Individual Thesis

AUTOMATED CONSTRUCTION OF CUSTOM AI AGENTS THROUGH A STRUCTURED MULTI-AGENT WORKFLOW

Stefanos Panteli

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

May 2026

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

Automated Construction of Custom AI Agents Through a Structured Multi-Agent Workflow

Stefanos Panteli

Supervising Professor
Dr. Yiannis Dimopoulos

The Individual Thesis was submitted in partial fulfillment of the requirements for the degree of Computer Science of the Department of Computer Science of the University of Cyprus.

May 2026

I declare that this dissertation and any accompanying software are my own original work, conducted in full compliance with the University of Cyprus Code of Conduct for Research (https://www.ucy.ac.cy/legislation/kodikas-deontologias-kai-politikes/), and with full accountability on my part for the accuracy, integrity, and validity of all content.

Acknowledgements

I would like to express my appreciation to my supervising professor, Dr. Yiannis Dimopoulos, for the trust he placed in me and for the opportunity to undertake this thesis topic. His guidance and valuable feedback throughout the development of this thesis helped shape the direction of the project and contributed to the development of both my technical and academic skills.

Abstract

Custom AI agents are increasingly used to support specialised tasks in real-life applications. However, building such agents remains difficult for developers because it involves more than just writing prompts or connecting an LLM to tools. A developer must interpret the user's requirements, design the workflow, define prompts, manage memory, write runnable code, validate outputs, and organise the wider package that supports the agent. This process is time-consuming and can become fragile when requirements are unclear, prompts are inconsistent, tools are misused, or models drift from their intended role. For this reason, a structured system is needed to reduce the manual effort, to guide construction, and make the produced agent easier to inspect, reproduce, and improve.

This thesis presents a structured multi-agent system for the automated construction of custom AI agents from natural-language user requests. The proposed system transforms an initial request into a reusable agent package through a staged workflow. The process begins by clarifying and refining the initial request. It then converts the refined request into a graph-based workflow and creates an initial code scaffold based on the generated graph. The system is designed as a supervised construction tool rather than a fully autonomous replacement of developers. Its purpose is to organise and speed up the agent-building process.

The implementation uses LangGraph-style orchestration, role-specific agents, structured outputs, prompt contracts, controlled tool access, and traceable intermediate artefacts. Each stage has a defined responsibility and passes a concrete artefact to the next stage, making the construction process easier to inspect, debug, and revise.

The result is a system that demonstrates how LLM collaboration can be used to construct runnable custom-agent packages with less manual effort. The work also identifies practical limitations, including dependence on model quality, incomplete guarantees for generated code, runtime cost, and the continued need for human supervision.

Keywords: AI Agents, Agentic AI, Multi-Agent Systems, LangGraph, Code Generation, Custom Agent Construction, Workflow, Tools, Prompt, Memory GitHub

Contents

CHAPTER 1. INTRODUCTION 1

1.1 Background and Motivation 1
1.2 Problem Statement 2
1.3 Scope 3
1.4 Thesis Organisation 4

CHAPTER 2. BACKGROUND AND FOUNDATIONS 5

2.1 Introduction 5
2.2 AI Agents and Agentic AI 6
2.3 Large Language Model Basics 11
2.4 Prompt Engineering, In-Context Learning, and RAG 13
2.5 Structured Outputs and Deterministic Validation 17
2.6 Tool Calling and Action Execution 19
2.7 Workflow Orchestration and Graphs 21
2.8 Memory and State Management 24
2.9 Multi-Agent Collaboration and Coordination Patterns 27
2.10 System-Level Reliability, Governance, and Ethical Considerations 29
2.11 LangChain Ecosystem Overview 31
2.12 Summary 34

CHAPTER 3. RELATED WORK 35

3.1 Introduction 35
3.2 Agents for Code Generation 35
3.3 Automatic Agent Builders and Agent-Construction Systems 38
3.4 Gap and Positioning of This Thesis 43

CHAPTER 4. SYSTEM SCOPE AND REQUIREMENTS 45

4.1 Introduction 45
4.2 Target Users and Use Cases 45
4.3 System Inputs, Outputs, and Produced Artefacts 47
4.4 Functional and Non-Functional Requirements 48
4.5 Safety Requirements 50
4.6 Summary 52

CHAPTER 5. DESIGN AND METHODOLOGY 53

5.1 Introduction 53
5.2 System Design 54
5.3 Agent Role Decomposition 57
5.4 Construction Methodology 58
5.5 Interfaces and Data Contracts 61
5.6 Prompt Contracts 62
5.7 Tool Access and Execution Control 64
5.8 Validation, Review, and Failure Handling 65
5.9 Artefact Management and Traceability 67
5.10 Summary 69

CHAPTER 6. IMPLEMENTATION 71

6.1 Introduction 71
6.2 Repository Structure and Build Environment 72
6.3 Main Workflow Implementation 76
6.4 Agent Implementation 77
6.5 Generated Package Structure 100
6.6 Summary 102

CHAPTER 7. END-TO-END WALKTHROUGHS AND SYSTEM DISCUSSION 104

7.1 Introduction 104
7.2 Evaluation Scope 105
7.3 Evaluation Criteria 106
7.4 End-to-End Walkthrough 108
7.5 Runtime and Cost Discussion 115
7.6 Results Discussion 116
7.7 Summary 117

CHAPTER 8. CONCLUSIONS AND FUTURE WORK 118

8.1 Introduction 118
8.2 Summary of the Thesis 119
8.3 Contributions 120
8.4 Limitations 121
8.5 Future Work 122
8.6 Conclusion 123

Chapter 1

Introduction

1.1 Background and Motivation **1**
1.2 Problem Statement **2**
1.3 Scope **3**
1.4 Thesis Organisation **4**

1.1 Background and Motivation

AI agents are becoming increasingly prominent across academic research and industrial applications. Recent applications of AI agents span areas such as agriculture [40, 50], manufacturing [28], healthcare [23], education [9], finance [27] and research automation [56]. These examples show that AI agents are designed around specific tasks and constraints, even in high-risk domains. Additionally, AI agents are gaining popularity for automating simple repetitive everyday tasks, such as email filtering and trip planning.

The value of such systems often depends on their level of customisation. A useful agent should be aligned with the task it is expected to perform, the information it must process, and the actions it may take. A generic language model can provide a starting point, but the final behaviour of a custom agent depends on the surrounding software structure that controls how the model is used.

Building this surrounding structure is a demanding software engineering task. The developer must understand the user's request, identify missing requirements, design prompts, select suitable libraries, implement source code, and test whether the result

behaves as intended. Prompt engineering is also part of this process, since prompts define part of the interaction contract [38]. Reliable agent behaviour depends on architectural choices such as component separation, explicit interfaces, validation, runtime limits, and logging [41].

The construction process becomes more complicated when the requested agent includes several interconnected behaviours. The system may need to manage external information, act, and produce structured outputs. Each added behaviour introduces further design, implementation, and testing work. As a result, the process can become time-consuming, particularly when the developer must repeatedly revise prompts or fix code to reach the desired result.

These challenges motivate the need for a structured and supervised construction process that can reduce the manual effort required to build custom agents. In such a process, part of the construction work can be divided among specialised construction agents. This thesis investigates how a multi-agent builder system can transform a natural-language request into a reusable and inspectable custom-agent package, while keeping the construction process bounded and human-supervised.

1.2 Problem Statement

The problem addressed in this thesis is the manual, repetitive effort required to construct custom AI agents from high-level user requests. Although users may describe the kind of agent they need in natural language, turning that request into an executable agent requires several design and implementation steps. A developer must interpret the request, decide what the agent should do, design the workflow, prepare prompts, implement source code, organise files, and test the resulting behaviour. This process can be slow, inconsistent, and difficult to reproduce. The proposed system addresses this problem by automating part of the custom-agent construction process. Its input is a natural-language user request for a custom AI agent, and its intended output is a runnable custom-agent package.

A key difficulty is that the input request is often much less detailed than the implementation that must be produced. A user may know the goal of the desired agent

but may not specify the software's design and implementation needs. These details must be clarified or inferred during construction. Consequently, the problem becomes more complex than direct code generation, because the system must move from an informal task description to a structured software artefact.

This thesis addresses the problem by treating custom-agent construction as a staged, supervised multi-agent process. The proposed system divides the construction task among specialised agents that refine the request, generate and implement code, produce prompts, and create the supporting project files. The aim is not to replace human developers, but to reduce the repetitive manual effort required to move from an informal request to a runnable and inspectable custom-agent package.

1.3 Scope

This thesis focuses on the supervised construction of bounded, task-specific custom AI agents from natural-language user requests. The proposed system comprises multiple specialised agents, each responsible for a specific part of the construction process. The intended output of the proposed system is a runnable custom-agent package containing generated source code, prompts, workflow logic, and supporting files.

The aim of the work is to reduce the manual effort required to translate a high-level request into an executable custom-agent package. To achieve this, the thesis focuses on a staged construction process that refines the request, generates the required implementation artefacts, and keeps the produced package inspectable.

The system does not claim to produce production-ready agents, guarantee correctness, or replace the developer. The generated package should be treated as an inspectable software artefact that may still require human review, testing, and adjustment before use.

The work assumes that the user can provide clarifications when needed and review important outputs during construction. Therefore, the goal is not full autonomy, but a bounded process that reduces the manual effort involved in moving from a high-level request to a runnable custom-agent package.

1.4 Thesis Organisation

The remainder of this thesis is organised as follows. *Chapter 2* presents the background and foundations required to understand the proposed system, including AI agents, large language models, prompt engineering, structured outputs, tool calling, workflow orchestration, memory, multi-agent collaboration, reliability, and the LangChain ecosystem. *Chapter 3* then reviews related work on code-generation agents and agent-construction systems and positions this thesis in relation to existing approaches. *Chapter 4* defines the system scope, target users, inputs, outputs, functional and non-functional requirements, and safety requirements. Following, *Chapter 5* presents the design and methodology of the proposed multi-agent builder system, including its architecture, agent roles, interfaces, prompt contracts, tool access, validation logic, and artefact management. *Chapter 6* describes the implementation of the system, including the repository structure, main workflow, implemented agents, and generated package structure, while *Chapter 7* discusses the system's execution, generated outputs, runtime behaviour, cost, and limitations. Finally, *Chapter 8* concludes the thesis and outlines possible future work.

Chapter 2

Background and Foundations

2.1 Introduction **5**
2.2 AI Agents and Agentic AI **6**
2.3 Large Language Model Basics **11**
2.4 Prompt Engineering, In-Context Learning, and RAG **13**
2.5 Structured Outputs and Deterministic Validation **17**
2.6 Tool Calling and Action Execution **19**
2.7 Workflow Orchestration and Graphs **21**
2.8 Memory and State Management **24**
2.9 Multi-Agent Collaboration and Coordination Patterns **27**
2.10 System-Level Reliability, Governance, and Ethical Considerations **29**
2.11 LangChain Ecosystem Overview **31**
2.12 Summary **34**

2.1 Introduction

The chapter is organised around the main concepts needed to explain the proposed system. *Subchapter 2.2 AI Agents and Agentic AI* defines how the two terms are defined in the literature and distinguishes their differences. *Subchapter 2.3 LLM Basics* introduces the model properties that most affect system design. *Subchapter 2.4, Prompt Engineering, In-Context Learning, and RAG*, explains how prompts define the expected behaviour of the agent and also examines how examples and retrieval-based prompting shape model outputs. *Subchapter 2.5 Structured Outputs and Deterministic Validation* then discusses the use of structuring and validation rules to constrain model responses, and S*ubchapter*

2.6 Tool Calling and Action Execution examines how tools enable agents to interact with their environment. *Subchapter 2.7 Workflow Orchestration and Graphs* presents graph-based agent representations, while S*ubchapter 2.8 Memory and State Management* focuses on how past information and knowledge is retained. *Subchapter* 2.*9 Multi-Agent Collaboration and Coordination Patterns* continues to examine how multi-agent systems coordinate and exchange information. Across these subchapters, reliability concerns are discussed for each component, hence, *Subchapter 2.10 System-Level Reliability, Governance, and Ethical Considerations* generalises these individual issues into wider agentic problems, while also addressing ethical concerns that arise from agentic systems. Finally, S*ubchapter 2.11, LangChain Ecosystem Overview*, briefly presents the ecosystem used to develop the proposed system, followed by a summary of the main foundations developed throughout the chapter in *Subchapter 2.12 Summary*.

2.2 AI Agents and Agentic AI

This subchapter examines the concepts of AI agents and agentic AI, which form an important part of the conceptual background of this thesis. Its purpose is to clarify how recent literature defines these terms and identify the main points of overlap. The discussion starts with AI agents, including a brief overview of their historical developments. Then, agentic AI is examined as a broader and more recent paradigm. Finally, the subchapter compares the two concepts and provides illustrative examples for easier differentiation between the two terms.

2.2.1 AI Agents

The concept of the agent was originally developed around the idea of an autonomous entity situated in an environment, able to perceive and act toward designated goals [11, 45, 47]. In this earlier literature, agents were already associated with core ideas that remain central today, including perception, action, goal-directed behaviour, and reasoning, even in changing environments [45, 47]. Recent work reinterprets the agent as a language-model-powered system that can reason over natural-language context, invoke tools, maintain memory, and operate more flexibly in digital environments, while still sustaining the core older view of an autonomous agent as an entity that perceives, decides, and acts in pursuit of goals [4, 8, 38, 50].

The literature defines the term AI Agent from various perspectives. Operational definitions describe an agent in terms of what the agent is expected to achieve, emphasising bounded task execution, explicit goals, and practical autonomy within a defined scope [2]. In contrast, capability-based definitions focus on what the agent is able to do, highlighting abilities such as planning, decision making, memory, tool use, and action [10, 38, 73]. On the other hand, architectural definitions shift attention to how agents are internally organised, describing them through interacting subsystems such as perception, reasoning, memory, and execution [8]. Finally, some use a combination of the above definitions to describe AI agents [4, 8, 22, 37, 44, 50]. These views frame the term from different levels of analysis and therefore emphasise different properties.

Regardless of differences in emphasis, a stable baseline can be identified across the definitions. AI agents are generally understood as autonomous or semi-autonomous systems that operate with some degree of independence to pursue a goal or a task [2, 4, 10, 22, 37, 38, 44, 50, 73]. Their behaviour is organised around explicit objectives, whether these are framed as bounded task execution or adaptive task completion within a defined or dynamic environment [2, 4, 22, 37, 38, 44, 50]. Across the literature, this behaviour is enabled by the agent's internal structure, which includes perception, reasoning and planning, memory, and action or execution [4, 8, 22, 37, 50, 73]. First, perception is the ability of the agent to understand its environment or task. Reasoning and planning transform the perceived information into decisions and actionable plans. Moreover, memory preserves immediate context and can also store longer-term experiences and accumulated knowledge. Lastly, action converts the decisions taken by the agent into concrete operations [4, 8, 22, 37, 50, 73]. Castrillo et al. [4] organise the agent around the previous capabilities, whereas Masterman et al. [38] reduce the minimum structure to just brain, perception, and action, showing that different authors often describe closely related ideas using different labels.

In such systems, the LLM typically serves as the agent's core reasoning engine, or "brain". It interprets inputs to generate plans, select actions, and may revise its behaviour based on observations. However, the agent should not be reduced to the language model alone. An agent comes from the incorporation of the reasoning model with memory, tool

interfaces, and control logic that together support a closed loop of perception, reasoning, and action [2, 4, 8, 10, 38, 50, 73].

Based on the common elements across the reviewed literature, this thesis treats an AI agent as *"an LLM-based software entity that pursues an explicit goal through an iterative loop of perceiving task-relevant context, reasoning about the next step, optionally consulting memory or invoking tools, and executing actions while adjusting to intermediate environmental feedback in order to complete a user-defined task within a bounded scope."* Here, perception is understood broadly as the intake of task-relevant context, which may come from the prompt itself, prior interaction history, human involvement, or other intermediate feedback available during execution. The above formulation captures the various emphases in the reviewed literature while remaining narrow enough to distinguish AI agents from the broader agentic AI systems discussed in the following subsection.

2.2.2 Agentic AI

Agentic AI is commonly presented in the literature as an answer to the limitations of narrower AI agent models [10, 50]. Recently, attention shifted toward systems that can pursue less tightly specified objectives over longer horizons [2, 10, 50]. Agentic AI can be seen as the development of AI agents toward a more complex entity. This shift is tied to AI agents, since they become less effective when tasks are dynamic, long-horizon, or multi-stage [10, 50].

In the literature, the term itself is used in at least two similar ways. In one usage, agentic AI refers to a more extensive class of more capable systems that combine planning, persistent memory, and iterative feedback-driven execution in dynamic and complex environments [8, 38, 41]. In another usage, it is defined more narrowly as a multi-agent organisation in which specialised agents communicate, collaborate, and execute delegated subtasks under an orchestration mechanism [2, 10, 50]. This second formulation can be understood as an extension of the more general view, in which similar capabilities are achieved through multiple coordinated agents.

These definitions collectively link agentic AI to systems that pursue complex and long-term goals [10, 41, 50]. Such systems are designed to stay oriented toward their objectives over extended durations, respond to intermediate outcomes, and persist in working towards a goal even if the path to that objective is not fully predetermined [10, 41]. This behaviour is typically structured around iterative cycles of interpretation, planning, action, evaluation, and revision [2, 8, 10]. A high-level objective may be decomposed into smaller sub-goals and adjusted as new information emerges or earlier assumptions are proven inadequate [2, 8, 10, 50]. Consequently, agentic AI is characterised by the integration of reasoning, action, feedback, and revision into a continuous, adaptable process.

Based on the common elements across the reviewed literature, this thesis treats agentic AI as *"an LLM-based autonomous system, or system of collaborative agents, that pursues complex goals with iterative planning, persistent or shared memory, tool-mediated execution, and adaptation to intermediate outcomes over extended workflows."* This definition is broad enough to include both closed-loop architectures and explicitly orchestrated multi-agent systems, while remaining specific enough to distinguish agentic AI from narrower AI agents that focus on bounded task execution within a more limited scope.

2.2.3 AI Agents and Agentic AI Comparison

The literature often treats AI agents and agentic AI as part of the same broader family of autonomous AI systems [22, 37, 38, 41, 44]. Both are described as goal-directed LLM-based systems that combine perception, reasoning, memory, action, and tool use to operate multiple steps [4, 8, 38, 50, 73].

The most consistent difference lies in scope. AI agents are usually described as entities that perform specific tasks within a defined scope [2, 50], whereas agentic AI is more often associated with longer-horizon goals, extended workflows, persistent or shared memory, goal decomposition, and coordinated execution across specialised agents or modules [10, 41, 50].

This distinction can be clarified through a simple illustrative example. Consider a WhatsApp-based assistant that receives receipt photos and extracts structured information such as date, total cost, merchant, location, and purchased items, to store the data in a spreadsheet. The assistant should then answer any possible user questions, such as how much was spent on groceries in a given month. Such a system is best understood as an AI agent, because it pursues a specific user-defined task within a relatively narrow scope. The same application could, however, move closer to agentic AI if it were extended into a more extensive expense-management system with longer-horizon goals and an expanded set of capabilities. These capabilities could include anticipatory monitoring of spending patterns, price comparisons across stores, online research for cheaper alternatives, prediction of replenishment needs, and personalised recommendations. As a result, the example illustrates that the difference between AI agents and agentic AI is not absolute but primarily concerns scope, scale, persistence, and architectural complexity.

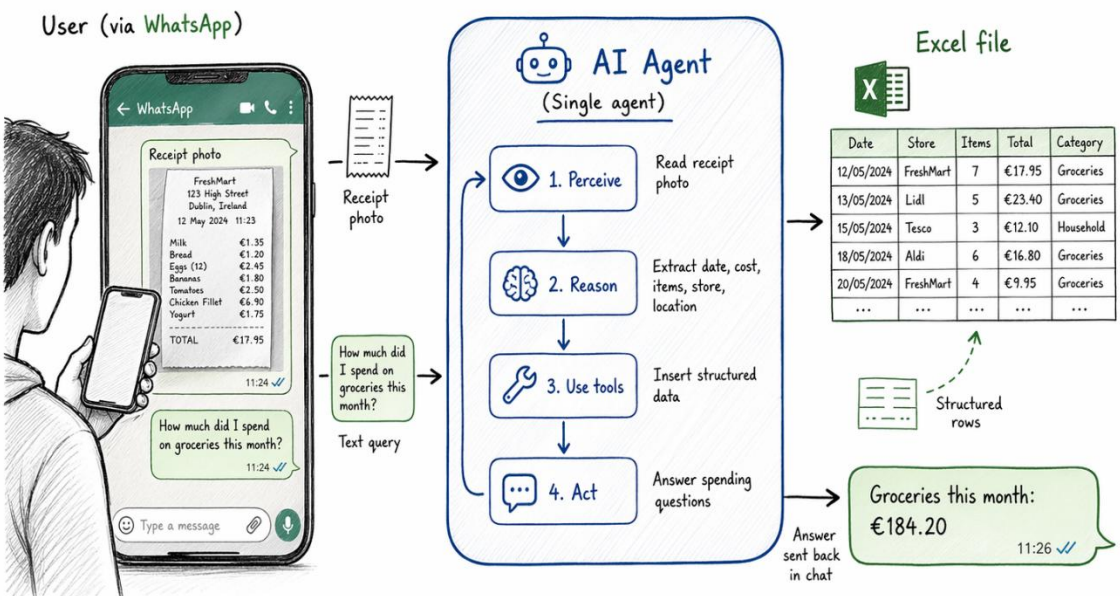

Figure 2-1. Example of an AI agent

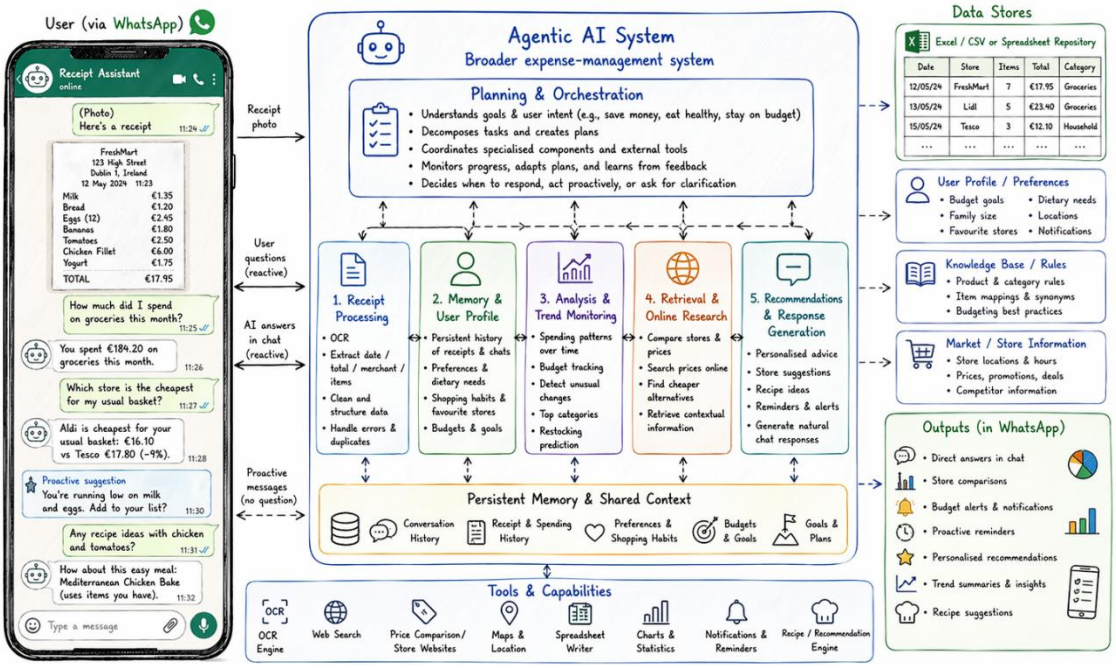

Figure 2-2. Example of an Agentic AI System

2.3 Large Language Model Basics

Large language models act as the primary reasoning component of many modern AI agents, but their practical use in real systems is restricted by several technical properties [4, 41]. They process text as token sequences and generate responses autoregressively [42, 58]. Their operation is bounded by a limited context window, which restricts how much information, in the form of tokens, can be passed to the model at once and creates difficulties in complex tasks that require long-range context [17, 42, 50]. Despite their strong capabilities, LLMs can fail. As a result, the present subchapter focuses only on aspects of LLM behaviour that directly affect system design, as they motivate subsequent choices.

