# POLITECNICO DI TORINO

## Master's Degree in Computer Engineering

Master's Degree Thesis

# Leveraging LLMs for Automated Technical Documentation of RPA Workflows: A UiPath Case Study

**Supervisors**

Prof. Paolo Garza

Dott. Giovanni Vanini

**Candidate**

Federico Spinoso

March 2026

## Abstract

Technical documentation proves to be crucial in the field of Software Engineering, serving as a key asset during the lifecycle of software in tasks such as maintenance, knowledge transfer, and governance. Despite its long-term importance, such tasks are often deprioritized due to the manual effort and time required. This may lead to nonexistent, incomplete, or outdated manuals that not only could reduce collaboration and clarity, but also possibly introduce operational risk. This problem is particularly relevant in the domain of Robotic Process Automation, where automation projects evolve rapidly and documentation tends to diverge from the actual implementation over time. Inspired by previous work on the subject, this thesis explores the use of LLMs specifically to automate the generation of technical documentation for RPA projects built with UiPath, a widely adopted enterprise platform. The work presents an end-to-end system that first extracts structured information from project files through deterministic analysis and then uses that information as input for LLM-based generation. Three commercially available Large Language Models are compared as documentation generators across several projects of varying complexity, and a dual evaluation approach is adopted: one based on quantitative text-similarity metrics and one based on qualitative assessment performed by a separate, more capable LLM acting as an evaluator. The results indicate that all tested models are capable of generating structured and usable documentation. However, traditional text-overlap metrics alone do not fully capture the overall quality of the generated output, whereas the inclusion of qualitative evaluation methods provides a more comprehensive assessment. This work was conducted in collaboration with Poseidon SB, a consulting company specializing in effective and rapid digitization of business processes through automation and other modern technologies.

# Table of Contents

**List of Tables** VI

**List of Figures** VIII

**Acronyms** IX

**1 Introduction** 1
1.1 Background . . . 1
1.2 Motivation . . . 2
1.3 Problem Statement . . . 3
1.4 Research Objectives . . . 4
1.5 Poseidon SB . . . 4

**2 State of the Art** 5
2.1 Software Documentation . . . 5
2.1.1 Different kinds of Documentation . . . 6
2.1.2 Technical documentation types . . . 7
2.2 Artificial Intelligence . . . 8
2.3 Natural Language Processing . . . 10
2.3.1 History of NLP . . . 11
2.3.2 Approaches to NLP . . . 12
2.4 Generative AI . . . 13
2.4.1 History of Generative AI . . . 14
2.5 Large Language Models . . . 15
2.5.1 History of LLMs . . . 16
2.5.2 Architecture . . . 17
2.5.3 Data preprocessing . . . 21
2.5.4 Attention Mechanisms . . . 22
2.6 Extensibility Techniques in Large Language Models . . . 24
2.6.1 Model Fine-Tuning . . . 24
2.6.2 Prompting Techniques . . . 25

2.6.3 Retrieval-Augmented Generation (RAG) . . . . . . . . . . . 25
2.7 Evaluation Metrics and Limitations of LLMs . . . . . . . . . . . . . 26
2.7.1 Evaluation Metrics . . . . . . . . . . . . . . . . . . . . . . . 26
2.7.2 Limitations of Current LLM Technologies . . . . . . . . . . 27
2.8 Agentic AI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
2.8.1 Multi-Agent Patterns and Applications . . . . . . . . . . . . 29
2.8.2 Popular LLM-Based Architectures and Frameworks . . . . . 30
2.8.3 Evaluation of LLM-Based Agents . . . . . . . . . . . . . . . 30
2.9 Existing approaches to Automated Software Documentation . . . . 32
2.9.1 Rule-Based Template Systems . . . . . . . . . . . . . . . . . 32
2.9.2 Information-Retrieval Systems . . . . . . . . . . . . . . . . . 33
2.9.3 AI and LLM-Based Systems . . . . . . . . . . . . . . . . . . 34
2.10 Automated Documentation in Robotic Process Automation (RPA) . 36
2.10.1 Overview of RPA . . . . . . . . . . . . . . . . . . . . . . . . 36
2.10.2 Applications and Use Cases of RPA . . . . . . . . . . . . . . 37
2.10.3 UiPath: A Leading RPA Platform . . . . . . . . . . . . . . . 37
2.10.4 Advantages of RPA . . . . . . . . . . . . . . . . . . . . . . . 38
2.10.5 Limitations and Challenges of RPA . . . . . . . . . . . . . . 38
2.10.6 Potential of LLMs in Automated RPA Documentation . . . 39

**3 Case Study: An AI-Powered Agent for Automated UiPath Documentation** 40
3.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
3.2 Requirements and Design Goals . . . . . . . . . . . . . . . . . . . 41
3.3 Technology Stack . . . . . . . . . . . . . . . . . . . . . . . . . . . 42
3.3.1 Core Technologies . . . . . . . . . . . . . . . . . . . . . . . . 42
3.3.2 LLM Integration . . . . . . . . . . . . . . . . . . . . . . . . 42
3.3.3 Evaluation Libraries . . . . . . . . . . . . . . . . . . . . . . 43
3.3.4 User Interface . . . . . . . . . . . . . . . . . . . . . . . . . . 43
3.4 System Architecture . . . . . . . . . . . . . . . . . . . . . . . . . 44
3.5 Input Layer: Project Retrieval . . . . . . . . . . . . . . . . . . . . 45
3.5.1 Local Project Input . . . . . . . . . . . . . . . . . . . . . . . 45
3.5.2 GitHub Integration . . . . . . . . . . . . . . . . . . . . . . . 45
3.6 Analysis Layer: Parsing and Structure Extraction . . . . . . . . . . 45
3.6.1 Project Parser . . . . . . . . . . . . . . . . . . . . . . . . . . 45
3.6.2 XAML Parser . . . . . . . . . . . . . . . . . . . . . . . . . . 46
3.6.3 Workflow Analyzer . . . . . . . . . . . . . . . . . . . . . . . 49
3.7 Generation Layer: LLM-Powered Documentation . . . . . . . . . . 50
3.7.1 LLM Client Abstraction . . . . . . . . . . . . . . . . . . . . 50
3.7.2 Prompt Engineering Strategy . . . . . . . . . . . . . . . . . 51
3.7.3 Generation Parameters . . . . . . . . . . . . . . . . . . . . . 53

3.7.4 Output Format and Delivery . . . . . 54
3.8 Evaluation Layer: Quality Assessment . . . . . 55
3.8.1 Metric-Based Evaluation . . . . . 55
3.8.2 LLM-as-Judge Evaluation . . . . . 56
3.9 Operational Workflow . . . . . 58
3.9.1 Command-Line Interface . . . . . 58
3.9.2 Execution Flow . . . . . 59
3.9.3 Web Interface . . . . . 59
3.10 Limitations and Considerations . . . . . 60

**4 Results and Analysis** 61
4.1 Experimental Setup . . . . . 61
4.1.1 Selected Models . . . . . 61
4.1.2 Judge Model Selection . . . . . 63
4.1.3 Test Projects . . . . . 64
4.1.4 Evaluation Protocol . . . . . 65
4.2 Metric-Based Evaluation Results . . . . . 66
4.2.1 BLEU Scores . . . . . 66
4.2.2 ROUGE Scores . . . . . 67
4.2.3 Word-Based F1 Scores . . . . . 67
4.2.4 Metric-Based Summary . . . . . 68
4.3 Limitations of Reference-Based Metrics . . . . . 69
4.3.1 The Paraphrasing Problem . . . . . 69
4.3.2 Structural vs. Semantic Equivalence . . . . . 69
4.3.3 Inability to Detect Hallucinations . . . . . 69
4.3.4 Dependence on Reference Quality . . . . . 70
4.3.5 Illustrative Cases . . . . . 70
4.4 LLM-as-Judge Evaluation Results . . . . . 71
4.4.1 Overall Scores by Model . . . . . 71
4.4.2 Per-Dimension Analysis . . . . . 71
4.4.3 Dimension Comparison Across Models . . . . . 74
4.4.4 Qualitative Observations from the Judge . . . . . 74
4.5 Comparative Analysis . . . . . 76
4.5.1 Model Ranking Comparison . . . . . 76
4.5.2 Project-Level Analysis . . . . . 76
4.5.3 Cost-Quality Analysis . . . . . 77
4.5.4 Project Complexity Impact . . . . . 77
4.6 Discussion . . . . . 78
4.6.1 Key Findings . . . . . 78
4.6.2 Threats to Validity . . . . . 79
4.6.3 Directions for Future Work . . . . . 81

**5 Conclusions** 82

**Declaration on the Use of AI Tools** 84

**Bibliography** 85

# List of Tables

2.1 Comparative overview of notable decoder-only transformer models . 20

3.1 Technology stack of the UiPath Documentation Generator . . . . . 43
3.2 Key fields extracted from UiPath `project.json` and their documentation relevance . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
3.3 XAML root elements and corresponding UiPath workflow types . . 46
3.4 Most common UiPath activity types recognized by the XAML Parser 48
3.5 Structure of the Workflow Analyzer output dictionary . . . . . . . . 49
3.6 Model alias resolution: user-friendly names mapped to unique API identifiers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
3.7 Components of the documentation generation prompt . . . . . . . . 51
3.8 Documentation output sections with expected content and intended audience . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
3.9 LLM inference parameters used for documentation generation . . . 53
3.10 Comparison of evaluation metrics used in the metric-based evaluation component . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
3.11 LLM-as-Judge evaluation dimensions with assessment criteria . . . 56
3.12 Example LLM-as-Judge evaluation prompt for the Truthfulness dimension . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
3.13 Command-line arguments of the UiPath Documentation Generator 58

4.1 Comparison of the LLMs used for documentation generation and evaluation, including per-token API pricing as of February 2026. . . 62
4.2 Characteristics of the three open-source UiPath test projects selected for evaluation. . . . . . . . . . . . . . . . . . . . . . . . . . . 65
4.3 Corpus-level BLEU scores (%) for each model and project . . . . . 66
4.4 ROUGE F1 scores (%) for each model and project . . . . . . . . . . 67
4.5 Word-based F1 scores (%) for each model and project . . . . . . . . 67
4.6 Average metric-based scores (%) across all three projects . . . . . . 68
4.7 Average word counts and length ratios (generated/reference) across all projects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

4.8 Illustrative cases where metric-based scores diverge from actual documentation quality . . . 70
4.9 Overall LLM-as-Judge scores by model and project . . . 71
4.10 Completeness scores by model and project . . . 71
4.11 Helpfulness scores by model and project . . . 72
4.12 Truthfulness scores by model and project . . . 72
4.13 Clarity scores by model and project . . . 73
4.14 Professionalism scores by model and project . . . 73
4.15 Average per-dimension LLM-as-Judge scores across all three projects 74
4.16 Model rankings by average metric-based score vs. average LLM-as-Judge score . . . 76
4.17 Average LLM-as-Judge scores by project, across all three models . . 76
4.18 Cost-quality comparison across models (API rates as of Feb 2026) . 77
4.19 Relationship between project complexity (number of workflows) and average overall judge score . . . 77

# List of Figures


2.1 Subfields of AI. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
2.2 Transformer architecture. . . . . . . . . . . . . . . . . . . . . . . . . 18
2.3 Tree of transformer-based architecture models. . . . . . . . . . . . . 20
2.4 Character-level tokenization example. . . . . . . . . . . . . . . . . . 21
2.5 Word-level tokenization example. . . . . . . . . . . . . . . . . . . . . 21
2.6 Subword-level tokenization example. . . . . . . . . . . . . . . . . . . 22
2.7 Multi-head attention mechanism illustration compared to single-head attention. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
2.8 Agent Architecture: An overview of core components. . . . . . . . . 31

3.1 System architecture of the UiPath Documentation Generator. The pipeline processes a UiPath project through four layers. . . . . . . . 44

# Acronyms

**AI**
: Artificial Intelligence

**AST**
: Abstract Syntax Tree

**BERT**
: Bidirectional Encoder Representations from Transformers

**BLEU**
: Bilingual Evaluation Understudy

**BPE**
: Byte-Pair Encoding

**CNN**
: Convolutional Neural Network

**CoT**
: Chain-of-Thought

**DL**
: Deep Learning

**DOM**
: Document Object Model

**GAN**
: Generative Adversarial Network

**GPT**

Generative Pre-trained Transformer

**GRU**

Gated Recurrent Unit

**LLM**

Large Language Model

**LoRA**

Low-Rank Adaptation

**LSTM**

Long Short-Term Memory

**ML**

Machine Learning

**MLM**

Masked Language Modeling

**MLP**

Multilayer Perceptron

**NLP**

Natural Language Processing

**NMT**

Neural Machine Translation

**NN**

Neural Network

**PEFT**

Parameter-Efficient Fine-Tuning

**RAG**

Retrieval-Augmented Generation

**RL**

Reinforcement Learning

**RNN**

Recurrent Neural Network

**ROUGE**

Recall-Oriented Understudy for Gisting Evaluation

**RPA**

Robotic Process Automation

**XAML**

eXtensible Application Markup Language

# Chapter 1

# Introduction

## 1.1 Background

The contemporary technological panorama is defined by what seems to be the sudden rise of Artificial Intelligence and Large Language Models (LLMs). Although this revolutionary breakthrough can be seen as a recent phenomenon, it actually lays its roots in the last decades of research, with the culmination of philosophical ambition, state-driven investments, and the recent convergence of singular economic and technical forces. Thinkers like Ramon Llull, back in the 13th century, already imagined systems of combinatorial logic to generate truths mechanically. Moreover, the dream of Leibniz was to create a *"universal language"* able to reduce reasoning to mere calculation. George Boole and his formalization of logic in the 19th century gave Alan Turing the foundations for his theoretical universal computing machine in the 20th century, and with the advent of digital computers, this vision began to become more concrete than it had ever been. While we can say that thinkers answered the *"what"*, the *"how"* was answered by political and military ambitions during the 20th century.

These incredible advancements in AI capabilities and more specifically in LLMs are thus not a single breakthrough, but rather the product of the convergence of three historical developments:

- Exponential **growth in computational power** according to Moore's Law, which brought the power necessary to train enormous models;

- The creation of an **unprecedented quantity of digital data (Big Data)**, fundamental to train complex AI models;

- **Refined algorithms to train neural networks**, allowing the combination of data availability and computational power to reach new goals.

It is then no surprise that the rise of such a powerful, non-neutral force has brought profound changes in every field. Such a disruptive technology has managed to completely revolutionize the way people approach every kind of task in a matter of a few years, becoming a persistent and in most cases reliable companion in everyday life. More and more people nowadays would rather ask for advice or assistance from a chatbot like ChatGPT instead of searching for information on traditional search engines. Many of the platforms and forums that used to be the go-to place for any kind of question in the most disparate fields are now experiencing a significant decrease in traffic due to these new powerful and increasingly more precise AI tools. What we are seeing here is not just a trivial trend but rather a completely new paradigm shift in the way people access any kind of information. As with every new technology, it is fundamental to understand how to best leverage its potential while being conscious of its limitations and potential risks.

## 1.2 Motivation

Having worked as a software developer, it becomes clear that Software Engineering is an extremely complex discipline that requires a great amount of technical knowledge and integrates multiple principles of engineering for each production stage, from design to testing and maintenance. Specifically in the area of software documentation production, the potential in using cutting-edge technologies in order to further boost its efficiency and effectiveness is enormous and undeniable. These recent technological advancements have made possible the automation of certain repetitive and time-consuming tasks, increasing not only the quality of the final product but also making the productivity of developers skyrocket.

In this context, this thesis aims to explore and test first-hand the utilization of such technologies, evaluating their effectiveness in helping generate good quality software documentation, with a specific focus on technical documentation for Robotic Process Automation (RPA) projects built with UiPath, the leading enterprise RPA platform. The choice of UiPath as the target platform is motivated not only by its widespread adoption in enterprise environments, but specifically by its extensive adoption by Poseidon SB, a consulting company specializing in effective and rapid digitization of business processes through automation and other modern technologies that actively contributed to the development of this work. This collaboration supplied the domain expertise and development infrastructure necessary to design realistic experiments and case studies and offered the necessary technical supervision that grounded the research in practical constraints and measurable outcomes.

## 1.3 Problem Statement

In Software Engineering, well-written documentation is crucial for the development process of software, since it is easily consultable whenever trying to understand what the software does, its architecture, design and implementation. Despite the critical importance of such assets in the field, it is very often overlooked or undervalued. Because of its very tedious and labour-intensive nature, developers tend to postpone it or even avoid it altogether. This could lead to a variety of negative outcomes, such as low code maintainability, poor knowledge transfer among team members, low code understandability and slower and less organized development processes overall. These issues are particularly acute in the RPA domain, where automation projects are frequently developed under tight delivery schedules by small teams, and where the visual, low-code nature of platforms like UiPath can create a false sense that the workflows are self-documenting. In practice, as projects grow in complexity, the absence of proper technical documentation becomes a critical bottleneck for maintenance and knowledge transfer. More broadly, these issues could be even more critical in the case of large-scale systems that were built on top of outdated software or legacy codebases, since nowadays only a niche group of experts can provide the right expertise to understand and maintain such systems. It becomes not only a matter of loss of time and productivity, but also heavily impacts the economic side of a project, because fixing bugs or adding new features to poorly documented software could take significantly more time.

However, well-documented software is essential in order to ensure maintainability, scalability, and ease of collaboration among team members. Moreover, comprehensive documentation can significantly reduce the learning curve for new developers joining a project, for example when the subject matter expert of the codebase may be absent or unavailable, enabling them to quickly understand the code and contribute effectively. Last but not least, clear and concise documentation could also serve as legal protection for software companies, as it can help demonstrate compliance with industry standards and regulations.

Recent advancements in AI Large Language Models present a promising solution to this problem, making the generation of high-quality, reliable and complete software documentation more accessible and seamless than it has ever been, easing the burden on software developers and other stakeholders and allowing them to increase their productivity.

## 1.4 Research Objectives

Thanks to models specialized in code understanding, the use of Generative AI and LLMs is already widespread when it comes to code generation. Only recently has the focus shifted towards its usage in software documentation generation. However, due to the dimensional and logical complexity of the systems involved, the produced documentation quality is not always complete, accurate or reliable enough to be used in real-world settings.

The main objective of this thesis is to go in depth and further explore the potential and capabilities of the currently available AI-powered tools and LLMs in the field of software documentation generation, specifically technical documentation for UiPath RPA projects, and to propose an appropriate approach for such specific tasks. To this end, the thesis presents the design, implementation, and evaluation of a documentation generation pipeline that combines deterministic analysis of UiPath project files with LLM-powered natural language generation, and compares the output quality of three frontier models using both traditional NLP metrics and an LLM-as-Judge evaluation methodology.

## 1.5 Poseidon SB

Poseidon SB is a process and IT consulting company, specialized in delivering projects with highly qualified teams on advanced technologies. With a management team that has over 20 years of experience in the industry, despite being a young company, it has gained significant experience and expertise in its first three years of operations with over 30 medium and large-sized Italian and international companies, consolidating its position especially in the local market.

Being based in Sicily, it has offices in two different locations (Catania and Palermo) and involves a highly qualified team of over 40 professionals.

As a benefit corporation, it was founded in 2022 with the aim of creating quality jobs in the South, promoting talent development and generating high-innovation employment opportunities that allow brilliant and passionate young people from the territory in the IT field to work in their own region. The company is specialized in Digital Process Automation (DPA) and Robotic Process Automation (RPA), offering solutions that streamline and automate business processes across procurement, HRIS and other domains, enabling clients to increase their efficiency and effectiveness, reduce manual effort and scale operations while maintaining transparency and control.

With a look towards the future, Poseidon SB is also approaching new frontiers in innovative fields such as Artificial Intelligence and Smart Automation, investing in research and development in order to offer its clients cutting-edge solutions.

# Chapter 2

# State of the Art

## 2.1 Software Documentation

What is exactly Software Documentation? Software Documentation is a crucial part of the software development life cycle that encompasses all the written materials created to describe, explain and maintain software applications. Its main objective is to unify all the information related to a software project, in order to provide clear guidelines that facilitate discussion between the main parties involved, like stakeholders, developers, product managers, designers and end-users.

To give a more informal explanation of what software documentation is, we can think about it as the instructions that come with a new electronic device. It is easy to say that someone with some expertise in the field could understand how to use the product without the need of reading any guidelines, because of their knowledge and experience gained in the field over the years. But for someone who sees the product as some kind of black box, without any clue of how it works, having clear instructions is indispensable. The same concept applies to the software industry: while it may be easier for experienced developers and users to navigate and quickly grasp the functionalities of a software application, newcomers and less experienced individuals may not have it that easy. Therefore, having well-structured and comprehensive documentation is essential to ensure that everyone, regardless of their level of expertise, can effectively use and understand the software.

While it is important to have good documentation to rely on, creating it such that it is clear and easy to navigate may be challenging for bigger and more complex projects. One of the issues with the task may be the tendency to rely on manual efforts where the developers and other personnel are the main parties responsible for writing and maintaining such artifacts, stretching them too thin as the complexity of the project grows. Moreover, in the era of information overload, excessive documentation can lead to the risk of overwhelming users, making it a

challenge rather than a help.

It is important to keep in mind that the ultimate goal of software documentation is to answer two specific questions clearly and transparently: “What does the software do?” and “Who is this documentation for?”. By addressing these two simple but at the same time fundamental questions, it is possible to create different types of documentation that can satisfy the needs of the target audience it is intended for, taking into consideration their level of expertise and familiarity with that specific area and the tools at the user’s disposal.

### 2.1.1 Different kinds of Documentation

Although we can identify many different types of documentation, in this section we will briefly explore the most common ones, with a specific focus on technical code documentation, being the object of this thesis.

#### Requirements Documentation

Requirements documentation describes what the software must do and serves as the communication bridge between stakeholders and the development team. It usually contains functional requirements (features and behaviors) and non-functional requirements (performance, security, usability, etc.). Requirements are often incomplete or ambiguous, so clear, well-documented requirements are essential to avoid misunderstandings, delays, and extra costs.

#### Architecture design documentation

Architecture documentation gives a high-level view of the system’s structure, constraints and rationale. It typically includes diagrams and descriptions of components, interactions and key patterns, enabling architects, engineers and stakeholders to evaluate the design and guide lower-level implementation. Good architecture docs answer questions such as: *“What constraints apply?”*, *“Which non-functional requirements must be met?”*, and *“Which architectural patterns are recommended?”*.

