploma.

(R).pdf IUIT01.transcript.pdf

IUIT01.profile.pdf

3.2 Issuing Credentials

When learners complete a program, the institution generates digital credentials of learners and passes them to the IU-SmartCert system. The system then organizes and publishes the digital fingerprint of those credentials to the Ethereum blockchain in such a way that learners can choose components of their credential to disclose and a relying party can validate a given credential (Fig. 2).

Constructing Merkle Tree

To begin this procedure, we use the schema and the digital fingerprint of each credential to construct a Merkle tree [13]. We first combine the identity of the credential with the content of the component and its type, mandatory or optional, defined by the schema, then uses SHA-256 to calculate the hash value of the combination and create a leaf node in the Merkle tree. We apply this procedure to all components of all input credentials and obtain corresponding leaf nodes. From those leaf nodes, we build the rest of the tree by concatenating two leaf nodes, calculating a new hash value resulting in a parent node, and continuing up toward the root of the Merkle tree. We can construct a Merkle tree from any arbitrary number of credentials and always results in a single root node, so an institution could issue credentials in batch to save time and effort.

Publishing Data as a Smart Contract

Once the Merkle tree is built, the IU-SmartCert system publishes the hash value of the root node and supporting data to the Ethereum blockchain as a smart contract so that a relying party can validate an issued credential independently without contacting the issuer. The smart contract (as shown in Fig. 3) consists of

– institute A read-only variable for the hash of the issuer name. The issuer name is the organization field in the issuer's X.509 certificate, and thus binds to the identity of the issuer.

– MTRoot A read-only variable for the hash of the root of the Merkle tree.

– revocationList A list of revoked credentials along with a reason of the revocation. The function is accessible only for the issuer which is the owner of the contract. See Sect. 3.5 for the revoking procedure.

– verify(bytes32[], bytes32) A function to check whether a component belongs to the Merkle tree represented by the root node stored in MTRoot of the contract.

– isValid(bytes32) A function to check whether a credential is revoked. If the credential is revoked, the function returns false with a reason, otherwise, returns true.

– revoke(bytes32, string) A function to revoke a credential. See Sect. 3.5 for the revoking procedure. It is worth noting that the smart contract stores only the root node of the Merkle tree constructed from the batch of credentials, and no other data about the credentials are published. When the deployment of the smart contract is confirmed, the system keeps the metadata to generate receipts for learners.

3.3 Generating Receipts

After publishing credentials to Ethereum network, the system generates a receipt for each credential and sends it along with the digital credential to the corresponding learner. A relying party will use it to verify and validate the credential.

A receipt contains metadata of the smart contract, a proof of existence in the Merkle tree of each component of the credential, and the X.509 certificate of the issuer.

The metadata in a receipt is to help a relying party to verify the identity of the issuer and the validity of the smart contract. The metadata consists of the address of the smart contract that manages the credential, the hash value of the transaction that deployed the smart contract and the identity of the issuer. To provide a verifiable identity, we use a hybrid approach [6] where we establish a binding between accounts in Ethereum and an X.509 public key certificate of the issuer. This X.509 certificate is issued by a trusted certificate authority (CA) of the traditional public key infrastructure (PKI), which comprises of hardware and software for creating, managing, distributing, using and revoking public keys. At registration time, CA verifies the identity of the issuer and writes its information into the X.509 certificate; then, CA digitally signs this certificate with its private key [1]. Therefore, originality of digital credentials in this system is guaranteed. Before using our system IU-SmartCert, the issuer needs to endorse the Ethereum account used to publish credentials by signing the respective address with the private key of the X.509 certificate. Then, the issuer input to the system the address of its Ethereum account, the signature, and its X.509 certificate chain. Later, replying parties like employers can retrieve from the receipt the information to verify them versus their trusted certificate authority, thereby authenticating the identity of the issuer and the validity of the smart contract.

In a receipt, the proof of existence of a component is extracted from the Merkle tree build in the issuing phase. Since each component corresponds to a leaf node of the tree, its proof of existence is a list of nodes in the Merkle tree required to recalculate the path from the corresponding leaf node to the root node [13]. In an example in Fig. 4, the proof of existence of component 3 is the hash value Hash 4 and the hash value Hash 12. From

these proofs, one can calculate the path from component 3 to the root node.