2.3.1 Tokens and Tokenisation

Tokens can be viewed as parts of, or whole words. The input text of an LLM is first represented as a sequence of tokens [42, 58]. A prompt may seem short, but it can consume many tokens when it contains code, JSON schemas, tables, examples, or tool descriptions [29, 43]. Since several LLM calls may happen during a single agent's execution, token usage can accumulate across steps, affecting throughput and cost [17, 43].

2.3.2 Context Window and Context Management

While tokenisation determines how text is represented and counted, the context window refers to the maximum number of tokens the model can consider within a single request [4, 42]. Consequently, context management is a core design issue for LLM-based agents, because long workflows can exceed the available context budget, forcing the system to decide which information to keep, summarise, retrieve, or discard [4, 17, 41]. However, these strategies require explicit policies, since summaries may omit important details and retrieval may require recovering stored information later [41]. In multi-agent systems, context management becomes even more complex because each agent may require a different subset of information [17, 41].

2.3.3 Autoregressive Generation, Decoding, and Output Variability

LLMs generate text autoregressively, meaning they produce the output sequence one token at a time by predicting the next token from the preceding tokens [42, 43, 58]. At each generation step, the model assigns probabilities to possible next tokens, and the decoding strategy determines how the next token is selected [18]. As a result, the same model may produce different outputs from the same prompt when different decoding strategies or sampling settings are used [18].

In turn, output variability can affect system design. A model may answer the same task with a different structure or level of detail [18, 42]. Recent work presents LLMs as chaotic systems, highlighting the model's sensitivity to its inputs [32]. In ordinary LLM use, this variability may be acceptable, however, in agentic systems, it can be deemed problematic, particularly when subsequent steps depend on the exact structure and content of the model's response [17, 41].

2.3.4 How LLMs Fail

Despite their usefulness, they exhibit many flawed behaviours. A key drawback is hallucination, in which the model produces fabricated or factually incorrect content while presenting it in fluent, plausible language [33, 42, 72]. Zhang et al. [72] describe a problem that derives directly from hallucination, hallucination snowballing, in which a model over-commits to an early mistake and then generates additional false claims to

remain consistent with it. This is important for agent workflows because an early error can shape later decisions [33, 72].

LLMs have shown biased behaviour [59]. Since LLMs are trained on copious amounts of text data, they can reproduce or even amplify the social biases present in human text [59]. Wan and Chang [59] show that LLM-generated texts can display gender, racial, and intersectional language-agency biases, with minority groups often described in less agentic terms. This matters for agentic systems because biased outputs can affect the user-facing text generated by the system [33, 59].

2.3.5 Practical Model Access and Deployment Constraints

In practice, many agentic systems access LLMs through external APIs. Based on the API provider, the model access can introduce various setbacks [42, 43]. First, API access itself causes network dependencies and latency. The provider can impose rate limits or be temporarily unavailable. These constraints are important because an agentic workflow may require several model calls, so a delay or failure in one call can affect the entire workflow [17, 41].

Model choice is also a crucial design constraint. Kaplan et al. [24] show that language model performance improves predictably with model size, dataset size, and training compute, suggesting that smaller or less capable models can exhibit considerable discrepancies relative to stronger ones. In an agentic system, weaker models may produce poorer plans, miss constraints more often, and generate less reliable outputs [24, 42]. Consequently, there is a trade-off between capability and cost. While powerful models may yield preferable outcomes, they entail greater costs [24, 42, 17].

Overall, these properties show that LLMs are powerful but constrained reasoning components [4, 41]. Their token-based operation, bounded context, probabilistic decoding, imperfect reliability, and differences in model capabilities explain why later system designs rely on prompting, validation, tool integration, memory, retrieval, and orchestration to achieve more dependable behaviour [4, 17, 41].

2.4 Prompt Engineering, In-Context Learning, and RAG

LLM-based agents are not shaped only by the model that powers them, but also by the information provided to the model at runtime. A prompt can guide the model's output [52, 62], thus, the prompt is considered a pylon component of agentic design.

2.4.1 Prompts as Behavioural Specifications

Prompts are the primary mechanism to control the LLM's behaviour at inference time [52, 62]. Instead of acting as simple instructions, a prompt defines how the model should interpret a task, what information is important, and how it is expected to respond within a given context [52, 62]. Accordingly, prompts do not only request content, but also shape the model's task interpretation, output form, and interaction style [52, 62].

As a result, prompts act as behavioural specifications that determine how the model operates during inference [52, 62]. The same underlying model can produce different results depending on how the prompt is structured [53]. This broader concern is also consistent with evidence that higher-quality prompt engineering is associated with higher-quality LLM output [25].

2.4.2 AI Agent Prompt Components

In agentic systems, prompts are usually designed for repeated use within a workflow. A reusable agent prompt must produce an output that is consistent and usable by the surrounding system [52, 62]. Schulhoff et al. [52] identify common prompt components, including examples, roles, output formatting, and additional information, while White et al. [62] demonstrate that prompt patterns can shape persona, templates, interaction style, and context control.

A key component is the role assigned to the model, which defines the perspective and responsibility it should assume, such as reviewer or assistant [52, 62]. The agent's objective is also often included in the prompt [62], along with relevant environmental context, presented as a set of inputs [30, 52]. Since LLMs operate only on the context supplied at inference time, prompts must provide sufficient information for the current task while excluding irrelevant details that could distract the model or unnecessarily consume the context window [52]. Agent prompts also typically contain instructions to

guide the agent's behaviour, specifying what the model must do or avoid [31, 62]. Yao et al. [66], for example, show that prompting can explicitly interleave reasoning and acting, enabling the model to follow a structured problem-solving process.

Additional core components include tools and output formats. If tools are available, prompts may also specify rules for their usage [66]. In agentic workflows, outputs typically need to follow a structured format, like a schema or tool-call record, to enable automated parsing by later components [52, 62].

2.4.3 Prompt Brittleness

Although prompts can effectively guide LLM behaviour, they are often brittle. Even minor changes in wording can lead to different outputs, even when the intended task remains the same [53]. This can be explained through the chaotic nature of LLMs [32].

Prompt brittleness becomes more apparent when prompts contain multiple requirements. Longer or information-dense prompts might enhance performance by offering valuable context [52, 75]. In structured workflows, prompts tend to be longer because they often require the model to meet multiple conditions at once [75]. In these cases, ambiguous constraints can lead to invalid outputs, and even slight format deviations can cause issues, especially when results are to be parsed or verified automatically [52, 75].

2.4.4 In-Context Learning

In-context learning refers to the ability of an LLM to adapt its behaviour based on information included in the prompt, without any parameter tuning [3, 52]. This information may consist of natural-language instructions, exemplars, or both. Brown et al. [3] distinguish between zero-shot, one-shot, and few-shot settings, depending on whether the model receives only a task description, one example, or several examples in context. Describing the expected output format can also be considered a form of learning from the provided context. As a result, in-context learning can reduce ambiguity, improve formatting consistency, and help the model follow role-specific behaviour more reliably [3, 52].

Although in-context learning is a potent prompting technique, it does not guarantee correctness. Its effectiveness depends on the quality of the instructions or examples and the clarity of the prompt [3, 52]. Poor or contradictory examples might teach incorrect patterns, and ambiguous instructions can be misunderstood.

2.4.5 Retrieval-Augmented Generation

Lewis et al. [30] introduce Retrieval-Augmented Generation, or RAG, as a way to combine parametric and non-parametric knowledge retrieved from stored documents. A typical RAG pipeline involves indexing an external document collection, retrieving the most relevant passages for a given input, injecting them into the prompt, and then generating the final response [1, 30]. Later work extends this idea by making retrieval more adaptive and more reliable. SELF-RAG [1] allows the model to decide when retrieval is needed and to critique retrieved passages and generated outputs, while CRAG [64] adds a retrieval evaluator that can trigger corrective actions when retrieved documents are weak or misleading.

Although RAG can be a powerful tool to incorporate up-to-date and specific information into the model's context, it does not guarantee correctness. If retrieval returns irrelevant passages, misses the critical evidence, or introduces weak supporting material, generation quality may still degrade [1, 64].

2.4.6 Prompt Techniques Used in Agentic Systems

The literature on prompting contains many techniques, and recent surveys provide broad taxonomies of these methods and their applications [48, 52]. The most relevant ones are those that help specialised agents maintain stable roles, produce structured outputs, decompose difficult tasks, interact with tools, and revise weak outputs before they are passed to later workflow steps.

Few-shot prompting [3], as already discussed, remains one of the most basic and influential techniques. Role prompting works similarly by conditioning the model to act from a defined perspective. This was already noted earlier at the level of prompt components, but it also appears as a distinct prompting strategy in practice [52, 62]. Xu et al. [63] extend this idea in ExpertPrompting, where the model is conditioned on a

detailed expert identity tailored to the instruction, thereby improving answer quality and specialisation. Related work has also explored automatic instruction generation.

For multi-step reasoning, chain-of-thought prompting is one of the most influential techniques. Wei et al. [61] show that providing exemplars with intermediate reasoning steps can help large language models solve more complex reasoning tasks. In agentic systems, the main value of this idea is that prompts can guide the model to break a difficult step into smaller reasoning stages. A related development is ReAct [66], where the model generates interleaved reasoning traces and actions. The intended outcome is not only better reasoning but also better task execution, since the model can use reasoning to decide what action to take next and can use environmental feedback, referred to as observations, to continue reasoning.

Another important direction is self-review or critique prompting. Madaan et al. [36] propose SELF-REFINE, in which the same model follows an iterative process of output generation, feedback, and refinement. The intended outcome is a prompt-guided feedback loop that improves the original output in a simple process.

White et al. [62] present a pattern-oriented view of prompt design that is relevant to agentic systems. Their catalogue includes Persona for role specification, Template for output structuring, Reflection for self-checking, Alternative Approaches for exploring multiple solution paths, Cognitive Verifier for dividing questions into smaller checks, Flipped Interaction for clarification before action, and Context Manager for regulating contextual information. Collectively, these patterns support a more methodical approach to prompt design that suits multi-step and role-specialised workflows.

2.5 Structured Outputs and Deterministic Validation

LLM-based agents often produce fluent natural language, but free-form text is not always sufficient since agentic systems frequently require intermediate artefacts that can be parsed, checked, and reused [41, 60]. Moreover, structured outputs help constrain the model's otherwise open-ended natural-language responses, making responses easier to parse and validate through the workflow [41, 60]. This subchapter examines structured

outputs and deterministic validation as mechanisms for turning probabilistic LLM responses into machine-readable artefacts that can support more reliable execution.

2.5.1 Structured Outputs as Workflow Interfaces

In an agentic workflow, structured outputs specify what a model's response should look like [41]. When the workflow contains multiple specialised agents, structuring the output can be pivotal, as each agent may depend on the previous agent's output to continue its task. Without this usable structure, the next component would need to interpret free-form text, thereby increasing ambiguity and making the workflow more fragile [41].

2.5.2 Structured Output Mechanisms and Failure Modes

Structured outputs can be implemented through several mechanisms. At the prompt level, the model can be instructed to return information in a specific format, such as JSON with a predefined schema [60]. In tool-using agents, the output may instead take the form of a structured function call or API request, with arguments represented in a machine-readable structure [35].

However, structured output does not remove the possibility of failure. The model may return an invalid JSON, omit required fields, or overlook schema constraints, such as the types of each field [35, 60]. It may also produce a syntactically valid structure that is still unsuitable for execution [35], because the actual values still depend on the model's reasoning and could therefore be hallucinated. Although structuring the output of the model is a necessary interface mechanism, it does not guarantee correctness. It must be combined with validation, parsing checks, and later review stages before the output is trusted by the rest of the workflow [35, 41].

2.5.3 Deterministic Validation as a Reliability Gate

Structured outputs provide limited control over LLM behaviour. However, as discussed, there is no guarantee that the model will actually produce the requested structure. Hence, a validation step is required. It introduces a degree of determinism at the structural level, because the system can assume that certain fields should exist and can be accessed predictably.

Deterministic validation is the step that checks whether the model actually produced this expected structure. A validator component applies fixed rules to the generated output, such as verifying that required fields are present and the values adhere to the specified data types [35]. Validation is critical for tool-using agents because structured outputs, such as tool calls, must be interpretable to ensure expected execution [35]. As a result, validation ensures that only outputs that conform to the expected structure are passed to later workflow steps. When validation fails, the output should be held back from the next step and corrected, because reliability in agentic workflows depends on explicit control, validation, and recovery boundaries between stages [41]. Deterministic validation acts as a gate between structured generation and workflow execution. Outputs that satisfy the expected structure can proceed, while invalid outputs must be corrected before they affect later stages [41].

2.6 Tool Calling and Action Execution

Tool calling is a central capability of modern AI agents because it extends the model beyond text-only generation and enables interaction with external data sources [38, 50]. Tools function both as observation interfaces that expose relevant environmental information to the agent and as action interfaces that translate internal reasoning into concrete external operations [4, 50]. Because tool calls can affect external state, dependable agentic systems place guardrails around tool execution [41]. The following subsections examine tools as extensions of agent capabilities, the basic tool-calling process, the reliability risks introduced by tool use, and the control and safety mechanisms needed to keep model-generated actions bounded and auditable.

2.6.1 Tools as Extensions of Agent Capabilities

On their own, LLMs are generative systems that do not directly interact with external systems unless additional mechanisms are provided [50]. Tool augmentation extends the LLM's capabilities beyond passive response generation [50]. Tools are the means through which an LLM can gather environmental information or act on its surroundings. In light of this, tools are not peripheral additions to an LLM-based agent but the main mechanism by which internal reasoning connects to external observation and action [4, 38].

2.6.2 The Tool-Calling Process

Tool calling begins when the agent determines that the current task cannot be completed reliably from the information already available in context, or when the next step requires an external action rather than further text generation [4, 50]. The model then produces a structured tool call that specifies the required operation and its arguments. The surrounding system then executes that request through the relevant tool interface [4, 50]. Once the result is obtained, it is passed back into the agent's context as a new observation, allowing the agent to continue reasoning under an updated state and decide what to do next [50, 66].

2.6.3 Reliability Risks in Tool Use

Although tool use extends agents' capabilities, it also introduces a new class of reliability problems since model outputs are no longer interpreted only as text but may be executed as actions that affect external systems [4, 41]. A tool-related error can have more serious consequences than an ordinary generation error. A hallucinated tool name, a malformed argument, or an unsafe call may trigger an operational failure [41]. An incorrect tool call could cause costly or irreversible environmental impact [41].

Lin et al. [33] describe these failures as part of execution hallucination in LLM-based agents. In their taxonomy, tool selection hallucination occurs when an agent selects a non-existent, unavailable, or unsuitable tool, whereas tool-calling hallucination occurs when the agent produces incorrect, incomplete, or unsupported tool-call arguments. Even when the tool call is syntactically valid, it may still be semantically unsuitable [41], as discussed in the previous subchapter. Tool-use failures can originate from the model's mistaken belief about what tools exist, what a tool can do, or how it should be called [33].

Another class of failures concerns repeated or misleading interactions. Tool-using agents may fall into unproductive loops by repeatedly invoking the same tool or continuing to act without making meaningful progress toward task completion [38, 41]. Even when a tool returns a valid result, later reasoning may still be derailed if subsequent steps build on a flawed interpretation of the observation [41].

Execution failures within the tool itself can also cause reliability issues. External tools may fail due to timeouts, rate limits, resource unavailability, or tool-specific runtime errors [41]. For side-effectful tools, this is especially important because retries or repeated calls may duplicate effects if the tools partially execute [41].

2.6.4 Control and Safety of Tool Execution

Because tool calls can affect external systems, agentic architectures usually place explicit controls at the tool boundary rather than executing raw model output directly [4, 41]. Reliable systems use structured tool schemas, argument validation, and restricted tool access so that only well-formed and authorised actions can proceed [4, 8, 41].

For side-effectful operations, additional safeguards are needed. These commonly include least-privilege permissions, sandbox or file restrictions, timeouts, bounded retries, and safe-halt behaviour when a call violates schema, policy, or environmental assumptions [8, 41]. Such controls help prevent duplicated effects, inconsistent intermediate states, and unsafe execution after partial failure [41].

Safe tool use also depends on traceability. Structured logging of tool arguments, outputs, and stopping conditions makes failures easier to inspect and replay, while human escalation remains important when risk or uncertainty is high [41].

2.7 Workflow Orchestration and Graphs

The term workflow is used in different ways across the agent literature. In some cases, the term refers to the plan composed by the agent to solve a task [46]. In other cases, it is used more structurally to describe a runtime arrangement of coordinated processing steps or a structured execution process composed of multiple operations [70]. This distinction is important because, in this thesis, the term workflow refers to the latter definition.

2.7.1 Workflow Graph Components

Since a workflow is regarded as an executable graph, defining its main components becomes more straightforward. At its core, the graph is made up of nodes and edges.

Nodes carry out specific tasks, while edges facilitate data flow, logic, and communication between these tasks.

Nodes are the executable units of the workflow. Depending on the system, a node may correspond to an LLM call, a retrieval step, a tool invocation, a validation stage, or a memory update [70]. In multi-agent settings, specialised agents can themselves appear as nodes. Edges, on the other hand, define how execution moves between nodes. In the simplest case, they express that one step follows another. More generally, however, they specify how control, data, or communication flows through the workflow [70].

A workflow graph does not necessarily execute in the same way every time it runs. Even when the workflow graph is static, edges can be conditional and lead execution along different paths depending on what has happened in earlier steps, such as a tool response or a verification result [70]. The same high-level graph can ultimately produce different runtime paths because later transitions may depend on the evolving information generated during the workflow's execution [70].

2.7.2 Common Executable Workflow Patterns

Although workflows can vary in their specific implementations, they are generally built from a small set of recurring architectural patterns. Yu et al. [69] explicitly outline several workflow modes, namely chain workflow, parallelisation workflow, routing workflow, orchestrator-workers, and evaluator-optimiser structures. A chain workflow is a structure in which a general task is broken into an ordered sequence of dependent steps, where each step uses the outputs from previous ones. This suits tasks that follow a steady forward progression. Parallelisation workflow is employed when multiple tasks are independent and can be processed concurrently. Routing workflow directs tasks to different specialised processors based on input type or characteristics. It is most useful when execution should follow different paths depending on the case. Orchestrator-workers describe a setup where a central LLM decomposes a complex task and delegates subtasks to specialised worker agents. Evaluator-optimiser involves one LLM generating an output and another evaluating and refining it based on feedback.

For implementation purposes, this thesis groups workflows into four practical graph-level types, namely *reactive conversational*, *linear pipeline*, *planner executor*, and *hybrid*. The reactive conversational type is the simplest form, since execution remains centred on a main conversational LLM node that can react to an input, call tools, and continue across turns. This makes it closely related to ReAct-style [66] interaction, although the label used here refers to the overall workflow shape rather than only the prompting technique. A linear pipeline corresponds to a fixed forward sequence of dependent stages with minimal branching and therefore aligns closely with the previously defined chain workflow. A planner-executor workflow separates deciding what should be done from carrying it out. In practice, this broader structure may incorporate more specific motifs such as routing, parallel execution, or orchestrator-worker delegation when the planner decomposes the task into subtasks, assigns them to specialised subgraphs, and then merges their outputs before determining the next step. Finally, a hybrid is used for workflows that combine fixed structural stages with a more flexible conversational or tool-using middle section. In this thesis, evaluator-optimiser is not defined as a separate top-level workflow category, since the same structural idea can be captured within the linear pipeline archetype by placing an evaluation stage after generation, so that earlier outputs are reviewed, validated, and, where necessary, refined before execution continues.

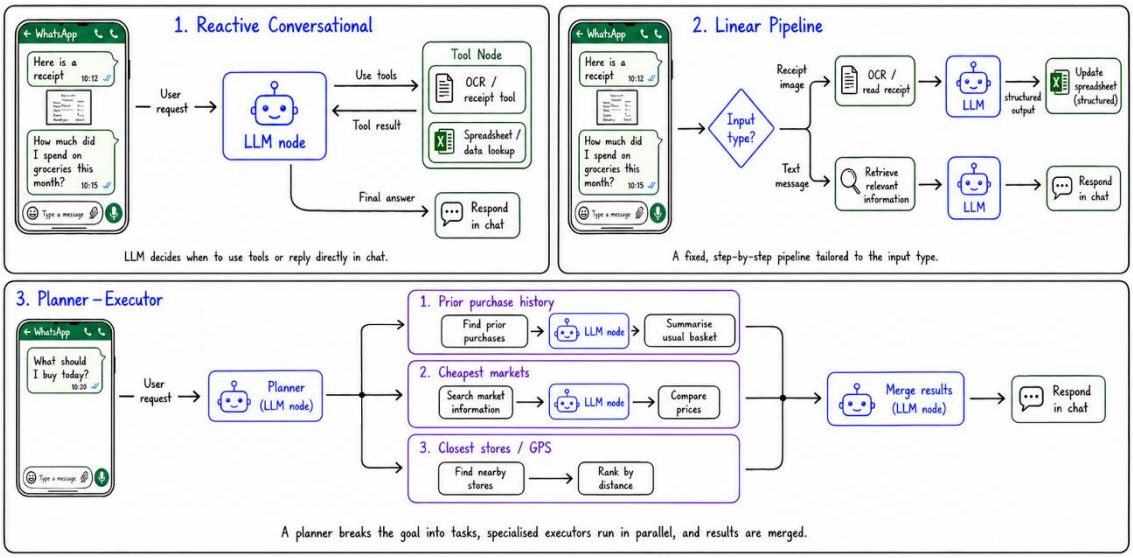

Figure 2-3. Graph-based Agent Workflow Patterns

2.7.3 Control, Failure Prevention, and Debugging

A workflow can still fail even when its individual nodes perform well, because the failure may lie in the structure that connects them. Recent work on workflow optimisation shows that a weak workflow structure can lead to problems such as missing verification, insufficient decomposition, fragile or incorrect control flow, excessive depth, and high communication overhead [70]. These issues show that workflow reliability depends not only on the quality of individual model calls, but also on how execution is organised at the graph level.

2.8 Memory and State Management

Memory and state management are central to LLM-based agents. Both allow information to remain available across multi-step execution [65], and since LLMs operate within finite context windows, they do not automatically retain all relevant information across interactions [41, 65]. Without explicit mechanisms for retaining, selecting, and reusing information, an agent may lose context, repeat previous work, or make decisions that contradict earlier observations [41, 65].

This subchapter examines memory and state management as foundations for agentic behaviour. It first clarifies the distinction between memory, state, and context. It then discusses short-term memory, which supports immediate reasoning during an ongoing task, followed by long-term memory, which preserves information beyond the current interaction. Shared memory in multi-agent systems is also examined, as it can support coordination among specialised agents. Finally, it reviews the main memory operations and reliability constraints of memory.

2.8.1 Memory and State as Agent Foundations

Memory is commonly treated as a core component of LLM-based agents, alongside perception, reasoning, planning, tool use, and action execution [4, 8, 65]. Its role is to retain information that may support later reasoning [65, 73]. This information may be temporary, such as recent observations in the current task, or persistent, such as knowledge and experience retained across sessions [8, 65]. Consequently, memory provides the agent with continuity beyond a single model call.