#### User documentation

User documentation explains how to use the software and helps end-users accomplish tasks. It should be searchable, consistent and easy to navigate. Common formats include:

- **Tutorial** — step-by-step guides for new users.
- **Thematic** — topic-organized chapters for intermediate users.
- **Reference** — comprehensive listings of features for advanced users.

#### Marketing documentation

Marketing documentation highlights the product's features, benefits and value propositions aimed at potential customers. Created by marketing teams, it includes brochures, websites, presentations and case studies to align expectations, showcase effectiveness and encourage adoption.

#### Technical documentation

Technical documentation is a comprehensive set of documents that provide detailed information about a software system's design, architecture, data structures, algorithms and other technical details. It serves as a reference for developers, testers and sometimes end-users, offering insights into the technical aspects of the software. This type of documentation is important because it provides detailed information about how the software works and what it can actually do. It helps developers, engineers, maintainers and other technical stakeholders understand the technical details of the software and provides guidance on how to use it effectively. Technical documentation can be useful for end-users as well, especially when it comes to information about the features and capabilities of the software, allowing them to make informed decisions about how to use it and achieve their goals. It is important to keep it clear as well as thorough, but not so verbose that it could become overwhelming and difficult to maintain.

### 2.1.2 Technical documentation types

#### Low-level documentation

Low-level documentation typically refers to basic methods where the specific implementation cannot be easily grasped by examining the code itself, or for code based on complex algorithms or data structures. Some examples are:

- **Inline comments**: inline comments are brief notes embedded within the source code that explain what a specific line or block of code does. They are intended to provide clarity on what and how the code works, but also why certain decisions were made during the development process.

- **Code documentation**: this type of documentation provides a detailed description of the codebase, including its structure, components, functions and classes. It may include diagrams, flowcharts and other visual aids to help developers understand how the code works and how different parts interact with each other.

### High-level documentation

Unlike low-level documentation, which describes portions of the code, high-level documentation provides a much wider overview of the entire system's architecture as a whole. This kind of documentation encompasses various formats and types including flowcharts, Unified Modeling Language (UML) diagrams illustrating the architecture, design documents outlining business logic and how the code aligns with the requirements specifications.

### Internal documentation

This kind of documentation is intended for internal use within an enterprise. Some examples include coding conventions and standards and process documents for how teams build software. Guidelines for setting up development environments may also fall under internal documentation.

### External documentation

External documentation is intended for developers and generally users outside the organization. It usually includes API references that describe the public classes, functions, methods and modules, along with configuration examples, integration notes and **README files**. These are text files that provide an overview of a software project, typically including information about its purpose, installation instructions, usage guidelines, contributing guidelines, license, contact information and other relevant details. They are often the first point of contact for users visiting a certain repository and developers who want to understand what the project is about and how to get started with it.

## 2.2 Artificial Intelligence

Artificial Intelligence (AI) is a discipline of computer science whose objective is to create machines capable of simulating human behavior by performing tasks that normally require human intelligence, such as learning, comprehension, decision making, problem solving, autonomy and creativity. More and more companies, both public and private, are racing to participate in this new global and irreversible revolution, with the risk of being left behind otherwise, their goal being to make the objects around us increasingly more intelligent. Nowadays it seems that AI is integrated into every aspect of our normal lives, from virtual assistants on our smartphones to recommendation systems that lay their foundation on the analysis of our behavior when it comes to routine activities such as shopping online, cinematic tastes and music preferences, so that they can suggest new products and services to be consumed. While all these applications of this "newly born" technology may

seem idyllic and flawless, it is important to keep in mind that there are still many challenges and much more research to be done in order to try to tackle all its limitations and eventually make it an ideal companion for everyday tasks. As a first step, it is essential to understand what exactly AI is, its history and evolution over time and how, while it may seem like a sudden and disruptive innovation and change in paradigm, it is actually the result of decades of research and the final convergence of multiple important milestones achieved in different fields of study.

In 1950, Alan Turing published the seminal paper *"Computing Machinery and Intelligence"* [1], where he asks the famous question: *"Can machines think?"*. He proposed the famous Turing Test as a way to measure a machine's ability to demonstrate intelligent behavior that is indistinguishable from that of a human. Later in 1956, the term *"Artificial Intelligence"* was coined by John McCarthy at the first-ever AI conference at Dartmouth College [2]. In the following decades, exactly in 1958 [3], Frank Rosenblatt created the Perceptron, the first computer based on a neural network that could actually "learn" through trial and error. In the 1980s, neural networks using backpropagation algorithms to train themselves became widely used in AI applications. In 1995, Stuart Russell and Peter Norvig published *"Artificial Intelligence: A Modern Approach"* [4], a comprehensive textbook that became one of the leading textbooks in the study of AI. In it, they delve into four potential goals or definitions of AI, differentiating between **thinking humanly**, **thinking rationally**, **acting humanly** and **acting rationally**. In 1997, IBM's Deep Blue beat world chess champion Garry Kasparov in a six-game match [5], (and rematch), marking a significant milestone in AI's ability to exceed human performance. In 2007, John McCarthy wrote a paper named *"What is Artificial Intelligence"* and proposed an often-cited definition: *"AI is the science and engineering of making intelligent machines, especially intelligent computer programs"* [6]. By this time, the era of big data and cloud computing had begun, enabling organizations to manage and process vast amounts of data, which would one day be used to train AI models. In 2011, IBM's Watson [7] won the quiz show Jeopardy! against champions Ken Jennings and Brad Rutter. In the following years, exactly in 2016, Google DeepMind's AlphaGo [8], powered by a deep neural network and reinforcement learning, beat world champion Go player Lee Sedol in a five-game match. Its significance lies in the fact that the number of possible moves was huge as the game progressed (over 14.5 trillion after just four moves). In 2022 we saw a rise in large language models, also known as LLMs, such as OpenAI's GPT. With these new generative AI practices, deep-learning models can be pre-trained on enormous amounts of data.

If not already obvious, AI is not just one single piece of technology, but rather a whole journey of different techniques and approaches that have evolved over time and culminated in the current state of the art we see today. AI is not just one thing, but rather an umbrella term (Figure 2.1), a superset that encompasses a variety of

techniques and approaches, each of which has its own strengths and weaknesses, and which are often used in combination to create more powerful and effective AI systems. In the next sections, we will explore some of the main approaches.

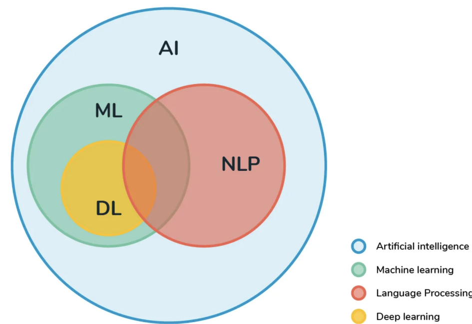


**Figure 2.1:** Subfields of AI.

## 2.3 Natural Language Processing

Natural Language Processing (NLP) is a subfield of computer science and AI that uses Machine Learning (ML) in order to enable computers to understand, interpret, and generate human language. NLP enables computers and digital devices in general to recognize, understand and generate text and speech by combining computational linguistics, the rule-based modeling of human language, together with statistical modeling, Machine Learning and Deep Learning (DL). Research in the field has enabled the beginning of the era of generative AI. NLP is already well integrated into our daily lives, more than we can imagine. The same technology powering search engines that are used seamlessly, like Google, prompting chatbots for customer service, filtering spam emails, and even our trusted virtual assistants like Alexa, Siri or Google Assistant. The benefits offered across many industries and applications are significant; to illustrate a few:

- **Automation of repetitive tasks -** NLP has proven to be extremely useful in fully or partially automating tasks like customer support, data entry and document handling. NLP-powered chatbots and virtual assistants can handle routine inquiries, freeing up human agents for more complex issues;
- **Data analysis -** NLP enhances data analysis by extracting insights from unstructured data, such as customer reviews, social media posts and news articles. Through text mining techniques, NLP can identify patterns, trends and sentiments that are not immediately obvious in large datasets, allowing businesses to better understand market trends, customer preferences and public opinion;

- **Enhanced search -** NLP improves search by enabling systems to understand the intent behind user requests, providing more precise and relevant results. Instead of relying only on keyword matching, NLP-powered search engines analyze the meaning of words and phrases, making it easier to find information even when queries are vague or complex;

- **Powerful content generation -** NLP powers advanced language models to generate text that is almost indistinguishable from human-written content, creating articles, reports, summaries and descriptions with only a few prompts. Such tools can assist humans in various tasks, from the most tedious ones like drafting emails to more complex ones such as writing code;

### 2.3.1 History of NLP

Natural Language Processing has a rich history that goes back to the 1950s. In 1950, Alan Turing published an extremely important article titled *Computing Machinery and Intelligence*, where he proposed what is now known as the Turing Test, as a criterion of intelligence for a machine. The proposed test includes a task involving automated interpretation and generation of natural language. We can recognize two main chapters in the history of NLP: the *symbolic* approach and the *statistical* approach.

The premise of symbolic NLP is summarized in an experiment conducted by John Searle's Chinese Room, where, given a collection of rules — in this case a Chinese phrasebook with questions and matching answers — the computer emulates NLP by applying the rules to the input data it confronts, without actually understanding the meaning. This approach dominated the field from the 1950s to the 1990s, with some famous examples. In the mid-1950s, the Georgetown experiment [9] demonstrated the feasibility of machine translation by automatically translating more than sixty Russian sentences into English. In the 1960s, Joseph Weizenbaum created ELIZA [10], an early natural language processing program that emulated a Rogerian psychotherapist. Using almost no information about human thought or emotion, ELIZA was sometimes able to provide human-like interactions. In the 1970s, programmers began to write *"conceptual ontologies"*, which structured real-world information so that it could be understood by machines. An example is MARGIE [11]. In the 1980s and early 1990s, the focus areas of the time included rule-based parsing, morphology, semantics, reference and other areas of natural language understanding. However, being rule-based systems that did not involve learning from data, their functions were highly limited and not scalable.

With the introduction of machine learning algorithms in the late 1980s and 1990s, NLP research shifted towards what are known as statistical methods. This new approach was also driven by the increase in computational power according to

Moore's Law [12]. The statistical approach heavily relies on making predictions based on probabilities learned from large amounts of data. In the 1990s, many early successes in statistical methods occurred in machine translation, due to the work of IBM researchers (Brown et al., 1990), who developed the first statistical translation systems by using existing bilingual text corpora produced by the Canadian Parliament [13]. In the 2000s, the growth of the web provided vast amounts of data, so research focused on unsupervised and semi-supervised learning algorithms, which could learn from data that was not hand-annotated. In 2003, the word n-gram model, at the time the best statistical algorithm, was outperformed by a multilayer Perceptron [14]. In 2010, Tomas Mikolov and his team applied a simple recurrent neural network with a single hidden layer to language modeling, and in the following years they developed the famous word2vec model [15]. The 2010s saw the rise of deep learning techniques and representation learning. That popularity was due partly to results showing that such techniques could achieve state-of-the-art results on many NLP tasks. To this day, NLP research continues to evolve rapidly, with advancements in deep learning, transformer architectures, and large language models driving significant improvements in natural language understanding and generation.

### 2.3.2 Approaches to NLP

Self-supervised learning (SSL) is specifically useful for supporting NLP because NLP requires vast amounts of labeled data to train AI models. Since these labeled datasets require time-consuming annotation, it could be prohibitively difficult to gather sufficient data using an approach that requires manually labeling data through a process involving manual labeling by humans. Self-supervised approaches can be more time-effective and cost-effective, as they replace some or all manually labeled training data. Three different approaches to NLP include:

- **Rule-based NLP -** Rule-based NLP systems rely on preprogrammed linguistic rules and patterns to process and understand natural language. The earliest applications were simple if-then decision trees, only able to provide answers in response to specific prompts. These systems use predefined sets of rules to analyze text, identify parts of speech, parse sentences, and extract meaning. Rule-based approaches can be effective for specific tasks with well-defined rules but often struggle with ambiguity and variability in natural language. They are not as scalable since there is no AI involved;

- **Statistical NLP -** Developed only later, statistical NLP uses probabilistic models and machine learning techniques to automatically extract, classify and label elements of text and voice data, assigning a statistical likelihood to each meaning of those elements. Statistical NLP introduced the essential technique

of mapping language elements, such as words and grammatical rules, into a vectorial representation, so that language could be modeled mathematically. This informed early NLP systems like spellcheckers and even T9 texting;

- **Deep learning NLP -** Only recently have deep learning models become the main protagonists in the panorama of NLP techniques, by making use of enormous volumes of raw and unstructured data. Deep learning can be seen as a further evolution of statistical NLP, with the key difference being that it relies on a neural network approach, using semantic networks and word embeddings to capture semantic relationships between words. Neural machine translation, based on then-newly invented sequence-to-sequence models with attention mechanisms, made obsolete intermediate steps like word alignment and phrase extraction, previously necessary for statistical machine translation.

## 2.4 Generative AI

Before diving into the specifics of Large Language Models (LLMs) and their applications in software engineering, it is crucial to understand the much broader context of Generative AI. Generative AI is a subfield of Artificial Intelligence that focuses on using generative models to produce new content, such as text, images, videos, audio, code and other forms of data. The process of generating new data is based on a preliminary phase of training on large datasets, where the model learns and understands new patterns to replicate, thanks to inputs that often come in the form of natural language prompts.

Such tools have exploded in popularity in recent years, made possible by advancements in transformer-based deep neural networks, particularly the advent of Large Language Models (LLMs). Some of the most used tools include ChatGPT, DALL · E, Stable Diffusion, Codex and GitHub Copilot. Although more and more useful tools are at the disposal of every kind of user, few are the companies that are able to build and train such models from scratch, due to the high computational costs and resource requirements. Generative AI includes a wide range of applications across various fields, from software development to healthcare, from entertainment to education.

However, such sudden and widespread adoption has raised many ethical questions and governance challenges. Not only that, but even if used ethically, generative AI models can inadvertently lead to unfortunate consequences like mass replacement of jobs and intellectual property violations. Moreover, due to the enormous computational power required, they have a significant environmental impact as well.

### 2.4.1 History of Generative AI

The origins of algorithmically generated content can be traced back to the development of the Markov Chain by Russian mathematician Andrey Markov in 1906 [16], which later enabled probabilistic text generation. Once trained on a text corpus, such chains could generate probabilistic text. By the early 1970s, researchers were experimenting with computers to extend generative techniques beyond text. Harold Cohen's AARON program [17] created original artworks using rule-based systems and heuristics during experiments. The terms "generative AI" and "planning" were used between the 1980s and 1990s in the context of automated planning. In the 1990s, such methods were still a relatively mature technology. They were mainly used to generate crisis action plans for military use, process plans for manufacturing and decision plans such as in prototype autonomous spacecraft during research and development.

Beginning in the late 2000s, the introduction of deep learning technology led to many improvements in image classification, speech recognition, natural language processing and other tasks across domains. Neural networks in this age were typically trained as discriminative models due to the difficulty of generative modeling. In 2014, Ian Goodfellow and his colleagues introduced Generative Adversarial Networks (GANs), which led to the production of the first practical deep neural networks capable of learning generative models [18], as opposed to discriminative ones, for complex data like images. They were the first ones to not only output classes but whole images, and thus enabled new creative applications in art, design and media production workflows across industries. In 2017, the Transformer architecture was introduced by Vaswani et al., enabling advancements in generative models compared to older LSTM approaches, leading to the first Pre-trained Transformer (GPT) [19], also known as GPT-1, developed by OpenAI and released in 2018.

In 2021, OpenAI released DALL · E [20], a transformer-based model capable of generating images from textual descriptions, marking an important milestone in AI-generated imagery. The following year, OpenAI launched what is today known as the most popular chatbot in the world, ChatGPT [21], based on the GPT-3.5 architecture, which could generate accurate human-like text, code and even images, making it an incredible success with rapid user adoption worldwide. It was shown that within five days of release, ChatGPT reached one million users, making it the fastest-growing consumer application ever and highlighting mass public interest. In December 2023, Google unveiled Gemini [22], joining the AI race and rapidly expanding competition. Anthropic also released Claude [23], an extremely valid competitor in performance to ChatGPT.

As of 2025, despite continued consumer growth, more enterprises began adopting generative AI tools to enhance productivity, automate tasks and drive innovation

across various industries, although leading analysts seemed to be skeptical about the actual economic impact in the short term, describing the period as entering the Gartner "Hype Cycle" towards the "Trough of Disillusionment" [24]. It appears difficult to predict what may happen in the following years, but one thing is clear: there is no going back from Generative AI now or in the future, and even if the current hype may die down and eventually plateau, such technology will continue to evolve and shape the future of technology and human interaction with it.

Although all Generative AI models share the common goal of generating new content, that does not mean they can all be classified as Large Language Models (LLMs), since LLMs are a specific and extremely important subset of the broader Generative AI family. In the next sections, we will explore the main concepts and techniques behind LLMs.

## 2.5 Large Language Models

One of the major advancements in the field of Artificial Intelligence has been the development and rise of Large Language Models (LLMs). These models represent the convergence of NLP and Deep Learning advancements and have revolutionized the way machines understand and generate human language, enabling technologies that surround us daily. In order to understand the true potential of such technologies we have to ask ourselves: what exactly are Large Language Models? LLMs are deep learning models trained with self-supervised machine learning on massive datasets of text, designed for natural language processing tasks. The largest and most capable LLMs are Generative Pre-trained Transformers (GPTs), which are based on the Transformer architecture and are now the backbone of many state-of-the-art NLP applications such as ChatGPT, Gemini and Claude. Such models can be fine-tuned for specific tasks like translation, summarization, question answering and text completion, or guided by prompt engineering. It is important to keep in mind that, although they are very powerful models able to acquire predictive power regarding syntax, semantics and even ontologies inherent in human language corpora, they also inherit inaccuracies, biases and limitations present in their training data.

Their peculiarity lies in the fact that they consist of billions to even trillions of parameters, operating as general-purpose models across multiple domains. LLMs represent a significant new technology paradigm in their ability to generalize across tasks without explicit or minimal task-specific supervision, enabling capabilities like conversational agents, code generation, knowledge retrieval, and automated reasoning that previously required specialized models. Their versatility is the winning factor that has led to their rapid adoption across industries.

### 2.5.1 History of LLMs

Before the advent of transformer-based models in 2017, some language models of the time were considered large relative to the hardware capabilities and data constraints of the time. In the early 1990s, IBM pioneered word alignment techniques for machine translation [25], laying the foundations for corpus-based language modeling. In 2001, a smoothed n-gram model like those employing Kneser-Ney smoothing was trained on 300 million words, achieving state-of-the-art perplexity on benchmark tests [26]. During the 2000s, the concept of the web as corpus emerged, since the production of massive amounts of datasets from web crawling enabled researchers to train statistical language models.

Moving beyond n-gram models, researchers began exploring the use of neural networks to learn language models. Deep neural networks marked an important breakthrough in language modeling. The shift was initiated by the development of word embeddings like Word2Vec [15] and GloVe [27], which represented words as dense vectors in a continuous space, capturing semantic relationships between words. Sequence-to-sequence models with attention mechanisms [28] also enabled more effective modeling of long-range dependencies in text, using LSTMs. In 2016, Google's Neural Machine Translation (GNMT) [29] system demonstrated its effectiveness in translation services, moving away from statistical phrase-based models. These early NMT systems used LSTM-based encoder-decoder architectures, as they were the state-of-the-art at the time, before the invention of transformers.

2017 marked a true turning point with the introduction of the Transformer architecture by Vaswani et al. at the NeurIPS conference by Google Research in their landmark paper *Attention is All You Need* [30]. The goal of the paper was to improve upon the limitations of recurrent and convolutional models by introducing a novel architecture based mainly on the attention mechanism, developed in 2014 by Bahdanau et al. The following year, in 2018, BERT (Bidirectional Encoder Representations from Transformers) was introduced by Devlin et al. at Google AI Language. Although the transformer has both encoder and decoder components, BERT only used the encoder part. Academics and industry quickly adopted BERT, although its decline in popularity began in 2023, following the rapid improvements of decoder-only models like GPT that could solve tasks through prompting.

Even though the decoder-only GPT-1 was introduced in 2018 by OpenAI, the real breakthrough came with GPT-2 in 2019 [31], which was actually deemed so powerful and potentially dangerous that it was not fully released to the public at first. But it was later in 2022, with the release of the popular chatbot ChatGPT based on the GPT-3.5 architecture, that LLMs gained massive public coverage. In 2023, GPT-4 [32] was released, marking another leap in capabilities and praised for its accuracy and multimodal capabilities. Other companies followed the trend, with Anthropic releasing Claude, Meta presenting LLaMA [33] and Google unveiling

Gemini later in 2023. Since then, most LLMs are based on transformer architectures and have been trained to be multimodal, handling different kinds of input data like text, images, audio and even 3D meshes.

### 2.5.2 Architecture

#### Early LLM Architectures

Before modern deep learning, **language models** were mainly statistical. Classic N-gram models predict the next word from the previous N-1 words using corpus counts and the Markov assumption. They worked well for small N but suffered severe data sparsity and could not capture long-range dependencies beyond modest context windows.

To address this, researchers turned to **recurrent neural networks (RNNs)**, which process sequences by maintaining a hidden state across timesteps. Early Elman networks [34] and later RNN language models [35] substantially reduced perplexity versus N-grams by remembering much more context. However, RNNs are harder to train due to vanishing/exploding gradients during backpropagation through time [36], limiting practical long-range learning.

**Long Short-Term Memory (LSTM)** networks [37] solved many training issues by introducing gated cells: input, forget and output gates regulate information flow, enabling persistent cell states and much longer-term memory. LSTMs dominated sequence tasks for years. The **Gated Recurrent Unit (GRU)** [38] is a simpler alternative that merges gates and states (update/reset gates), offering similar performance with fewer parameters and faster training. Both are examples of **gated RNNs**, yet they remain inherently sequential, limiting parallelism and making very long-range dependency modeling costly.