Finally, the system generates one receipt for each credential as a file in JSON data format (as shown in Fig. 5) with the following fields:

– issuedOn: the time when the credential was published to the blockchain

– transactionHash: the hash value of the transaction that deployed the smart contract that manages the credential

– contractAddress: the address of the smart contract that manages the credential

– credentialID: the identity of the credential

– components: the data to prove the authenticity of the credential. For each component of the credential, the data includes

• name: the name of the component

• mandatory: a boolean value indicating whether the component is mandatory or not

• proof: the proof of existence of the component in the Merkle tree

• hash: a hash value of the concatenation of the credential's identity, the type, and the content of the component.

– issuer: the verifiable identity of the issuer

• ethereumAccount: the Ethereum account of the issuer

• ethereumAccountSignature: the signature of the issuer's Ethereum account endorsed by the private key of the X.509 certificate

• IssuerCertificateChain: the chain of the X.509 certificates of the issuer in PEM format

3.4 Exchanging Credentials

In our proposed system, learners can exchange their credentials with a selective disclosure option. In other words, learners can choose components to share, and also can choose components not to share while following the schema defined by issuers (Requirement R3).

The exchanging credentials begin when an employer requests the learner to present his credential. First, the learner takes his credential and picks the most relevant components to share. This selection is possible because credentials in our system are organized into two parts a mandatory component and a list of optional components. And since the type of each component is defined by the issuer, the learner can freely pick and skip some optional components without invalidating the credential.

After the selection, the learner needs to generate a new receipt which is the proof of existence for those selected components. He can upload his original credential's receipt to the IU-SmartCert system, select the chosen components, and download a new receipt. On the other hand, the learner can make the new receipt on his own without using the system. He can open his receipt in a text editor, remove the sections corresponding to the components that are not chosen, and save the file as a new receipt.

For example, in Fig. 5 there are two valid receipts of one credential. The receipt on the left is used when the learner would like to show all of the three components of his credential. The receipt in the right is used when the learner chooses not to share the ScienceProfile component. Finally, the learner sends the new receipt and related files to the employer.

3.5 Revoking Credentials

Like other credential management systems, the IU-SmartCert allows issuers to revoke issued credentials (Requirement R4). While in IU-SmartCert learners can select components to present to relying parties, they always have to present the mandatory component of the credential otherwise the credential is invalid. With that structure, issuers to revoke an issued credential by marking its mandatory component as revoked. In detail, we store in the smart contract that manages the issued credentials a list of revoked components along with the reason for their revocation. And only the owner of the smart contract, i.e. the issuer of the credential, can add a record to that revocation list. Later, any relying party can check

the status of a credential by checking its mandatory component against the revocation list. In Fig. 3 the data and the functions for the revocation are the revocationList, the revoke, and the isValid functions.

3.6 Verifying a Credential

An employer or any relying party can verify a credential issued by IU-SmartCert from the files, the receipt and the Ethereum network without contacting the issuing institution. The credential verifying process includes the following steps:

1. Check the receipt information and the issuer's information

– The issuer's X.509 certificate – The signature of the Ethereum account used to issue the credential.

– Validity of the owner of the smart contract on Ethereum.

– The name of the issuer in the smart contract and the name on the X.509 certificate.

2. Check the integrity of all components of the credential. This step checks the number, the type and the hash value of the files against the value stored on the receipt.

3. Check the validity of the components in the certificate. This step checks the hash value and proofs to confirm whether the credential belongs to the merkle tree whose root node is published on the Ethereum blockchain.

4. Check whether the certificate is revoked. This step checks if the mandatory component of the credential in the list of revoked credentials stored in the smart contract on the Ethereum blockchain. The above procedure is implemented in IU-SmartCert with a progress bar for each step as shown in Fig. 6. On the other hand, relying parties can perform the verifying procedure on their own. Once received a credential and its receipt, a relying party can extract all the data to verify a credential without the involvement of the issuer. Then with a connection to the Ethereum blockchain and any publicly available tools to verify a digital signature and a X.509 certificate, the relying party can finish the verification procedure. That helps us fulfill the requirements R2 and R5.