State is connected to memory but is more immediate and specific. In agent systems, it represents the current active record of execution, including information the agent has at a given moment, such as the current goal, recent observations, intermediate outputs, tool responses, ongoing progress, and pending decisions [41, 65]. As the process advances, the state evolves because each new input can update the information the agent needs to consider next [41, 65].

Context is even narrower. It refers to the selected part of the state or memory that is provided to the LLM during a model call, because LLMs reason only over the information in their current input [4, 65]. As a result, memory management also requires context assembly, which involves selecting what should and should not be included in the prompt [41, 65]. Including too little context can cause the agent to miss relevant constraints, while including too much can introduce irrelevant information, increase token usage, and make later reasoning less focused [41, 65].

2.8.2 Short-Term Memory

Short-term memory refers to information retained for the immediate task or interaction [8, 65]. It keeps recent and directly relevant information available for the current process, such as recent messages, observations, tool outputs, and intermediate records from earlier steps [65]. This type of memory is usually temporary, limited, and closely tied to the current execution [8, 65]. Its main role is to preserve enough recent context for the agent to continue reasoning and acting coherently during an ongoing task [4, 65].

2.8.3 Long-Term Memory

Long-term memory refers to information retained beyond the immediate task [8, 65]. Unlike short-term memory, which mainly supports the current execution, long-term memory stores persistent information across sessions, including accumulated knowledge, historical interactions, learned patterns, and user preferences [65]. Its purpose is to support continuity over longer time spans, allowing an agent to reuse past information [4, 65].

In LLM-based agents, long-term memory is often implemented externally [4, 41]. External memory may take the form of document repositories, databases, logs, knowledge

bases, or other persistent stores that can be queried when relevant information is needed [4, 41].

2.8.4 Shared Memory in Multi-Agent Systems

In multi-agent systems, memory may be used as a shared resource across several specialised agents [8, 50]. Shared memory enables agents to operate from a common knowledge base, thereby supporting coordination when different agents contribute to the same broader objective [8, 50].

Shared memory can improve coordination by allowing agents to complement direct message passing with a persistent shared record [8]. A common memory space can preserve shared goals, previous actions, tool results, constraints, and decisions that should remain visible across the system [8, 41]. This can reduce duplicated work, help agents maintain a consistent view of the task, and make coordination more coherent across multiple steps [8, 50].

2.8.5 Memory Operations and Reliability Constraints

Memory in agentic systems is an active process that determines how information enters, changes, and leaves the agent's information space [41, 65]. Yang et al. [65] describe four basic memory operations, specifically writing new memory units, reading or retrieving relevant memory, updating existing memory, and deleting obsolete or irrelevant memory.

Memory operations, however, create reliability challenges [41, 65]. Stored information may become stale, incorrect, duplicated, or irrelevant to the current task [41]. In addition, unverified model outputs may contaminate memory if they are stored without validation [41]. Furthermore, Lin et al. [33] describe memory-update hallucinations, in which the model incorrectly modifies or deletes memory.

These risks can be reduced through explicit memory management policies [41, 65]. Reliable memory systems need rules for what should be written, when stored information should be updated, how retrieval should be filtered, and when old information should be removed or ignored [41, 65]. Nowaczyk [41] argues that reliable memory requires governance over the whole memory lifecycle. Stored items should remain traceable to

their source, time-sensitive information should be refreshed or expired, untrusted content should be filtered before reuse, long histories should be compressed without losing auditability, and temporary reasoning traces should be separated from durable knowledge stores [41]. These controls help prevent outdated, weak, or unverified memories from influencing later reasoning [41].

2.9 Multi-Agent Collaboration and Coordination Patterns

Multi-agent collaboration is used when tasks require several specialised forms of reasoning [2, 8, 38, 50, 73]. Agents are assigned distinct responsibilities so that the overall task can be decomposed, coordinated, checked, and refined through a structured process [6, 19].

This subchapter examines multi-agent collaboration as an architectural mechanism for organising distributed work among specialised agents [2, 8, 38, 50]. It first discusses role-based task decomposition and coordination [6, 19]. It then examines review, aggregation, and conflict handling [6, 19, 41]. Finally, it considers collaboration failure modes [19, 31, 33, 41].

2.9.1 Role-Based Task Decomposition and Coordination

Role-based task decomposition is one of the main ways multi-agent systems organise collaborative work [6, 7, 19, 69, 73]. Instead of treating the task as a single undifferentiated problem, the system decomposes the broader objective into smaller responsibilities that can be distributed across multiple agents [69, 73]. These agents may differ in their roles, models, tool access, and task responsibilities, allowing the system to combine complementary capabilities when one agent alone is insufficient for a complex task [2, 8, 50, 73].

Coordination then depends on how these roles interact. Three patterns are mainly used, namely sequential handoff, horizontal, and vertical. First, sequential handoff refers to the step-by-step coordination in which one role produces an artefact that serves as input to the next role [19, 69]. This kind of coordination is predictable because each role has a defined position in the workflow and each handoff has an expected output. In horizontal

coordination, agents participate more equally in a shared discussion or decision-making process [7, 38, 73]. This pattern is useful when the task benefits from multiple perspectives, such as consulting, tool-use discussion, or information synthesis [7]. Vertical or supervisor-led coordination involves one agent acting as the master, which decomposes the goal and assigns subtasks to worker agents [2, 50, 69, 73]. This pattern is useful when the workflow requires clear task ownership, controlled handoffs, and a single point for monitoring progress or deciding whether the plan should be revised.

2.9.2 Review, Aggregation, and Conflict Handling

Multi-agent coordination also requires mechanisms for deciding whether intermediate outputs should be accepted or rejected [6, 19, 41]. Specialised agents often produce artefacts that influence later stages of the workflow [19, 21, 41]. If these artefacts are passed forward without review, an early mistake can propagate and escalate throughout the workflow [19, 33, 41]. Owing to that, review stages are commonly used to inspect intermediate outputs before they are reused by the system [6, 19, 21, 41].

Review can take several forms [6, 14, 19, 21, 41, 69, 74]. In software-oriented systems, one agent may generate an artefact while another agent checks it against requirements, tests, or expected behaviour [19, 21]. A related review pattern is LLM-as-a-judge, in which an LLM or an agent evaluates and critiques the outputs of other components [14, 74].

Aggregation is needed when several agents contribute partial outputs to the same result, because the workflow must decide how those contributions merge into a usable output [2, 7, 50, 73]. Aggregation requires selecting relevant information, removing duplication, preserving constraints, and keeping unresolved uncertainty visible when the available agent outputs do not fully agree [33, 41].

Conflict handling concerns the decision rule used when agents produce incompatible or uncertain outputs [5, 41, 50]. Agents may disagree [33, 41], hence, the workflow needs an explicit resolution mechanism, such as a reviewer agent, a supervisor decision, a revision loop, or human escalation for high-risk cases [5, 6, 19, 21, 41]. Without such a

mechanism, disagreement may be ignored, weak outputs may be accepted, or the most recent agent response may overwrite earlier concerns [33, 41].

2.9.3 Collaboration Failure Modes

Multi-agent collaboration introduces failure modes that do not appear in the same form in single-agent systems because agents can influence one another [5, 33, 41, 50]. A local mistake may propagate across the collaboration process [19, 33, 41].

One failure mode is role drift, where an agent diverges from its designated responsibility or switches to a different role [5, 31, 41]. CAMEL [31] reports role flipping in autonomous communicative agents, suggesting that role prompts alone may not ensure stable boundaries. Another issue is unproductive communication, where agents exchange messages without improving outcomes or progressing towards termination [5, 31, 38, 41]. CAMEL [31] reports repeated instructions, weak responses, and infinite loops, while Masterman et al. [38] show that systems without a clear leader can spend much effort directing each other. A third failure is error propagation through weak handoffs [19, 33, 41]. MetaGPT [19] identifies cascading hallucinations as an issue in naively chained LLM-based multi-agent systems, where logical inconsistencies can build up across agents.

2.10 System-Level Reliability, Governance, and Ethical Considerations

Reliability has already appeared throughout this chapter because each major component of an agentic system introduces its own failure modes. This subchapter treats reliability as a system-level property rather than as a property of any single component. The following subsections examine system-level concerns and ethical, fairness, and governance considerations.

2.10.1 System-Level Concerns

System-level concerns arise because agentic workflows integrate multiple uncertain components into a single execution. Once agentic components are linked, the relevant concern is no longer only whether a single module fails, but how the architecture manages interactions between components [5, 33, 41]. Nowaczyk [41] argues that reliability in

agentic AI is chiefly architectural, thus, system-level concerns refer to risks that emerge from interactions between components rather than from a single module in isolation.

One system-level concern involves generated artefacts. The concern is not merely that an artefact may be wrong, but that it may appear structured as expected or coherent to be accepted by the workflow. Empirical work on AI-assisted code generation reports high error rates and security concerns [12, 68]. The literature often includes observer and reviewer roles [6, 19], further showing that intermediate results should be checked.

A second concern is opacity. Hahn et al. [16] argue that transparency becomes more difficult when AI agents carry out complex multi-step operations, use external tools, or comprise multiple interacting agents. Opacity is an aspect of reliability, as the source of a failure may be hidden across nested steps [5, 33, 50] and should be traceable. Without observability, such failures may be noticed only at the final output, making it difficult to locate the source of the problem [33, 50].

2.10.2 Reliability Techniques

Reliability techniques aim to reduce the chance that local errors become system-level failures. During execution, bounded retries, stopping rules, tool-call limits, and fallback behaviour can prevent repeated ineffective actions, unsafe loops, or uncontrolled tool use [5, 41]. After generation, review mechanisms such as self-correction, validator agents, and human intervention can correct outputs before they propagate to later workflow stages [6, 14, 33, 36, 74].

Observability is also necessary because reliability controls are difficult to apply when the workflow is opaque. A dependable agentic system should record the intermediate artefacts it produces [41, 50]. These records make failures easier to inspect, reproduce, and contain. Standardised workflow execution representations can also support reliability by making workflows easier to debug and compare across systems [69].

2.10.3 Ethical, Fairness, and Governance Considerations

Ethical concerns arise because agentic systems can make decisions, use tools, and affect external environments with only partial human supervision [10, 16, 26]. In multi-agent

systems, failure attribution and accountability can become difficult because several components may contribute to an error, even though the system as a whole caused the outcome [26, 50]. Responsible design should preserve human responsibility rather than treating the agent as an independent moral or legal actor [16, 26].

Fairness is also important because biased outputs can be reused across later stages of an agentic workflow. A biased model response may be treated as a recommendation or a memory entry, allowing bias to propagate through the system [50, 59].

Governance concerns are closely related to transparency, privacy, security, and oversight. Agentic systems may process user data, store information in memory, call external tools, or produce actions that are difficult to inspect after the fact [2, 10, 16]. As a result, governance should include clear limits on what the system is allowed to do, what data it may store, when human approval is required, and how actions are logged for later inspection [26, 50].

2.11 LangChain Ecosystem Overview

The previous subchapters examined the conceptual foundations needed to design LLM-based agentic systems. This subchapter now introduces the LangChain ecosystem as the implementation framework used to realise these ideas in practice.

Recent work treats the tools provided by the LangChain ecosystem as complementary parts of an agentic AI toolchain. LangChain[1] provides reusable components for building LLM applications, including prompt templates, model interfaces, tools, retrievers, memory modules, chains, agents, and output parsers [39, 49]. LangGraph[2] extends this component-based approach into a graph-based orchestration framework, where stateful workflows can be represented by nodes, edges, conditional routing, cycles, and multi-agent execution [39, 49]. LangSmith[3] then supports observability, debugging, evaluation, and monitoring of LLM workflows [39, 49].

[1] LangChain, https://docs.langchain.com/oss/python/langchain/overview

[2] LangGraph, https://docs.langchain.com/oss/python/langgraph/overview

[3] LangSmith, https://docs.langchain.com/langsmith/home

2.11.1 LangChain as the Component Layer

LangChain is the component layer that supports individual LLM-driven operations, while LangGraph supports the overall orchestration structure [39]. It provides reusable abstractions for model access, prompt templates, messages, tools, agents, chains, retrievers, memory, and output parsing [13, 39, 49, 57]. These abstractions are useful at the node level [49].

The most pivotal abstractions are message handling and tools. In LangChain, the communication between models is represented with structured message abstractions. A message can come from different sources. Specifically, a system message can define a set of rules the model should follow, a human message can represent the user's query or feedback, an AI message is often wrapped around the model's response, and a tool message represents the chosen tool's output. On the other hand, tool abstraction can convert any normal Python function into a callable tool for the LLM. The model has to make a structured tool call that includes the tool's name and arguments in order for the surrounding system to execute it and return the answer wrapped in a ToolMessage [49, 50].

2.11.2 LangGraph as the Orchestration and Stateful Execution Layer

LangGraph is the most relevant component of the ecosystem because it provides the graph-based execution model that coordinates the multi-step agent behaviour. LangGraph connects directly to the workflow model introduced in *Subchapter 2.7*.

The StateGraph abstraction is central to this model. A typical LangGraph workflow begins by defining the state schema [39], which determines what information the workflow carries forward during execution. The state is what allows a LangGraph workflow to remain coherent across several steps, by containing information such as the user request, intermediate outputs, tool results, and environmental feedback. As each node runs, it can update part of this state and pass the updated state to the next node.

LangGraph also supports tool execution within the executable graph. Tool nodes can be defined to support tool execution in the graph. By placing tool execution inside the graph, the system can make tool use traceable and bounded by workflow logic.

The same graph structure also supports multi-agent workflows. A specialised agent can be represented as a node, as a subgraph, or as a tool-like component. Sapkota et al. [49] identify LangGraph as suitable for multi-agent workflows because its graph model supports coordinated transitions and adaptive execution.

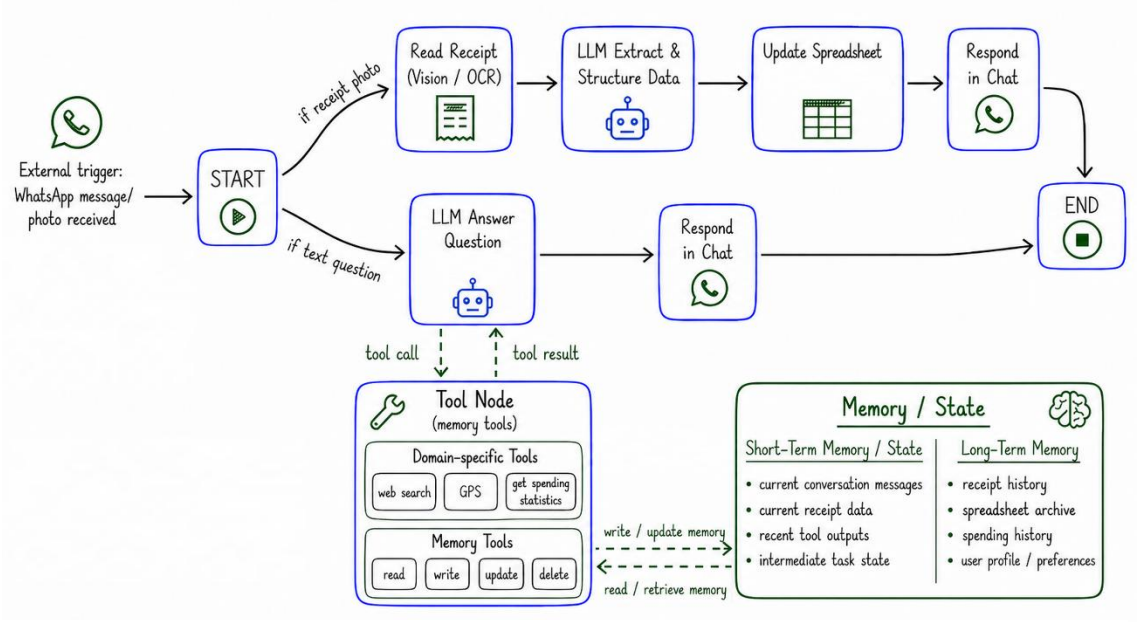

Figure 2-4. LangGraph-based Workflow for an AI Agent

2.11.3 LangSmith as the Observability Layer

LangSmith is the observability layer of the LangChain ecosystem. Its role is to make LLM applications easier to inspect and monitor [39, 49]. As discussed in *Subchapter 2.10,* observability is important for graph-based agentic systems, where failures may occur at any step of the workflow's execution.

LangSmith enables traceability in the executable graph-based workflow. LangSmith can record inputs, outputs, intermediate steps, model calls, tool invocations, errors, and execution metadata, allowing developers to inspect how the workflow unfolded [39, 49].

This supports debugging because a failure can be traced back to the node, prompt, tool call, or intermediate output where it first appeared.

Monitoring is another relevant function. Sapkota et al. [49] describe LangSmith as supporting real-time tracing, latency and token accounting, prompt and session inspection, error logging, trace visualisation, and production quality assurance.

2.12 Summary

This chapter established the background and foundational concepts needed to understand the design of LLM-based agentic systems. It first clarified the relationship between AI agents and agentic AI, viewing AI agents as limited, goal-oriented software entities, while agentic AI involves longer-horizon goals, coordination, persistent or shared memory, and adaptive execution. It then reviews the key technical mechanisms that make such systems possible. LLMs provide the reasoning core but, due to their token-based operation, limited context windows, probabilistic output, hallucination risks, and deployment constraints, they cannot be fully relied upon as standalone components. Techniques like prompt engineering, in-context learning, and retrieval-augmented generation shape model behaviour and provide relevant context, while structured outputs and deterministic validation transform model responses into machine-readable formats suitable for subsequent workflow stages. Tool calling expands agent capabilities beyond text generation by enabling interaction with external systems. Workflow orchestration organises multi-step processes using graph structures. Memory and state management preserve the information needed for continuity over ongoing tasks, while multi-agent collaboration divides responsibilities, exchanges information, and coordinates within broader workflows. System reliability, governance, and ethics highlight that failures can stem from interactions among reasoning, tools, memory, validation, and orchestration, not just from isolated model outputs. The chapter concludes by introducing the LangChain ecosystem, a framework for operationalising many of these concepts, with LangGraph supporting stateful graph execution and LangSmith enabling tracing, debugging, and evaluation. This foundational overview prepares future chapters to discuss related work, present the proposed multi-agent system, and explain how its design addresses the challenges outlined here.

Chapter 3

Related Work

3.1 Introduction **35**
3.2 Agents for Code Generation **35**
3.3 Automatic Agent Builders and Agent-Construction Systems **38**
3.4 Gap and Positioning of This Thesis **43**

3.1 Introduction

Chapter 2 established the conceptual and technical foundations needed to understand LLM-based agentic systems, both as theoretical constructs and as implemented software systems. This chapter now turns from background concepts to related systems in the literature. Its purpose is to examine prior work that is closest to the system developed in this thesis and to clarify where the proposed approach fits within the existing literature.

The reviewed work is organised around two main areas. The first area concerns code-generation agents, described in *Subchapter 3.2*. The second area concerns automatic agent builders and agent-construction systems, described in *Subchapter 3.3*. These works are closer to the present thesis because they treat agent design itself as part of the automation problem.

3.2 Agents for Code Generation

This subchapter reviews three agentic code-generation systems, specifically MetaGPT [19], AgentCoder [21], and CodeAgent [71]. These works are selected because they show

how agentic systems can move from direct code generation toward more structured and realistic software-development workflows.

3.2.1 MetaGPT [19]

MetaGPT is a multi-agent framework for collaborative software development. Its main idea is to organise AI agents according to software engineering Standard Operating Procedures (SOPs), rather than allowing agents to interact through unconstrained free-form conversation. The system models a software development team through specialised roles such as Product Manager, Architect, Project Manager, Engineer, and QA Engineer, with each role responsible for a specific part of the development workflow.

The workflow begins from a user requirement and proceeds through a sequence of structured stages. The Product Manager analyses the requirement and produces a Product Requirements Document (PRD). The Architect translates the requirements into system design artefacts, such as file lists, data structures, and interface definitions. The Project Manager decomposes the work into tasks, the Engineer implements the required classes and functions, and the QA Engineer generates test cases to support code quality. MetaGPT uses structured communication through documents, diagrams, a shared message pool, and explicit task assignments, enabling agents to exchange task-relevant artefacts.

MetaGPT provides design lessons, such as that multi-agent work can be controlled through specialised roles, structured artefacts, and predefined handoffs rather than through open-ended agent conversation.

3.2.2 AgentCoder [21]

AgentCoder is a multi-agent framework for code generation that focuses on improving generated code through testing and self-optimisation. AgentCoder uses a smaller agent structure comprising three specialised agents: the Programmer, Test Designer, and Test Executor. The main purpose of this design is to separate code generation from test generation and execution.

The workflow begins with the Programmer agent, which generates code from the given programming requirement. Separately from this task, the Test Designer agent generates test cases without seeing the produced code. The separation is important because tests produced after seeing the generated code may become biased toward that implementation and may fail to expose its weaknesses. The Test Executor agent then runs the generated tests against the generated code in a local execution environment. If all tests pass, the code is returned as the final output. If execution produces errors or failed assertions, the feedback is sent back to the Programmer agent, which revises the code. This loop continues until the code passes the tests or the iteration budget is reached.

AgentCoder contributes to this thesis by showing the value of separating implementation, validation, and feedback into distinct responsibilities. It also supports the design choice of giving coder agents bounded implementation tasks, while using separate testing or review steps to provide focused feedback before code is passed further through the construction pipeline.

3.2.3 CodeAgent [71]

CodeAgent is an agent framework for repository-level code generation. CodeAgent targets coding tasks where the generated code must fit into an existing repository with documentation, dependencies, and a runtime environment. This is important because real software development often requires understanding how new code interacts with existing classes, functions, files, and project-specific constraints.

The system provides the agent with access to five programming tools that support different stages of the coding process. These include Website Search, Documentation Reading, Code Symbol Navigation to inspect relevant classes or functions, a Format Checker to format generated code, and a Code Interpreter to run code and receive execution feedback. CodeAgent demonstrates that tools can aid in code generation.

3.2.4 Summary of Code-Generation Agents

Three key design lessons from these systems are especially relevant to the proposed system. First, separating roles in code generation is advantageous, as the code-generating component should not also handle requirement definition or review its own output [19,

21]. Second, generated code must be supported by structured context, which includes requirements, file contents, surrounding code, and implementation constraints [21, 71]. Third, code generation should be followed by validation and refinement mechanisms to identify weak or incomplete implementation artefacts before they are used by later stages [19, 21, 71].

3.3 Automatic Agent Builders and Agent-Construction Systems

This subchapter reviews systems that automatically create, configure, or search agents, agent teams, and multi-agent workflows. These works are closer to the present thesis than general code-generation agents because they treat agent design itself as part of the automation problem. However, they approach this problem in different ways. AgentVerse [7], AutoAgents [6], and AIME [54] focus on runtime agent composition, team formation, or on-demand actor creation during task execution. AgentBuilder [55] instead focuses on creating a task-specific agent through role generation, model selection, and self-evolution. A different direction is taken by Automated Design of Agentic Systems (ADAS) [20] and MAS-GPT [67], which represent agentic systems or multi-agent workflows as executable code. The purpose of this subchapter is to examine what these systems contribute to automatic agent construction.

3.3.1 AgentVerse [7]

AgentVerse is a multi-agent collaboration framework that uses AI agents to solve tasks through dynamically formed specialised agent groups. It is relevant to the proposed system because it dynamically creates AI agents based on the given task and environmental feedback. Specifically, the system can generate and recruit agents with task-specific expertise and then use them within a collaborative execution process. AgentVerse treats agent composition as part of the task-solving procedure.