A key advance was the **attention mechanism** [28] for sequence-to-sequence models: instead of compressing sources into a single vector, attention lets the decoder focus on relevant encoder positions. This idea led to **self-attention**, removing the need for recurrence by directly attending to all input positions — paving the way to transformer architectures that efficiently model long-range dependencies in parallel.

#### The Transformer Architecture

In the original paper "*Attention is All You Need*", the Transformer was introduced as an encoder–decoder architecture for machine translation: the encoder produces contextualized representations of the input, while the decoder generates the output autoregressively by attending to encoder outputs and previous decoder states (see Figure 2.2). Unlike RNNs, transformers process all positions in parallel, greatly improving training efficiency on modern accelerators.

The core mechanism is **self-attention**, which lets each token compute a representation by attending to all other tokens in the sequence. Each transformer layer then applies a position-wise feed-forward network (two linear layers with a ReLU between) to every position independently, adding nonlinearity after attention. Layers use residual connections plus layer normalization [39] so that a sublayer implementing $F(x)$ yields $\text{LayerNorm}(x + F(x))$, which stabilizes training and preserves gradients (Ba et al., 2016).

Because transformers lack recurrence or convolution, they require **positional encodings** to inject token-order information; the original model used fixed sinusoidal encodings added to token embeddings. A key advantage of self-attention is its short path length between tokens — every position can directly attend to any other in a constant number of sequential operations — making long-range dependency learning easier. The trade-off is computational cost: vanilla self-attention has quadratic time/space complexity $O(n^2)$ per layer in sequence length, which motivates many later efficient-attention variants.

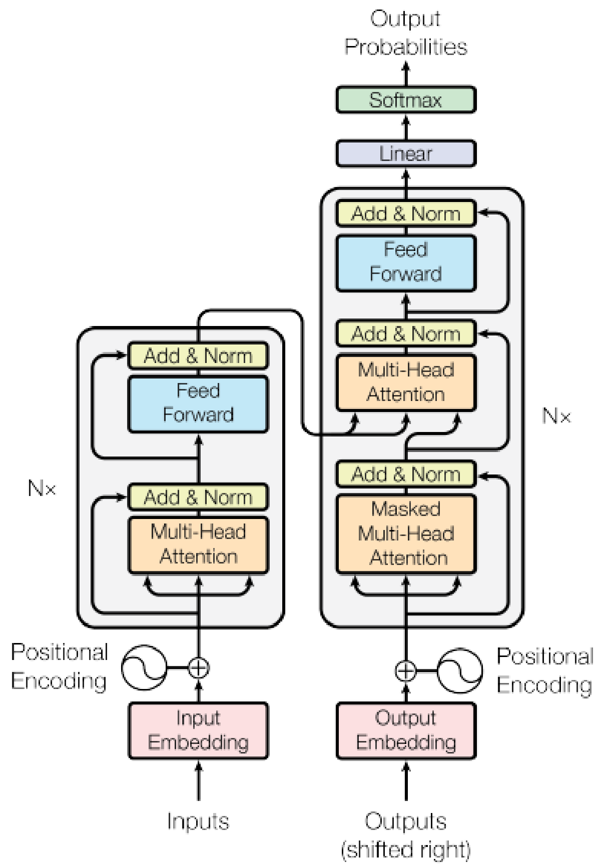


**Figure 2.2:** Transformer architecture.

### Transformer-Based Model Architectures

The modularity of the transformer architecture allows it to be configured for different kinds of LLMs. We can categorize transformer-based LLMs into three principal categories, many of which are shown in Figure 2.3.

### Encoder-Only Transformers

These are transformer stacks that function like deep bidirectional feature extractors. The prime example is BERT [40], where the bidirectional conditioning is enabled by BERT's training objective of Masked Language Modeling (MLM) — randomly masking some parts of the input and training the model to predict them using context from all positions. Another task is Next Sentence Prediction (NSP), where the model learns to predict whether one sentence follows another. BERT's architecture is a deep stack of transformer encoder layers with learned positional embeddings and the usual multi-head self-attention. Being encoder-only, BERT excels at understanding tasks such as classification, entailment, question answering, etc.

### Decoder-Only Transformers

These models consist of a transformer decoder stack, without an encoder, and are designed for autoregressive language modeling, where the input sequence is seen as the beginning of the output sequence and is processed and extended. The GPT (Generative Pre-Training) series is the archetype of decoder-only LLMs. A decoder-only transformer makes use of **self-attention** layers with a causal mask, limiting attention only to previous tokens. It may also include a final feed-forward and output softmax layer to predict the next-token probabilities. Nowadays the main players that have developed most of the prominent decoder-only LLMs are OpenAI, Anthropic, Google DeepMind, and Meta, as shown in Table 2.1.

### Encoder–Decoder (Sequence-to-Sequence) Transformers

This architecture uses both an encoder and a decoder as in the original "Attention is All You Need" paper. A prominent example is T5 [41], which frames every NLP task as text-to-text: inputs and outputs are strings, so tasks like translation, summarization, classification or QA share the same interface (e.g., "translate English to Sentiment: <sentence>" → "positive"/"negative"). The encoder encodes the input while the decoder generates the target text, making the model flexible across tasks via prompting. T5 achieved strong results by training on the large C4 corpus and scaling models (up to ∼11B parameters), demonstrating the versatility of the encoder–decoder setup.

| Model | Year | Parameters | Key features and innovations |
|---|---|---|---|
| GPT-1 (OpenAI) | 2018 | 117M | First large-scale application of the Transformer decoder for language modeling; demonstrated transfer learning for NLP. |
| GPT-2 (OpenAI) | 2019 | 1.5B | Scaled-up decoder-only model trained on WebText; coherent long-form generation and few-shot capabilities. |
| GPT-3 (OpenAI) | 2020 | 175B | Massive scaling; emergent zero-/few-shot behaviour via prompting; trained on large mixed corpora. |
| GPT-4 (OpenAI) | 2023 | 1T (est.) | Multimodal (text + images); improved reasoning and factuality; large context windows (8k–32k tokens); RLHF. |
| GPT-5-2 (OpenAI) | 2025 | undisclosed | Series of iterative generations with expanded reasoning, tool use and very large context windows (reports vary). |
| LLaMA 2 (Meta) | 2023 | 7B–70B | Released for research/commercial use; improved fine-tuning and alignment; competitive on many benchmarks. |
| LLaMA 3.x (Meta) | 2024–2025 | up to 405B | Extended context support (e.g. 128k), multilingual improvements and efficiency optimizations in later variants. |
| Claude 3–4 (Anthropic) | 2024–2025 | undisclosed | Multiple variants focused on safe alignment, multimodal reasoning, coding and long-context document analysis. |
| Gemini 1–2 (Google) | 2023–2025 | undisclosed | Successors to PaLM with multimodal capabilities, tool integration and features targeting agentic workflows. |

**Table 2.1:** Comparative overview of notable decoder-only transformer models

**Figure 2.3:** Tree of transformer-based architecture models.

### 2.5.3 Data preprocessing

Before a Transformer model can actually process textual data, the raw text must be transformed into a numerical form through various preprocessing steps, like converting text into a sequence of tokens, constructing a vocabulary of tokens, adding any necessary special tokens for specific tasks, and including positional encodings to make use of word order information.

#### Tokenization

Tokenization is the process of splitting text into simple units called tokens, which are the atomic elements of the model's input sequence. Transformers operate on token sequences and each token typically corresponds to a meaningful chunk of text such as a character, a word, or a subword fragment. The choice of tokenization granularity has an impact on model performance and vocabulary size:

**Character-level tokenization:** Each character is a token. This avoids out-of-vocabulary (OOV) issues and is robust to typos, but produces long sequences, is less semantically informative, and is inefficient for common words.

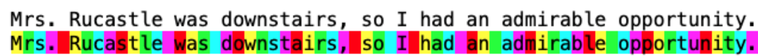


**Figure 2.4:** Character-level tokenization example.

**Word-level tokenization:** Each word is a token, yielding intuitive, shorter sequences and straightforward embeddings. Drawbacks include large vocabularies, OOV tokens for unseen words, and poor handling of rare or morphologically related words.

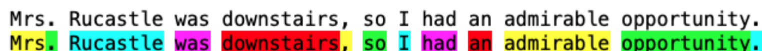


**Figure 2.5:** Word-level tokenization example.

**Subword-level tokenization:** Text is split into subword units (e.g., prefixes/suffixes). This gives a compact vocabulary while allowing composition of unseen words. A common method is Byte-Pair Encoding (BPE) [42], which iteratively merges frequent symbol pairs to build useful subword tokens.

```
Mrs. Rucastle was downstairs, so I had an admirable opportunity.
Mrs. Rucastle was downstairs, so I had an admirable opportunity.
```

**Figure 2.6:** Subword-level tokenization example.

In contemporary Transformer models, *subword tokenization* has become the de facto standard for its ability to handle open vocabularies and reduce data sparsity.

### Vocabulary Construction

A fixed vocabulary of model and special tokens is constructed for the chosen tokenization scheme: word-level vocabularies typically retain the top-N words (others → [UNK]), while subword methods (BPE/WordPiece) merge units to a target size so that rare words are decomposed into known subwords. The vocabulary size is selected to balance excessive splitting against unnecessary model growth.

### Positional Encodings

Transformers have no inherent notion of token order, so a positional encoding is added to token embeddings. The original Transformer used fixed sinusoidal encodings:

$$\begin{aligned} PE(pos,2i) &= \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right), \\ PE(pos,2i+1) &= \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right), \end{aligned} \tag{2.1}$$

for indices $i$ covering the model dimensions. Later models often use learned position embeddings instead of these fixed sinusoids.

## 2.5.4 Attention Mechanisms

The Transformer architecture relies exclusively on attention to model dependencies between tokens. In fact, Vaswani et al. [30] describe the Transformer as a model *"relying entirely on an attention mechanism to draw global dependencies between input and output."* At a high level, an attention layer takes three inputs: a set of *queries* $Q$, *keys* $K$, and *values* $V$. These are typically obtained by applying learned linear projections to the input token embeddings. Given $Q \in \mathbb{R}^{N \times d_k}$, $K \in \mathbb{R}^{M \times d_k}$, and $V \in \mathbb{R}^{M \times d_v}$, the **scaled dot-product attention** is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V, \tag{2.2}$$

where the softmax is applied row-wise. The softmax normalization ensures that each row of $QK^\top$ sums to 1, so the $i$-th output is a weighted sum of the rows of $V$

with weights given by the $i$-th row of the softmaxed score matrix. This formulation is more efficient than earlier additive attention and compatible with optimized matrix operations.

### Multi-Head Attention

To enrich the model's ability to focus on different aspects of the input, the Transformer uses **multi-head attention**. Instead of performing a single attention function, the inputs $Q$, $K$, $V$ are linearly projected $h$ times into different subspaces. For each head $i \in \{1, \ldots, h\}$, one computes the linear projections

$$QW_i^Q, \quad KW_i^K, \quad VW_i^V,$$

where $W_i^Q, W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$ and $W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$. Each head applies scaled dot-product attention:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V).$$

The outputs of all heads are concatenated and projected using $W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O. \tag{2.3}$$

This allows the model to jointly attend to information from different representation subspaces.

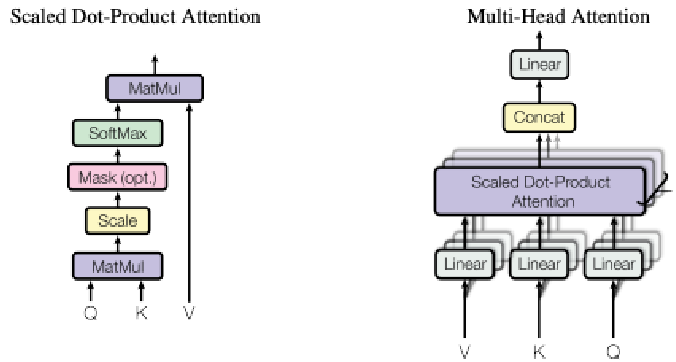


**Figure 2.7:** Multi-head attention mechanism illustration compared to single-head attention.

### Self-Attention

**Self-attention** (or intra-attention) refers to the case where $Q = K = V$. Each token attends to all tokens in the same sequence, including itself. The encoder stack of the Transformer consists of multiple layers of multi-head self-attention. This design enables each token's representation to be updated based on a global context without requiring recurrence.

#### Encoder–Decoder (Cross) Attention

In sequence-to-sequence settings, the decoder includes **encoder–decoder** (or **cross**) attention. Here, the queries come from the decoder layer, while the keys and values come from the encoder's output. This allows the decoder to attend over the entire input sequence at each step. Unlike self-attention, cross-attention typically does not use masking.

#### Masked (Causal) Attention

For autoregressive generation tasks, the decoder must use **masked** (or **causal**) self-attention. This prevents a token from attending to future tokens by applying a mask matrix $M$ before the softmax:

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(M + \frac{QK^\top}{\sqrt{d_k}}\right)V, \tag{2.4}$$

where $M_{ij} = -\infty$ for $j > i$ and 0 otherwise. This ensures that each token can only attend to itself and preceding tokens.

## 2.6 Extensibility Techniques in Large Language Models

Extending the already vast capabilities of LLMs to additional tasks or domains involves various adaptation methods. In broad terms, extensibility can be achieved either by modifying - or more precisely, **fine-tuning** - model parameters, or by altering how the model is prompted and augmented.

### 2.6.1 Model Fine-Tuning

Fine-tuning adapts a pre-trained LLM to a target task by continuing training on task-specific data and updating weights. Full fine-tuning yields strong performance but is costly for very large models. Parameter-efficient fine-tuning (PEFT) reduces cost by freezing most weights and learning a small set of additional parameters. PEFT methods allow efficient specialization with far fewer parameters and reduced memory and compute requirements. Examples:

- *Sentiment analysis*: a LoRA adapter [43] (e.g. $r = 4$ per layer) on GPT-3 adds only millions of trainable parameters vs. full re-training of 175B, and can match full fine-tuning performance.

- *Domain adaptation*: a BERT model fine-tuned for legal text using adapters [44] or prefix tuning [45] adapts quickly to domain language while keeping most weights frozen.

### 2.6.2 Prompting Techniques

Prompting has proved to be an extremely powerful paradigm for leveraging Large Language Models. By carefully and specifically engineering text inputs, users can induce an LLM to perform a desired task or exhibit certain behaviors. In this section, we review state-of-the-art prompting methods.

#### Zero-Shot Prompting

Zero-shot prompting gives the model a concise task description without examples; the LLM must rely on pretraining to perform the task. Implementation is simple (clear instruction + prompt), but results are sensitive to wording and are typically less accurate than example-based or fine-tuned methods.

#### Few-Shot Prompting

Few-shot prompting includes a small set of input–output demonstrations in the prompt so the model can infer the task pattern at inference time. It often boosts accuracy versus zero-shot [46] but is sensitive to example choice and order, and is inefficient because demonstrations must be sent on every call.

#### Chain-of-Thought Prompting

Chain-of-Thought (CoT) [47] [48] prompts ask the model to produce intermediate reasoning steps before the final answer, improving multi-step reasoning for sufficiently large models. CoT can be used in few-shot or zero-shot forms (e.g., "Let's think step by step.") [49]; it increases token usage and latency and may propagate incorrect intermediate steps, so outputs should be validated.

### 2.6.3 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation augments a generative LLM with an external retriever so that responses are conditioned on retrieved documents [50]. Typical flow: form query → retrieve top-k passages → provide passages + query to the generator. RAG improves factuality and updatability but depends on retriever quality and increases prompt length and latency.

## 2.7 Evaluation Metrics and Limitations of LLMs

### 2.7.1 Evaluation Metrics

Large Language Models are evaluated using a range of metrics that depend on the task. These automatic metrics complement human evaluations of quality, coherence or factuality, and specialized measures such as hallucination rate. Below we define some popular metrics.

#### Perplexity

Perplexity (PPL) measures how well a probabilistic model predicts a token sequence. For $X = (x_1, \dots, x_T)$ under model $p_\theta$:

$$\mathrm{PPL}(X) = \exp\left(-\frac{1}{T}\sum_{i=1}^{T} \log p_\theta(x_i \mid x_{<i})\right). \tag{2.5}$$

Lower PPL indicates better predictive fit (perfect prediction gives $\mathrm{PPL} = 1$). It is commonly reported during training and validation for autoregressive models.

#### Accuracy (Precision, Recall, F1)

Accuracy is the fraction of correct labels:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}. \tag{2.6}$$

For imbalanced or structured tasks, precision $= TP/(TP+FP)$, recall $= TP/(TP+FN)$, and the F1 score (harmonic mean of precision and recall) are typically reported instead.

#### BLEU

BLEU [51] evaluates overlap between a candidate and one or more references using $n$-gram precision with a brevity penalty:

$$\mathrm{BLEU}_N = BP \cdot \exp\left(\frac{1}{N}\sum_{n=1}^{N} \log p_n\right), \tag{2.7}$$

where $p_n$ are modified $n$-gram precisions and $BP$ penalizes overly short candidates. BLEU is widely used for translation but does not capture paraphrastic or semantic equivalence well.

### ROUGE

ROUGE-N [52] is a recall-oriented $n$-gram overlap metric between reference $R$ and candidate $G$:

$$\text{ROUGE-N} = \frac{\sum_{g \in \mathcal{G}_n} \min(\text{Count}_G(g), \text{Count}_R(g))}{\sum_{g \in \mathcal{G}_n} \text{Count}_R(g)}, \tag{2.8}$$

i.e., the fraction of reference $n$-grams covered by the candidate. Like BLEU, it is an overlap metric and ignores deeper semantics.

### Human Evaluation

Automatic metrics miss aspects like coherence, factuality and style, so human ratings remain the gold standard. Common protocols include scalar ratings (e.g., 1–5) on fluency/relevance, pairwise preference tests, or targeted error annotations on sampled outputs.

### Hallucination Rate

A hallucination is output containing information not supported by the input or references. The hallucination rate is the proportion of generated items that contain such unsupported or false statements [53]. Measuring it typically requires human or automated fact-checking and is crucial for knowledge-grounded tasks.

## 2.7.2 Limitations of Current LLM Technologies

Despite their enormous capabilities, contemporary LLMs exhibit well-documented weaknesses. Since LLMs learn statistical patterns rather than grounded knowledge, they can fail to maintain factual accuracy when asked follow-up questions or when summarizing source documents. Together, these phenomena undermine trust in applications that require reliable output.

### Interpretability and Robustness

LLMs remain largely “black boxes”: common proxies (e.g., attention) do not provide reliable explanations of model decisions [54], hindering trust in high-stakes settings. They are also brittle — small paraphrases, typos or adversarial perturbations can cause large swings in output quality or correctness — so deployment requires careful validation and robust input handling.

### Compute, Cost and Privacy

Training and serving modern LLMs demand substantial compute and energy, concentrating capability in well-resourced organizations and raising environmental and cost barriers for wider research or deployment. Large models also risk memorizing and exposing sensitive training data, so rigorous data governance, auditing and privacy-preserving measures are required when models are trained on proprietary or personal information [55].

### Alignment, Bias and Ethical Concerns

Aligning LLM behavior to human values is an open problem: RLHF and filters reduce but do not eliminate undesirable behaviors, and adversarial prompts can still elicit harmful outputs. Models inherit biases from training corpora, which can produce unfair or offensive results; therefore, thorough bias evaluation, transparency about training data, and application-specific safeguards are necessary to mitigate ethical risks.

## 2.8 Agentic AI

Agentic AI denotes a class of systems that extend large language models (LLMs) with autonomy: persistent memory, explicit planning, tool use, and decision-making policies. Compared to static pipelines, agentic systems dynamically choose actions, break down goals, invoke external tools, and revise strategies based on observations. Agentic systems share a small set of composable components, as shown in Figure 2.8.

### Agent Module

The central controller interprets the current goal and context, produces a plan (a sequence of subgoals or tool calls) via an LLM or planner, and decides the next action and whether to continue, retry, or halt. Design choices include making the module purely prompt-driven, PEFT-tuned, or hybrid (a neural planner combined with symbolic procedures).

### Tools and Tooling Layer

Tools provide deterministic or environment-side capabilities; typical examples (following common agent toolkits) include `Search_Engine()` for document or web retrieval, `Knowledge_Base()` for querying indexed corpora or vector stores, `CodeInterpreter()` for running and inspecting code or data, and `Prompt_Generator()` for synthesizing task-specific prompts; implementations may

also expose calculators, API-call wrappers, or many more specialized tools. Tools are wrapped to return structured observations that the agent consumes.

#### Memory (Short- and Long-Term)

Memory supports statefulness:

- **Short-term memory:** a transient buffer of recent interactions used for context and coherence.
- **Long-term memory:** an indexed store of persistent facts, preferences, or outcomes (text, embeddings, multimodal artifacts).

Memory access is typically retrieval-driven: the agent issues a query (semantic or keyword) to fetch relevant entries and may update memory by appending, merging, or overwriting items after reflection.

#### Planner and Reasoner

Planning styles vary by complexity:

- **One-step planning:** generate an entire decomposition at once (useful for short deterministic tasks).
- **Multi-step / iterative planning:** generate, execute, observe, and refine plans in a closed loop (better for uncertain or long-horizon tasks).
- **Reflection / Self-Criticism:** periodically summarize outcomes and revise stored strategies or memory entries.

### 2.8.1 Multi-Agent Patterns and Applications

Agentic systems can be composed into multi-agent architectures with different interaction modes:

- **Cooperative:** specialized agents (researcher, verifier, executor) collaborate and share state to solve complex problems.
- **Adversarial / Competitive:** agents simulate opponents for robust planning or evaluation (useful in benchmarking).
- **Hybrid:** hierarchical or parallel mixtures where a commander agent delegates to workers and aggregates results.

### 2.8.2 Popular LLM-Based Architectures and Frameworks

Over the past two years, numerous architectures have been proposed to realize agentic behavior with LLMs. A selection of influential examples includes:

- **AutoGPT (autonomous task agents)** [56]. An open-source pattern for recursive task decomposition, tool usage and self-evaluation where an LLM loops with minimal human input to accomplish high-level goals; it shows the feasibility of sustained autonomy but also highlights stability, loop and guardrail challenges.
- **LangGraph / graph-based workflows** [57]. Represents agent logic as directed graphs of nodes (LLM calls, tools, decision points), supporting branching, parallelism and more robust control flows than a linear ReAct loop — useful for long-running or stateful agents.
- **Multi-agent collaboration frameworks** [58], [59]. Frameworks like MetaGPT and CrewAI provide orchestration primitives for teams of specialized agents (roles/personas) that communicate and coordinate to solve complex tasks, improving coherence on multi-step projects.
- **Agent orchestration libraries** [60], [61]. Tools such as Microsoft AutoGen and LangChain supply messaging, async execution, tool integration and common agent patterns (commander/worker, requester/responder), reducing engineering overhead for multi-agent or tool-augmented systems.