The framework operates through four main stages: expert recruitment, collaborative decision-making, action execution, and evaluation. During expert recruitment, the system generates descriptions of expert agents that are suitable for the current task. These agents then participate in collaborative decision-making, exchanging information and proposing a solution or action. The selected action is executed in the environment, and the evaluation

stage judges whether the current result is satisfactory. If the result is unsatisfactory, feedback can be used in the next round to adjust the group composition or the collaboration process.

AgentVerse contributes to this thesis by demonstrating that agent roles can be dynamically generated from the task and refined through feedback. However, its agents function as temporary collaborators for specific tasks within a fixed framework, rather than being reusable, flexible agent systems.

3.3.2 AutoAgents [6]

AutoAgents is a multi-agent task-solving framework that automatically generates a temporary team of specialised agents for a given task. The generated agents are not standalone agentic systems, but role-specific collaborators used within the AutoAgents execution process.

The framework is divided into two main stages: a Drafting Stage and an Execution Stage. In the Drafting Stage, predefined agents such as the Planner, Agent Observer, and Plan Observer collaborate to generate a temporary agent team and an execution plan for them to follow. The observers assess whether the generated agents are suitable for the task and if the execution plan is coherent and properly assigned. The generated agents follow a predefined template, which requires the agent's prompt, description, toolset, and suggestions. The toolset is selected from an existing pool of tools. In the Execution Stage, the generated agents carry out the plan through self or collaborative refinement, while an Action Observer coordinates and monitors the execution.

AutoAgents contributes two useful ideas to this thesis. First, it shows that generated agents should be described through explicit components. Second, its refinement mechanisms show that generated outputs can be improved through feedback before the process continues. The proposed system adapts these ideas at the level of agent specification and construction-stage refinement.

3.3.3 AIME [54]

AIME is a multi-agent framework that aims to make task execution more adaptive than static plan-and-execute systems. Instead of generating a fixed plan and assigning work to predefined agents, AIME combines dynamic planning, on-demand actor creation, ReAct-style [66] execution, and shared progress management. It is relevant to this thesis because it treats agent capabilities, task allocation, and execution state as elements that can change during the run, rather than as fixed decisions made only at the beginning.

The framework contains four main components: the Dynamic Planner, the Actor Factory, Dynamic Actors, and the Progress Management Module. The Dynamic Planner decomposes the user request into a hierarchy of subtasks and maintains the task list in a shared memory component. The Actor Factory then creates a specialised Dynamic Actor for each dispatched subtask by selecting an appropriate persona, knowledge source, and a toolset. The Dynamic Actor executes the assigned subtask through a ReAct-style [66] loop of reasoning, action, and observation. The Progress Management Module acts as a shared state component that records the task hierarchy, subtask statuses, progress updates, and completion criteria.

For this thesis, AIME is useful because it emphasises shared execution state and feedback-based adaptation. However, the actors created by AIME remain temporary executors for subtasks inside a runtime task-solving framework. The proposed system takes the broader ideas of shared state, task-specific configuration, and feedback-guided refinement, but applies them to a construction pipeline that produces reusable agent artefacts.

3.3.4 AgentBuilder [55]

AgentBuilder is a framework for automating the creation of task-specific agents that can improve over time through feedback. It focuses on constructing a task-specific agent and then allowing that agent to evolve through task execution and feedback. The overall pipeline begins with task analysis, continues with role building and task execution, and then uses task feedback to support self-evolution.

The Role Builder module constructs the task-specific agent through several role components. These include the agent's character, skills, constraints, memory pool, and

knowledge base. The LLM Manager dynamically selects suitable models from an LLM pool according to the task and integrates their outputs through weighted aggregation. The self-evolution mechanism improves the agent after each task execution by using feedback from generated task trajectories. At the practical level, this refinement is also described through iterative prompt optimisation and enhancement to the agent's planning workflow.

AgentBuilder contributes to this thesis by demonstrating that agent construction can be treated as a structured process of specification and refinement. Its main value here is not the exact set of role components, but the broader principle that an agent should be defined, constrained, executed, and improved through feedback.

3.3.5 ADAS [20]

Automated Design of Agentic Systems (ADAS) is a framework that formulates agent design as an optimisation problem, comprising three main components: a search space, a search algorithm, and an evaluation function. The search space defines which agentic systems can be represented, the search algorithm explores possible designs, and the evaluation function measures how well each candidate agent performs on the target objective.

The paper proposes Meta Agent Search as a concrete ADAS algorithm. In this approach, a meta-agent iteratively creates new agentic systems in code. The archive is initially seeded with baseline agents, such as Chain-of-Thought or Self-Refine agents, and the meta-agent uses this archive to design new candidate agents. Each candidate is described in high-level terms and then implemented as code. The candidate agent is evaluated on validation tasks. If runtime errors occur, the meta-agent reflects on the error and repairs the implementation before the candidate is added back to the archive so that later iterations can build on previous discoveries.

A key feature of ADAS is that the generated agentic systems are represented in code. In the paper's implementation, the meta-agent is provided with a small framework that includes basic abstractions such as information objects, foundation-model modules, prompt formatting, and model querying. The meta-agent then implements a Python function called *forward*, which defines how the candidate agent processes the task.

ADAS is important for this thesis because it shows that agentic workflow design can be treated as a code-generation problem. This supports the idea that agent construction can produce executable artefacts rather than only natural-language role descriptions. However, ADAS searches for agent designs that perform well on predefined task domains and evaluation functions. The generated agents are primarily code-defined workflows for solving benchmark-style tasks, rather than complete, reusable agent implementations.

3.3.6 MAS-GPT [67]

MAS-GPT is a system for generating multi-agent systems from a user query. Its main idea is to simplify the construction of multi-agent systems by treating them as a generative language task: the input is a user query, and the output is an executable multi-agent system represented as Python code. Unlike approaches that manually define a fixed collaboration structure, MAS-GPT trains an LLM to generate different multi-agent workflows for each query in a single inference step.

The generated multi-agent system is represented as a Python function named *forward*. Within this function, agent prompts are defined, LLM calls are represented as function calls, and communication between agents is implemented by passing previous outputs into later prompts.

MAS-GPT is important for this thesis because it supports the central idea that a natural-language request can be translated into an executable agentic structure. It shows that multi-agent workflows do not always need to be manually designed, but can be generated from the task itself. However, MAS-GPT generates a query-specific forward function for answering the current query. The agents within this function are lightweight, prompt-defined LLM modules, whose behaviour is primarily determined by role instructions rather than by source files, persistent memory, or custom tool implementations. The proposed system applies the broader idea of generating executable agentic structures to a construction pipeline that produces reusable agent files, prompts, tools, and supporting artefacts.

3.3.7 Summary of Agent-Construction Systems

The reviewed agent-construction systems show that automatic agent design can take several forms. AgentVerse [7], AutoAgents [6], and AIME [54] generate or adapt agents during task execution, mainly so that a current task can be solved by a suitable runtime team or actor configuration. AgentBuilder [55] moves closer to explicit agent construction by defining a task-specific agent through role components and improving it through feedback. ADAS [20] and MAS-GPT [67] take a more executable direction by representing agentic workflows or multi-agent systems as code.

Across these systems, three design lessons are especially relevant to the proposed system. First, generated agents need explicit specifications, such as roles, personas, tools, constraints, knowledge sources, memory assumptions, or output expectations [6, 54, 55]. Second, agent construction benefits from staged workflows, where planning, role generation, execution, observation, and refinement are separated into distinct steps [6, 54, 55]. Third, feedback is central, since evaluation, observer agents, progress tracking, self-refinement, self-evolution, and repair mechanisms all show that generated outputs should be reviewed and improved before they are trusted [6, 7, 55, 20].

The most relevant direction for this thesis is the move from natural-language agent descriptions toward executable representations. ADAS [20] and MAS-GPT [67] show that agentic workflows can be represented as Python code, in which prompts, LLM calls, intermediate outputs, refinement steps, and aggregation logic are arranged via explicit control flow.

3.4 Gap and Positioning of This Thesis

The reviewed literature on code-generation agents shows the value of role separation and feedback-driven refinement [19, 21, 71]. Similarly, agent-construction systems show that agent roles, prompts, workflows, and coordination structures can be generated and adapted [6, 7, 54, 55, 20, 67]. These works therefore provide important foundations for the system developed in this thesis.

However, a gap remains between systems that use agents to solve tasks and those that construct reusable agents as their final output. Most of the agent-construction systems reviewed in *Subchapter 3.3* generate temporary runtime collaborators, task-specific role

templates, or lightweight executable workflows for solving the current task. Even systems that represent workflows as code, such as ADAS [20] and MAS-GPT [67], mainly generate code-defined reasoning structures within a predefined framework, rather than a complete, reusable agent implementation with separate source files, prompt artefacts, tool implementations, supporting files, and an execution-ready project structure.

This thesis is positioned in this gap. The proposed system is a bounded multi-agent builder that takes a natural-language request and constructs reusable custom AI agent artefacts. It combines specialised construction agents, staged workflow control, structured intermediate outputs, code and prompt generation, validation, and refinement. In contrast to systems that generate temporary task-solving teams, the target output is a reusable agent package that includes all surrounding implementation artefacts.

Chapter 4

System Scope and Requirements

4.1 Introduction **45**
4.2 Target Users and Use Cases **45**
4.3 System Inputs, Outputs, and Produced Artefacts **47**
4.4 Functional and Non-Functional Requirements **48**
4.5 Safety Requirements **50**
4.6 Summary **52**

4.1 Introduction

The previous chapters established the conceptual foundations of LLM-based agentic systems and positioned the proposed system against related work. This chapter now defines the scope and requirements of the system developed in this thesis. Its purpose is to clarify what the system is expected to accept, what it is expected to produce, which behaviours it must support, and which limits constrain its operation.

The rest of this chapter is organised as follows. *Subchapter 4.2* defines the target users and use cases addressed by the system. *Subchapter 4.3* describes the system inputs, outputs, and produced artefacts. *Subchapter 4.4* presents the functional and non-functional requirements that guide the construction workflow. *Subchapter 4.5* defines the main safety requirements. Finally, *Subchapter 4.6* summarises the chapter and links these requirements to the design decisions discussed in the next chapter.

4.2 Target Users and Use Cases

This subchapter defines the target users and provides example use cases of the proposed system. Its purpose is to clarify what type of user interaction the system assumes and what kinds of custom agents it is expected to construct. This matters because the proposed system is not a fully autonomous developer that receives a vague request and produces a verified final product without further involvement. Instead, it is a bounded construction workflow that depends on user clarification, review, and acceptance at important points of the process.

4.2.1 Target Users

The proposed system targets users with either full technical knowledge or moderate general knowledge of the agent they want to construct. The user must be able to describe the desired behaviour, answer clarifying questions, review pivotal intermediate decisions, and accept or reject portions of the generated output.

Human involvement is a crucial control point in the system, through clarification, verification, and approval. The system may ask the user to clarify ambiguous requirements, confirm key assumptions, or approve intermediate artefacts before a stage can continue. Vague or incomplete task descriptions can affect the quality of the responses. Prior work shows that multi-agent systems may fail when task specifications are unclear or verification is incomplete [5], and that errors in LLM-based agents can propagate across multi-step workflows before becoming visible in the final output [33]. For this reason, the target user should be able to describe the intended agent behaviour, answer clarification questions, and inspect the generated artefacts before treating the produced agent as usable.

The system is most suitable for users who understand the task they want to automate and can judge whether the produced agent matches their intention. For example, a user may not know how to implement a LangGraph workflow, but they should know whether they want the agent to store preferences, ask follow-up questions, use a database, analyse images, or generate recommendations. The important requirement is not expert programming ability, but the ability to provide meaningful feedback during construction and recognise when the produced package needs correction or further testing.

4.2.2 Supported Use Cases

The proposed system supports requests where the desired agent can be described as a workflow with expected behaviour. The use cases below are not intended to cover all possible AI agent designs, but to illustrate the types of construction requests that fall within the system's scope.

A first example is a receipt and spending assistant, as introduced in *Subchapter 2.2*. A user may ask for an agent that receives receipt photos, extracts structured information, stores the results in a spreadsheet or similar repository, and later answers relevant questions. A second example is a menu recommendation assistant. A user may ask for an agent that stores a person's food and drink preferences and provides suggestions when a menu is provided.

The use case shows why clarification may be needed, since menus can appear in many formats, so the system should clarify whether the produced agent should support photos, text, PDFs, links, or only one of these input forms.

4.3 System Inputs, Outputs, and Produced Artefacts

This subchapter defines the input-output contract of the proposed system. The system begins with a natural-language request and attempts to transform it into a produced agent package. During this process, it may also require user clarification, approval, or review at specific stages. The outputs of the system include both final artefacts, such as generated source files and prompt files, and intermediate artefacts, such as refined requests, workflow structures, logs, and file snapshots.

4.3.1 System Inputs

The system receives inputs about the requested agent and configuration variables. The initial user request is the most crucial input, providing a natural-language description of the desired agent. The configuration also requires two additional inputs. First, the orchestrator mode manages how clarification questions are handled during stages that need extra information. When enabled, clarification requests are routed through the

clarification orchestrator. When disabled, clarification questions are passed directly to the user. The second input is the prompt review mode, which specifies whether the prompt review is conducted by an LLM reviewer, the user, or both.

4.3.2 Interactive Construction Inputs

In addition to the initial system inputs, the construction workflow may request user input during execution. These inputs control points that help the system reduce ambiguity and verify important intermediate results before later stages continue. Some agents may ask the user to clarify the intended behaviour of the produced agent, so that the original request can be converted into a clearer construction input. The user may also be asked to approve, reject, or revise selected intermediate outputs.

4.3.3 Produced Agent Artefacts

The main output of a successful construction run is a produced agent package. This package is stored as a generated project folder and contains the artefacts needed to inspect or run the produced agent. The core artefacts are the generated source code and the prompt file. The source code includes the main Python implementation file for the produced agent, based on the LangGraph framework. The prompt file stores the prompt templates used by the generated agent. The package may also include any necessary supporting files. Depending on the requested agent, these may include data files, configuration files, folders, local storage resources, or other files required for execution.

4.3.4 Intermediate and Observability Artefacts

The system also creates intermediate artefacts during construction. These artefacts help later stages operate, but they are not final files in the produced agent package. The system may also store logs and file snapshots after each agent finishes its task. These artefacts support observability and traceability by making it easier to inspect how the produced agent package was constructed and where errors may have occurred.

4.4 Functional and Non-Functional Requirements

This subchapter defines the main requirements that guide the behaviour and quality of the proposed system. Functional requirements (FR) describe what the construction workflow

must do to transform a user request into a produced agent package. Non-functional requirements (NFRs) describe the qualities the workflow should maintain throughout this process.

4.4.1 Functional Requirements

FR 1. Refine and clarify the user request.

The system must accept an initial natural-language request and transform it into a clearer construction input, using clarification questions when needed.

FR 2. Produce a workflow representation.

The system must convert the refined request into a workflow representation that describes the main steps, tools, memory needs, and execution structure of the produced agent.

FR 3. Generate the source code of the produced agent.

The system must create the generated project folder and produce the main Python implementation file for the agent. This file should contain the workflow structure, node functions, routing logic, helper functions, and any required tool-related code.

FR 4. Generate and review prompt templates.

The system must create and store prompt artefacts used by the agent. The prompt artefacts should contain the instructions used by the produced agent. The system should support prompt reviewing according to the selected prompt review mode.

FR 5. Create the supporting execution environment.

The system must create any additional files or folders required by the produced agent.

FR 6. Support selected user approval and revision points.

The system must allow users to clarify, approve, or revise at key stages.

FR 7. Preserve intermediate construction outputs.

The system must preserve or expose important intermediate outputs. This supports later inspection and debugging.

FR 8. Terminate with a produced package or a clear stopping point.

The system must not continue indefinitely. It should end with a produced agent package when construction succeeds or stop at a clear point when construction cannot continue.

4.4.2 Non-Functional Requirements

NFR 1. Modularity.

The system should separate construction responsibilities into specialised stages or agents. This supports clearer responsibility boundaries and makes the construction process easier to inspect and modify. Replacing a stage with an equivalent should not affect the rest of the system.

NFR 2. Traceability.

The system should preserve enough intermediate information to make the construction process inspectable.

NFR 3. Execution control.

The system should use execution limits where appropriate, including recursion limits, review limits, and clear stopping conditions. This reduces the risk of repeated processing or construction attempts that make no progress.

NFR 4. Maintainability and practical usability.

The produced package should be organised so that a user can inspect, run, test, and modify it after generation. This includes separating source code, prompt artefacts, and supporting files in a clear project structure.

4.5 Safety Requirements

This subchapter defines the main safety requirements of the proposed system. The system does not claim production-grade security, but safety requirements remain necessary because it processes user instructions, generates executable code, creates files, and may produce agents that use tools or access local storage. The requirements below define basic limits to keep generated artefacts inspectable and to reduce unsafe behaviour during construction.

4.5.1 User Input and Prompt Injection

A user request may include instructions that conflict with the intended behaviour, attempt to bypass constraints, or ask the system to ignore previous instructions. This risk is related to prompt injection, where user input can override the original developer instructions in a prompt [52]. The mitigation is to separate user-provided task content from system-level construction instructions, since prompt-based defences and guardrails can reduce prompt-hacking risk even though they do not fully solve it [52]. The user request should specify the agent being built, but it should not override the system prompts or role constraints.

4.5.2 Generated Code Safety

Generated code should not be assumed to be correct, secure, or ready for deployment. Empirical studies show that AI-assisted code generation varies in validity, correctness, security, reliability, maintainability [15, 68], and can contain security weaknesses [12]. The mitigation is to keep the generated code inspectable and to include verification stages that ask an LLM reviewer to identify potential implementation, reliability, or security issues. This remains only a partial safeguard because LLM-based repair can fix some weaknesses in generated code but does not prove that the resulting code is safe or correct [12].

4.5.3 File and Tool Boundaries

File and tool use should remain constrained to the intended construction context. The construction workflow should avoid unrestricted file operations, arbitrary command execution, and destructive actions outside the generated project directory. Sensitive tools, such as file deletion, file writing, external API calls, or database modification, should have explicit privileges and access permissions. This follows the broader architectural principle that tool-using agents should rely on constrained interfaces, schema validation, least-privilege access, and explicit control around actions that affect external systems [41].

4.5.4 Repeated Actions and Execution Control

Repeated actions should be controlled during construction. Multi-agent systems may repeat steps, fail to recognise termination conditions, terminate prematurely, or perform

incomplete verification [5]. Repeated processing can waste resources [5], and tool-using architectures should manage side-effectful actions through explicit runtime controls such as budgets, termination conditions, and safeguards [41]. The mitigation is to use recursion limits, review limits, tool-call limits, and clear stopping conditions. The system should stop either with a produced package or at a clear stopping point, rather than continuing indefinitely.

4.5.5 Runtime and Validation Failures

Runtime and validation failures should be handled before later stages reuse their outputs. Since the construction workflow depends on LLM calls and structured intermediate outputs, failures may occur when a model response is malformed, a schema cannot be parsed, a required field is missing, an API call fails, or a rate limit is reached. If these failures are ignored, later stages may build on invalid artefacts and produce incorrect files. The mitigation is to validate structured outputs before reuse, retry recoverable failures within limits, stop on non-recoverable failures, and return a clear error state when construction cannot safely continue [41].

4.6 Summary

This chapter defined the scope and requirements of the proposed system. It identified the expected users, and possible use cases, the system inputs and outputs, and the main functional, non-functional, and safety requirements. These requirements frame the system as a bounded, human-supervised construction workflow that produces inspectable agent packages rather than verified production-ready agents. The next chapter uses this scope to define the design principles and methodology that guide automated agent construction.

Chapter 5

Design and Methodology

5.1 Introduction **53**
5.2 System Design **54**
5.3 Agent Role Decomposition **57**
5.4 Construction Methodology **58**
5.5 Interfaces and Data Contracts **61**
5.6 Prompt Contracts **62**
5.7 Tool Access and Execution Control **64**
5.8 Validation, Review, and Failure Handling **65**
5.9 Artefact Management and Traceability **67**
5.10 Summary **69**

5.1 Introduction

Chapter 4 defined the requirements of the proposed system, including its scope, expected inputs and outputs, functional behaviour, quality constraints, and safety limits. This chapter now explains the design and methodology used to satisfy those requirements. Its purpose is to describe how the proposed system is organised, how the construction process is structured, and which design principles guide the transformation of a natural-language request into a reusable custom-agent package.

The chapter focuses on design-level decisions, starting with *Subchapter 5.2,* which presents the overall system design and the main construction pipeline. *Subchapter 5.3* explains the role decomposition used by the specialised construction agents. *Subchapter*

5.4 describes the construction methodology followed by the builder. *Subchapter 5.5* discusses the interfaces and data contracts that allow information to move between stages. *Subchapter 5.6* treats prompts as behavioural contracts for the proposed team of agents. *Subchapter 5.7* explains the design of tool access and execution control. *Subchapter 5.8* discusses validation, review, and failure handling as gates between construction stages. *Subchapter 5.9* explains how artefacts are managed and traced during construction. Finally, *Subchapter 5.10* summarises the chapter and links these design choices to the implementation described in *Chapter 6*.

5.2 System Design

The proposed system is designed as a multi-agent builder for custom AI agents. It receives a natural-language request and transforms it into an inspectable custom-agent package containing source code, prompt templates, and supporting files when required.

At the system level, the builder follows a staged construction pipeline. The initial request is refined into a clearer construction input, converted into a workflow design, translated into a code scaffold, annotated, implemented, extended with prompts, and completed with supporting files. This staged structure follows the broader principle that complex LLM-based software tasks benefit from decomposition into specialised roles, intermediate artefacts, and explicit review or feedback stages [6, 17, 19, 21].

This subchapter presents the system design at three levels. First, it defines the objective of the builder as the production of an inspectable custom-agent package. Second, it describes the high-level pipeline through which construction artefacts are progressively created. Third, it defines the design's boundary by clarifying that the generated package remains a human-inspectable output rather than a production-verified agent.

5.2.1 Design Objective

The design objective is to transform a natural-language request into a reusable and inspectable custom-agent package. The system treats the user request as the start of the construction process. Since a request may be incomplete, ambiguous, or too high-level, it must be converted into intermediate artefacts that later stages can use. This follows the

broader view that complex agentic tasks are easier to control when organised through intermediate representations and staged execution [34, 69, 70, 73].

The produced package is expected to contain the generated agent's main implementation file, prompt file, and any supporting files needed for execution. The focus on inspectability is important because LLM-generated code, prompts, and workflow decisions may appear coherent while still containing errors. Moreover, the system does not claim that the generated agent is production-ready or formally verified. This boundary is necessary because AI-generated code may still contain errors and should remain subject to inspection and testing [12, 15, 68].

5.2.2 High-Level Pipeline

The system is organised as a sequence of specialised construction stages, forming a sequential handoff pattern as discussed in *Subchapter 2.9*. Each stage receives an artefact from the previous stage and transforms it into a more implementation-ready artefact.

The main stages are the Input Refiner, Workflow Refiner, Code Scaffolding, Code Annotator, Software Engineer, Prompt Engineer, and File Handler. The Input Refiner clarifies the original request because unclear task specifications can affect later stages [5, 33], while the Workflow Refiner converts the refined request into a graph-based workflow representation. Both agents can ask the user clarifying questions when the available context is insufficient to continue. These clarifications first pass through the Clarification Orchestrator, which attempts to answer them through the prior conversation. Code Scaffolding creates the initial source-code structure based on the generated workflow. The Code Annotator adds implementation guidance to the code, enabling the Software Engineer to implement it through Coder subagent coordination. The Prompt Engineer creates and reviews prompt templates used by the agent, and the File Handler then creates the supporting files or folders needed.

Stage	Purpose	Main output
Input Refiner	Clarifies the original request	Refined request
Workflow Refiner	Converts the request into a graph-based design	Workflow representation
Code Scaffolding	Creates the initial project structure	Python code scaffold
Code Annotator	Adds implementation guidance	Annotated scaffold
Software Engineer	Implements the generated code using Coder agents	Implemented Python file
Prompt Engineer	Creates and reviews prompt templates	Completed prompt file
File Handler	Creates supporting files	Supporting files and completed package

Table 5-1. Main construction stages of the proposed system.

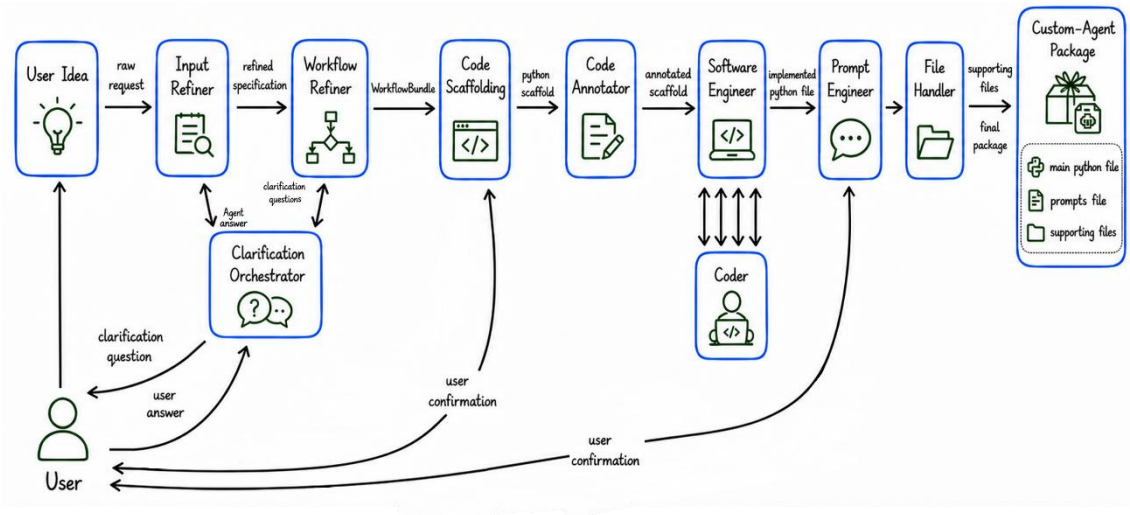

Figure 5-1. High-level design of the proposed multi-agent construction pipeline.

The pipeline structure, shown in *Table 5-1* and *Figure 5-1*, keeps each stage focused on a limited responsibility. The user interactions in the figure represent clarification and approval points, which help keep the construction process aligned with the user's intention before the final package is produced.

5.2.3 Design Boundary

The system design has clear limits. It does not guarantee that the generated agent is correct, secure, or ready for deployment. It also does not fully verify the generated source code, prove the correctness of the produced workflow, or remove the need for human review. These limits are necessary because LLM-based agents can fail through hallucinated reasoning or faulty intermediate artefacts [33]. Generated code also remains vulnerable to functional and security weaknesses even when produced by advanced models [12, 15, 68].

These limits are not weaknesses of the design, but part of the system boundary. They keep the proposed system aligned with the requirements defined in *Chapter 4*. The system is designed to make agent construction more structured and inspectable, not to remove all uncertainty from LLM-generated artefacts. The following subchapters explain how this design is supported.

5.3 Agent Role Decomposition

As discussed, the system is divided into specialised agents, each assigned a clear, independent task. The strict separation provides a level of inspectability and control, so failures can be easily traced back to the source.

5.3.1 Single Responsibility Principle

The role decomposition follows a single-responsibility design. Each construction agent is assigned one responsibility and is expected to produce an output that matches that responsibility. This design reduces the need for any single agent to manage the whole construction problem, which would make the construction process harder to inspect, harder to correct, and more vulnerable to error propagation across stages [33, 41]. By separating responsibilities, the system keeps each agent focused on a narrower task and makes its output easier to evaluate.

The separation is also useful for prompt design and tool control. Since each agent has a narrower role, its prompt can define a more specific task. Moreover, access to tools can be limited to the minimum set needed for execution.

5.3.2 Artefact Handoff

Each construction agent is responsible for producing a defined artefact usable by the next stage. Sequential handoff makes the construction process more explicit because later agents receive only the expected inputs instead of free-form text. This ownership structure makes handoffs inspectable. If a later stage produces a weak result, the construction process can be inspected to determine whether the problem came from the current role or from the artefact it received. This is consistent with the broader reliability principle that agentic systems require explicit interfaces and traceable intermediate artefacts so that errors can be identified before they propagate to later stages [33, 41].

5.4 Construction Methodology

The construction methodology defines how the proposed system progresses from an initial user request to the production of a custom AI agent. While *Subchapter 5.2* described the system at the level of its main stages, this subchapter explains the transformation process followed by those stages. The central idea is that the request is gradually transformed into a sequence of intermediate artefacts that become more specific, structured, and executable as the workflow progresses. The following subsections describe the main transformation stages.

5.4.1 Requirement Refinement and Clarification

The first transformation is the refinement of the initial user request into a clearer construction input. The original request may describe the desired agent in broad, informal, or incomplete terms. Therefore, requirement refinement reduces ambiguity before the system commits to a workflow design.

This stage is handled by the Input Refiner. It receives the original user request and produces a refined, agent-creation-ready request. The process first corrects grammar, spelling, and formatting errors while preserving the meaning and vocabulary of the original text. Schmidtová et al. [51] showed that spelling mistakes can degrade LLM performance in machine-translation prompt settings; hence, the correction step may increase the performance. The corrected request is then used in a clarification loop, where the agent asks clarification questions or makes safe assumptions in order to reduce

ambiguity in the user's request. Iterative refinement then takes place, in which the request is rewritten as a more explicit construction input, based on the user's feedback.

Clarification is used only when the corrected request lacks enough information for reliable workflow generation. If Clarification Orchestrator mode is enabled, clarification questions first pass to the orchestrator, which attempts to answer them from prior conversation memory. If no answer can be inferred from memory, the question is passed to the user. After refinement, the user can approve the refined request or request changes.

This stage is important because unresolved ambiguity can propagate through the rest of the construction process. A vague request may lead to an undesirable outcome. Prior work on agentic systems shows that errors can propagate across stages [33], while multi-agent systems can also fail when task specifications are incomplete or ambiguous [5]. This is why requirement refinement is the first step of the construction process.

5.4.2 Workflow Derivation

The second transformation converts the refined construction input into a workflow representation. At this stage, the system defines how the produced agent should run, including the workflow type, main steps, tool use, memory needs, and possible user interaction points.

The Workflow Refiner receives the refined request and focuses only on the produced agent's workflow. If execution behaviour is still unclear, it may ask clarification questions, which can pass through the Clarification Orchestrator before reaching the user. Once the workflow is clear enough, the Workflow Refiner produces a structured workflow bundle.

The workflow bundle describes the produced agent through a root workflow graph and optional subgraphs, which may represent subagents or more complex internal processes within a node. The graphs include a workflow type, a memory flag, a name, a description, and a set of nodes and edges. The supported workflow types are as discussed in *Subchapter 2.7*. The memory flag indicates whether the produced agent needs persisted state, such as messages or any other information that should persist across runs.

5.4.3 Scaffold-First Construction

The third transformation is the conversion of the workflow representation into an initial code scaffold. The system transitions from workflow design to source files, translating the root graph and all subgraphs into a structured Python file that later agents can complete. This scaffold approach provides the construction process with a stable implementation framework before detailed code generation begins.

The scaffold-generation utility creates the project folder and writes the initial Python file for the generated agent, along with an empty prompt file. The Python file follows a predictable LangGraph-style structure, including schemas, tools, LLM definitions, helper functions, node functions, conditional functions, graph construction, graph compilation, and a testing section. Many parts are initially placeholders, but they give later stages a concrete file to inspect and complete.

5.4.4 Incremental Implementation

The fourth transformation implements the source code. The system uses the scaffold as a shared file and completes it through smaller implementation stages rather than a one-off generation.

Before implementation begins, the Code Annotator prepares the scaffold for coding. It adds implementation guidance directly to the source file. This guidance includes node docstrings, proposed state schemas, helper function stubs, tool function stubs, and LLM configuration decisions, such as whether an LLM should use tools or structured output. The Code Annotator turns a structural scaffold into an implementation plan embedded in the code.

The Software Engineer coordinates the full implementation. It inspects the annotated file, chooses which function should be implemented next, and delegates bounded coding tasks to Coder subagents. Each Coder receives the Software Engineer's instructions and the function name to implement. In turn, the Software Engineer can review the returned implementation and either accept or reject it. Both the Software Engineer and Coder agents pass from a static review process before submitting their implementations.

5.4.5 Prompt Review and Support Files

The final transformation completes the implemented source code into a usable custom-agent package. Although the main Python file has been implemented, the produced agent also needs prompt templates that match the code and may require supporting files.

The Prompt Engineer creates the prompt templates used by the produced agent. Within the agent, a two-step review process takes place. First, the prompt is statically checked to ensure the instructions adhere to the intended behaviour. Second, a mock response is generated to assess the model's actual behaviour.

The File Handler then creates the remaining supporting artefacts. These may include JSON files for persistent memory, storage folders, configuration files, or other helper files required by the produced agent. Keeping file creation separate makes side-effectful operations, such as file creation and modification, easier to inspect and keeps them within the generated project context.

This final stage turns the implemented code into a completed custom-agent package. The package contains the main source file, the prompt file, and any required supporting files. It also marks the end of the construction methodology.

5.5 Interfaces and Data Contracts

Interfaces and data contracts define how information moves through the construction workflow. Since the system is composed of several LLM-driven agents, later stages cannot depend only on vague natural-language outputs. Each stage must receive information in a form that is expected, hence, the system uses explicit inputs and outputs for each stage of the process.

5.5.1 Stage Interfaces and Inter-Agent Data Movement

The construction workflow is organised around explicit stage interfaces. Each agent receives a defined input and transforms it into an output artefact for the next stage. Agents therefore communicate mainly through sequential handoffs rather than open-ended

dialogue. The expected output artefact of each stage is summarised in *Table 5-1*, while the transformation process is explained in more detail in *Subchapter 5.4*.

5.5.2 Intra-Agent Data Contracts

Data contracts also exist within each agent. In the implementation, each graph node receives the current state as input and returns a state update. Thus, nodes operate through predefined state fields such as messages.

LLM outputs are also constrained through structured-output contracts where possible. Instead of relying on free-form text, several agents use schemas to define the fields that the model should return. For example, the Workflow Refiner can produce a structured workflow bundle. This connects to the use of structured outputs and deterministic validation discussed in *Subchapter 2.5*. Tool calls form another type of data contract. Similar to the structured outputs, the LLM must produce a structured tool call with the expected tool name and arguments.

5.5.3 Contract Failures and Recovery

Contract failures mainly occur when an LLM output does not match the expected structure or interface. For example, an LLM may return malformed structured output, omit a required schema field, or provide tool-call arguments that do not match the expected signature. The system reduces these risks through schema enforcement for both structured outputs and tools, as well as bounded retries when a schema mismatch occurs.

5.6 Prompt Contracts

Prompt contracts define how the behaviour of each builder agent is specified before execution. In the proposed system, prompts are treated as a foundational component of agentic design. They are design artefacts that define each construction agent's role, task scope, input context, expected output, constraints, and rules for tool use or structured output when applicable.

The rest of this subchapter explains prompt contracts at three levels. First, S*ubsection 5.6.1* describes the repeated structure used by the system's prompts. *Subsection 5.6.2*

explains why these prompts are implemented as templates with runtime placeholders. Finally, *Subsection 5.6.3* discusses prompt review and iterative correction, since prompt brittleness can affect the reliability of later construction stages [52, 62].

5.6.1 Prompt Contract Structure

The prompts follow a repeated structure and specify what the agent is, what information it receives, what action it must perform, what constraints it must adhere to, and what output it must return. This aligns with the prompt components discussed in *Subchapter 2.4*.

This structure is visible across the system's prompts. They commonly use sections such as role, objective, inputs, instructions, available tools, output rules, and output format. These sections make the prompt easier to inspect and reduce ambiguity about the agent's responsibility. The structure also helps preserve role boundaries across the construction workflow.

5.6.2 Prompt Templates and Runtime Inputs

The system's prompts are implemented as templates rather than fixed static prompts, which is necessary because the same agent role can be used across different construction runs and user requests. A fixed prompt cannot represent the changing runtime context without being rewritten for each execution.

Template placeholders allow a stable prompt contract to be combined with dynamic inputs. For example, prompts may include placeholder fields such as {user_request}, which are filled before each model call. Prompt templates allow the role, rules, and output expectations of the agent to remain stable, while the task-specific information changes according to the current state of the context.

5.6.3 Prompt Review and Iterative Correction

As discussed in S*ubchapter 2.4*, prompt wording, formatting, missing constraints, and unclear output rules can affect LLM behaviour [52, 62]. More specifically, a weak prompt may lead to role drift or weak handoffs between agents [31, 41]. It may also cause

malformed outputs, incorrect tool use, or broken formatting when the prompt does not clearly define output expectations, tool-use rules, or runtime constraints [33, 52, 62].

The review process checks both behavioural and technical issues. At the behavioural level, it checks whether the LLM complies with its role, objective, constraints, and expected output defined by the prompt. At the technical level, it checks whether the prompt is compatible with the code that uses it, including output format requirements, structured-output expectations, tool-use rules, missing placeholders, and fragile *.format(...)* usage. This process is iterative since a minor change in the prompt can significantly change the LLM's response [53].

5.7 Tool Access and Execution Control

This subchapter explains tool design at four levels. *Subsection 5.7.1* explains why builder agents need tools, while *Subsection 5.7.2* discusses role-based tool access and why agents should receive only the tools required for their responsibilities. *Subsection 5.7.3* explains tool boundaries and permissions, especially for file-modifying tools. Finally, *Subsection 5.7.4* describes how tool calls are executed and how tool-side errors are returned to the workflow as observable results.

5.7.1 Why Builder Agents Need Tools

Builder agents need tools because several parts of the construction process require executable operations. An LLM can describe that a file should be read or a function should be implemented, but it cannot perform these actions unless the surrounding system exposes them through tools. Tools connect the agent's decision-making with controlled operations in the construction environment [4, 38, 50].

In the proposed system, tools support several construction needs. The Software Engineer uses tools to call Coder agents to implement selected functions. The Coder uses tools to submit a function-level implementation and may use external web search when extra information is needed. The File Handler uses tools to create supporting files or folders required by the produced agent.

5.7.2 Role-Based Tool Access

Tool access follows the responsibility of each agent. The system does not give every agent the same tools, because each tool expands the set of actions that an LLM can request. If an agent receives tools unrelated to its role, the chance of wrong tool selection or unnecessary side effects increases. Moreover, controlled tool access is part of broader reliability and safety design in agentic systems [41]. In addition, a wider range of available tools requires more tool descriptions and argument schemas in the model context, increasing token use and making the active context harder to manage [17, 41].

5.7.3 Tool Boundaries and Permissions

Tool boundaries define where and how tools are allowed to operate. Some tools, such as web search, are lower risk than other, more sensitive, tools that modify source code, create files, call other agents, or terminate an execution path. Tools that can change artefacts require stricter boundaries.

5.7.4 Tool Execution and Error Handling

Tool execution is controlled by the surrounding graph or runtime. The LLM produces a structured tool call that includes the tool name and arguments. The system then validates and executes the call, returning the result to the workflow as a tool response.

Errors can occur at two levels. The first level is the LLM side, where the model may request the wrong tool or produce arguments that do not match the expected schema. The system reduces these failures through typed tool schemas, argument validation, and bounded retries when the model's output cannot be parsed or validated. The second level is within the tool itself, where the tool receives a valid call but cannot complete the operation. For example, an external search service may fail. Since the tools are predefined and tested as part of the system, such failures are treated as runtime conditions. The tool returns an error response, which is passed back to the workflow so the next step can decide whether to retry, revise the request, stop, or continue with another action.

5.8 Validation, Review, and Failure Handling

Validation, review, and failure handling define how the construction workflow prevents weak or malformed artefacts from moving forward unchecked. Deterministic validation has already been discussed in S*ubchapter 2.5*, where schema checks and required-field validation were treated as reliability gates for structured outputs. The present subchapter, therefore, focuses on the design role of review and failure handling around the produced artefacts, rather than repeating the full discussion of deterministic validation.

Review is handled through two main mechanisms. Human review is used when the system needs confirmation of intention or design suitability. Similarly, LLM-based review is used to reduce the chance that errors propagate without inspection. These mechanisms reflect the reliability techniques discussed in S*ubchapter 2.10*, where self-correction, validator agents, and human intervention were presented as ways to reduce the risk that weak artefacts propagate through an agentic workflow [6, 33, 36, 41]. The two review styles are used as complementary gates. Human reviewers can confirm intention and suitability but may not be able to verify every technical detail of the generated package, whereas LLM reviewers can help detect such issues [6, 19, 21, 36, 41].

5.8.1 Human Review and Approval Gates

The human review process serves primarily as a checkpoint for intention alignment, with its role in quality assurance being secondary. It allows the user to reject or revise important artefacts before later stages depend on them. In the proposed system, human review appears at selected control points. After the Input Refiner refines the request, the user can approve or revise the refined construction request. The user can also review the proposed graph structure by the Workflow Refiner. During annotation by the Code Annotator, the user may also approve the proposals before the implementation begins. Finally, in the Prompt Engineer, the user can review the generated prompts, depending on the configuration.

5.8.2 LLM-Based Review and Judge Gates

An LLM-based review is used when another model checks an artefact before the workflow continues. These review gates can help identify weaknesses of the generated outputs. For these cases, the system uses LLM-based judgement as a controlled review

mechanism. In the Coder agent, an internal reviewer checks the proposed function implementation before it is returned to the Software Engineer. The Software Engineer also performs a QA-style final review of the implemented file. The Prompt Engineer similarly supports LLM-based review of the generated prompts.

These review gates follow the broader use of critique, self-review, validator agents, and review-based refinement discussed in S*ubchapters 2.4*, *2.9*, and *2.10* [6, 36, 41]. However, LLM reviewers are not treated as perfect judges. Their output is treated as feedback for revision, not as proof of correctness. This keeps LLM-based review useful as a design gate while preserving the need for bounded retries, human approval where needed, and later inspection of the produced artefacts.

5.8.3 Validation Outcomes and Failure Handling

Validation and review gates produce two operational outcomes. If an artefact is approved, it is passed to the next stage. If it is rejected with feedback, the producing agent revises the artefact using the provided feedback and resubmits it for another check.

These mechanisms do not guarantee that the final package is correct or production-ready. Their purpose is to create explicit decision gates during construction. Each gate decides whether an artefact should be accepted, revised, or exposed for later inspection. This reduces the chance that weak artefacts silently propagate through the workflow [33, 41], while keeping the construction process bounded and inspectable [5, 41].

5.9 Artefact Management and Traceability

Artefact management is central to the proposed system because the system produces a custom-agent package that can be inspected after construction, including the generated source code, prompt templates, and supporting files. Traceability supports the same design goal during construction. The system creates and preserves intermediate artefacts such as the refined request, workflow bundle, code scaffold, annotations, prompt proposals, review comments, logs, and file snapshots. These records make the construction path visible from the original request to the completed package. They also help identify where problems originate from.

This subchapter first explains the final package artefacts produced by the system. It then discusses the intermediate artefacts created during the construction workflow. Finally, it explains how logs and file snapshots support debugging, comparison between stages, and later inspection.

5.9.1 Final Package Artefacts

The final output of the construction workflow is a generated project folder for the produced agent. This folder contains the files needed to inspect and run the agent after construction. The core artefacts are the main Python file and the prompt file, which contains the prompt templates used by the generated agent. The package may also include supporting files created by the File Handler. These files depend on the requested agent. For example, an agent that stores preferences or records may require local JSON files. Keeping these supporting artefacts inside the generated project folder makes the package easier to inspect and prevents required runtime resources from being scattered across unrelated locations.

5.9.2 Intermediate Construction Artefacts

Intermediate construction artefacts keep a record of each step between the original user request and the final package. They are not all part of the produced agent at runtime, but they explain how the package was constructed and why later stages received the information they used. The first intermediate artefacts are created during requirement and workflow refinement. Further intermediate artefacts are created during code and prompt construction. These artefacts support the handoffs discussed in Su*bchapter 5.5*. If a problem appears in the final package, the intermediate artefacts help narrow the source of the issue.

5.9.3 Logs, Snapshots, and Traceability

Logs and file snapshots record different parts of the construction process. File snapshots are mainly useful for code-related stages because they preserve the generated source file after scaffolding, annotation, implementation, prompt generation, and file handling. They allow the user to compare how the file changed across stages. Logs preserve the returned state of each main construction agent, including the message history when messages are

part of the agent's state, reviewer comments, or tool responses that would not be visible from a code snapshot. Together, logs and snapshots make the construction process easier to inspect, debug, and compare across stages.

5.10 Summary

This chapter presented the design and methodology of the proposed system. The system was described as a staged multi-agent builder that transforms a natural-language request into an inspectable custom-agent package. Instead of generating the entire package in a single step, the builder breaks construction into refinement, workflow design, scaffolding, annotation, implementation, prompt generation, and file handling.

The chapter first defined the overall system design and its boundary. The system aims to make agent construction structured and inspectable. It then explained the role decomposition used by the builder. Each construction agent owns a specific responsibility and produces an artefact for the next stage, making the workflow easier to inspect.

The chapter also described the system's construction methodology. A user request is refined into a clearer construction input, converted into a workflow bundle, translated into a code scaffold, annotated with implementation guidance, implemented through bounded coding tasks, and completed with prompts and supporting files. Interfaces and data contracts were then presented as the mechanism that keeps information movement between stages explicit. These contracts include stage inputs and outputs, graph state, structured LLM outputs, file paths, and validated tool-call arguments.

Prompt contracts and tool access were also treated as design controls. Prompt contracts define each agent's role, input context, task boundaries, and output expectations. Tool access is assigned according to role responsibility, so agents only receive the tools needed for their part of the construction process. All tools are executed by the surrounding system rather than directly by the model, while sensitive tools are additionally bounded by the generated project context.

Finally, the chapter explained how output review, failure handling, and artefact management support the construction process. Human review is mainly used for intention alignment and design approval, while LLM-based review supports technical and structural checks. Logs and snapshots make the construction path inspectable and help locate where problems first appeared.

Chapter 6

Implementation

6.1 Introduction **71**
6.2 Repository Structure and Build Environment **72**
6.3 Main Workflow Implementation **76**
6.4 Agent Implementation **77**
6.5 Generated Package Structure **100**
6.6 Summary **102**

6.1 Introduction

Chapter 5 presented the design and methodology of the proposed multi-agent builder system. *Chapter 6* explains how that design was implemented. The focus is on the concrete structure of the implementation, including the repository layout, the main workflow entry point, the implemented builder agents, the generated agent package, and the runtime configuration used to execute the system.

This chapter is organised as follows. *Subchapter 6.2* describes the repository structure and build environment. *Subchapter 6.3* explains the main workflow implementation and the order in which the construction stages are executed, while S*ubchapter 6.4* presents the implementation of each agent and construction stage, including their inputs, outputs, prompts, tools, structured outputs, workflow nodes, and failure handling. *Subchapter 6.5* describes the structure of the generated custom-agent package. Finally, *Subchapter 6.6* summarises how the implementation maps the design of *Chapter 5* into a runnable multi-agent construction system.

6.2 Repository Structure and Build Environment

The system is implemented as a Python project with separate folders for the builder agents, generated agents, main orchestration code, and shared utility functions. The repository also includes configuration files needed to run the workflow. This subchapter describes how the implementation is organised, how the build environment is prepared, how model and tool providers are configured, and which shared utilities support the rest of the system.