### 2.8.3 Evaluation of LLM-Based Agents

Evaluating autonomous LLM-driven agents calls for a broad, multi-dimensional perspective that goes beyond single-number summaries. Central to any assessment is the agent's ability to complete intended tasks: a **Task Success Rate** (measured as a binary fraction or as a graded score to give partial credit on complex objectives) captures whether the agent actually achieves its goals. At the same time, an agent's degree of independence matters — an **Autonomy Ratio** (the percentage of tasks completed without human intervention) quantifies human-in-the-loop dependence and helps balance autonomy against reliability.

Complementing these outcome measures are efficiency and reasoning-oriented evaluations. **Efficiency** metrics — step count, time-to-completion, API calls, and compute/memory usage — reveal resource and latency costs and enable fair comparisons across architectures and deployment settings. **Rationality** or plan quality examines the coherence and optimality of an agent's strategy (for example, by comparing plans to expert trajectories or via human/model ratings of intermediate reasoning), penalizing unnecessary detours or illogical steps. Finally,

score-based analyses (task returns, learning curves, and trade-off plots) and human assessments of **human-likeness and safety** (perceived naturalness, acceptability, and counts of safety violations) are indispensable for understanding robustness, user trust, and risk. In practice, rigorous evaluation combines automated metrics with targeted human judgments to produce a balanced picture of both performance and practical suitability.

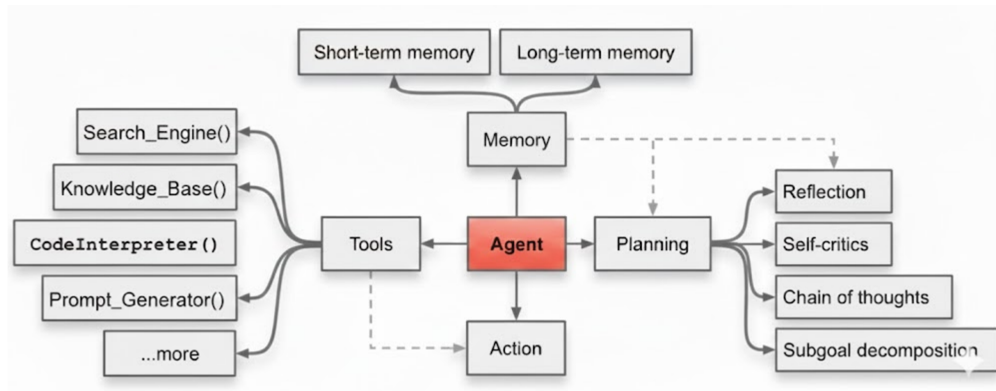


**Figure 2.8:** Agent Architecture: An overview of core components.

## 2.9 Existing approaches to Automated Software Documentation

Automated software documentation encompasses a spectrum of techniques for producing human-readable descriptions of code. At one end of this spectrum sit rule-based generators that leverage program structure and handcrafted templates to produce consistent, controlled comments. At the other end are retrieval and neural approaches that either reuse existing human-written text or synthesize new explanations by learning statistical and semantic patterns from large corpora. More recently, large language models (LLMs) and transformer-based architectures have blurred these boundaries by combining broad contextual understanding with the ability to generate fluent, varied documentation.

Each family of approaches carries distinct trade-offs: template systems offer predictability and grammatical correctness but limited adaptability; information-retrieval methods can reuse natural, proven explanations yet depend heavily on matching corpus examples; learning-based and LLM approaches provide greater flexibility and contextual reasoning but raise concerns about factual accuracy, dataset bias, and evaluation. Practical tools often adopt hybrid strategies — augmenting templates with retrieval priors or grounding model outputs in static analysis — to balance fluency, faithfulness, and coverage. The remainder of this chapter surveys such paradigms in turn.

### 2.9.1 Rule-Based Template Systems

Rule-based documentation systems rely on fixed templates or grammars to transform program entities like methods, classes, and functions into natural-language text. These systems go back to the early days of programming languages: for example, **Javadoc** [62], which was introduced by Sun Microsystems in the mid-1990s, was designed to generate API documentation from special comments embedded in Java code. Javadoc uses structured comment blocks to fill predefined textual templates, emitting simple HTML pages. Similarly, another important tool is **Doxygen** [63], first released in 1997, which is a language-agnostic documentation generator that parses annotated comments in the most popular general-purpose languages such as C, C++, Java, Python, etc., and produces cross-referenced manuals. These tools simply apply human-written templates or markup rules. Since traditional and non-automated documentation tends to be outdated or missing due to cost and oversight, tools like Javadoc and Doxygen automatically export the developer's comments into a user-friendly form. Overall, early automated documentation was largely template-driven, using either comment markup or pattern-based summarization.

**Template Structures and Examples**

Template-driven documentation generators operate with fixed sentence or comment patterns. In a typical API documentation tool, the template might be implicit: for a method comment, a one-line description is followed by `@param` and `@return` lines. For example, a generated Javadoc comment might look like:

```
/**
 * Computes the sum of two integers.
 *
 * @param a the first integer
 * @param b the second integer
 * @return the sum of a and b
 */
public int add(int a, int b) { ... }
```

This output follows a template: a plain English description and standard tags, even if written by a developer. Many templates like this are simply concatenated or slightly aggregated to form longer comments. In all cases, the structure of the output is largely predetermined by the template: the generator only substitutes specific identifiers or values extracted from the code.

**Advantages and Limitations of Template-Based Approaches**

Template-based systems produce fluent, predictable and controllable text because the phrasing is predefined and editable. By operating on parsed program structure they generate consistent, interpretable documentation quickly and without large training data. Their main limitations are brittleness and low adaptability: novel code patterns require manual rule additions, outputs can feel repetitive or formulaic, and templates cannot infer implicit intent or wider project context. For these reasons, template generators remain popular for API documentation when reliability and control are more important than flexibility.

### 2.9.2 Information-Retrieval Systems

Information-retrieval (IR) approaches generate code documentation by reusing and adapting existing natural language content, rather than synthesizing descriptions from scratch. These methods leverage the observation that similar code snippets often have similar documentation. By treating source code and associated texts as retrievable units, IR-based systems find relevant human-written descriptions and repurpose them as documentation for new code. Unlike generation-based techniques that directly translate code structure into English, IR-based systems

focus on searching large corpora for matches. Approaches like latent semantic indexing (LSI) and vector space modeling (VSM) have been employed to generate documentation for classes and methods [64]. Key examples of such systems include clone-based comment recommenders like:

- **CloCom (Clone-and-Comment)**: CloCom [65] scans a large corpus for clones tolerant to renaming, matching by structural features (AST nodes, token sequences) rather than raw text. For each target method it retrieves the closest clone's comment from a corpus of well-commented projects and suggests it as documentation — effective when near-duplicates exist.
- **AutoComment**: AutoComment [66] mines Stack Overflow Q&A to build a code–description database. Given new code, it searches for similar examples using code-similarity measures and, if a close match is found, transfers the corresponding Q&A explanation as a comment — useful for common tasks well represented online.

#### Advantages and Limitations of IR-Based Approaches

IR systems reuse human-authored content, are simple to build (index + retrieval), scale cheaply with more corpus data, and provide traceability to source examples. Their main drawbacks are limited coverage for novel or rare code patterns, brittleness to contextual differences (variable names, edge cases), inability to invent explanations beyond retrieved text, and inconsistent tone and detail when sourcing from diverse repositories.

### 2.9.3 AI and LLM-Based Systems

In the last few years, Large Language Models (LLMs) have revolutionized how information is handled and how software is developed. These models, trained on vast amounts of text data, have demonstrated remarkable capabilities in understanding and generating human-like text, becoming not only a mere convenience but a vital asset in the software development lifecycle. Because code is always growing in complexity and size, developers increasingly rely on LLMs to assist with tasks such as code generation, debugging, and indeed, code documentation. Code documentation using LLMs has huge potential to improve developer productivity, given the tedious and time-consuming nature of the task. Compared to previous rule-based or retrieval-based methods, LLMs can generate more context-aware, fluent, and human-like documentation by learning from large-scale code corpora. In this way, LLMs can capture subtle patterns and conventions in code comments that are difficult to encode manually or with tools that simply copy existing text or fill templates. There are many open-source and commercial models that have

been developed over the years, such as the GPT series, LLaMA, Claude, and many others, whose performance varies and will be discussed in the following sections.

### Foundation Models and Code Summarization

Early code summarization used rule-based systems that were reliable but rigid. Learning-based models enabled context-sensitive summaries, while Transformers improved long-range dependency modeling. Foundation models like GPT-3 and Codex revolutionized automated documentation, generating summaries comparable to or better than developer-written ones [64]. Codex, fine-tuned on GitHub code, produces human-like documentation, while GPT-3, GPT-4, and LLaMA outperform original comments [67] [68]. However, performance varies by language. Overall, LLMs show strong potential for high-quality documentation generation.

### Global Structure Analysis

Global structure analysis constructs a repository-wide project tree preserving semantic hierarchy: the root is the repository, intermediate nodes are directories, and file nodes correspond to files. Each file is parsed with an AST extractor to harvest meta-information (type, name, signature, etc.) for classes and functions, which become atomic documentation units attached as leaf nodes. Call relationships are extracted to provide richer context. This combined structural and reference view, treated as a DAG, supplies both local implementation details and global usage patterns, enabling the model to generate fine-grained, repository-grounded documentation.

### Evaluating Documentation

Evaluating the quality of generated documentation remains a complex and inherently multi-dimensional task, which makes reliable, scalable automatic assessment difficult [69]. In fact, most studies still rely on human judgments — extremely costly and often using bespoke rubrics that differ across papers [70], [71], [72] — and occasionally complement these with reference-based metrics such as BLEU or ROUGE, despite their weak correlation with human evaluations. Overall, as of now, there is no widely accepted, comprehensive methodology for measuring documentation quality.

### Datasets

For training and evaluating models that generate or evaluate software documentation, large-scale paired datasets provide the supervised examples needed to learn mappings from code structure and semantics to concise, accurate natural-language

summaries; enable evaluation on realistic developer-facing tasks; and expose models to diverse languages, coding styles, and domain conventions. Below we list the most commonly used resources and provide a focused description of each.

- **DocString** [73]: A corpus of Python functions paired with developer-written docstrings from open-source repositories. Useful for function-level docstring generation and reflecting idiomatic styles (Google, NumPy, reST).
- **CodeSearchNet** [74]: A multilingual benchmark pairing code snippets with natural-language queries and short descriptions across languages (Python, Java, JavaScript, PHP, Ruby, Go). It provides standardized splits and evaluation scripts for retrieval and summarization.
- **CodeXGLUE** [75]: A unified benchmark suite combining multiple datasets and tasks (summarization, completion, clone detection, translation, etc.), allowing comparison using common metrics (BLEU, CodeBLEU) and community baselines.

## 2.10 Automated Documentation in Robotic Process Automation (RPA)

### 2.10.1 Overview of RPA

Robotic Process Automation (RPA) refers to the use of software "robots" or agents to automate repetitive, rule-based tasks that are usually performed by humans on computer interfaces. These software bots emulate human interactions with digital systems, executing tasks like data entry, form processing, or triggering responses in various applications. The essence of RPA is working at the user interface level without requiring changes to the underlying IT infrastructure, making it a flexible automation approach for existing systems. By automating routine processes, RPA can operate 24/7 with high accuracy and speed, significantly reducing processing times and minimizing errors. Rather than aiming to *replace* human workers, RPA is designed to **augment** them: it frees employees from monotonous and structured tasks so that they can focus on more complex, creative, and value-adding activities. Today's leading RPA software solutions — including UiPath, Blue Prism, and Automation Anywhere — provide user-friendly environments for building automation workflows without extensive coding.

### 2.10.2 Applications and Use Cases of RPA

RPA applies across industries to automate repetitive processes that interact with multiple software systems. Common use cases include:

- **Finance & Accounting:** Automating invoice processing, accounts payable/receivable, reconciliations and report generation by extracting and entering data across systems.
- **Human Resources:** Streamlining onboarding/offboarding, payroll and benefits administration by collecting form data and updating HR systems automatically.
- **Customer Service:** Handling routine inquiries, logging tickets, sending acknowledgements, and transferring data between email and CRM systems to free agents for more complex issues.
- **Supply Chain & Logistics:** Updating inventory, processing orders and synchronizing shipment or procurement data across ERP and logistics platforms.
- **Healthcare Administration:** Automating patient record entry, scheduling and claims/billing tasks to reduce administrative burden.
- **IT & Data Management:** Performing routine maintenance, account provisioning, data migrations and nightly batch operations without manual intervention.

### 2.10.3 UiPath: A Leading RPA Platform

Among the various RPA platforms, **UiPath** has emerged as one of the leading solutions in both industry adoption and technical capabilities. UiPath provides an end-to-end automation suite that combines core RPA functionality with additional tools for process discovery, analytics, and AI integration. UiPath's architecture is composed of several important components, some of which are:

1. **UiPath Studio**: a desktop visual designer for building automation workflows with drag-and-drop activities, supporting sequences, branching, loops and error handling — often without code.
2. **UiPath Robots**: attended or unattended agents that execute deployed workflows, automating user actions on target systems.
3. **UiPath Orchestrator**: a web platform for deploying, scheduling, monitoring and managing robots at scale, offering logging, queues and security controls.

This modular architecture (Studio–Robot–Orchestrator) is representative of modern RPA platforms and highlights how UiPath balances **usability** and **scalability**. Business users or analysts can design processes in Studio using high-level activities, while IT administrators use Orchestrator to ensure reliability, security, and integration into the enterprise environment.

### 2.10.4 Advantages of RPA

RPA brings several practical benefits when applied to suitable, rule-based processes:

- **Improved efficiency and speed:** Bots execute repetitive tasks much faster than humans and can run continuously, reducing cycle times and accelerating throughput.
- **Higher accuracy and consistency:** Automated execution removes many manual errors and produces uniform outputs once correctly configured.
- **Cost savings and scalability:** After initial development costs, running bots is inexpensive at scale; capacity can be increased by deploying additional robot instances rather than hiring staff.
- **Enhanced compliance and auditability:** Deterministic bot behavior and detailed logs improve traceability and help meet regulatory requirements in sectors like finance and healthcare.
- **Better employee experience:** By offloading monotonous tasks, RPA lets staff focus on higher-value, creative work, often improving morale and productivity.

These advantages make RPA an attractive, low-invasiveness option for automating stable, high-volume processes without heavy changes to existing systems.

### 2.10.5 Limitations and Challenges of RPA

Despite its clear benefits, RPA also has constraints that must be considered: it is best suited for structured, rule-based tasks and cannot natively handle unstructured inputs or complex judgment without complementary AI components. Bots that mimic UI interactions are fragile and very sensitive to interface or workflow changes, so maintaining many bots in a dynamic environment can incur significant effort. While RPA can avoid deep integrations, scaling across legacy systems, remote desktops or complex workflows often requires careful engineering and may be less stable than API-based automation. Licenses, development effort and ongoing maintenance can be expensive. Successful deployment requires developers, IT support and organizational buy-in, and resistance to change or lack of expertise can hinder adoption. Bots may access sensitive data, so least-privilege, auditing and regulatory controls are essential to avoid breaches or compliance violations. Finally, processes with many exceptions or poor documentation are poor candidates for RPA, as automating flawed processes risks limited benefit.

### 2.10.6 Potential of LLMs in Automated RPA Documentation

One emerging way to enhance RPA development and maintenance is the use of **Large Language Models** to generate and assist with technical documentation for RPA software. Creating and updating documentation for automated workflows can be a time-consuming task: each RPA process ideally comes with technical specifications, user guides, and maintenance notes. LLMs offer the potential to **automate the generation of these documents** by analyzing the RPA workflows or code and producing human-readable descriptions. Recent research has demonstrated the feasibility of using LLMs to produce software documentation in many other popular languages. An area that has not yet been largely explored is the application of similar techniques in the RPA domain, in which an LLM could read a UiPath workflow (XML or JSON files defining the process steps and actions) and then produce a well-structured technical document explaining what the bot does, step by step.

By automating documentation, organizations can ensure that their RPA solutions come with up-to-date guides and technical specifications, which in turn **facilitates governance and knowledge transfer**. Moreover, when processes change, the documentation can be regenerated to reflect updates, closing the gap that often exists between implementation and documentation. Such documentation can also be useful as a means to translate processes from a specific RPA platform (e.g., UiPath) to another one (e.g., Automation Anywhere), by providing a high-level description of the process that can then be re-implemented in the target platform.

However, it is important to acknowledge the limitations and considerations of this approach. LLM-generated content should be validated by humans, as these models might sometimes produce inaccurate or nonspecific explanations if the prompt or context is insufficient. Ensuring data security is another concern: if an RPA workflow contains sensitive business logic or data and it is used as input to an LLM, organizations must manage that risk. Despite these caveats, the intersection of RPA and LLM technology is a frontier that could significantly augment the development lifecycle.

# Chapter 3

# Case Study: An AI-Powered Agent for Automated UiPath Documentation

## 3.1 Motivation

Technical documentation plays an extremely important role in the lifecycle of Robotic Process Automation (RPA) solutions. In enterprise contexts, as previously stated, documentation is essential for activities such as development, maintenance, knowledge transfer, and the overall long-term sustainability of automation programs. Despite its importance, documentation is often perceived as a secondary task, resulting in artifacts that are incomplete, outdated, or inconsistent with the actual implementation.

This issue is particularly evident in projects developed with UiPath, where the functional logic of an automation is distributed across multiple workflow files written in XAML (eXtensible Application Markup Language), configuration objects stored in JSON format, and various dependencies managed through NuGet packages. As projects evolve, manual documentation may quickly diverge from the source code, increasing the risk of misunderstandings, technical debt, and operational errors. Furthermore, the effort required to keep documentation aligned with the implementation often leads teams to overlook its importance.

The motivation behind this case study is to investigate whether automation itself, combined with artificial intelligence, can be leveraged to address this problem. By applying deterministic code analysis and leveraging LLM capabilities, the documentation process can be transformed from a manual, error-prone, and tedious activity into a systematic and easily repeatable pipeline. In this perspective,

documentation is treated as a dynamic asset that can be generated on demand and kept continuously aligned with the existing system.

The system presented in this chapter, which will be referred to as the **UiPath Documentation Generator**, serves as an experimental platform to explore how automated pipelines, combining deterministic parsing with LLM-powered generation, can interpret an entire project and produce structured documentation with considerably less human intervention required. This case study therefore bridges the gap between the theoretical concepts discussed in previous chapters and a practical, real-world application, demonstrating the feasibility and potential impact of LLMs in the automation of technical documentation in RPA projects.

## 3.2 Requirements and Design Goals

Before diving into the architecture of the system, it is essential to outline the requirements and design goals that led to the final implementation. These were elaborated based on the existing documentation created at Poseidon SB and on the theoretical concepts discussed in Chapter 2.

1. **Automation**: the generation process should require minimal human intervention, taking a UiPath project as input and producing complete documentation as output without any additional manual steps;

2. **Comprehensiveness**: the generated documentation should cover all essential aspects of a UiPath project, without omitting any critical information or important concepts;

3. **Accuracy**: the documentation should reflect the actual content of the input files, avoiding or at least minimizing hallucinations. This is crucial since LLMs can sometimes generate incorrect content [53];

4. **Readability**: the output should be written in clear, professional technical language suitable for different stakeholders;

5. **Modularity**: the system architecture should be modular, allowing individual components to be replaced or enhanced independently, without affecting the rest of the pipeline;

6. **Multi-model support**: the system should enable comparative analysis of documentation quality across different models from various providers;

7. **Evaluation capability**: the system should include built-in mechanisms for assessing the quality of generated documentation through quantitative metrics and qualitative LLM-based evaluation.

## 3.3 Technology Stack

The UiPath Documentation Generator is implemented as a Python-based application, usable mainly through the command line or via a highly convenient web interface. As an extremely popular language, Python offers an extensive ecosystem of libraries for XML processing, natural language processing, and API integration, as well as widespread adoption in the AI community, making it a smart choice for this project.

### 3.3.1 Core Technologies

#### lxml

Since XAML is an XML-based format, the `lxml` library [76] is used for parsing XAML workflow files. It provides high-performance XML processing capabilities that are essential for navigating the complex structure of UiPath workflow definitions. Unlike simpler parsers, `lxml` supports full XPath 1.0 queries and handles large XML documents efficiently, which is critical when processing enterprise UiPath projects containing multiple workflow files.

#### GitPython

The `GitPython` library [77] enables interaction with Git repositories, allowing the system to clone UiPath projects directly from GitHub or other Git hosting platforms. This allows the system to automate the retrieval of source code more seamlessly and efficiently, supporting both local and remote project inputs.

### 3.3.2 LLM Integration

#### Anthropic Claude API

The primary LLM integration uses the Anthropic Python SDK to access Claude models [23], which have proven to be more suitable for documentation generation tasks. Claude Sonnet 4.6 is used as the default model for documentation generation. The system uses the Messages API with configurable parameters for temperature (defaulting to 0.3 for deterministic, technical output) and maximum token count (set to 16,000 to accommodate comprehensive documentation).

#### OpenRouter

To enable a more complete comparison, the system also integrates with OpenRouter [78], a unified API gateway that provides access to models from multiple providers including OpenAI, Google, Meta, and others. This integration allows the system to

route requests to different models through a single interface. The ability to compare outputs across models is particularly important for evaluating which architectures produce the best documentation for RPA projects.

### 3.3.3 Evaluation Libraries

#### NLTK

The Natural Language Toolkit (NLTK) [79] is used for computing BLEU scores [51] during the evaluation phase. NLTK provides robust implementations of sentence and word tokenization as well as BLEU score calculation, which are necessary for comparing generated documentation against reference documentation.