6.2.1 Repository Layout

The repository is organised around distinct folders, as shown in *Figure 6-1*. The *agents/* folder contains the implemented builder agents, each placed in its own subfolder. Each agent folder follows a similar internal structure, containing the agent implementation file, its prompt file, a README file, and a *graphs/* folder for storing the visual representation of the agent's workflow. The *creations/* folder is home to the custom agents produced by the system. Each generated agent is placed in a separate subfolder under *creations/*, and normally includes the generated Python agent file, a prompt file, and any additional files created during construction. This separation keeps the builder implementation distinct from the artefacts it produces.

The *logs/* directory stores intermediate artefacts produced during debug runs. The logs are grouped by execution date, with each run stored under a date-specific folder. For each top-level workflow stage, the system records the returned output state in a text file. It also stores snapshots of the generated Python file after the stages that modify or extend it, including code scaffolding, code annotation, software implementation, prompt generation, and file handling. These logs make the construction process easier to inspect, since they show both the intermediate agent states and the evolution of the generated implementation file across the pipeline.

The *main_workflow/* folder contains the end-to-end workflow entry point. Its *main.py* file is responsible for calling the construction stages in sequence, coordinating the overall generation process, and storing any produced artefacts. The *utils/* folder contains shared

helper code used across the implementation. In particular, it includes common runtime utilities and the code-building logic used to create the initial structure of generated agents.

At the root level, the repository also includes configuration and support files. The requirements.txt file defines the Python dependencies, *.env* stores local API and provider configuration, and *.env.example* provides the expected configuration format. The repository also includes support files such as *create_agent.py*, which creates a new builder-agent folder structure under the *agents/* directory.

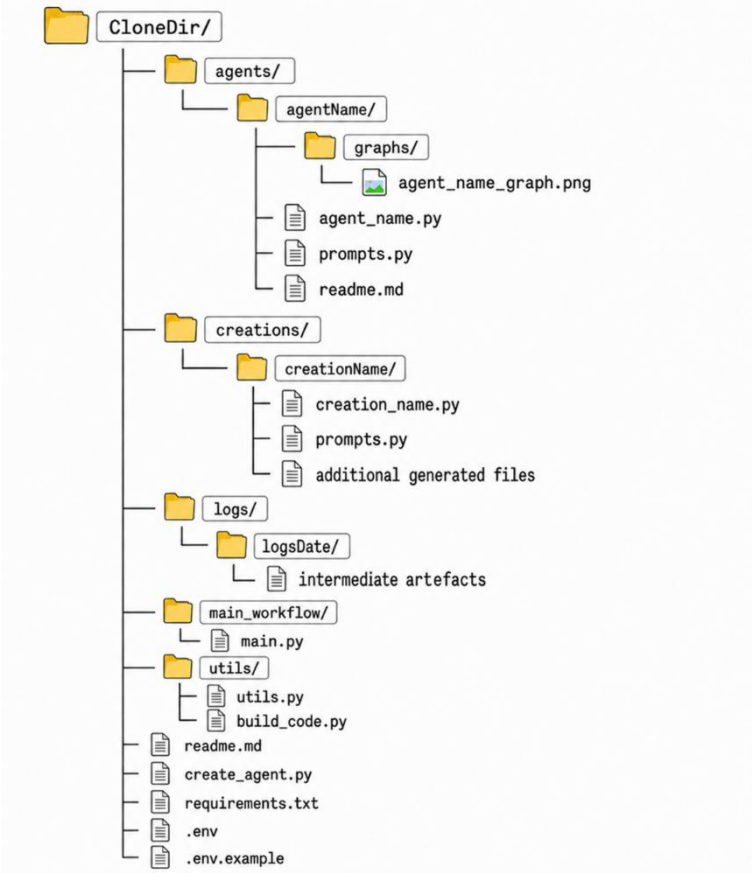

Figure 6-1. Repository structure.

6.2.2 Build Environment

The system is developed and executed in a Python environment. Python 3.11 or later is recommended, since the implementation uses modern Python typing features, Pydantic schemas, LangChain components, and LangGraph workflow definitions. The required packages are defined in the *requirements.txt* file and should be installed in a virtual environment before running the system. This keeps the project dependencies separate

from the global Python installation and makes the execution environment easier to reproduce.

The repository also requires the project root to be included in *PYTHONPATH* because the agents, workflow files, and utility modules are stored in separate folders and import one another during execution. Setting *PYTHONPATH* to the repository root allows these imports to work consistently, whether the user runs an individual agent in isolation or executes the full workflow from *main_workflow/*.

After the environment is prepared, the system can be run either at the level of an individual agent or through the complete workflow. Individual agents can be executed for development and testing, while the full construction process starts from the main workflow entry point. In a complete run, the system reads the user request, invokes the construction stages in sequence, and writes the generated agent package under the *creations/* directory.

6.2.3 Environment and Provider Configuration

The system uses environment variables to configure model access, LangSmith traceability, and external services such as web search through an API. These variables are stored locally in the *.env* file, while the *.env.example* provides the expected structure without exposing private keys. The environment file keeps sensitive configuration values outside the source code and allows the same implementation to run with different model providers or settings with minimal changes.

The main provider configuration is based on the variable *PROVIDER*. The selected provider is then used to read the corresponding API key and base URL through the environment variables *{PROVIDER}_API_KEY* and *{PROVIDER}_BASE_URL*. The model is defined through the variable *MODEL_NAME*. This structure keeps API keys outside the codebase, making it easier to switch to a different model or provider with minimal changes.

The environment file also stores optional tool and observability settings. For example, *TAVILY_API_KEY* is used by agents that can perform web search, while LangSmith-

related variables can be used to trace and inspect workflow executions. The *DEBUG* flag controls whether additional runtime information is printed and whether intermediate logs and file snapshots are stored.

6.2.4 Shared Runtime Utilities

The *utils/* folder contains shared code used across the builder agents and the main workflow. These utilities are used to avoid repeating common runtime logic inside every agent.

A central utility is the custom *myChatOpenAI* wrapper. This class extends LangChain's standard *ChatOpenAI* interface and reads the base URL, API key, and model name from the environment configuration. Moreover, this wrapper takes the model temperature as an argument. As a result, model instances can be created in a consistent way without hardcoding provider-specific details for each LLM instance.

Another crucial utility is the *safe_invoke* function, which wraps LLM calls with retry and error-handling logic. It retries recoverable failures such as rate limits and Pydantic validation errors when appropriate. The *safe_invoke* utility is vital because the full workflow relies on numerous LLM calls, and a temporary API failure should not immediately terminate the construction process.

The utilities also include helper functions for tool and output handling. The *will_tool_call* function checks whether a model response intends to call a tool, while *parse_tool_arguments* attempts to normalise and parse tool-call arguments into valid JSON. Other helpers include *clean_llm_output*, which removes unwanted markdown or wrapper tags from model-generated code, and *read_state_file*, which loads the current generated Python file from the file path stored in the agent state. These functions support the agents that read, modify, review, or execute changes on generated code.

Finally, the shared utilities include debugging and helpers to identify approvals. The *print_function_name* function prints the currently executing function when debug mode is enabled, making the execution path easier to follow.

6.3 Main Workflow Implementation

The end-to-end construction workflow is implemented in *main_workflow/main.py*. While the overall construction pipeline was presented at the design level in *Chapter 5*, the implementation in *main.py* defines how this pipeline is executed in practice. The file acts as the central entry point of the system and is responsible for receiving the user request, invoking the builder stages in order, passing intermediate outputs between them, and recording debug artefacts when debug mode is enabled.

The *main()* function accepts three main inputs. The first is *user_request*, which contains the natural-language request for the custom agent. The second is the orchestrator flag, which controls whether clarification questions are routed through the Clarification Orchestrator first. The third is *prompt_review_mode*, which determines how the Prompt Engineer reviews generated prompts. The available modes allow review by the LLM, by the user, or by both.

Before invoking the agents, the workflow creates a runtime configuration for each stage that is compatible with LangGraph execution and LangSmith-based trace inspection. This configuration includes a recursion limit and configurable metadata, namely *user_id*, *run_name*, and *thread_id*. The recursion limit defines the maximum number of graph steps that an agent invocation may execute before being stopped, preventing accidental infinite loops or uncontrolled graph execution. This configuration parameter is set to 150 steps to give agents that require repeated tool use or review loops enough room. The *user_id* identifies the logical user or calling context of the run, while *run_name* provides a readable name for the stage in LangSmith. The *thread_id* identifies the specific execution thread used by LangGraph's state and checkpointing mechanisms when enabled. In this implementation, the *thread_id* is built using the agent's name, the execution date, and a generated UUID. This separates each agent invocation and minimises the chance of state collisions across workflow runs. The code implementation of the configuration is shown in *Figure 6-2*.

```python
uuid_: str = str(uuid.uuid4())
date: str = datetime.now().strftime('%d-%m-%y')
config: Callable[[str], Dict] = lambda agent_name: {
    'recursion_limit': 150,
    'configurable': {
        'user_id': 'main',
        'run_name': 'main',
        'thread_id': f'main:{agent_name}:{date}:{uuid_}',
    }
}
```

Figure 6-2. Configuration per agent invocation.

The implemented execution follows the construction order defined by the system design. The Input Refiner first receives the raw user request and returns a refined specification. The Workflow Refiner then converts the refined request into a WorkflowBundle. The *create_file* function uses this bundle to create the initial generated Python and prompt files. For each generated file, the workflow then invokes the Code Annotator, Software Engineer, Prompt Engineer, and File Handler in order.

When debug mode is enabled, the main workflow stores additional artefacts for inspection. The *print_to_file* helper stores the agents' output states in text files, while *copy_file* stores snapshots of the generated Python file after an agent that can modify it finishes its execution. These logs make it possible to inspect how the generated implementation changes after scaffolding, annotation, software implementation, prompt generation, and file handling, making debugging and later analysis of the construction process easier.

6.4 Agent Implementation

The main workflow is implemented through a set of specialised agents and construction stages. Each component is responsible for a specific part of the agent-building process. Although the components serve different purposes, most implemented agents follow a similar structure. They define an input schema, initialise one or more LLM instances, define the workflow graph, use prompt templates to control model behaviour, and

compile the workflow as a LangGraph graph. Some agents also bind tools to their LLMs, while others rely on structured outputs to constrain the model response into a predefined schema. The following subsection first describes this common implementation pattern. The remaining subsections then explain each agent and construction stage in the order used by the main workflow.

6.4.1 Common Implementation Pattern

The agents are implemented as LangGraph workflows. Each agent defines a StateGraph around an input state schema that describes the data available during execution. Depending on the agent, the schema may be a TypedDict, a Pydantic BaseModel, or LangGraph's MessagesState. The state typically contains the user request, the current file path, previous messages, intermediate outputs, or agent-specific fields required by the workflow. MessagesState is used when an agent needs to maintain a message history, since it already defines the standard messages key used by LangGraph message-based workflows. The state schema acts as the local data contract for the agent.

Each graph is composed of nodes and edges, as discussed in *Subchapter 2.7*. After the nodes and edges are defined, each graph is compiled into an application object that can be invoked by the main workflow or by another agent.

The agents use prompt templates stored in their own *prompts.py* files. These templates define the model's role, the task it must complete, the input context it receives, and the expected response format. During execution, the relevant prompt is formatted with values from the current state, such as the clarified user request, the generated code file, or previous review comments.

Several LLM calls use structured output schemas. In these cases, the model instance is configured with *.with_structured_output(...)*, so the response must match a Pydantic schema or equivalent structured model. Structuring the output is applied when later stages need machine-readable outputs, such as workflow graphs. Although structured outputs reduce ambiguity at handoff points, they do not guarantee that the generated content is semantically correct.

Some LLM instances are configured with *.bind_tools(...)*, giving the model access to specific functions exposed through LangChain tool interfaces. Tool access is assigned based on the agent's responsibilities. The model decides when to request a tool call, but the tool's execution is performed by the surrounding system.

This repeated implementation pattern gives each agent a consistent code structure while still allowing each one to define its own state fields, LLM calls, tools, nodes, routing rules, and failure-handling behaviour.

6.4.2 Input Refiner

The Input Refiner is the first agent invoked by the main workflow. Its role is to transform the raw user request into a refined construction input by resolving missing information through clarification and producing a clearer agent-creation-ready description that can be passed to the Workflow Refiner.

The agent defines three schemas. The input schema is a TypedDict with two fields: the raw user request and the orchestrator flag. The internal intermediate schema extends LangGraph's MessagesState, allowing the graph to store a message history during execution. It also stores the orchestrator flag, the clarification question-answer list, the corrected original request, previous refinement attempts, and user requests made during the refinement review. The final output schema is a Pydantic model with two fields, namely, the corrected original request and the user-approved refined text.

The Input Refiner uses three LLM instances. The correction model uses 0 temperature and produces a minimally corrected version of the original request. The clarification model uses a temperature of 0.8 and is bound to the Tavily search tool, allowing it to request external context when the request or the user's answers are not clear enough. The refinement model uses a temperature of 0.7 and produces the final agent-creation-ready request after clarification is complete.

The graph contains five nodes, as shown in *Figure* 6-3. Execution starts at the node correct, moves to clarify, and then the flow is conditional. If a tool call is needed, the graph moves to tools and then returns to clarify with the tool result. If more clarification

is needed, the graph loops back to clarify to continue the process. If the model decides that no further clarification is needed, the execution moves to refine. After a refinement attempt, the user either approves the result, in which case execution moves to *parse_output*, or requests changes, in which case the graph repeats the refinement step.

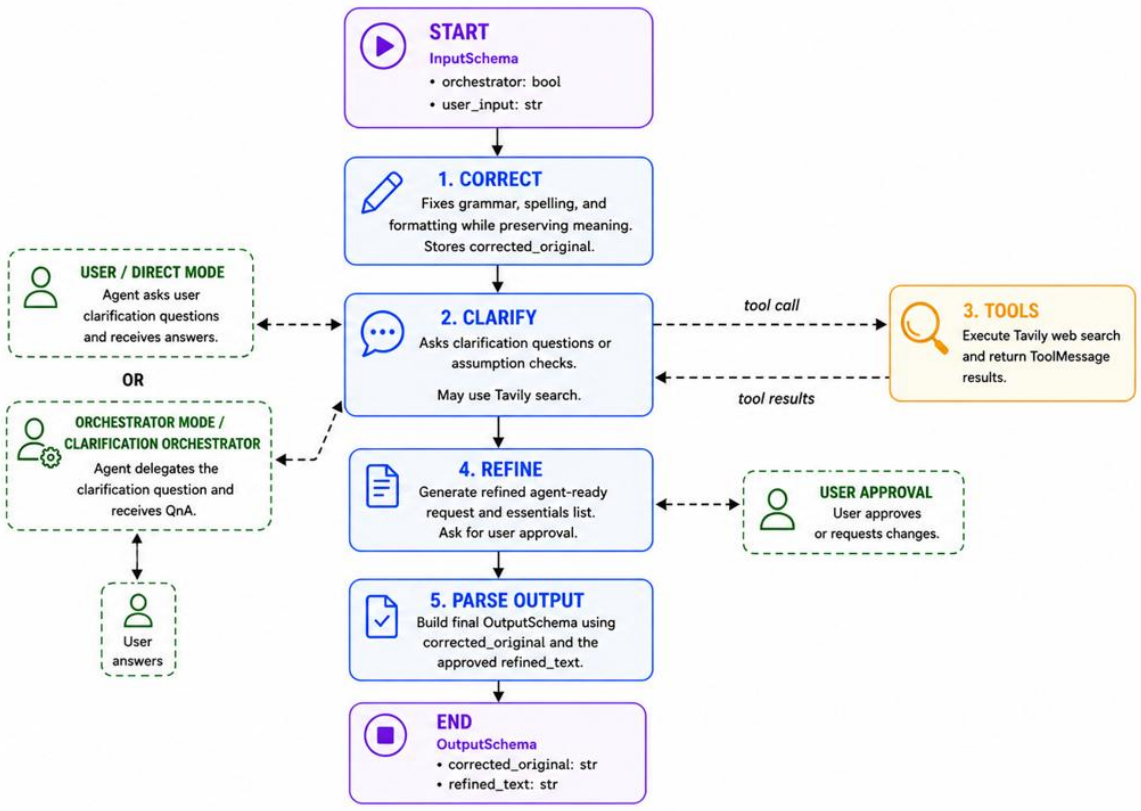

Figure 6-3. Input Refiner's Workflow

The correction node receives the raw request, formats the correction prompt, and invokes the correction model. The prompt instructs the model to fix grammar, spelling, and formatting while preserving the original meaning, vocabulary, and structure. The returned text becomes the corrected request stored in the workflow state. If correction fails, the node falls back to the original user request so the workflow can continue.

The clarification node receives the corrected request, previous clarification history, and any previous tool results. It formats the relevant prompt and calls the clarification model. The prompt asks the model to decide whether the request is clear enough for refinement, whether a clarification or assumption check is needed, or whether web search should be used. If the model requests the tool, the graph routes execution to the tool node, which

executes Tavily and returns the result to the clarification node. If the model asks a clarification question, the answer is collected either directly from the user or through the Clarification Orchestrator, depending on the orchestrator flag. When the Clarification Orchestrator is used, it is invoked with a fixed thread ID. The same thread ID is also used by the Workflow Refiner, so answers collected during one refinement stage can remain available to the other. In practice, this thread ID allows the Clarification Orchestrator's checkpointed state to be reused across invocations within the same session, provided the thread ID is the same.

The refinement node runs when the clarification model determines that the collected information is sufficient. It formats the refinement prompt using the conversation history of the clarifications, previous refinement attempts, and any user change requests. The prompt requests the model to produce one agent-creation-ready request, including a clear refined paragraph and a short list of construction essentials such as role, scope, inputs, outputs, constraints, and any user preferences. The refined response is shown to the user for approval. If the user approves it, the workflow continues to the output node. If the user requests changes, the workflow repeats the refinement node and includes the latest request in the next prompt.

The final node just returns the corrected original request and the approved refined text in the agent's output schema. The workflow then terminates, and the agent returns these values to the main workflow. The main workflow uses the refined text as the clarified construction input for the Workflow Refiner.

6.4.3 Workflow Refiner

The Workflow Refiner receives the refined construction input produced by the Input Refiner and converts it into a buildable workflow representation. Its role is to clarify only workflow-related uncertainties and produce a structured workflow that can be used by the code scaffolding stage.

The agent defines several schemas for representing the produced workflow. A workflow graph needs nodes and edges. A node has a name, a description, and optionally a subgraph reference. Similarly, an edge contains the names of both the source node and target node,

as well as a description of the transition. A workflow graph contains the workflow type, as discussed in S*ubchapter 2.7*, a memory flag indicating whether the generated workflow relies on persisted state, a graph name, a description, and finally a list of nodes and edges. The workflow bundle then combines the root workflow graph with any optional subgraphs. The input schema extends LangGraph's MessagesState, so it can store the message sequence and tool results. It also includes the orchestrator flag and the refined request received from the Input Refiner. The output schema contains the final workflow bundle.

The Workflow Refiner uses two LLM instances. The clarification model uses a temperature of 0.7 and is integrated with the Tavily search tool, allowing it to request external context when the workflow shape depends on information not already available. The workflow engineer model also uses a temperature of 0.7 but is configured with a structured output. Its response is validated against the workflow bundle schema.

The graph contains three nodes: clarify, tools, and *create_workflow*. Execution starts at the clarification node. Similar to the Input Refiner, if the clarification model requests a web search, the control moves to the tools node and then returns with the tool result. If more clarification is needed, the graph loops back to the clarification node. Once the workflow requirements are clear, execution moves to the workflow creation node. This final node produces the final workflow bundle, after which the graph terminates. The graph is visualised in *Figure 6-4*.

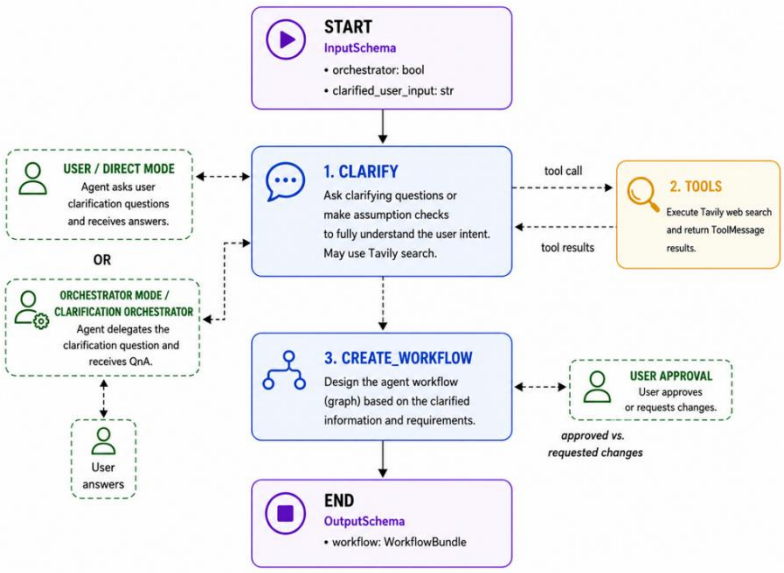

Figure 6-4. Workflow Refiner's Workflow

The clarification node receives the refined request and the current clarification history. It formats the workflow-clarification prompt and invokes the clarification model. The prompt focuses the model on how the generated agent should run, rather than on domain content. It asks the model to resolve workflow-level issues, such as the trigger, workflow type, memory needs, user interaction pattern, and tool use. If the model requests web search, the graph routes execution to the tool node, which executes Tavily and returns the result. If the model asks a clarification question, the answer is collected directly from the user or through the Clarification Orchestrator. As in the Input Refiner, the orchestrator is invoked with the fixed clarification thread ID, allowing checkpointed clarification state to be reused across the refinement stages of the same session.

The workflow creation node runs after workflow clarification is complete. It builds a conversation history from the messages stored in the state and formats the workflow-creation prompt. That prompt instructs the workflow engineer model to produce a concrete graph representation. Within the workflow bundle, the model is prompted to include execution tags in the node descriptions. These tags help subsequent stages form the code structure.

The workflow engineer returns the workflow bundle through structured output. After the model produces a workflow, the result is shown to the user for approval. If the user requests changes, the workflow creation node stores the previous workflow attempt together with the user's request and repeats the generation step. If the user approves the result, the node performs a minor post-processing step before returning the output. It checks the root graph and any subgraphs and adds missing start or end nodes and their connecting edges when needed. This keeps the produced graph compatible with the scaffolding stage, even if the model omitted these boundary nodes.

Failure handling is kept local to each node. If clarification fails, the clarification node returns the existing state as is, allowing the graph to continue the clarification loop rather than stopping immediately. During workflow creation, schema-related failures can trigger another generation attempt. Other recoverable model errors are handled by the safe invocation utility through its retry mechanism. If an error is still raised after that handling,

it is treated as unrecoverable for that node, and the run requires inspection because a valid workflow bundle may not have been produced.

The final output of the Workflow Refiner is the workflow bundle that was accepted by the user. The main workflow stores this bundle and passes it to the code scaffolding function. At that point, the construction process moves from a refined natural-language request to a concrete graph structure that can be translated into initial Python files.

6.4.4 Clarification Orchestrator

The Clarification Orchestrator is a support agent used by the Input Refiner and Workflow Refiner. Its role is to answer clarification questions from previously collected user answers when possible, reducing the need to ask the user the same or similar question again.

The agent defines three main schemas. The QnA schema stores a clarification question, its answer, and a justification for that answer. The input schema contains the current clarification question and the accumulated question-answer memory. The output schema returns the final QnA item created during the invocation. The agent also defines a coordinator schema for the LLM response, which includes a confidence score, any questions answered from memory, and any questions that still require user input because the agent could not infer the answer from the memory.

The agent uses one LLM instance. The coordinator model uses temperature 0 to make responses more deterministic, reducing unsupported inference. The coordinator is also configured with structured output, so its responses are validated against the coordinator schema. The prompt instructs the model to answer only from prior human-provided information and to leave unsupported questions unanswered rather than guessing.

The graph is small and contains two nodes: *answer_question* and *return_answer*. Execution starts with the answer node, as shown in *Figure 6-5*. This node formats the prompt with the new clarification question and converts previous QnA memory into messages. Prior questions are represented as AI messages, while prior answers are represented as human messages, so the model is guided to treat only user-provided

answers as valid evidence. The coordinator model is then invoked through the safe invocation utility.

The model response determines whether the user must be asked directly. If the model returns a confidently supported answer, it is used. If confidence is low or some parts remain unanswered, the node asks the user for the missing information and appends the new response to the stored QnA memory. If structured-output validation fails, the node also falls back to asking the user directly. Any newly obtained answer is stored with a justification, so later clarification questions can reuse it.

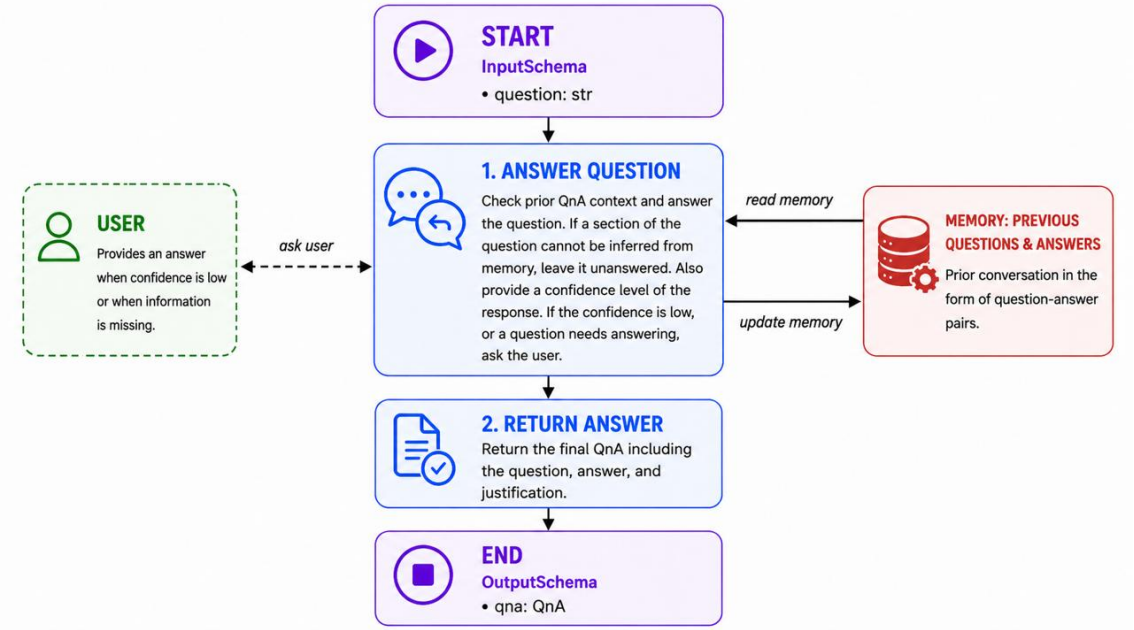

Figure 6-5. Clarification Orchestrator's Workflow

The return node simply returns the latest QnA item from the state. The graph is compiled with a MemorySaver checkpointer, allowing accumulated clarification answers to persist across invocations when the same thread ID is used. This is why the Input Refiner and Workflow Refiner invoke the orchestrator with the same clarification thread ID during the same construction session.

6.4.5 Code Scaffolding

Code Scaffolding is the first fully deterministic construction stage after the Workflow Refiner. Its role is to convert the accepted workflow bundle into the initial files of the

generated agent. Unlike the previous stages, it is not implemented as a LangGraph agent and does not call an LLM. It is implemented as a utility function that receives the workflow representation, writes the first version of the generated Python file or files, and creates the corresponding prompt files.

The main input to this stage is the workflow bundle produced by the Workflow Refiner. The function accepts the workflow object directly and converts it into a dictionary before code generation begins. The output of the stage is a list of generated Python file paths, which are returned to the main workflow so that later stages can process each generated file.

First, the workflow bundle is passed to an internal workflow-building function. This function reads the root workflow graph and any subgraphs and creates a code string for each graph based on the name provided by the Workflow Refiner. Second, the file-creation function creates files with these code strings in the *creations/* directory. The folder name is derived from the root workflow name, so each generated agent package is placed under its own project folder.

The generated Python file follows a fixed template with sections for imports, schemas, tools, LLM definitions, helper functions, nodes, conditional functions, graph construction, and graph compilation. Many sections initially contain placeholders, giving later stages a stable structure to annotate and complete.

The workflow nodes are transformed into Python node functions. Start and end nodes are not converted into ordinary functions, because they are represented through LangGraph's START and END constants. For the remaining nodes, the scaffold generator checks the node description. If the description indicates an LLM-based or subgraph-based step, the generated function includes placeholders for prompt formatting, invocation, post-processing, return values, and error handling. If the node represents code-only execution, the generated function contains just a placeholder to indicate implementation is needed. In both cases, the node description becomes the function docstring, so the implementation guidance produced by the Workflow Refiner is preserved inside the generated code.

For nodes whose descriptions indicate model use, the scaffolding stage also creates initial LLM declarations using the shared model wrapper. It also inserts placeholder prompt calls based on the node names and creates an empty prompt file with the same base name as the generated Python file. The prompt content is not written at this stage; it is completed later by the Prompt Engineer.

Edges in the workflow bundle are translated into LangGraph edge definitions. If a node has only one outgoing edge, the scaffolding stage creates a normal graph edge. If a node has multiple outgoing edges, the stage creates a conditional routing function and connects it through conditional edges. The conditional function is initially left with placeholders for the actual routing logic, because later stages must decide how the generated graph should inspect the state and choose the next node. Start and end edge names are also normalised so that they map correctly to LangGraph's START and END.

The result of Code Scaffolding is an initial project folder containing scaffolded Python files and matching prompt files. The main workflow then passes each generated Python file to the Code Annotator, where the structural scaffold is enriched with implementation guidance, before sending it to the remaining implementation stages, one file at a time. The Python files created can be viewed as a shared-memory component of the agents, since they have full read access, and selected agents have restricted write access.

6.4.6 Code Annotator

The Code Annotator receives the scaffolded Python file produced by Code Scaffolding and enriches it with implementation guidance. Its role is not to complete the code, but to make the scaffold more informative before the Software Engineer starts implementation. It does this by adding node docstrings, schemas, helper-function stubs, tool-function stubs, and LLM configuration changes.

The input schema extends LangGraph's MessagesState, so the graph can store user feedback during each approval loop. It also contains the path of the file being annotated, the refined user request, the workflow bundle, and the temporary proposal for the current step. The graph does not define a separate output schema, since this agent's output is the side effect of updating the generated Python file in place.

The Code Annotator uses five LLM instances, all configured with structured output. The docstring generator uses a temperature of 0.5 and is responsible for proposing docstrings for the generated node functions. The schema generator uses a temperature of 0.2 and returns the schemas to be used within the workflow, including the Main Agent state schema, which is analogous to the combined input and output schemas. The helper-function and tool-function generators also use 0.2 for temperature and return proposed function stubs, so they get implemented in the following steps of the construction process. The LLM-modifier generator uses temperature 0.8 and returns proposed model configurations, including temperatures, tool bindings, and structured-output bindings.

The graph follows a repeated proposal-approval-update pattern, as shown in *Figure 6-6*. Each proposal node reads the current generated file, formats its prompt using available information, the current code, and any user feedback provided during the process, then invokes its structured-output model. The proposal is shown to the user for approval. If the user approves, the graph moves to the update node for that step, which applies the accepted changes directly to the Python file. If the user requests some changes, the graph repeats the same proposal node using the latest feedback. After each successful update, the agent clears the temporary messages and current proposal before moving to the next group.

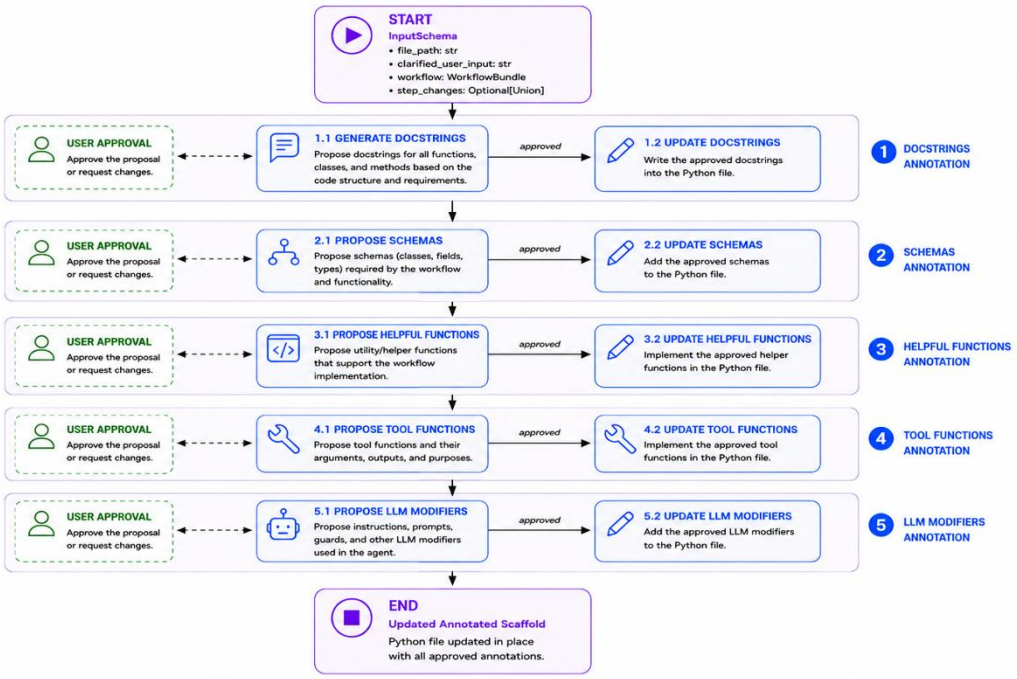

Figure 6-6. Code Annotator's Workflow

The first group concerns node docstrings. The docstring proposal node asks the model to inspect the scaffolded node functions and propose detailed docstrings for them. The prompt guides the model to focus on what each node should do, what state values it should read and return, and which helper functions or tools may be needed. If the user approves the proposal, the update node reads the target file, isolates the Nodes section, and injects the generated docstrings into the existing node functions. This turns the scaffolded nodes into more detailed implementation instructions for the next stage.

The second group concerns schemas. The schema proposal node reads the newly annotated file and asks the model to propose the schemas needed by the generated agent. The prompt requires the main Agent State schema to be included and asks for any supporting schemas needed by complex state values. If approved, the update node replaces the Schemas section of the generated file with the proposed schema definitions. The main state schema is placed last, so it can refer to any supporting schemas defined before it.

The third group concerns helpful functions. The proposal node prompts the model to identify deterministic helper functions that would clarify or reduce repetition in the node implementations. These are ordinary Python functions, not LLM tools. If approved, the update node inserts the proposed function stubs under the Helpful Functions section. The inserted functions include signatures and docstrings, but no implementations, because the actual coding is handled later by the Software Engineer and Coder agents.

The fourth group concerns tool functions. The model is asked to identify tool functions that LLM nodes may need to call. If approved, the update node inserts the proposed stubs under the Tools section and decorates them as LangChain tools. As with helper functions, the tool bodies remain placeholders at this stage.

The final group concerns LLM modifiers. Before invoking the LLM-modifier model, the node detects the current LLM names, tool names, and schema names from the file. The model then proposes the correct temperature and whether each LLM should use tool binding, structured output, or neither. The node validates the proposal against the detected

names, so an undefined LLM, tool, or schema is not silently accepted. If the user approves the valid proposal, the update node replaces the LLM section with the updated model definitions.

Failure handling is mostly local to each proposal node. If a proposal fails, the node returns the current state so the same stage can be retried. If the user does not approve a proposal, the graph loops back to the same proposal node with the user's feedback. For LLM modifiers, additional validation prevents references to undefined LLMs, tools, or schemas before the update is applied.

The result of the Code Annotator is an annotated generated Python file. The main workflow then passes this file to the Software Engineer, which uses the annotated file to begin implementation.

6.4.7 Software Engineer

The Software Engineer receives the annotated Python file and turns it into an implemented file by coordinating Coder subagents and applying accepted code changes. Its role is to manage the implementation process rather than write all the code directly. It calls coders for function-level implementation, reviews their returned code, approves or rejects outputs, applies approved changes to the file, manages imports, and runs a final QA pass before completion.

The input schema extends LangGraph's MessagesState, allowing tool calls and tool results to be stored as messages during the implementation loop. It also contains the file path of the generated Python file and the number of QA review cycles already performed. The agent does not return a structured output; instead, it updates the Python file in place.

The Software Engineer also uses several internal schemas and global runtime stores. The coder-output schema extends the Coder's output with approval and rejection flags. A separate comment schema stores the feedback given by the Software Engineer on rejected coder outputs, and the code-issues schema stores QA feedback from the final reviewer. These values provide the agent with short-term memory, allowing the agent to remember

which functions are waiting for approval, which were rejected, which imports were requested, and which QA issues remain unresolved during the current run.

The agent uses three LLM instances. The tool-section adder uses a temperature of 0.4 and is responsible for preparing the generated file for proper tool handling before implementation begins. The main Software Engineer model also uses temperature 0.4 and is bound to the implementation tools. These tools allow it to call a Coder, approve or reject coder outputs, apply approved code, add imports, make small local replacements, submit the file for review, and mark QA issues as resolved. The QA model uses temperature 0.6 and is configured with structured output, so its response is validated against the code-issues schema.

The bound tools define the actions available to the Software Engineer model. The *call_coder* tool invokes a Coder subagent for one existing function at a time. The *approve_function_code* tool writes an accepted coder implementation into the target file, while *disapprove_and_comment_on_coder_code* stores rejection feedback so the same function can be attempted again with more context. The *approve_function_proposals* tool inserts accepted helper-function or tool-function proposals of the Coders into the generated file. The *add_imports* tool adds requested import lines while avoiding exact duplicates, and the *replace_code* tool supports small local edits that do not require another coder call. Finally, *submit_final_code* signals that the file is ready for QA review, and *code_issue_resolved* removes issues from the current QA issue list after the Software Engineer has addressed them. If issues are unresolved, the tool *submit_final_code* will return an error prompting the Software Engineer to resolve them.

The graph contains five nodes, as shown in *Figure 6-7*. Execution starts at the tool-section node and then moves to the main Software Engineer node. From there, the flow depends on the model's next tool call. If no tool call is produced, the graph returns to the Software Engineer node. If an approval or import-related tool is called, execution moves to the custom approval tool node. It needs a custom tool node because the provided LangGraph ToolNode executes all tool calls in parallel, and the two mentioned tools modify the file, which could cause race conditions. If any other tool is called, execution moves to the standard ToolNode. If the model calls the final submission tool, the flow moves to the

QA node. Both tool-execution paths return to the Software Engineer node, while the QA node either ends the workflow if no issues were raised or the review limit has been reached. Otherwise, if issues exist in the code, the Software Engineer node regains execution to resolve them.

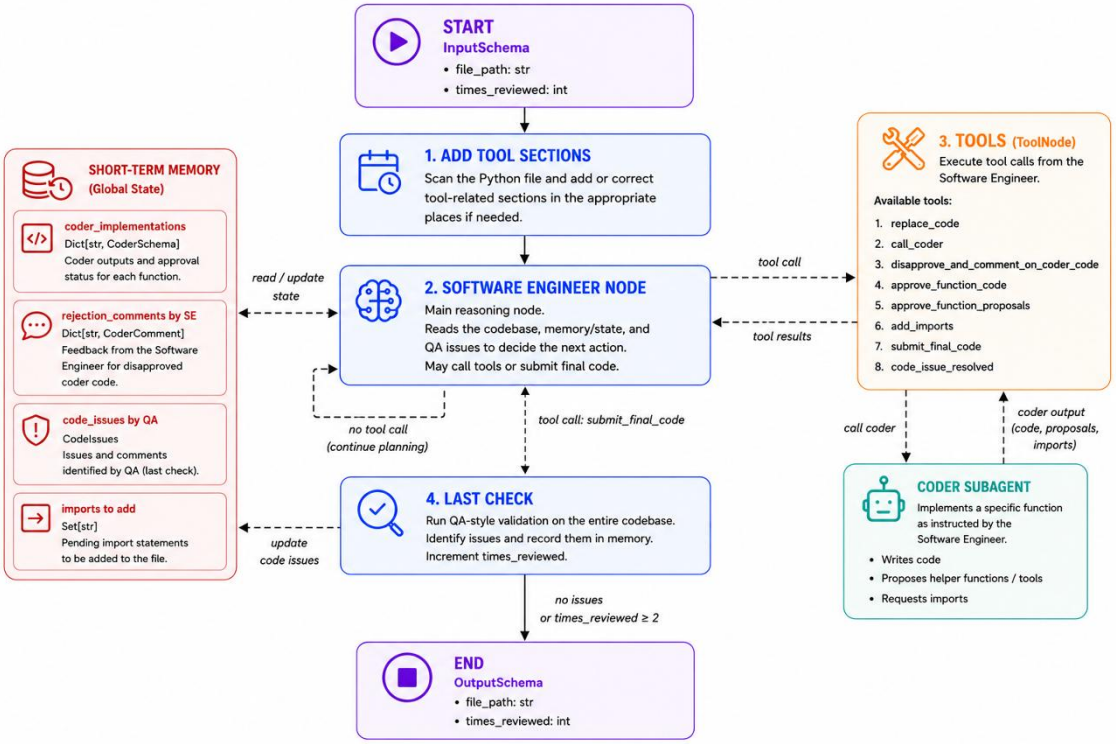

Figure 6-7. Software Engineer's Workflow

The first node prepares the file for tool-aware execution. The prompt instructs the model to modify only the tool-handling structure of the file. This includes missing tool nodes, conditional routing, and custom handler stubs when required. The specific step is crucial because the tool functions were created after the workflow was defined; hence, the graph does not yet support tool calling.

The main Software Engineer node controls the implementation loop. It reads the current file and formats the main Software Engineer prompt with the file contents, recent tool messages, pending coder outputs, rejected coder outputs, collected import requests, and any QA issues. The model then decides the next action through tool calling. Most substantive implementation work is delegated through the Coder tool, while the Software Engineer is instructed to use direct replacement only for small local edits.

The Coder tool is the main mechanism used by the Software Engineer. When called, it checks that the requested function exists in the target file, prepares the Coder input, and invokes a new Coder instance. The Coder receives the file path, the function name, and the Software Engineer's instructions. If the specific Coder provided an implementation that was rejected, the previous attempts and the rejection comments are also provided in-context. Its returned implementation is stored in the Software Engineer's runtime state rather than being written to the file immediately. The Software Engineer must then review the result and either approve it or reject it with constructive feedback.

The tool-execution nodes apply the Software Engineer's decisions. The custom approval-tool node executes tool calls that apply approved code, create stubs for approved proposed helper or tool functions, or add imports. It also prevents repeated import additions within a short recent-message window, reducing the risk of import spam. Other tool calls are handled through LangGraph's standard ToolNode. Tool results are returned as tool messages, and the graph then returns to the Software Engineer node, allowing the model to continue from the updated state.

The approval tools are the point where coder outputs become part of the generated file. When the Software Engineer approves a function implementation, the corresponding tool locates the existing function section in the target file and replaces it with the Coder's implementation. If the Coder proposed additional helper functions or tools, the Software Engineer may approve them separately, and they are inserted into the appropriate sections of the file. Requested imports are tracked during the implementation loop and can be inserted into the import block through the relevant tool, while duplicate imports are avoided.

The QA node runs after the Software Engineer submits the file as complete. It formats the final-check prompt using the current file contents and invokes the QA model with structured output. The model returns general comments and, if any, a list of concrete code issues. The node stores the issues and increments the review count. If no issues are reported, the workflow ends. If issues are raised and the file has not reached the review limit, control returns to the Software Engineer node so the model can address the issues

and mark them as resolved through a tool. The workflow also terminates once the review count reaches the defined limit.

Failure handling is local to the nodes and tools. Tool failures are returned as tool messages or error strings, allowing the Software Engineer model to react in the next step. The Coder tool also handles the case where a requested function does not exist by returning an error-like coder response instead of invoking a coder on an undefined target. The final QA loop is bounded by the review count, so the workflow does not continue indefinitely when issues remain unresolved.

The result of the Software Engineer stage is the implemented Python file. The file is not guaranteed to be perfect or error-free, but it has passed through function-level coder delegation, approval or rejection of coder outputs, and a bounded QA loop. The main workflow then passes the implemented file to the Prompt Engineer, which completes the prompt artefacts required by the generated agent.

6.4.8 Coder

The Coder is a subagent invoked by the Software Engineer. Its role is to implement one existing function inside the generated Python file and return a candidate implementation for review.

The input schema extends LangGraph's MessagesState, allowing the graph to store tool results and model responses during the coding loop. It also contains the file path, the function name to implement, the Software Engineer's instructions, and the previous outputs and the Software Engineer's comments, if any. The schema also defines the latest submitted implementation and the reviewer's comments for a review loop before the final implementation is submitted. The output schema is a Pydantic model containing the implemented function code, optional helper or tool function proposals, and optional import lines. A function-proposal schema is also defined, which is used when the Coder wants to request an additional helper function or tool needed by the implementation.

The Coder uses three LLM instances. The brainstormer uses a temperature of 0.8 and proposes possible implementation approaches before coding begins. The main coder

model uses temperature 0.5 and is bound to Tavily search and the output tool. Tavily can be used once for external reference, while the output tool is used to submit the final function implementation, proposals, and imports. The reviewer uses a temperature of 0.2 and checks the submitted function before it is returned to the Software Engineer.

The graph contains six nodes, as shown in *Figure 6-8*. Execution starts with the brainstorming node and then moves to the coder node. If the model requests Tavily, the graph executes the search and returns to the coder node. If the model submits the implementation through the output tool, execution moves to the output-parsing node. In contrast, if the model only responds in plain language, the graph loops back to the coder node. After submission, the implementation is reviewed by the reviewer model. If the reviewer accepts it, the graph moves to the output node and terminates; otherwise, if the reviewer raises an issue, the flow returns to the coder node with reviewer feedback.

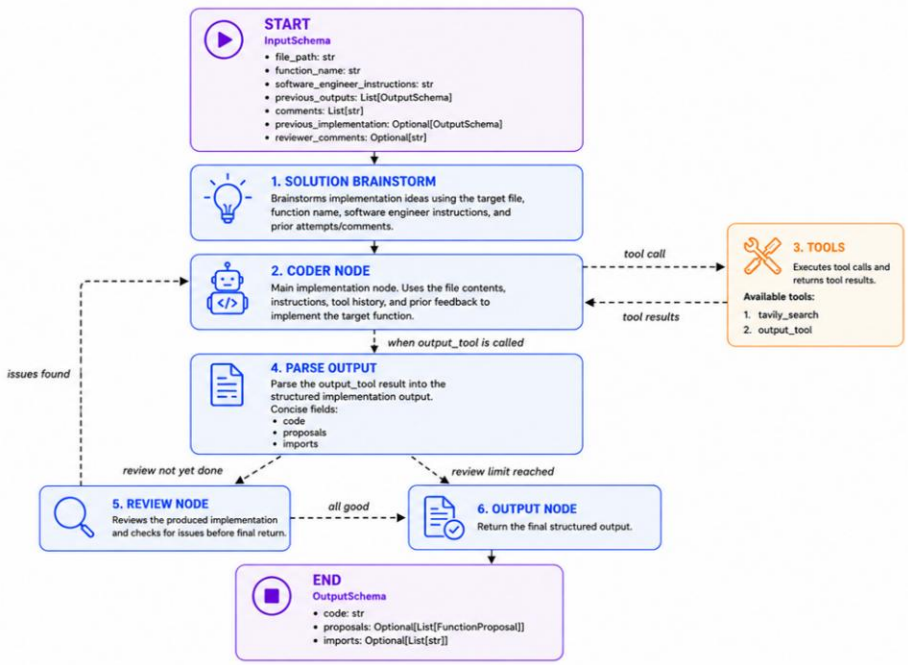

Figure 6-8. Coder's Workflow

The brainstorming node is used to propose a small number of possible implementation approaches before actual coding begins. The result is stored as a message and becomes context for the coding node.