#### ROUGE Score

The `rouge-score` library implements ROUGE metrics [52], providing recall-oriented evaluation of content overlap between generated and reference documentation. These metrics complement BLEU by focusing on how much of the reference content is captured in the generated output.

### 3.3.4 User Interface

#### Streamlit

A web-based GUI is built using Streamlit [80], a Python framework for creating interactive web applications. The GUI provides visual access to all system features that are accessible via CLI, including documentation generation, standalone quality evaluation, and document comparison, making it a convenient way to access the tool.
Table 3.1 provides a summary of the complete technology stack and the role of each component within the system.

| Component | Technology | Role |
|---|---|---|
| Language | Python 3.8+ | Core implementation |
| XML Parsing | lxml | XAML workflow parsing |
| Git Integration | GitPython | Remote repository cloning |
| Primary LLM | Anthropic Claude API | Documentation generation |
| Multi-model Gateway | OpenRouter API | Cross-provider comparison |
| NLP Metrics | NLTK | BLEU score computation |
| Recall Metrics | rouge-score | ROUGE score computation |
| Web Interface | Streamlit | Optional GUI |

**Table 3.1:** Technology stack of the UiPath Documentation Generator

## 3.4 System Architecture

The UiPath Documentation Generator follows a modular pipeline architecture that processes a UiPath project through a series of well-defined stages and can be divided into four layers, each responsible for a specific macro-function within the documentation lifecycle:

1. **Input Layer**: receives and prepares the UiPath project, either from a local directory or a remote Git repository;
2. **Analysis Layer**: parses and analyzes all project files, extracting structured information from XAML workflows and JSON configuration files;
3. **Generation Layer**: transforms the structured analysis data into human-readable documentation using LLMs;
4. **Evaluation Layer**: assesses the quality of the generated documentation through both quantitative and qualitative metrics.

Figure 3.1 illustrates the overall system architecture and the data flow between components.

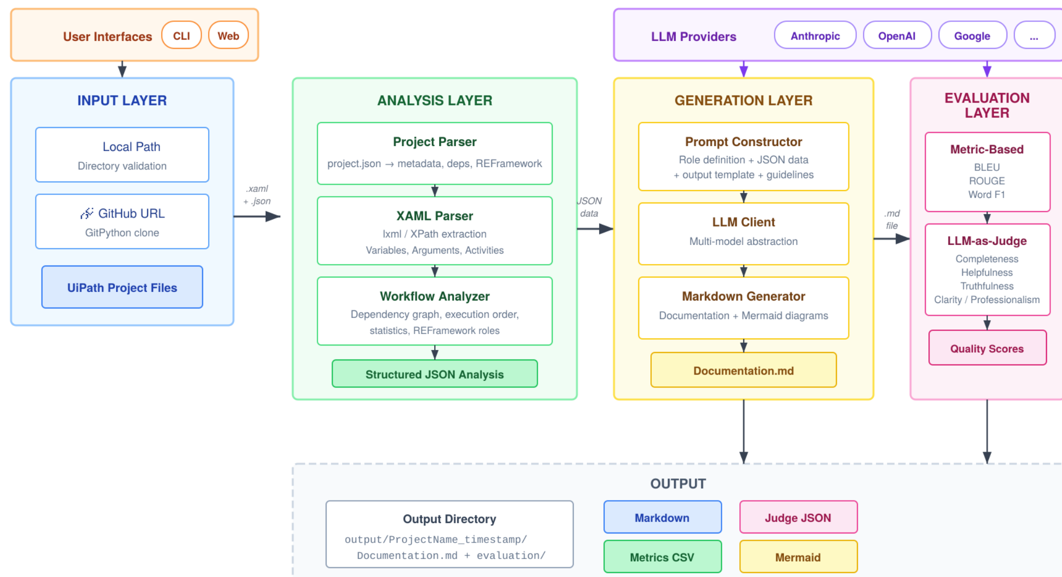


**Figure 3.1:** System architecture of the UiPath Documentation Generator. The pipeline processes a UiPath project through four layers.

## 3.5 Input Layer: Project Retrieval

The first stage of the pipeline is mainly responsible for retrieving the UiPath project according to the specified input method, such as a local directory path or a remote Git repository URL.

### 3.5.1 Local Project Input

When a local path is provided, the system validates that the directory actually exists and contains the expected UiPath project structure. The minimum requirement is the presence of at least one `.xaml` file, which represents a UiPath workflow. The presence of a `project.json` file, while not strictly mandatory, enables the extraction of essential metadata such as the project name, version, dependencies, and main entry point.

### 3.5.2 GitHub Integration

By using GitPython to clone the repository into a temporary directory, a specific branch can be selected, which is useful when projects maintain separate branches for development and production environments. Upon successful cloning, the temporary directory is treated identically to a local project path for all subsequent processing stages.

## 3.6 Analysis Layer: Parsing and Structure Extraction

The analysis layer is the core deterministic component of the system. It is responsible for extracting all relevant structural and semantic information from the UiPath project files without relying on any AI-based inference. This separation of concerns ensures that the factual data presented to the LLM is accurate and complete, mitigating the risk of hallucination in the final documentation.

The analysis layer consists of three main components: the **Project Parser**, the **XAML Parser**, and the **Workflow Analyzer**, which orchestrates the entire analysis process.

### 3.6.1 Project Parser

The Project Parser component is responsible for extracting project metadata from the `project.json` file that is present in every UiPath project. Table 3.2 illustrates some important information that can be extracted from this file.

| JSON Field | Type | Documentation Use |
|---|---|---|
| `name` | String | Project title in documentation header |
| `description` | String | Executive summary and overview |
| `projectVersion` | String | Version tracking and traceability |
| `main` | String | Identifies execution entry point |
| `dependencies` | Object | Lists external libraries and integrations |
| `studioVersion` | String | Records development environment version |
| `expressionLanguage` | String | Indicates VB.NET or C# expressions |
| `projectType` | String | Classifies as Process, Library, or Test |
| `targetFramework` | String | Specifies .NET runtime compatibility |

**Table 3.2:** Key fields extracted from UiPath `project.json` and their documentation relevance

### 3.6.2 XAML Parser

UiPath stores workflow definitions as XAML files, each of which contains the complete definition of a workflow, including its type, variables, arguments, activities, error handling constructs, and references to other workflows.

The parser uses `lxml` to build a Document Object Model (DOM) tree of each XAML file and then traverses this tree using XPath queries to extract the following information.

#### Workflow Type Detection

The root element of a XAML file determines the workflow type. Table 3.3 summarizes the mapping between XAML root structures and their corresponding workflow types, each of which has specific use cases and implications for how the workflow is structured and executed.

| XAML Root/Child Element | Workflow Type | Typical Use Case |
|---|---|---|
| `<Activity><Sequence>` | Sequence | Simple linear processes |
| `<Activity><Flowchart>` | Flowchart | Complex branching logic |
| `<Activity><StateMachine>` | State Machine | REFramework, state-driven processes |

**Table 3.3:** XAML root elements and corresponding UiPath workflow types

### Variable Extraction

The parser extracts all variable declarations, capturing the variable name, data type (specified through XAML type arguments), default value, and scope. The data type extraction process requires special handling because UiPath encodes types using CLR (Common Language Runtime) type references. For instance, a variable of type `System.Data.DataTable` appears in XAML as a `TypeArgument` attribute referencing the fully qualified .NET type. The parser normalizes these type references by stripping namespace prefixes and assembly qualifications, converting them into human-readable type names (e.g., `DataTable`, `String`, `Int32`, `Boolean`).

### Argument Extraction

Arguments define the input/output interface of a workflow, enabling data exchange between workflows when one invokes another. The parser identifies arguments and classifies them by direction:

- `InArgument<T>`: data passed into the workflow;
- `OutArgument<T>`: data returned from the workflow;
- `InOutArgument<T>`: data passed in and returned with optional modifications.

Arguments are particularly important for understanding how workflows communicate with each other in modular UiPath projects. The parser records each argument's name, type, and direction, creating a clear specification for each workflow that the LLM can use to describe data flow in the generated documentation.

### Activity Extraction

Activities are the fundamental building blocks of UiPath workflows, representing atomic actions such as assigning values, invoking methods, performing UI actions, or controlling flow. The parser identifies activities by examining element tags and their `DisplayName` attributes, filtering out all elements that are not activities. For each activity, the parser records the activity type, display name, and more.

The parser maintains an exclusion list of XAML element patterns that should not be treated as activities, such as `Variable`, `Literal`, `VisualBasicValue`, `VisualBasicReference`, `Argument`, and various internal UiPath designer elements. This filtering ensures that only the relevant workflow steps are reported, avoiding noise and clutter in the final documentation. Table 3.4 lists the most commonly encountered UiPath activity types and how they are categorized by the parser.

| Activity Type | Category | Description |
|---|---|---|
| `Assign` | Data | Assigns a value to a variable |
| `If` / `FlowDecision` | Control Flow | Conditional branching |
| `ForEach` / `While` | Control Flow | Iterative loops |
| `InvokeWorkflowFile` | Invocation | Calls a sub-workflow |
| `TryCatch` | Error Handling | Exception handling block |
| `RetryScope` | Error Handling | Automatic retry on failure |
| `LogMessage` | Logging | Writes to the execution log |
| `Click` / `TypeInto` | UI Automation | User interface interactions |
| `ReadRange` / `WriteRange` | Excel | Spreadsheet data operations |
| `SendOutlookMailMessage` | Email | Sends email through Outlook |

**Table 3.4:** Most common UiPath activity types recognized by the XAML Parser

### Dependency Detection

In order to understand how workflows invoke one another, the parser identifies `InvokeWorkflowFile` activities, which are the UiPath mechanism for calling other workflows, and extracts the referenced workflow file paths. This information is later used to construct a project-wide dependency graph that reveals the overall hierarchy and execution flow across the entire project.

### Error Handling Analysis

The parser specifically identifies error handling constructs, including:

- **TryCatch** activities and their associated `Catch` blocks, with the specific exception types being handled;
- **RetryScope** activities, which automatically retry failed operations a specified number of times before throwing the exception.

This information is essential for documenting how robust and resilient the entire automation actually is. Enterprise RPA projects typically implement multi-layered error handling with specific exception types caught at different levels of the workflow hierarchy.

### 3.6.3 Workflow Analyzer

The Workflow Analyzer acts as the coordinator between the Project Parser and the XAML Parser to produce a complete project analysis. It is responsible for:

1. **File discovery**: recursively scans the project directory for all `.xaml` files while excluding temporary and version control directories;

2. **Coordinated parsing**: invokes the Project Parser and XAML Parser on each discovered file, collecting results into a unified data structure;

3. **Dependency graph construction**: builds a directed graph that maps each workflow to the set of workflows it invokes;

4. **Execution order determination**: starting from the main entry point workflow, the analyzer performs a depth-first traversal of the dependency graph to determine a plausible execution order;

5. **REFramework analysis**: if the project is identified as a REFramework implementation, the analyzer categorizes workflows into their framework roles and identifies configuration files;

6. **Statistics computation**: calculates aggregate metrics such as the total number of workflows, variables, arguments, activities, and error handlers, as well as a frequency distribution of activity types.

Its output is a comprehensive JSON-serializable dictionary containing all extracted information, which serves as the input to the generation layer. Table 3.5 shows the top-level structure of this output and the information contained in each section.

| Key | Content |
|---|---|
| `project_info` | Name, version, description, entry point, framework, expression language, dependencies |
| `workflows` | List of per-workflow analysis results (type, variables, arguments, activities, error handlers) |
| `dependency_graph` | Adjacency list mapping each workflow to its invoked sub-workflows |
| `execution_order` | Ordered list of workflows based on dependency traversal |
| `statistics` | Aggregate counts: total workflows, variables, arguments, activities, error handlers, activity frequency |
| `reframework_info` | (If applicable) Framework role assignments and configuration file references |

**Table 3.5:** Structure of the Workflow Analyzer output dictionary

## 3.7 Generation Layer: LLM-Powered Documentation

The generation layer transforms the structured analysis data into human-readable technical documentation using LLMs. This is where the system leverages the natural language generation capabilities discussed in Chapter 2 to produce fluent, well-organized documentation from raw technical data.

### 3.7.1 LLM Client Abstraction

A key architectural decision was the creation of a unified LLM client abstraction layer that separates the documentation generation logic from any specific LLM provider. The `LLMClient` class provides a consistent interface for sending prompts and receiving completions, regardless of whether the underlying provider is Anthropic, OpenAI, Google, Meta, or any other model accessible through OpenRouter.

The client handles:

- **Provider detection**: automatically determines the appropriate provider based on the model identifier;
- **Model resolution**: maps user-friendly model names to their unique API identifiers, as shown in Table 3.6;
- **API key management**: supports API keys from environment variables, configuration files, or direct parameter passing;

- **Error handling**: manages API-specific error responses, rate limits, and timeout conditions with appropriate retry logic.

| User Alias | Provider | Canonical API Identifier |
|---|---|---|
| `claude-sonnet` | Anthropic | `claude-sonnet-4-5-20250929` |
| `claude-haiku` | Anthropic | `claude-haiku-4-5-20251001` |
| `gpt-4o` | OpenRouter | `openai/gpt-4o` |
| `gpt-4o-mini` | OpenRouter | `openai/gpt-4o-mini` |
| `gemini-flash` | OpenRouter | `google/gemini-2.0-flash-001` |
| `gemini-pro` | OpenRouter | `google/gemini-2.5-pro-preview` |
| `llama-3` | OpenRouter | `meta-llama/llama-3.3-70b-instruct` |

**Table 3.6:** Model alias resolution: user-friendly names mapped to unique API identifiers

### 3.7.2 Prompt Engineering Strategy

**Prompt Structure**

The generation prompt is composed of several components, each serving a specific purpose in guiding the LLM's output. Table 3.7 shows these components and provides representative excerpts from the actual prompt template used in the system.

| **Component** | **Representative Content (excerpt)** |
|---|---|
| Role Definition | *"You are a technical documentation expert specializing in RPA (Robotic Process Automation) and UiPath projects. Generate comprehensive technical documentation based on the following project analysis data."* |
| Data Injection | *"PROJECT ANALYSIS DATA:* `{analysis_data_json}` *"* |
| Output Template | *"Structure the documentation with the following sections: 1. Executive Summary, 2. Project Information, 3. Architecture Overview, 4. Dependencies and Libraries, 5. Workflows and Process Flow [. . . ]"* |
| Diagram Instructions | *"Include a Mermaid graph showing workflow dependencies. Use* `graph TD` *syntax with nodes for each workflow and edges for InvokeWorkflowFile calls."* |
| Anti-Hallucination | *"If information is missing or unclear, write 'Not specified' rather than making assumptions or inventing details."* |
| Style Guidelines | *"Write in clear, professional technical language. Focus on what each component does and why, not just how. Use proper Markdown formatting with headers, tables, and code blocks where appropriate."* |

**Table 3.7:** Components of the documentation generation prompt

### Structured Data Injection

The complete project analysis data, serialized as a JSON object, is embedded directly in the prompt so that it serves as a form of contextual grounding, ensuring that the LLM has access to all factual information about the project. By providing pre-extracted, verified data rather than raw XAML files, the system reduces the risk of the LLM misinterpreting complex XML structures or hallucinating technical details. The inclusion of this reduced representation of the project data was made possible by the deterministic analysis performed in the previous layer.

### Output Template

The prompt specifies a detailed documentation structure with eleven sections, each with specific instructions about what information to include, ensuring consistent and comprehensive output regardless of the model used (Table 3.8).

| Section | Expected Content | Primary Audience |
|---|---|---|
| Executive Summary | High-level purpose, scope, and key characteristics | Stakeholders |
| Project Information | Name, version, framework, entry point, metadata | All |
| Architecture Overview | System design, patterns (e.g., RE-Framework), data flow | Architects |
| Dependencies | NuGet packages, versions, and their roles | Developers |
| Workflows & Flow | Per-workflow descriptions, Mermaid dependency graph | Developers |
| Variables & Arguments | Data types, scopes, directions, inter-workflow contracts | Developers |
| Activities & Logic | Key activities, business logic, decision points | Developers |
| Error Handling | TryCatch blocks, exception types, retry strategies | Developers |
| Configuration | Settings files, environment parameters, constants | Operations |
| Technical Details | Expression language, .NET framework, Studio version | Operations |
| Project Statistics | Aggregate counts and activity frequency distribution | All |

**Table 3.8:** Documentation output sections with expected content and intended audience

#### Anti-Hallucination Instructions

A crucial component of the prompt is the explicit instruction to avoid hallucinated information. The prompt explicitly states: *"If certain information is missing from the data, note it as "Not specified" rather than making assumptions."* This instruction directly addresses the tendency of LLMs to produce plausible-sounding but factually incorrect information [53].

#### Generation Guidelines

The prompt concludes with explicit guidelines that instruct the LLM to:

- Write in clear, professional technical language;
- Focus on both the *what* and the *why*, not only the *how*;
- Be concise but comprehensive;
- Use proper Markdown formatting;
- Include Mermaid diagrams [81] where helpful for visualization;
- Mark missing information as *"Not specified"* rather than making assumptions;
- Highlight relevant design patterns such as REFramework or Dispatcher-Performer.

### 3.7.3 Generation Parameters

Table 3.9 lists the configurable parameters used for model inference and their default values with justifications:

| **Parameter** | **Default** | **Rationale** |
|---|---|---|
| Temperature | 0.3 | Lower values encourage deterministic, factual output while allowing natural language variation. |
| Max Tokens | 16,000 | Sufficient for comprehensive documentation of most projects; within the output limits of modern LLMs. |
| System Prompt | Role + Template | Establishes domain expertise and output structure. |
| Stop Sequences | None | Allows the model to determine natural completion. |

**Table 3.9:** LLM inference parameters used for documentation generation

### 3.7.4 Output Format and Delivery

The system generates documentation in Markdown format, which was chosen for several reasons:

- Rendered by most code hosting platforms such as GitHub, enabling documentation to be stored alongside the project source code;
- Support for embedded Mermaid diagrams, which are used to visualize workflow dependency graphs;
- Easy conversion to other formats using tools such as Pandoc;
- Lightweight and version-control friendly, allowing documentation changes to be tracked alongside code changes;
- Easy to read by LLMs during evaluation, as it preserves structure that can be used for quality assessment.

The generated documentation is saved in an organized, timestamped directory structure, which enables multiple documentation versions generated across different models or project versions to coexist:

```
output/
  <ProjectName>_<timestamp>/
    <ProjectName>_Documentation.md
    evaluation/
      metrics_results.csv
      llm_judge_results.json
```

## 3.8 Evaluation Layer: Quality Assessment

The integrated evaluation layer provides two complementary approaches for assessing documentation quality.

### 3.8.1 Metric-Based Evaluation

The metric-based evaluation component compares generated documentation against a human-written reference document using established NLP metrics. The system computes the following metrics:

#### BLEU Score

The system calculates both sentence-level and corpus-level BLEU scores [51] using smoothed $n$-gram precision. Sentence-level BLEU averages the scores across all generated sentences, where each sentence is evaluated against the full set of reference sentences. Corpus-level BLEU treats the entire document as a single sequence, providing a holistic measure of lexical similarity.

#### ROUGE Scores

ROUGE-1, ROUGE-2, and ROUGE-L F1 scores [52] are computed to measure content coverage from a recall-oriented perspective. ROUGE-L, which measures the longest common subsequence, is particularly informative for documentation evaluation as it captures structural similarity beyond simple $n$-gram overlap.

#### Word-Based Metrics

The system also computes precision, recall, and F1 scores based on word-level overlap between the generated and reference documents. These metrics provide an intuitive measure of the vocabulary overlap between the two documents. Table 3.10 provides a comparative overview of the evaluation metrics used, highlighting their complementary strengths.

| Metric | Focus | Orientation | Strength |
|---|---|---|---|
| BLEU (sentence) | $n$-gram precision | Precision | Lexical accuracy per sentence |
| BLEU (corpus) | $n$-gram precision | Precision | Document-level similarity |
| ROUGE-1 | Unigram overlap | Recall | Content coverage at word level |
| ROUGE-2 | Bigram overlap | Recall | Phrase-level similarity |
| Word F1 | Word overlap | Balanced | Vocabulary agreement |

**Table 3.10:** Comparison of evaluation metrics used in the metric-based evaluation component

### 3.8.2 LLM-as-Judge Evaluation

Although useful, reference-based metrics have limitations when evaluating documentation quality. To address these limitations, the system includes an LLM-as-Judge evaluation component. This approach uses a separate LLM instance to evaluate the generated documentation across five quality dimensions, inspired by the multi-dimensional evaluation frameworks proposed in recent literature [69].

#### Evaluation Dimensions

The five dimensions assessed are summarized in Table 3.11, which also lists the specific criteria used by the judge model for each dimension:

| **Dimension** | **Assessment Criteria** |
|---|---|
| Completeness | Coverage of workflows, dependencies, variables, arguments, error handling, configuration, and project metadata. Penalizes missing sections or undocumented components. |
| Helpfulness | Usefulness for intended audiences (developers, architects, stakeholders). Evaluates actionable information, appropriate detail level, and practical guidance. |
| Truthfulness | Factual accuracy relative to the provided project analysis data. Checks for hallucinated features, incorrect specifications, or fabricated technical details. |
| Clarity | Writing quality, logical organization, consistent terminology, appropriate use of formatting, and readability of technical explanations. |
| Professionalism | Adherence to professional standards: neutral tone, consistent style, suitability for stakeholder review, and absence of informal language or subjective commentary. |

**Table 3.11:** LLM-as-Judge evaluation dimensions with assessment criteria

Each dimension is evaluated independently through a dedicated LLM call with a structured evaluation prompt. The prompt includes the dimension definition, specific evaluation criteria, the documentation to evaluate, and explicit instructions to provide a numerical score (0–10) along with identified strengths, weaknesses, and reasoning. Table 3.12 illustrates the structure of the evaluation prompt used for the Truthfulness dimension as a representative example.

| **Section** | **Content (excerpt)** |
|---|---|
| System Role | *"You are an expert evaluator of technical documentation for RPA (Robotic Process Automation) projects. Evaluate the following UiPath project documentation on the dimension of **Truthfulness**."* |
| Criteria | *"Consider: Are all documented features actually present in the project? Are technical specifications (types, counts, dependencies) accurate? Are there any fabricated or assumed details not supported by the source data?"* |
| Input | *[Full generated documentation inserted here]* |
| Response Format | *"Respond with a JSON object containing: score (integer 0–10), strengths (list of strings), weaknesses (list of strings), reasoning (string)."* |

**Table 3.12:** Example LLM-as-Judge evaluation prompt for the Truthfulness dimension

### Judge Independence

The judge model operates independently of the generation model in order to remain neutral and minimize bias in the evaluation. By default, the system uses Claude Sonnet 4.6 for evaluation regardless of which model generated the documentation. Users can override the judge model if desired, for example to cross-validate using a model from a different provider.

### Score Interpretation

The individual dimension scores are averaged to produce an overall quality score, which is categorized according to four quality levels:

- **Excellent** (8–10): exceeds professional documentation standards;
- **Good** (6–7): meets requirements well with minor areas for improvement;
- **Adequate** (4–5): meets basic requirements with some gaps;
- **Needs Improvement** (0–3): major issues requiring significant revision.

The combination of per-dimension scores with an aggregate rating provides both detailed feedback useful for improving specific aspects and a high-level quality indicator suitable for reporting. This dual approach helps bridge the gap between detailed information useful for developers and summary assessments needed by project managers and stakeholders.

## 3.9 Operational Workflow

This section describes the end-to-end operational workflow of the system, illustrating how the components described in previous sections interact with each other in order to produce documentation from a UiPath project.

### 3.9.1 Command-Line Interface

The primary interface is a command-line application that accepts a project path (either local or a GitHub URL) and optional parameters for model selection, output configuration, and evaluation options. Table 3.13 lists the available command-line arguments:

| Argument | Default | Description |
|---|---|---|
| `project_path` | (required) | Local path or GitHub URL |
| `--model` | `claude-sonnet` | LLM model for generation |
| `--branch` | `main` | Git branch (for GitHub URLs) |
| `--output-dir` | `./output` | Output directory for documentation |
| `--benchmark` | None | Path to reference doc for metrics |
| `--llm-judge` | False | Enable LLM-as-Judge evaluation |
| `--judge-model` | `claude-sonnet` | Model to use for LLM evaluation |

**Table 3.13:** Command-line arguments of the UiPath Documentation Generator

A typical invocation may follow this pattern:

```
python main.py https://github.com/UiPath/ReFrameWork \
    --model claude-sonnet \
    --llm-judge \
    --output-dir ./docs
```

### 3.9.2 Execution Flow

The execution proceeds through the following stages:

1. **Input resolution**: if a GitHub URL is detected, the repository is cloned to a temporary directory; otherwise, the local path is validated;
2. **Project analysis**: the Workflow Analyzer is instantiated with the project root and performs the complete analysis, parsing `project.json` and all discovered `.xaml` files;
3. **Documentation generation**: the Markdown Generator is instantiated with the selected model and provider, receives the analysis data, constructs the prompt, and invokes the LLM API;
4. **Output storage**: the generated Markdown documentation is saved to the timestamped output directory;
5. **Evaluation** (optional): if metric-based benchmarking is requested, the generated documentation is compared against the provided reference. If LLM-as-Judge evaluation is requested, the documentation is assessed across the five quality dimensions;
6. **Cleanup**: temporary directories created for GitHub clones are removed.

### 3.9.3 Web Interface

The Streamlit-based web interface provides the same functionality through a more convenient and visually appealing interface with three modes: **Generate Documentation**, which allows users to select the input method, model, and evaluation options, then generate and download documentation; **Evaluate Documentation**, which enables uploading existing documentation for standalone quality assessment using the LLM-as-Judge component; and **Compare Documents**, which supports uploading reference and generated documents for quantitative comparison using NLP metrics.

## 3.10 Limitations and Considerations

While the UiPath Documentation Generator demonstrates the feasibility and clear advantages of AI-assisted documentation generation for RPA projects, several limitations must be acknowledged.

#### LLM Dependency

The quality and style of the generated documentation are inherently tied to the capabilities and limitations of the underlying LLM, including the potential for subtle inaccuracies.

#### Context Window Constraints

Very large projects with dozens of workflows may produce analysis data that approaches the context limits of smaller models, although this is mitigated by deterministic parsing.

#### Evaluation Limitations

The LLM-as-Judge approach, while practical, introduces its own biases and may not correlate perfectly with human expert assessments.

#### Security Considerations

Submitting UiPath project data to external LLM APIs may raise data privacy concerns for organizations with sensitive business logic.

#### Dynamic Content

The system captures a snapshot of the project at a specific point in time. It does not automatically track or document changes over time. For that purpose, manual re-execution is required after updates.

#### Deterministic Parsing vs LLM Analysis

The decision to perform all structural analysis deterministically through code-based parsing rather than delegating XAML interpretation to the LLM, although more complex, guarantees that the extracted data faithfully represents the file content, eliminating the risk of LLM hallucinations. UiPath XAML files are extremely verbose, and deterministic parsing compresses them into a compact JSON representation. This token efficiency directly translates to lower API costs when processing large projects with many workflow files.

# Chapter 4

# Results and Analysis

This chapter presents the experimental results obtained from evaluating the UiPath Documentation Generator across multiple LLM providers and UiPath projects. The evaluation follows a dual methodology: first, traditional reference-based NLP metrics such as BLEU, ROUGE, and Word F1 are applied to quantify lexical similarity between generated and human-written documentation; second, an LLM-as-Judge approach is employed to assess qualitative dimensions that the previous metrics cannot capture. By contrasting these two evaluation paradigms, this chapter demonstrates both the capabilities and the limitations of current automated evaluation methods for documentation generation tasks.

## 4.1 Experimental Setup

### 4.1.1 Selected Models

Three generation models and one judge model were carefully selected for this evaluation, each from a different provider or occupying a distinct role in the experimental design. The selection criteria encompassed several considerations: the generation models should represent the current frontier of commercially available LLMs suitable for production documentation tasks, they should span different providers in order to avoid single-vendor bias, and they should differ meaningfully in pricing to enable a cost-quality analysis. The judge model, conversely, was deliberately chosen to be more capable than any of the generation models, in order to reduce evaluation errors and improve alignment with expert qualitative judgment.

### Generation Models

**Claude Sonnet 4.6** (Anthropic) is the balanced mid-tier model in Anthropic's Claude family, positioned between the lightweight Haiku and the flagship Opus variants. It supports a 1M-token context window and is widely recognized for strong instruction-following, nuanced reasoning, and code understanding. At $3.00/$15.00 per million input/output tokens, it represents the mid-range price point among the three generators.

**GPT-5.2** (OpenAI) is OpenAI's flagship model released in January 2026, representing the current state of the art from the GPT family. It features a 400K-token context window and has been praised for strong agentic capabilities, advanced reasoning, and competitive benchmark performance across coding and language tasks. Its inclusion ensures that the evaluation reflects the latest capabilities from the most widely adopted commercial LLM provider. At $1.75/$14.00 per million input/output tokens, it is competitively priced relative to its capabilities.

**Gemini 2.5 Pro** (Google) is Google DeepMind's flagship reasoning model, notable for its 1M-token native context window, the largest among the three models tested, and its strong performance on coding and multimodal benchmarks. Its inclusion tests whether Google's approach to long-context processing and "thinking-native" architecture translates into high-quality documentation generation. At $1.25/$10.00 per million input/output tokens, it is the most affordable of the three generators on a per-token basis.

Table 4.1 summarizes the key characteristics and pricing of the three generation models and the judge model. The pricing comparison reveals meaningful differences: Gemini 2.5 Pro is the most economical option, with input costs roughly 58% lower than Claude Sonnet 4.6 and 29% lower than GPT-5.2. Output costs follow a similar pattern, with Gemini 2.5 Pro at $10.00 per million tokens compared to $14.00 for GPT-5.2 and $15.00 for Claude Sonnet 4.6. These differences become significant at scale, and are therefore worth considering.

| **Model** | **Role** | **Context** | **Input ($/1M)** | **Output ($/1M)** |
|---|---|---|---|---|
| Claude Sonnet 4.6 | Generator | 200K | 3.00 | 15.00 |
| GPT-5.2 | Generator | 400K | 1.75 | 14.00 |
| Gemini 2.5 Pro | Generator | 1M | 1.25 | 10.00 |
| Claude Opus 4.6 | Judge | 1M | 5.00 | 25.00 |

**Table 4.1:** Comparison of the LLMs used for documentation generation and evaluation, including per-token API pricing as of February 2026.

### 4.1.2 Judge Model Selection

The LLM-as-Judge evaluation requires a model that can perform qualitative assessment across multiple dimensions simultaneously, maintaining consistency over a large number of evaluations while providing structured and quantifiable scores. Selecting an appropriate judge model is therefore a critical design decision that directly affects the reliability of the evaluation.

**Claude Opus 4.6** (Anthropic) was selected as the primary judge model; it is currently the most capable model in Anthropic's lineup and one of the strongest reasoning models commercially available. A fundamental design principle behind this choice was that the judge model should be strictly more capable than any of the generation models in order to better identify subtle quality differences, detect inaccuracies, and assess overall coherence in the generated documentation, thereby reducing evaluation errors and improving alignment with human judgment. At $5.00/$25.00 per million input/output tokens, Opus 4.6 is the most expensive model used in this study.

It is important to acknowledge that Claude Opus 4.6 shares a provider with Claude Sonnet 4.6, one of the three generation models. This introduces a potential concern regarding self-preference bias, the tendency for an LLM to overestimate the quality of outputs produced by models from its own family. Recent research has shown that this bias is a measurable phenomenon [82]. More broadly, Zheng et al. [83] systematically examined position, verbosity, and self-enhancement biases in LLM judges and found that, despite these limitations, strong LLM judges can achieve over 80% agreement with human preferences, comparable to inter-human agreement levels. Ye et al. [84] proposed a comprehensive framework that categorizes twelve distinct bias types in LLM-as-a-Judge settings, including self-preference.

Several design choices in this study follow these recommendations. First, Opus 4.6 and Sonnet 4.6 are architecturally distinct models with different training procedures and capability profiles; the judge is not evaluating its own outputs but those of a separate, less capable sibling model. Second, the evaluation is structured around explicit rubrics with detailed scoring criteria for each dimension, which constrains the judge's discretion and reduces the space for implicit bias. Third, the judge prompt is entirely different from the generation prompt, further separating the two roles. Nevertheless, the same-provider concern cannot be fully eliminated without empirical cross-provider validation, which is identified as a priority direction for future work in Section 4.6.3.

### 4.1.3 Test Projects

As test cases, three open-source projects publicly available on GitHub with varying size and complexity were selected. They represent a diverse set of automation scenarios commonly encountered in RPA practice, ranging from a single-workflow AI-powered document processing pipeline to a multi-stage document manipulation framework with many different interconnected workflows. All three projects are summarized in Table 4.2.

**REFrameWork**[1] is UiPath's official Robotic Enterprise Framework template, a production-grade, state-machine-based automation designed for transactional queue-driven processes. It implements a four-state architecture (Init, Get Transaction Data, Process Transaction, End Process) with two-tiered exception handling (Business Rule vs. System Exceptions), configurable retry logic, and centralized settings management via `Config.xlsx`. Being written in Visual Basic .NET, it represents the most architecturally complex project in the test set, with deeply nested invocation chains and a built-in test framework. Its widespread adoption in the UiPath community makes it a strong candidate for evaluating how well the documentation generator handles such common but complex patterns.

**doc-manipulation**[2] is a comprehensive document processing pipeline that demonstrates an end-to-end business scenario: email ingress, data extraction from password-protected Excel files, Word document generation from templates, and email egress. This project is the largest in the test set in terms of workflows. Also written in Visual Basic .NET, it showcases multiple execution strategies (single loop, multi-loop, REFramework-based) for processing the same data, along with extensive error handling. The project's multi-layered architecture and rich inter-workflow dependencies make it particularly challenging for automated documentation generation.

**DocumentUnderstanding_Demo1**[3] is a UiPath Document Understanding demonstration project that implements the full document processing pipeline: OCR digitization via OmniPage, intelligent classification using keyword-based and ML classifiers, multi-strategy data extraction (Form, Regex, and Machine Learning extractors), human-in-the-loop validation via Validation Station, and structured export to Excel. Despite consisting of a single workflow, it integrates seven NuGet packages including AI Center connectivity and represents a fundamentally different automation paradigm: AI-powered document intelligence rather than rule-based process automation. Written in C#, it tests the generator's ability to document modern UiPath capabilities that differ significantly from traditional RPA patterns.

[1] `https://github.com/UiPath/ReFrameWork`

[2] `https://github.com/AndreiBarbuOz/uipath-doc-manipulation`

[3] `https://github.com/sunilsm7/uipath-document-understanding`

| Project | Workflows | Activities | Language | Pattern |
|---|---|---|---|---|
| REFrameWork | 13 | 401 | VB .NET | State Machine |
| doc-manipulation | 27 | 744 | VB .NET | Multi-strategy |
| DU_Demo1 | 1 | 77 | C# | Sequence (AI) |

**Table 4.2:** Characteristics of the three open-source UiPath test projects selected for evaluation.

The decision to select open-source projects rather than proprietary enterprise codebases was driven by two important considerations: first, *reproducibility*: public repositories are easily accessible, making it straightforward to replicate the experiments. Second, *accessibility*: although enterprise RPA projects are more representative of real use cases, they typically encode sensitive information and business logic, making them difficult to obtain for academic work. The implications of this choice are also discussed in Section 4.6.2.

### 4.1.4 Evaluation Protocol

The complete evaluation protocol consists of the following steps:

1. For each of the 3 projects, generate documentation using each of the 3 models, producing a total of 9 documentation artifacts;
2. For each artifact, run the LLM-as-Judge evaluation across 5 quality dimensions (Completeness, Helpfulness, Truthfulness, Clarity, Professionalism) using Claude Opus 4.6;
3. Aggregate and compare results across models and projects;
4. Analyze score distributions to identify model-specific and project-specific patterns.

All three generation models were called using the same prompt template, a temperature of 0.3, and a maximum output token count of 16,000, as described in Chapter 3. The LLM-as-Judge evaluation was executed with temperature 0.2 to maximize consistency. The only variable across experiments was the model itself, ensuring that observed differences in output quality are attributable to model capabilities rather than other factors.

## 4.2 Metric-Based Evaluation Results

This section presents the results of the quantitative metric-based evaluation, where each generated documentation artifact is compared against a corresponding reference document using established NLP metrics. Since the three test projects are open-source community samples that either came with scarce or no official technical documentation, the reference documents were written specifically for this evaluation by expert UiPath developers at Poseidon SB, who analyzed each project's source code, configuration files, and README materials in detail before drafting the references. The resulting documents follow the same eleven-section structure used by the generator. The LLM-as-Judge evaluation (Section 4.4), by contrast, operates entirely without reference documents, assessing each artifact on its own merits.

### 4.2.1 BLEU Scores

Table 4.3 reports the corpus-level BLEU scores for each model across the three test projects. BLEU measures $n$-gram precision between the generated and reference texts, penalizing outputs that are significantly shorter than the reference.

| **Model** | **REFrameWork** | **doc-manip.** | **DocUnderst.** | **Avg.** |
|---|---|---|---|---|
| Claude Sonnet 4.6 | 6.74 | 5.02 | 7.10 | 6.29 |
| GPT-5.2 | 8.42 | 7.23 | 12.77 | 9.47 |
| Gemini 2.5 Pro | 11.89 | 10.92 | 13.92 | 12.24 |

**Table 4.3:** Corpus-level BLEU scores (%) for each model and project

BLEU scores are overall well below the thresholds typically associated with high-quality machine translation, reflecting an obvious mismatch between BLEU's design for translation evaluation and the documentation generation task. Interestingly, the ranking of models by BLEU is the *inverse* of the LLM-as-Judge ranking: Gemini 2.5 Pro achieves the highest average BLEU (12.24%), while Claude Sonnet 4.6 scores the lowest (6.29%).

This inversion is largely explained by output length, since Claude Sonnet 4.6 generates substantially longer documentation than Gemini 2.5 Pro. BLEU's brevity penalty does not fully compensate for this disparity, and the $n$-gram precision component is diluted by the larger volume of generated text that, while adding genuine value, does not overlap with the specific phrasing of the reference.

### 4.2.2 ROUGE Scores

ROUGE metrics provide a recall-oriented perspective, measuring how much of the reference content is captured in the generated output. Table 4.4 presents ROUGE-1 (unigram), ROUGE-2 (bigram), and ROUGE-L (longest common subsequence) F1 scores.

| Model | Metric | REFrameWork | doc-manip. | DocUnderst. | Avg. |
|---|---|---|---|---|---|
| Claude Sonnet 4.6 | ROUGE-1 | 49.71 | 40.79 | 48.56 | 46.35 |
| | ROUGE-2 | 17.45 | 15.11 | 17.87 | 16.81 |
| | ROUGE-L | 19.44 | 15.58 | 18.15 | 17.72 |
| GPT-5.2 | ROUGE-1 | 53.92 | 53.86 | 60.61 | 56.13 |
| | ROUGE-2 | 15.98 | 17.13 | 20.63 | 17.91 |
| | ROUGE-L | 20.75 | 18.53 | 22.59 | 20.62 |
| Gemini 2.5 Pro | ROUGE-1 | 59.12 | 58.65 | 60.09 | 59.29 |
| | ROUGE-2 | 21.73 | 18.24 | 20.26 | 20.08 |
| | ROUGE-L | 25.59 | 21.18 | 22.94 | 23.24 |

**Table 4.4:** ROUGE F1 scores (%) for each model and project

ROUGE scores follow a pattern consistent with the BLEU results: Gemini 2.5 Pro leads on all three ROUGE variants, followed by GPT-5.2, with Claude Sonnet 4.6 scoring the lowest. ROUGE-1 scores are substantially higher than ROUGE-2 and ROUGE-L, since individual technical terms overlap more easily than exact multi-word phrases. The gap between ROUGE-1 and ROUGE-2 is notably large, indicating that while the generated documentation uses some of the same vocabulary as the reference, the specific phrasing and sentence construction differ substantially. This phenomenon is discussed in Section 4.3.1.

### 4.2.3 Word-Based F1 Scores

Word-based precision, recall, and F1 scores provide an intuitive measure of vocabulary overlap between the generated and reference documents. Unlike BLEU and ROUGE, these metrics operate on unordered word sets, capturing whether the same technical terms and concepts appear in both documents regardless of phrasing.

| Model | REFrameWork | doc-manip. | DocUnderst. | Average |
|---|---|---|---|---|
| Claude Sonnet 4.6 | 36.22 | 28.71 | 32.24 | 32.39 |
| GPT-5.2 | 31.38 | 30.30 | 31.24 | 30.97 |
| Gemini 2.5 Pro | 38.33 | 33.33 | 35.59 | 35.75 |

**Table 4.5:** Word-based F1 scores (%) for each model and project

Word-based F1 scores present a different picture from BLEU and ROUGE. Here, Gemini 2.5 Pro again leads, but Claude Sonnet 4.6 comes in second place. This is because Word F1 operates on unordered word sets, making it less sensitive to output length: Sonnet's longer documentation introduces more unique vocabulary from the reference, boosting recall even though its precision is lower due to the additional content not present in the reference.

### 4.2.4 Metric-Based Summary

Table 4.6 provides an overall view of the average metric-based scores across all three projects for each model.

| **Model** | **BLEU** | **ROUGE-1** | **ROUGE-2** | **ROUGE-L** | **Word F1** |
|---|---|---|---|---|---|
| Claude Sonnet 4.6 | 6.29 | 46.35 | 16.81 | 17.72 | 32.39 |
| GPT-5.2 | 9.47 | 56.13 | 17.91 | 20.62 | 30.97 |
| Gemini 2.5 Pro | 12.24 | 59.29 | 20.08 | 23.24 | 35.75 |

**Table 4.6:** Average metric-based scores (%) across all three projects

The metric-based results consistently rank Gemini 2.5 Pro first across all five metrics, followed by GPT-5.2, with Claude Sonnet 4.6 last. This ranking is the exact inverse of the LLM-as-Judge ranking (Table 4.9), where Claude Sonnet 4.6 leads and Gemini 2.5 Pro trails. A key factor appears to be the relationship between output length and metric scores. Table 4.7 reports the average word counts for the generated documentation alongside the reference documents.

| **Model** | **Avg. Generated Words** | **Avg. Length Ratio** |
|---|---|---|
| Reference | 1,272 | 1.00 |
| Claude Sonnet 4.6 | 4,720 | 3.70 |
| GPT-5.2 | 2,547 | 1.99 |
| Gemini 2.5 Pro | 1,973 | 1.55 |

**Table 4.7:** Average word counts and length ratios (generated/reference) across all projects

Claude Sonnet 4.6 generates documentation that is on average 3.7 times longer than the reference, while Gemini 2.5 Pro produces output only 1.55 times the reference length. Since reference-based metrics reward lexical overlap with the reference, shorter outputs that more closely match the reference's scope and phrasing naturally score higher, even though the additional content in longer outputs may actually provide genuine value.

## 4.3 Limitations of Reference-Based Metrics

Before presenting the LLM-as-Judge results, some considerations must be addressed. While BLEU and ROUGE metrics are well-established in NLP evaluation, their application to documentation generation reveals the need for complementary evaluation approaches.

### 4.3.1 The Paraphrasing Problem

Consider the following example, where both descriptions accurately document the same UiPath workflow behavior:

> *Reference*: "The workflow initializes the application settings by reading the Config.xlsx file and storing the configuration values in the Config dictionary."
>
> *Generated*: "During startup, configuration parameters are loaded from an Excel-based settings file (Config.xlsx) and populated into a dictionary object for runtime access."

Despite containing identical information, these two sentences would receive moderate to low *n*-gram overlap scores due to different word choices. This phenomenon is especially pronounced in this research task, where the same concept can be described using multiple different but equally valid terminologies and sentence structures.

### 4.3.2 Structural vs. Semantic Equivalence

BLEU and ROUGE operate at the *n*-gram level and cannot assess whether the generated documentation is *semantically* equivalent to the reference. A generated document might reorganize information into a different but equally valid structure, describe workflows in a different order, or use alternative representation formats such as tables or diagrams. All of these changes would reduce metric scores without necessarily reducing the overall quality of the documentation.

### 4.3.3 Inability to Detect Hallucinations

Reference-based metrics cannot detect hallucinated content, because even if a generated document includes fabricated workflow descriptions or incorrect technical specifications alongside accurate content, the metrics may still report relatively high overlap scores, as the correct portions contribute positively. On the other hand, a document that is entirely accurate but uses different terminology would score lower.

### 4.3.4 Dependence on Reference Quality

Human-written documentation may itself be incomplete, outdated, or inconsistent with the actual implementation, as discussed in Chapter 1. Evaluating generated documentation against an imperfect reference introduces systematic bias: a generated document that is more complete or accurate than the reference would paradoxically receive lower scores.

These observations align with findings in the broader literature, where BLEU and ROUGE have been shown to correlate weakly with human judgments for text generation tasks that permit significant paraphrastic variation [51, 52, 69].

### 4.3.5 Illustrative Cases

To concretely demonstrate these limitations, Table 4.8 presents cases from the experimental results where metric scores and actual documentation quality diverge.

| **Case** | **BLEU** | **Judge** | **Explanation** |
|---|---|---|---|
| Gemini – REFrameWork | 11.89 | 7.6 | Highest BLEU, lowest judge score. Concise output achieves high lexical overlap, but the judge found incomplete coverage and low truthfulness (6.0). |
| Sonnet – doc-manipulation | 5.02 | 8.8 | Lowest BLEU, highest judge score. Exhaustive documentation dilutes $n$-gram precision, yet the judge rated it 9/10 on completeness and helpfulness. |
| Gemini – doc-manipulation | 10.92 | 7.8 | Second-highest BLEU, second-lowest judge score. Truthfulness rated 5.0/10, the lowest, due to inferred variable names and configuration details not present in the input data. |

**Table 4.8:** Illustrative cases where metric-based scores diverge from actual documentation quality

These cases concretely demonstrate that reference-based metrics not only fail to capture documentation quality but can actively reward outputs of lower qualitative value.

## 4.4 LLM-as-Judge Evaluation Results

This section presents the results of the LLM-as-Judge evaluation, where Claude Opus 4.6 assesses each generated documentation artifact across five quality dimensions: Completeness, Helpfulness, Truthfulness, Clarity, and Professionalism. Each of the 9 documentation artifacts (3 projects $\times$ 3 models) was evaluated independently, with the judge receiving only the generated documentation and the structured project data used to produce it.

### 4.4.1 Overall Scores by Model

Table 4.9 presents the overall quality scores (averaged across all five dimensions) for each model and project, along with the grand average.

| Model | REFrameWork | doc-manip. | DocUnderst. | Average |
|---|---|---|---|---|
| Claude Sonnet 4.6 | 8.6 | 8.8 | 8.8 | 8.73 |
| GPT-5.2 | 8.4 | 8.6 | 8.2 | 8.40 |
| Gemini 2.5 Pro | 7.6 | 7.8 | 8.0 | 7.80 |

**Table 4.9:** Overall LLM-as-Judge scores by model and project

The results rank Claude Sonnet 4.6 first with the highest average score, followed by GPT-5.2 and Gemini 2.5 Pro. All three models produce documentation that the judge rates above 7.5 on a 10-point scale, confirming the high potential of LLM-powered documentation generation across providers.

### 4.4.2 Per-Dimension Analysis

A more granular view of the evaluation results is obtained by examining scores across individual dimensions.

#### Completeness

Completeness measures whether the documentation covers all essential aspects of the UiPath project, including all workflows, dependencies, variables, arguments, error handling mechanisms, and configuration details.

| Model | REFrameWork | doc-manip. | DocUnderst. | Average |
|---|---|---|---|---|
| Claude Sonnet 4.6 | 9.0 | 9.0 | 9.0 | 9.0 |
| GPT-5.2 | 8.0 | 8.0 | 7.0 | 7.7 |
| Gemini 2.5 Pro | 7.0 | 8.0 | 7.0 | 7.3 |

**Table 4.10:** Completeness scores by model and project

Completeness is the dimension where Claude Sonnet 4.6 performs best against the other two models, achieving a perfect 9.0 across all three projects. The judge noted that Sonnet's documentation consistently covered all workflows with individual sections, provided exhaustive argument tables, and documented error handling mechanisms in exceptional detail. GPT-5.2 performed well on the larger projects (8.0 for both REFrameWork and doc-manipulation) but received a lower score for DocumentUnderstanding_Demo1, where the judge identified gaps in variable documentation and missing decision point conditions. Gemini 2.5 Pro showed the most variation, with its weakest performance on the REFrameWork project (7.0), where the judge flagged insufficient NuGet package listings and missing variable documentation as significant gaps.

### Helpfulness

Helpfulness evaluates whether the documentation is useful for its intended audiences: developers maintaining the automation, architects reviewing the design, and stakeholders requiring a high-level understanding.

| Model | REFrameWork | doc-manip. | DocUnderst. | Average |
|---|---|---|---|---|
| Claude Sonnet 4.6 | 9.0 | 9.0 | 9.0 | 9.0 |
| GPT-5.2 | 8.0 | 9.0 | 8.0 | 8.3 |
| Gemini 2.5 Pro | 8.0 | 9.0 | 8.0 | 8.3 |

**Table 4.11:** Helpfulness scores by model and project

Helpfulness scores are uniformly high across all models, suggesting that all three are effective at producing documentation that serves practical purposes. The doc-manipulation project received the highest helpfulness scores from all models (9.0 across the board).

### Truthfulness

Truthfulness assesses the factual accuracy of the generated documentation relative to the actual project data. This is the dimension most directly related to the hallucination problem discussed in Chapter 2.

| Model | REFrameWork | doc-manip. | DocUnderst. | Average |
|---|---|---|---|---|
| Claude Sonnet 4.6 | 7.0 | 8.0 | 8.0 | 7.7 |
| GPT-5.2 | 8.0 | 8.0 | 9.0 | 8.3 |
| Gemini 2.5 Pro | 6.0 | 5.0 | 8.0 | 6.3 |

**Table 4.12:** Truthfulness scores by model and project

GPT-5.2 clearly outperforms Claude Sonnet 4.6 on this dimension, suggesting that it is more conservative in its claims and more faithful to the provided data, using expressions such as "not specified in metadata" rather than inferring or fabricating details.

In the case of Gemini 2.5 Pro's low truthfulness score on doc-manipulation (5.0), the judge identified instances where the model inferred variable names, configuration details, and workflow behaviors that were not present in the structured input data.

The uniformly high scores for DocumentUnderstanding_Demo1 (8.0–9.0) likely reflect the project's simpler structure, which provides fewer opportunities for the models to introduce fabricated information.

### Clarity

Clarity evaluates the writing quality, logical organization, consistent use of terminology, appropriate formatting, and overall readability of the generated documentation.

| **Model** | **REFrameWork** | **doc-manip.** | **DocUnderst.** | **Average** |
|---|---|---|---|---|
| Claude Sonnet 4.6 | 9.0 | 9.0 | 9.0 | 9.0 |
| GPT-5.2 | 9.0 | 9.0 | 9.0 | 9.0 |
| Gemini 2.5 Pro | 9.0 | 9.0 | 9.0 | 9.0 |

**Table 4.13:** Clarity scores by model and project

Clarity is the only dimension where all three models scored a uniform 9.0, indicating that all three frontier models are equally capable of producing well-organized, clearly written technical documentation with consistent formatting.

### Professionalism

Professionalism assesses whether the documentation adheres to professional standards: neutral tone, consistent style, suitability for stakeholder review, and absence of informal language or subjective commentary.

| **Model** | **REFrameWork** | **doc-manip.** | **DocUnderst.** | **Average** |
|---|---|---|---|---|
| Claude Sonnet 4.6 | 9.0 | 9.0 | 9.0 | 9.0 |
| GPT-5.2 | 9.0 | 9.0 | 8.0 | 8.7 |
| Gemini 2.5 Pro | 8.0 | 8.0 | 8.0 | 8.0 |

**Table 4.14:** Professionalism scores by model and project

Although the judge noted that Gemini's documentation occasionally included slightly more informal phrasing and less rigorous formatting consistency compared to the other two models, all outputs were deemed suitable for professional use.

### 4.4.3 Dimension Comparison Across Models

Table 4.15 summarizes the average per-dimension scores across all projects for a direct model-to-model comparison.

| Model | Compl. | Help. | Truth. | Clarity | Prof. | Overall |
|---|---|---|---|---|---|---|
| Claude Sonnet 4.6 | 9.0 | 9.0 | 7.7 | 9.0 | 9.0 | 8.73 |
| GPT-5.2 | 7.7 | 8.3 | 8.3 | 9.0 | 8.7 | 8.40 |
| Gemini 2.5 Pro | 7.3 | 8.3 | 6.3 | 9.0 | 8.0 | 7.80 |

**Table 4.15:** Average per-dimension LLM-as-Judge scores across all three projects

The consolidated view reveals several patterns. Claude Sonnet 4.6 dominates on four of five dimensions but is surpassed by GPT-5.2 on Truthfulness, the dimension most directly related to hallucination avoidance. This creates an interesting trade-off: the most complete and helpful documentation is not necessarily the most factually conservative.

Clarity emerges as a "solved" dimension at the frontier level, with all three models achieving a uniform 9.0. The most differentiating dimensions are Truthfulness (a spread of 2.0 points) and Completeness (a spread of 1.7 points), suggesting that these are the aspects of documentation generation where model choice matters most.

### 4.4.4 Qualitative Observations from the Judge

Beyond numerical scores, the LLM-as-Judge provides structured textual feedback in the form of identified strengths, weaknesses, and reasoning for each dimension. This section summarizes the most significant findings across all runs.

#### Common Strengths

Across all three models, the judge consistently identified the following strengths:

- Comprehensive workflow documentation, with each workflow receiving its own dedicated section including purpose, arguments, and invocation relationships;
- Effective use of Mermaid diagrams to visualize workflow dependency graphs, which the judge praised as particularly valuable for understanding complex inter-workflow relationships;
- Professional tone and consistent Markdown formatting throughout all documentation artifacts;

- Thorough error handling documentation, including TryCatch block inventories, exception type classification, and retry mechanism descriptions;
- Actionable production-readiness recommendations that go beyond mere description to identify gaps and suggest improvements.

### Model-Specific Observations

**Claude Sonnet 4.6** received the highest praise for exhaustive coverage. The judge noted that Sonnet provided detailed argument tables for all workflows and comprehensive exception classifications. However, it occasionally inferred variable names and behaviors not explicitly present in the structured data, penalizing its Truthfulness scores.

**GPT-5.2** was distinguished by its conservative approach. The judge noted that GPT-5.2 more consistently used explicit disclaimers rather than inferring content to fill gaps. This contributed to its leading Truthfulness scores, though it occasionally resulted in less actionable documentation.

**Gemini 2.5 Pro** produced the most concise documentation. While this enhanced focus, it sometimes sacrificed completeness. More critically, Gemini's tendency to infer details not present in the input data resulted in the lowest Truthfulness scores.

### Common Weaknesses

The most frequently identified weaknesses across models included:

- Configuration details presented at a surface level, with actual key-value pairs, default values, and environment-specific settings often missing or only partially enumerated;
- Missing or unresolved references to external workflows: in the doc-manipulation project, all models identified five externally referenced workflows that were absent from the project, but none provided documentation for what those workflows should contain;
- Input/output specifications frequently left vague, with document source paths, output file naming conventions, and exact data flow endpoints not fully specified.

## 4.5 Comparative Analysis

This section briefly summarizes the findings from both evaluation approaches, examining the relationship between metric-based scores and judge-based assessments, and providing an overall comparison of the three models.

### 4.5.1 Model Ranking Comparison

Table 4.16 compares the model rankings produced by each evaluation method. A ranking disagreement between the two methods further supports the argument that metric-based evaluation alone is insufficient.

| Model | Rank (Metrics Avg.) | Rank (Judge Avg.) |
|---|---|---|
| Claude Sonnet 4.6 | 3 | 1 |
| GPT-5.2 | 2 | 2 |
| Gemini 2.5 Pro | 1 | 3 |

**Table 4.16:** Model rankings by average metric-based score vs. average LLM-as-Judge score

The ranking comparison confirms a complete inversion between the two evaluation paradigms: the metric-based ranking is the mirror image of the judge-based ranking. Only GPT-5.2 maintains the same relative position (2nd).

### 4.5.2 Project-Level Analysis

Table 4.17 presents the average scores across all three models for each project, revealing how project characteristics affect documentation quality.

| Project | Compl. | Help. | Truth. | Clarity | Prof. | Overall |
|---|---|---|---|---|---|---|
| REFrameWork | 8.0 | 8.3 | 7.0 | 9.0 | 8.7 | 8.20 |
| doc-manipulation | 8.3 | 9.0 | 7.0 | 9.0 | 8.7 | 8.40 |
| DocUnderstanding | 7.7 | 8.3 | 8.3 | 9.0 | 8.3 | 8.33 |

**Table 4.17:** Average LLM-as-Judge scores by project, across all three models

The project-level analysis reveals an interesting pattern: Truthfulness is inversely correlated with project complexity. This suggests that as project complexity increases, models are more likely to introduce inaccurate claims about inter-workflow relationships, configuration details, or variable behaviors that are not explicitly present in the structured data.

At the same time, doc-manipulation receives the highest Helpfulness scores (9.0 average), despite being the largest project. Its clearly staged pipeline (ingress → extract → process → outgress) provides a natural narrative structure that all three models leveraged effectively, resulting in documentation that the judge found particularly actionable.

### 4.5.3 Cost-Quality Analysis

Table 4.18 provides a practical cost comparison based on the published API pricing of each model.

| Model | Input ($/1M) | Output ($/1M) | Judge Score |
|---|---|---|---|
| Claude Sonnet 4.6 | 3.00 | 15.00 | 8.73 |
| GPT-5.2 | 1.75 | 14.00 | 8.40 |
| Gemini 2.5 Pro | 1.25 | 10.00 | 7.80 |

**Table 4.18:** Cost-quality comparison across models (API rates as of Feb 2026)

While Claude Sonnet 4.6 is the most expensive model and achieves the highest overall score, GPT-5.2 offers a more balanced solution, whereas Gemini 2.5 Pro, despite being the most affordable option, shows a more significant quality gap. Overall, GPT-5.2 may represent the best cost-quality compromise.

### 4.5.4 Project Complexity Impact

To understand how project complexity affects documentation quality, Table 4.19 examines the relationship between the number of workflows and the overall judge scores.

| Project | Workflows | Avg. Overall Score |
|---|---|---|
| DocumentUnderstanding_Demo1 | 1 | 8.33 |
| REFrameWork | 13 | 8.20 |
| doc-manipulation | 27 | 8.40 |

**Table 4.19:** Relationship between project complexity (number of workflows) and average overall judge score

Interestingly, overall quality does not degrade monotonically with project size. The largest project actually achieves the highest average overall score (8.40), while the smallest project falls in the middle (8.33). This suggests that the documentation generator handles projects of varying complexity without significant quality degradation in the range tested. However, as noted in the Truthfulness analysis,

complexity does affect factual accuracy, with simpler projects receiving higher Truthfulness scores. The implication is that larger projects receive documentation that is comprehensive and well-organized but contains more inferred or potentially inaccurate details.

## 4.6 Discussion

### 4.6.1 Key Findings

The experimental results yield several important findings that inform both the practical deployment of LLM-powered documentation systems and the broader understanding of evaluation methodologies for generated text.

First, all three generation models produce documentation that the judge rates with very high scores, confirming the viability of LLM-powered documentation generation for RPA workflows regardless of the chosen provider.

Second, the complete inversion of model rankings between the two approaches demonstrates that BLEU, ROUGE, and word-overlap metrics are not merely insufficient but potentially misleading for evaluating generated technical documentation. This extends beyond the weak positive correlations typically reported for translation tasks in the NLP literature [51, 52], suggesting that the documentation generation task has unique characteristics that fundamentally violate the assumptions underlying reference-based evaluation.

Third, the evaluation dimensions that most differentiate the models are Completeness and Truthfulness, with spreads of 1.7 and 2.0 points respectively. Clarity, by contrast, is a uniformly solved problem at the frontier level. This suggests that future prompt engineering and system design efforts should focus on improving content coverage and factual accuracy rather than writing quality or formatting.

Fourth, there is a meaningful trade-off between Completeness and Truthfulness. Claude Sonnet 4.6 leads on Completeness (9.0 average) but trails GPT-5.2 on Truthfulness (7.7 vs. 8.3). Models that attempt to provide more comprehensive documentation are also more likely to infer or fabricate details not present in the input data.

Fifth, the LLM-as-Judge approach provides rich, actionable feedback beyond numerical scores. The per-dimension breakdown with structured strengths, weaknesses, and reasoning gives developers specific guidance on how to improve documentation quality, making the evaluation itself a useful development tool. The judge identified concrete issues that would be invisible to traditional NLP metrics.

Sixth, project complexity affects documentation quality asymmetrically across dimensions. While overall scores remain stable across different project sizes, Truthfulness degrades as project complexity increases, suggesting that the hallucination risk grows with the number of inter-workflow relationships the model must describe.

### 4.6.2 Threats to Validity

#### Internal Validity

Several factors may affect the internal validity of the experimental results. The most significant is **prompt optimization bias**: the documentation generation prompt was developed and iteratively refined using Claude Sonnet 4.6 as the primary model. While the same prompt was applied identically to all generator models, its overall structure is likely better aligned with Anthropic's instruction-following patterns.

A second concern relates to **same-provider bias in the judge model**. Claude Opus 4.6 and Claude Sonnet 4.6 are both developed by Anthropic, which raises the possibility that the judge may implicitly favor documentation stylistic patterns or structural conventions characteristic of Anthropic models. Although the two models are architecturally distinct and the evaluation uses explicit rubrics to constrain scoring, subtle biases in how the judge weighs certain phrasing choices or organizational patterns cannot be fully ruled out. This limitation is acknowledged as one of the most significant threats to the validity of the qualitative evaluation, and future work should incorporate a cross-provider judge to provide independent validation of the scores.

Finally, **temperature variability** introduces measurement noise. Although the median of three runs was reported for each model-project pair, LLM outputs are inherently non-deterministic at non-zero temperatures. Different runs may produce documentation with different structures, levels of detail, and emphases, which affects both metric-based and judge-based scores.

#### External Validity

The generalizability of these results is constrained by several factors. The three test projects, while diverse in structure and complexity, represent a limited sample of the broader UiPath ecosystem. Results may not generalize to all project types, particularly those using advanced features such as AI Center integration, long-running workflows, or extensive Orchestrator queue management.

A related and significant limitation concerns the nature of the test projects themselves. Ideally, the evaluation would have been conducted on production-grade enterprise automation code, as this would best reflect the real-world conditions under which the documentation pipeline is intended to operate. However, access to such codebases was not possible due to confidentiality and data privacy constraints imposed by the partner organizations. As a result, the test projects used in this evaluation, while representative of common UiPath patterns, may not fully capture the challenges that the system would face in a production enterprise environment.

The reference documentation used for metric-based evaluation, while expert-written, represents one possible "correct" documentation for each project. Different

experts might produce structurally different but equally valid documentation, which would affect metric-based scores in ways that do not reflect actual quality differences. This inherent subjectivity of the ground truth is a well-known limitation of reference-based evaluation in NLP.

Finally, LLM capabilities improve rapidly. The specific model versions tested, Claude Sonnet 4.6, GPT-5.2, and Gemini 2.5 Pro, represent a snapshot of the competitive landscape as of early 2026. Future versions of these models may perform significantly differently on the same tasks.

#### Construct Validity

The LLM-as-Judge approach, while addressing many limitations of metric-based evaluation, introduces its own construct validity concerns. The five evaluation dimensions, Completeness, Helpfulness, Truthfulness, Clarity, and Professionalism, while grounded in the literature [69], may not capture all aspects of documentation quality that human reviewers would consider important. Dimensions such as navigability, visual layout quality, or alignment with organizational documentation standards are not assessed.

More fundamentally, the evaluation does not include a formal human expert assessment to serve as ground truth for the LLM-as-Judge scores. While the produced artifacts were reviewed by human experts, a formal human evaluation framework would provide the most reliable validation of both evaluation approaches and is a natural next step for future work. The absence of human evaluation means that the LLM-as-Judge scores should be interpreted as a proxy for quality rather than as an absolute measure of it.

#### Scope Limitation: Process Design Documents

Among the most critical artifacts in the enterprise RPA domain are *Process Design Documents* (PDDs), which describe the business process to be automated from a functional perspective: the as-is process flow, business rules, decision logic, exception scenarios, input/output specifications, and the expected to-be automated behavior. The current architecture cannot produce PDDs, and early experimentation confirmed that attempting to do so yields mostly hallucinated results, since the information required for a PDD is simply not encoded in UiPath source files. This is not a shortcoming of the implementation but an inherent boundary of any code-analysis-based documentation approach: the system documents *how* the automation operates, whereas a PDD documents *why* it exists and *what* the business process requires, information that originates entirely outside the codebase. In principle, an LLM could assist in drafting PDDs if provided with appropriate business context such as process maps, stakeholder interviews, or existing as-is

documentation, but this would require a fundamentally different input pipeline and prompt design and is therefore beyond the scope of this thesis.

### 4.6.3 Directions for Future Work

The limitations identified in this chapter suggest several concrete directions for future research that could strengthen and extend the evaluation presented here.

The most immediate improvement would be to enhance the LLM-as-Judge evaluation through **cross-provider judge validation**, by deploying a second judge model from a different provider, such as GPT-5.2 or Gemini 2.5 Pro, to independently evaluate the same documentation artifacts using identical rubrics. This cross-provider validation protocol would substantially increase confidence in the qualitative evaluation results and could be further extended to include multiple judges in a panel configuration, where the final score is derived from aggregated or majority-vote assessments.

A second important direction concerns **evaluation on enterprise-grade codebases**. As discussed in the threats to validity, the test projects used in this study, while structurally representative of common UiPath patterns, do not fully capture the complexity and scale of production enterprise automation workflows. Evaluating the system's performance on this class of projects would provide a much stronger indication of its practical readiness for deployment in industrial settings.

Additionally, incorporating **formal human expert evaluation** alongside the automated metrics and LLM-as-Judge assessments would provide the most reliable ground truth for documentation quality. A structured human evaluation protocol, where experienced RPA developers rate the generated documentation on the same dimensions used by the automated judge, would enable direct calibration of the LLM-as-Judge methodology and identify any systematic discrepancies between machine and human quality assessments.

# Chapter 5

# Conclusions

The purpose of this thesis was to explore whether the use of Large Language Models could efficiently and effectively automate the production of technical documentation artifacts in the field of Robotic Process Automation developed through UiPath. Such a task is considered of vital importance for the lifecycle of software, given that it is very often neglected and deprioritized. Although the goal of automating documentation is not new, ranging from template-driven tools to information-retrieval methods and, more recently, transformer-based generation, its actual application to the RPA ecosystem still remains uncharted territory.

To address this research question, the UiPath Documentation Generator was built in collaboration with Poseidon-SB, a consulting company specializing in the effective and rapid digitization of business processes through automation and other modern technologies. The system was designed as an end-to-end pipeline that follows the principle of separation of concerns: first, the Analysis Layer parses XAML and JSON files into much more compact representations, ensuring that all essential details are captured for each workflow. This information is then passed to the Generation Layer, which was engineered with specific prompt techniques to produce complete and comprehensive Markdown documentation. This multi-stage design not only ensures that information can be promptly passed to the LLM while maintaining full context at the project level, but also drastically reduces the risk of hallucination typical of LLMs. These findings are confirmed by the high truthfulness scores observed across all tested generators.

The comparative evaluation across Claude Sonnet 4.6, GPT-5.2, and Gemini 2.5 Pro demonstrated that all three frontier models are capable of producing documentation that is structurally coherent, technically grounded, and aligned with professional standards, and in some cases even superior to human-written references, showing the ability to capture more subtle yet equally important details that are sometimes underrepresented. This confirms the overall viability of the approach, regardless of the chosen provider. Equally important is what emerged

from the Evaluation Layer: quantitative metrics such as BLEU and ROUGE correlate poorly with a qualitative approach such as LLM-as-Judge, which instead provides richer and more actionable feedback through per-dimension scoring and structured reasoning. This establishes the judge-based approach as a valuable complement to human expert review, though not yet a complete replacement.

These results confirm our first hypothesis. However, although encouraging, they come with acknowledged limitations. Confidentiality constraints not only prevented testing on production-grade enterprise codebases, but the sample size of projects was also limited. Moreover, the fact that the same LLM provider was used for both the generation phases and the evaluation could raise concerns about internal bias in the judge's assessments. Addressing these gaps through cross-provider judge validation, formal human evaluation, and enterprise-scale experiments represents the most immediate path for future work.

Taken together, the findings strongly suggest that LLM-driven documentation is approaching a level of maturity where it can significantly reduce the documentation burden in organizations adopting RPA at scale. As models continue to improve in reasoning and context handling, and as evaluation methodologies become more robust, the vision of documentation as a continuously generated, always up-to-date artifact appears increasingly within reach.

# Declaration on the Use of AI Tools

In accordance with the academic integrity policies of Politecnico di Torino, the author declares that AI tools were used during the preparation of this thesis for proofreading purposes, including vocabulary refinement, grammar checking, and improvement of sentence structures. All intellectual content, research design, system implementation, experimental analysis, and scientific conclusions presented in this work are entirely the result of the author's own effort.

# Bibliography

[1] A. M. Turing. «Computing Machinery and Intelligence». English. In: *Mind*. New Series 59.236 (1950), pp. 433–460. ISSN: 00264423. URL: http://www.jstor.org/stable/2251299 (cit. on p. 9).

[2] John McCarthy, Marvin Minsky, Nathaniel Rochester, and Claude E. Shannon. *A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence*. Proposal submitted August 31, 1955. Aug. 1955. URL: https://dl.acm.org/doi/10.1609/aimag.v27i4.1904 (cit. on p. 9).

[3] Frank Rosenblatt. «The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain». In: *Psychological Review* 65.6 (1958), pp. 386–408. DOI: 10.1037/h0042519. URL: https://doi.org/10.1037/h0042519 (cit. on p. 9).

[4] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. USA: Prentice Hall Press, 2009. ISBN: 0136042597. URL: http://dl.acm.org/citation.cfm?id=1671238 (cit. on p. 9).

[5] Feng-hsiung Hsu. *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton, NJ: Princeton University Press, 2002. ISBN: 978-0691090658. URL: https://press.princeton.edu/books/paperback/9780691118185/behind-deep-blue (cit. on p. 9).

[6] John McCarthy. *What is Artificial Intelligence?* 2007. URL: http://www-formal.stanford.edu/jmc/whatisai.pdf (visited on 12/13/2025) (cit. on p. 9).

[7] David Ferrucci et al. «Building Watson: An Overview of the DeepQA Project». In: *AI Magazine* 31.3 (2010), pp. 59–79. DOI: 10.1609/aimag.v31i3.2303. URL: https://doi.org/10.1609/aimag.v31i3.2303 (cit. on p. 9).

[8] David Silver et al. «Mastering the game of Go with deep neural networks and tree search». In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. DOI: 10.1038/nature16961. URL: https://doi.org/10.1038/nature16961 (cit. on p. 9).

[9] Leon E. Dostert. «The Georgetown-IBM Experiment». In: *Machine Translation of Languages*. Ed. by William N. Locke and A. Donald Booth. Cambridge, MA: MIT Press, 1955, pp. 124–135 (cit. on p. 11).

[10] Joseph Weizenbaum. «ELIZA—a computer program for the study of natural language communication between man and machine». In: *Communications of the ACM* 9.1 (1966), pp. 36–45. DOI: 10.1145/365153.365168. URL: https://doi.org/10.1145/365153.365168 (cit. on p. 11).

[11] Roger C. Schank, Neil M. Goldman, Charles J. Rieger III, and Christopher K. Riesbeck. «Inference and Paraphrase by Computer». In: *Journal of the ACM* 22.3 (July 1975), pp. 309–328. DOI: 10.1145/321892.321893. URL: https://doi.org/10.1145/321892.321893 (cit. on p. 11).

[12] Gordon E. Moore. «Cramming more components onto integrated circuits». In: *Proceedings of the IEEE* 86.1 (1998). Reprint of the 1965 paper, pp. 82–85. DOI: 10.1109/JPROC.1998.658762. URL: https://doi.org/10.1109/JPROC.1998.658762 (cit. on p. 12).

[13] Peter F. Brown, John Cocke, Stephen A. Della Pietra, Vincent J. Della Pietra, Fredrick Jelinek, John D. Lafferty, Robert L. Mercer, and Paul S. Roossin. «A Statistical Approach to Machine Translation». In: *Computational Linguistics* 16.2 (June 1990), pp. 79–85. URL: https://aclanthology.org/J90-2002/ (cit. on p. 12).

[14] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. «A Neural Probabilistic Language Model». In: *Journal of Machine Learning Research* 3 (2003), pp. 1137–1155. URL: https://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf (cit. on p. 12).

[15] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. «Efficient Estimation of Word Representations in Vector Space». In: *Proceedings of Workshop at ICLR*. 2013. URL: https://arxiv.org/abs/1301.3781 (cit. on pp. 12, 16).

[16] Andrey A. Markov. «Rasprostranenie zakona bol'shih chisel na velichiny, zavisyaschie drug ot druga [Extension of the limit theorems of probability theory to a sum of variables connected in a chain]». In: *Izvestiya Fiziko-matematicheskogo obschestva pri Kazanskom universitete* 15.4 (1906), pp. 135–156 (cit. on p. 14).

[17] Harold Cohen. «What is an Image?» In: *Proceedings of the 6th International Joint Conference on Artificial Intelligence (IJCAI)*. Vol. 2. Morgan Kaufmann Publishers Inc. 1979, pp. 1028–1057 (cit. on p. 14).

[18] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML]. URL: https://arxiv.org/abs/1406.2661 (cit. on p. 14).

[19] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. «Improving Language Understanding by Generative Pre-Training». In: *OpenAI* (2018). URL: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf (cit. on p. 14).

[20] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. *Zero-Shot Text-to-Image Generation*. 2021. arXiv: 2102.12092 [cs.CV]. URL: https://arxiv.org/abs/2102.12092 (cit. on p. 14).

[21] OpenAI. *Introducing ChatGPT*. Blog Post. Nov. 2022. URL: https://openai.com/blog/chatgpt (cit. on p. 14).

[22] Gemini Team and Google. *Gemini: A Family of Highly Capable Multimodal Models*. Tech. rep. Google DeepMind, 2023. URL: https://arxiv.org/abs/2312.11805 (cit. on p. 14).

[23] Anthropic. *The Claude 3 Model Family: Opus, Sonnet, Haiku*. Tech. rep. 2024. URL: https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf (cit. on pp. 14, 42).

[24] Gartner. *Gartner Predicts 30% of Generative AI Projects Will Be Abandoned After Proof of Concept By End of 2025*. July 2024. URL: https://www.gartner.com/en/newsroom/press-releases/2024-07-29-gartner-predicts-30-percent-of-generative-ai-projects-will-be-abandoned-after-proof-of-concept-by-end-of-2025 (cit. on p. 15).

[25] Peter F. Brown, Stephen A. Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. «The Mathematics of Statistical Machine Translation: Parameter Estimation». In: *Computational Linguistics* 19.2 (1993). Ed. by Julia Hirschberg, pp. 263–311. URL: https://aclanthology.org/J93-2003/ (cit. on p. 16).

[26] Joshua T. Goodman. *A Bit of Progress in Language Modeling*. Tech. rep. MSR-TR-2001-72. Microsoft Research, 2001. URL: https://www.microsoft.com/en-us/research/publication/a-bit-of-progress-in-language-modeling/ (cit. on p. 16).

[27] Jeffrey Pennington, Richard Socher, and Christopher D Manning. «GloVe: Global Vectors for Word Representation». In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics. Doha, Qatar, 2014, pp. 1532–1543. URL: https://aclanthology.org/D14-1162 (cit. on p. 16).

[28] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. «Neural Machine Translation by Jointly Learning to Align and Translate». In: *arXiv preprint arXiv:1409.0473* (2014). Published in ICLR 2015. URL: https://arxiv.org/abs/1409.0473 (cit. on pp. 16, 17).

[29] Yonghui Wu et al. «Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation». In: *arXiv preprint arXiv:1609.08144* (2016). URL: https://arxiv.org/abs/1609.08144 (cit. on p. 16).

[30] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. «Attention is All You Need». In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017, pp. 5998–6008. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf (cit. on pp. 16, 22).

[31] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. «Language Models are Unsupervised Multitask Learners». In: *OpenAI blog* 1.8 (2019), p. 9 (cit. on p. 16).

[32] OpenAI. *GPT-4 Technical Report*. Tech. rep. OpenAI, 2023. URL: https://arxiv.org/abs/2303.08774 (cit. on p. 16).

[33] Hugo Touvron et al. «LLaMA: Open and Efficient Foundation Language Models». In: *arXiv preprint arXiv:2302.13971* (2023). URL: https://arxiv.org/abs/2302.13971 (cit. on p. 16).

[34] Jeffrey L Elman. «Finding Structure in Time». In: *Cognitive Science* 14.2 (1990), pp. 179–211. DOI: 10.1207/s15516709cog1402_1. URL: https://doi.org/10.1207/s15516709cog1402_1 (cit. on p. 17).

[35] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. «Recurrent Neural Network Based Language Model». In: *Interspeech* 2 (2010), pp. 1045–1048. URL: https://www.isca-archive.org/interspeech_2010/mikolov10_interspeech.pdf (cit. on p. 17).

[36] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. «Learning Long-Term Dependencies with Gradient Descent is Difficult». In: *IEEE Transactions on Neural Networks* 5.2 (1994), pp. 157–166. DOI: 10.1109/72.279181. URL: https://doi.org/10.1109/72.279181 (cit. on p. 17).

[37] Sepp Hochreiter and Jürgen Schmidhuber. «Long Short-Term Memory». In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735 (cit. on p. 17).

[38] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. «Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation». In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1724–1734. DOI: 10.3115/v1/D14-1179. URL: https://aclanthology.org/D14-1179 (cit. on p. 17).

[39] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. «Layer Normalization». In: *arXiv preprint arXiv:1607.06450* (2016). URL: https://arxiv.org/abs/1607.06450 (cit. on p. 18).

[40] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. «BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding». In: *arXiv preprint arXiv:1810.04805* (2018). Published in NAACL 2019. DOI: 10.18653/v1/N19-1423. URL: https://arxiv.org/abs/1810.04805 (cit. on p. 19).

[41] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. «Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer». In: *CoRR* abs/1910.10683 (2019). arXiv: 1910.10683. URL: http://arxiv.org/abs/1910.10683 (cit. on p. 19).

[42] Rico Sennrich, Barry Haddow, and Alexandra Birch. «Neural Machine Translation of Rare Words with Subword Units». In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*. 2016, pp. 1715–1725. URL: https://aclanthology.org/P16-1162/ (cit. on p. 21).

[43] Edward J. Hu, Yelong Shen, Phil Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. «LoRA: Low-Rank Adaptation of Large Language Models». In: *Proceedings of the 10th International Conference on Learning Representations (ICLR)*. 2022. URL: https://openreview.net/forum?id=nZeVKeeFYf9 (cit. on p. 24).

[44] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Blake Morrill, Google Corrado, Dale Schuurmans, Danielle Michrowski, and Sylvain Gelly. «Parameter-Efficient Transfer Learning for NLP». In: *Proceedings of the 36th International Conference on Machine Learning (ICML)*. 2019, pp. 2790–2799. URL: http://proceedings.mlr.press/v97/houlsby19a.html (cit. on p. 25).

[45] Xiang Lisa Li and Percy Liang. «Prefix-Tuning: Optimizing Continuous Prompts for Generation». In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics (ACL)*. 2021, pp. 4582–4597. URL: https://aclanthology.org/2021.acl-long.353/ (cit. on p. 25).

[46] Tom Brown et al. «Language Models are Few-Shot Learners». In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 1877–1901. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf (cit. on p. 25).

[47] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. «Chain-of-Thought Prompting Elicits Reasoning in Large Language Models». In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 24824–24837. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf (cit. on p. 25).

[48] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. «Self-Consistency Improves Chain of Thought Reasoning in Language Models». In: *arXiv preprint arXiv:2203.11171* (2022). Published in ICLR 2023. URL: https://arxiv.org/abs/2203.11171 (cit. on p. 25).

[49] Takeshi Kojima, Shixiang Shane Gu, Machel Reinhart, Yutaka Matsuo, and Yusuke Iwasawa. «Large Language Models are Zero-Shot Reasoners». In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 35. 2022, pp. 22199–22213. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/8bb0d291acd4acf06ef112099c16f326-Paper-Conference.pdf (cit. on p. 25).

[50] Patrick Lewis et al. «Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks». In: *Advances in Neural Information Processing Systems*. Vol. 33. 2020, pp. 9459–9474. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf (cit. on p. 25).

[51] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. «BLEU: a Method for Automatic Evaluation of Machine Translation». In: *Proceedings of ACL* (2002), pp. 311–318. URL: https://aclanthology.org/P02-1040.pdf (cit. on pp. 26, 43, 55, 70, 78).

[52] Chin-Yew Lin. «ROUGE: A Package for Automatic Evaluation of Summaries». In: *Proc. of ACL Workshop on Text Summarization* (2004), pp. 74–81. URL: https://aclanthology.org/W04-1013.pdf (cit. on pp. 27, 43, 55, 70, 78).

[53] Ziwei Ji et al. «Survey of Hallucination in Natural Language Generation». In: *ACM Computing Surveys* 55.12 (Mar. 2023), pp. 1–38. ISSN: 1557-7341. DOI: 10.1145/3571730. URL: http://dx.doi.org/10.1145/3571730 (cit. on pp. 27, 41, 53).

[54] Sarthak Jain and Byron C. Wallace. «Attention is not Explanation». In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*. 2019, pp. 3543–3556. URL: https://aclanthology.org/N19-1357/ (cit. on p. 27).

[55] Nicholas Carlini et al. «Extracting Training Data from Large Language Models». In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 2633–2650. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/carlini-extracting (cit. on p. 28).

[56] Toran Bruce Richards. *AutoGPT: An Autonomous GPT-4 Experiment*. 2023. URL: https://github.com/Significant-Gravitas/AutoGPT (cit. on p. 30).

[57] LangChain Inc. *LangGraph: Composable Multi-Agent Workflow Framework for LLMs*. 2024. URL: https://github.com/langchain-ai/langgraph (cit. on p. 30).

[58] MetaGPT Contributors. *MetaGPT: The Multi-Agent Framework*. https://github.com/geekan/MetaGPT. 2023 (cit. on p. 30).

[59] CrewAI Contributors. *CrewAI: Framework for Autonomous Agent Collaboration*. https://github.com/joaomdmoura/crewAI. 2024 (cit. on p. 30).

[60] AutoGen Contributors. *AutoGen: Enabling Next-Gen Large Language Model Applications*. https://github.com/microsoft/autogen. 2023 (cit. on p. 30).

[61] LangChain Contributors. *LangChain: Building Applications with LLMs through Composability*. https://github.com/langchain-ai/langchain. 2023 (cit. on p. 30).

[62] Sun Microsystems. *Javadoc Tool*. 1995. URL: https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html (cit. on p. 32).

[63] Dimitri van Heesch. *Doxygen: Documentation Generator*. 1997. URL: https://www.doxygen.nl/ (cit. on p. 32).

[64] Junaed Younus Khan and Gias Uddin. *Automatic Code Documentation Generation Using GPT-3*. 2022. arXiv: 2209.02235 [cs.SE]. URL: https://arxiv.org/abs/2209.02235 (cit. on pp. 34, 35).

[65] Edmund Wong, Taiyue Liu, and Lin Tan. «CloCom: Mining existing source code for automatic comment generation». In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 2015, pp. 380–389. DOI: 10.1109/SANER.2015.7081848 (cit. on p. 34).

[66] Edmund Wong, Jinqiu Yang, and Lin Tan. «AutoComment: Mining question and answer sites for automatic comment generation». In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2013, pp. 562–567. DOI: 10.1109/ASE.2013.6693113 (cit. on p. 34).

[67] Shubhang Shekhar Dvivedi, Vyshnav Vijay, Sai Leela Rahul Pujari, Shoumik Lodh, and Dhruv Kumar. *A Comparative Analysis of Large Language Models for Code Documentation Generation*. 2024. arXiv: 2312.10349 [cs.SE]. URL: https://arxiv.org/abs/2312.10349 (cit. on p. 35).

[68] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. «Comparing Code Explanations Created by Students and Large Language Models». In: *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. ITiCSE 2023. ACM, June 2023, pp. 124–130. DOI: 10.1145/3587102.3588785. URL: http://dx.doi.org/10.1145/3587102.3588785 (cit. on p. 35).

[69] Henry Tang and Sarah Nadi. «Evaluating Software Documentation Quality». In: *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 2023, pp. 67–78. DOI: 10.1109/MSR59073.2023.00023 (cit. on pp. 35, 56, 70, 80).

[70] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. «A Human Study of Comprehension and Code Summarization». In: *2020 IEEE/ACM 28th International Conference on Program Comprehension (ICPC)*. 2020, pp. 01–12. DOI: 10.1145/3387904.3389258 (cit. on p. 35).

[71] Xing Hu, Qiuyuan Chen, Haoye Wang, Xin Xia, David Lo, and Thomas Zimmermann. «Correlating Automated and Human Evaluation of Code Documentation Generation Quality». In: *ACM Transactions on Software Engineering and Methodology* 31 (May 2022). DOI: 10.1145/3502853 (cit. on p. 35).

[72] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. «On the evaluation of neural code summarization». In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. ACM, May 2022, pp. 1597–1608. DOI: 10.1145/3510003.3510060. URL: http://dx.doi.org/10.1145/3510003.3510060 (cit. on p. 35).

[73] Antonio Valerio Miceli Barone and Rico Sennrich. «A Parallel Corpus of Python Functions and Documentation Strings for Automated Code Documentation and Code Generation». In: *Proceedings of the 8th International Joint Conference on Natural Language Processing (IJCNLP-2017, Short Papers)*. 2017. URL: https://aclanthology.org/I17-2053 (cit. on p. 36).

[74] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. «CodeSearchNet Challenge: Evaluating the State of Semantic Code Search». In: *CoRR* abs/1909.09436 (2019). URL: https://arxiv.org/abs/1909.09436 (cit. on p. 36).

[75] Shuai Lu et al. «CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation». In: *CoRR* abs/2102.04664 (2021). URL: https://arxiv.org/abs/2102.04664 (cit. on p. 36).

[76] Stefan Behnel. *lxml: XML and HTML Processing with Python.* High-performance XML/HTML parser based on libxml2. 2024. URL: https://lxml.de/ (cit. on p. 42).

[77] Giteous and Contributors. *GitPython: A Python Library for Git Repositories.* 2024. URL: https://github.com/gitpython-developers/GitPython (cit. on p. 42).

[78] OpenRouter. *OpenRouter: A Unified Interface for LLMs.* Accessed: 2026. 2024. URL: https://openrouter.ai/docs (cit. on p. 42).

[79] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit.* O'Reilly Media, 2009. ISBN: 978-0596516499. URL: https://www.nltk.org/book/ (cit. on p. 43).

[80] Snowflake Inc. *Streamlit: A Faster Way to Build and Share Data Apps.* Open-source Python framework for interactive web applications. 2024. URL: https://streamlit.io/ (cit. on p. 43).

[81] Knut Sveidqvist and Contributors. *Mermaid: Generation of Diagrams and Flowcharts from Text.* 2024. URL: https://mermaid.js.org/ (cit. on p. 53).

[82] Koki Wataoka, Tsubasa Nakamura, and Ryuji Ogawa. «Self-Preference Bias in LLM-as-a-Judge». In: *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP).* 2024. URL: https://arxiv.org/abs/2410.21819 (cit. on p. 63).

[83] Lianmin Zheng et al. «Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena». In: *Advances in Neural Information Processing Systems 36 (NeurIPS 2023).* 2023. URL: https://arxiv.org/abs/2306.05685 (cit. on p. 63).

[84] Jiayi Ye et al. «Justice or Prejudice? Quantifying Biases in LLM-as-a-Judge». In: *arXiv preprint arXiv:2410.02736* (2024). URL: https://arxiv.org/abs/2410.02736 (cit. on p. 63).