The coder node formats its prompt using the full file contents, the Software Engineer instructions, the possible implementations suggested by the brainstorming model, prior

attempts and comments by the Software Engineer, the tool history, and reviewer feedback when available. The prompt instructs the model to implement only the requested function, avoid inline imports, respect the generated agent's state schema, and submit the final result through the output tool. If more context is needed, the model may use Tavily. If the model is ready, it calls the output tool with the function implementation, optional proposals, and optional imports.

The output-parsing node handles the final submission. It extracts the *output_tool* call, normalises missing proposal or import fields, removes inline import lines when they are also listed separately, and stores the submitted implementation in the state. The review node then checks the implementation against the current file and the Software Engineer's instructions. If the reviewer approves, the output node returns the implementation. Otherwise, the review is passed back into the coder node for another attempt.

The Coder graph is compiled with a memory checkpointer, and the Software Engineer invokes it using the function name as the thread identifier. This allows repeated invocations of the same function to retain all prior implementation attempts by the Coder, along with the corresponding rejection comments.

The result of the Coder stage is a candidate implementation for one function, together with any proposed helper functions, tools, or imports. These values are returned to the Software Engineer, which decides whether to approve the code, reject it with feedback, or approve any proposed additions before updating the generated Python file.

6.4.9 Prompt Engineer

The Prompt Engineer receives the implemented Python file and creates the prompt templates required by the agent. Its role is to extract the prompt names used in the code, generate the missing prompt templates, review and test them, and write the final prompts into the generated prompt file.

The input schema is a TypedDict containing the file path, the list of prompts, the active prompt index, an error flag, and the review mode. The review mode can be set to LLM review, user review, or both. The agent also defines a Prompt schema, which stores each

prompt name, the suggested prompt, formatting-related code changes, comments, the latest test response, and review counters.

The Prompt Engineer uses five LLM instances. The prompt-generation model uses a temperature of 0.4 and generates a prompt for the active prompt name. The prompt-review model uses a temperature of 0.25 and checks the generated prompt. The formatter uses a temperature of 0.8 and a structured output to produce formatting values for the prompt that comply with the *.format(...)* method. The tester uses a temperature of 0.7 and generates a mock response from the formatted prompt created by the formatter. The response-review model uses a temperature of 0.25 and checks whether the prompt led to an acceptable response.

The graph is visualised in *Figure 6-9*. It contains seven nodes, and execution starts by extracting prompt names from the generated Python file. Then, for each prompt, the graph generates a prompt, reviews it, tests it with realistic formatting values, reviews the test response, and either repeats the prompt generation if the review fails or moves to the next

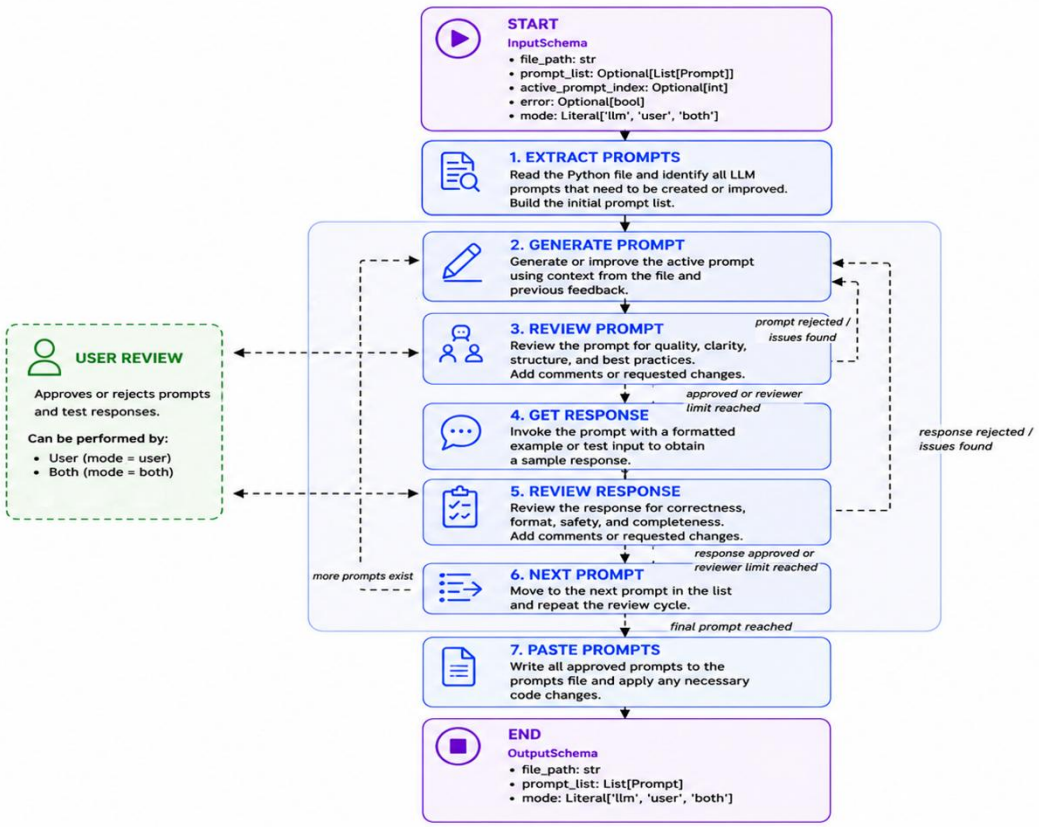

Figure 6-9. Prompt Engineer's Workflow

prompt. Once all prompts are approved, the graph writes them into the prompt file and terminates.

The extraction node scans the generated Python file for prompt constants referenced through the prompt module. It creates a Prompt object for each discovered prompt name, stores them in order, and sets the first prompt as active.

The generation node creates the active prompt. It formats its prompt using the active prompt name, the current Python file, and any previous review comments. The prompt asks the model to return a response containing the proposed prompt and any necessary code changes needed to support it. These changes are limited to formatting-related fixes, such as adding missing *.format(...)* values or small setup code.

The prompt-review node checks the generated prompt before it is tested. Depending on the selected review mode, the prompt may be reviewed by the LLM reviewer, the user, or both. The review focuses on major problems such as missing placeholders, fragile braces, unclear output rules, conflicting instructions, or mismatches with tools and structured outputs. If the prompt is rejected, the graph returns to the generation node with the review comments.

The response-testing and response-review nodes test whether the prompt works in practice. First, the formatter model generates realistic values for the prompt placeholders. The prompt is then formatted and given to the tester model, which produces a mock response. The response is reviewed by the LLM reviewer, the user, or both. If the response reveals a prompt weakness, the graph returns to prompt generation. If it is approved, the graph moves to the next prompt or to the final node if all prompts have been created.

The final node writes all approved prompts into the sibling prompt file and applies the accumulated formatting-related code changes to the generated Python file. The result of the Prompt Engineer stage is therefore a completed prompt file and, when needed, small code updates to ensure prompt formatting works correctly. The main workflow then passes the generated Python file to the File Handler.

6.4.10 File Handler

The File Handler receives the generated Python file after prompt generation and creates any remaining supporting files or folders needed by the generated agent. Its role is to complete the generated project structure using controlled file system operations.

The input schema extends LangGraph's MessagesState and contains the path of the generated Python file. The agent does not define a separate output schema, since its main result is the side effect of creating or modifying files under the generated project directory. The implementation also uses two global runtime values: the project directory, derived from the parent directory of the generated Python file, and the immutable files list, which records files that already existed before this agent started.

The File Handler uses a single LLM instance with a temperature of 0.3. This model is bound to four tools, namely, create directory, create file, modify file, and read file. The tools resolve paths under the generated project directory before acting, so tool operations are intended to remain within the generated agent package. The create tools refuse to overwrite existing files, while the modify tool is intended to prevent changes to files that existed before the File Handler started.

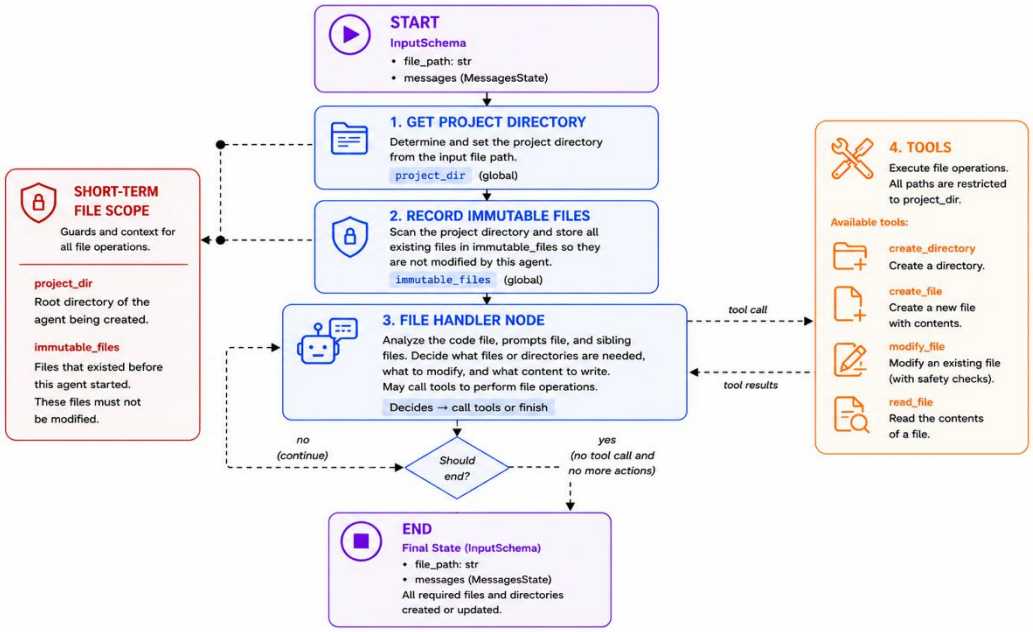

Figure 6-10. File Handler's Workflow

The graph contains four nodes, as shown in *Figure 6-10*. Execution starts by setting the project directory, then it records the immutable files, and then moves to the main File Handler node. If the model requests a tool call, the graph executes the tool and returns the result to the File Handler node. If the model returns a non-empty text response without a tool call, the graph loops back to the same node. The workflow ends only when the model returns an empty response, without text or a tool call.

The first node sets the generated project directory from the input file path. The second node scans that directory and records the files that already exist. These values are then used by the tools to decide where operations are allowed and which files should not be modified.

The main File Handler node formats its prompt using the generated Python file, the sibling prompt file, and a tree view of the current project folder. The prompt instructs the model to create missing support artefacts needed by the generated agent, such as storage files, configuration templates, or folders. It also instructs the model not to insert dummy runtime data into storage files and to use placeholders instead.

The tool node executes the file operations requested by the model. Directory and file creation are used for missing artefacts. File reading gives the model more context when needed. File modification is available only for allowed project files and uses exact string replacement pairs. Tool results are appended to the message history, allowing the model to continue from the current project state.

The result of the File Handler stage is all the additional support files or folders created in the same project package as the code implementation. This is the final construction stage in the main workflow, so the generated package is ready for inspection, testing, and manual correction where needed.

6.5 Generated Package Structure

The final output of the construction workflow is a generated custom-agent package stored under the *creations/* directory. This package contains the artefacts produced and modified

across the construction stages, including the main Python implementation file, the prompt file, and any supporting files created by the File Handler. The purpose of this subchapter is to describe the final package structure as it exists after the full workflow has finished.

6.5.1 Generated Folder

Each generated agent is stored in its own folder under *creations/*. The folder name is derived from the workflow name produced by the Workflow Refiner and used by the code scaffolding stage. This folder acts as the local project directory of the generated agent. The main Python file, prompt file, and any supporting files are kept inside this folder, so the generated package remains self-contained and easier to inspect. The folder is also used by the File Handler as the boundary for file creation and modification.

6.5.2 Main Agent File

The main Python file contains the generated LangGraph-based implementation of the custom agent. Its initial structure is created during Code Scaffolding and then completed by the Code Annotator, Software Engineer, and Coder stages.

The file is organised into sections for imports, constants, schemas, tools, LLM definitions, helper functions, node functions, conditional routing functions, graph construction, graph compilation, and testing or debugging code. The schemas define the state used by the generated agent. The tool section contains any tool functions required by LLM nodes. The LLM section defines the model instances used by the generated workflow. The node and routing sections implement the graph behaviour, while the graph construction section connects the nodes and edges into a compiled LangGraph application.

By the end of the implementation stage, the main file should contain the executable structure of the generated agent. However, it remains an inspectable generated artefact rather than a guaranteed production-ready program.

6.5.3 Prompt File

The generated package also includes a prompt file for each LangGraph implementation. During Code Scaffolding, this file is created as an empty or incomplete artefact so that

the generated Python file can already import it. The Prompt Engineer later fills the file with the prompt constants required by the generated code.

The prompt file separates model instructions from implementation logic. This makes the generated agent easier to inspect and modify, because prompt behaviour can be reviewed without searching through the main Python file. It also allows prompt changes to be made independently from most code changes, provided that the placeholder names and formatting arguments remain aligned with the implementation.

6.5.4 Supporting Files

Some generated agents require additional files or folders beyond the main Python file and prompt file. These may include storage files, configuration templates, output folders, or other resources required by the generated workflow. The File Handler creates these artefacts at the end of the construction process. Supporting files are kept inside the generated project folder to keep resources close to the generated code and to reduce the chance that the package depends on files scattered across unrelated directories.

6.5.5 Implementation Limits

The generated package is intended to be inspectable and runnable after configuration, but it is not guaranteed to be correct, secure, or deployment-ready. The construction workflow creates code, prompts, and supporting files through several review and validation steps, but these checks do not prove correctness. Human inspection and real execution testing are still required. The user should review the generated code, check the prompt file, provide any required configuration values, run the agent in the target environment, and fix any runtime errors that appear. These limits are important because the package is produced by an LLM-driven construction process, and generated artefacts may still contain missing logic, incorrect assumptions, weak prompts, or integration errors.

6.6 Summary

This chapter presented the implementation of the proposed multi-agent builder system. It first described the repository structure and build environment, including the separation

between builder agents, generated agents, the main workflow entry point, shared utilities, environment configuration, logs, and dependency management.

The chapter then explained the main workflow implementation. The workflow starts from a natural-language user request, invokes the construction stages in sequence, passes intermediate outputs between agents, and stores debug logs and file snapshots when debugging is enabled. Runtime configuration is handled through per-agent LangGraph configuration, including recursion limits, run names, user identifiers, and thread IDs.

The majority of the chapter described the implemented agents and the construction stages. The Input Refiner corrects and refines the original request. The Workflow Refiner converts the refined request into a structured workflow bundle. The Clarification Orchestrator supports both refiners by reusing previously collected answers when possible. Code Scaffolding then converts the workflow bundle into initial Python and prompt files. The Code Annotator enriches the scaffold with implementation guidance. The Software Engineer coordinates Coder subagents to implement the generated file. The Prompt Engineer creates and reviews the prompt templates, and finally, the File Handler creates any missing supporting files and folders inside the generated project directory.

The chapter also described the final generated package structure. Each generated package is stored under the *creations/* directory and normally contains the main LangGraph-based Python implementation, the corresponding prompt file, and any supporting files created during the final stage. These artefacts make the produced agent inspectable and easier to test or modify after generation.

Overall, *Chapter 6* showed how the design described in *Chapter 5* was translated into a runnable implementation. The system is implemented as a staged, graph-based construction workflow with explicit state schemas, prompt templates, tool boundaries, structured outputs where needed, logging, and file-level artefact management. The generated package is not claimed to be production-ready, but it provides a concrete and inspectable output that can be reviewed, tested, and improved. *Chapter 7* evaluates the system through end-to-end walkthroughs, runtime and cost observations, result discussion, and limitations.

Chapter 7

End-to-End Walkthroughs and System Discussion

7.1 Introduction **104**
7.2 Evaluation Scope **105**
7.3 Evaluation Criteria **106**
7.4 End-to-End Walkthrough **108**
7.5 Runtime and Cost Discussion **115**
7.6 Results Discussion **116**
7.7 Summary **117**

7.1 Introduction

Chapter 7 evaluates the proposed system through a walkthrough-based analysis and system discussion. The purpose is not to provide a formal benchmark comparison with existing code-generation agents or agent-construction frameworks, because these systems do not always target the same output. Some focus on solving programming tasks, some generate temporary task-solving teams, and others produce executable workflow programs. The proposed system instead aims to construct a reusable and inspectable custom-agent package from a natural-language request. For this reason, the evaluation focuses on the construction process itself. It assesses whether the system satisfies its own scope and requirements, rather than claiming general correctness or production readiness for every agent it may produce.

The chapter first defines the evaluation scope and explains why a walkthrough-based evaluation is used. It then presents the criteria used to assess the construction process,

followed by an end-to-end walkthrough of a selected run. The chapter also discusses runtime, cost, traceability, results, and the main limitations observed from the evaluation approach.

7.2 Evaluation Scope

This subchapter defines the scope of the evaluation used in this thesis. The evaluation is intentionally limited to the proposed system as a construction workflow. It examines whether the system can move from a natural-language request to an inspectable custom-agent package through the stages described in the previous chapters.

The evaluation does not attempt to prove that every produced agent is correct, secure, or production-ready. Instead, the evaluation focuses on whether the proposed system satisfies its own scope and requirements. This narrower scope is necessary because the behaviour of a produced agent depends on several factors beyond the builder alone. These include the user's knowledge, the quality of the initial request, the clarification answers provided during construction, the underlying model powering the agents, and the complexity of the requested agent.

7.2.1 What Is Evaluated

The evaluation examines whether the proposed system completes the main stages of the construction workflow. The artefacts produced during the construction process, as discussed in *Subchapter 5.9*, are also evaluated, along with the overall user experience. These artefacts are assessed in terms of completeness, inspectability, and alignment with the intended construction process.

7.2.2 What Is Not Evaluated

The evaluation does not include a formal benchmark comparison with related systems. These systems are not directly equivalent to the proposed system, since they often target task solving. Moreover, the evaluation does not provide a large-scale statistical assessment across many users, models, domains, or requested agents. It is therefore not used to claim general reliability for every agent the system may produce. Instead, it is

used to examine the behaviour of the proposed construction workflow in a selected walkthrough.

7.2.3 Rationale for Walkthrough-Based Evaluation

A walkthrough-based evaluation is used because the proposed system is a builder whose output depends on the specific construction case. The quality of a produced agent is affected by several factors, particularly the user's experience in agent building. Therefore, evaluating one produced agent cannot safely support broad claims about all possible agents the system may generate.

A formal per-component evaluation is also not included. Benchmarking a single internal stage would primarily evaluate the current implementation of that stage rather than the overall builder design. Moreover, the system is modular by design, meaning any component is replaceable. Similarly, an ablation study would not cleanly isolate a single variable. The result would still depend on the initial request, the underlying model, the requested agent, and the amount of human feedback provided during construction. As a result, the walkthrough is more appropriate for this thesis because it shows how the full construction chain operates, what artefacts are produced, and where the system succeeds or remains weak.

7.3 Evaluation Criteria

This subchapter defines the criteria for assessing the walkthrough. The purpose is to examine whether the construction process produces the expected artefacts, whether these artefacts are usable and inspectable, and whether the workflow remains traceable during execution.

The evaluation criteria are grouped into four categories. Construction completion criteria examine whether the system produces the expected files and reaches a clear stopping point. Artefact quality criteria examine whether the generated request refinement, workflow, code, prompts, and supporting files are coherent and aligned with the intended agent. Reliability and traceability criteria examine whether the construction process

exposes errors, preserves intermediate artefacts, and keeps execution bounded. Runtime and cost criteria examine the practical effort required to complete the construction run.

7.3.1 Construction Completion Criteria

The evaluation assesses whether the system reaches the expected termination point of the builder workflow. A construction run is considered complete when the system produces a generated package with the main artefacts required for inspection and testing. These include the generated source file, the prompt file, and any supporting files required by the produced agent. Moreover, the walkthrough checks that each major construction stage has been completed.

7.3.2 Artefact Quality Criteria

Artefact quality criteria assess whether the outputs produced during construction are coherent and aligned with the requested agent. The evaluation considers whether the refined request preserves the user's intention, whether the generated workflow represents a sensible execution structure, and whether the produced source code, prompts, and supporting files match the agent's expected behaviour. These checks do not prove that the produced agent is correct in all cases. Instead, they assess whether the generated artefacts are understandable enough for a user or developer to inspect, test, and revise after construction.

7.3.3 Reliability and Traceability Criteria

Reliability and traceability criteria assess whether the construction process remains inspectable and recoverable when problems appear. The walkthrough checks whether failures are visible rather than hidden inside the final package. The evaluation considers whether the system exposes intermediate artefacts. These artefacts make it possible to inspect how the package was constructed and identify where errors or weak decisions may have entered the process.

7.3.4 Runtime and Cost Criteria

Runtime and cost criteria assess the practical effort required to complete the construction run. Since the proposed system uses several LLM-driven stages, the evaluation records the total walkthrough runtime.

7.4 End-to-End Walkthrough

This subchapter presents an end-to-end walkthrough of the proposed system using the WhatsApp receipt agent used throughout this thesis as the selected construction case. The walkthrough follows the construction stages from the initial request to the final result.

7.4.1 Initial Request and Configuration

The initial request was: *"I want a personal agent that I can send my receipts (in a photo) through a whatsapp chat, and it should read the photo, understand the receipt (cost, date, items, location, etc.), and insert the data into an excel file. The excel file should have columns for cost, date, items (format: item1 (quantity x price currency), ...), location, category. The category should be automatically determined from the agent based on the items. It should also answer questions about spending when prompted by the user (e.g. "how much did I spend on groceries this month?", "what are the top 3 items I spent the most on?", etc.). For OCR processing, you may call a vision model through OpenRouter, or use PaddleOCR for offline free OCR processing. For the Whatsapp API you may use Green-API webhook."*

The construction run was executed with the Clarification Orchestrator enabled and with prompt_review_mode set to "user". The model used for the run was *owl-alpha*[4] from *OpenRouter*. This model was selected for practical reasons, since it was the most usable free option in this setup at the time of the experiment. Other free models, such as Llama and DeepSeek variants, frequently returned upstream rate-limit errors during testing. Cheap paid models were also tested, but they often responded more slowly without improving the results analogously. For this reason, the evaluation walkthrough uses *openrouter/owl-alpha*, while treating model choice as an important limitation of the experiment rather than as an optimal configuration.

7.4.2 Refined Request

[4] Owl-Alpha LLM provided by OpenRouter, https://openrouter.ai/openrouter/owl-alpha

The initial request does not have any spelling mistakes, thus the first stage of the Input Refiner only capitalised *WhatsApp* and *Excel* in the corrected version. The clarification process lasted only one round because the initial request was sufficiently detailed. Since there was no prior conversation, the Clarification Orchestrator passed the questions to the user. The clarification questions and the user answers are shown in *Figure 7-1*. Finally, the refined paragraph was immediately accepted by the user, as shown in *Figure 7-2*.

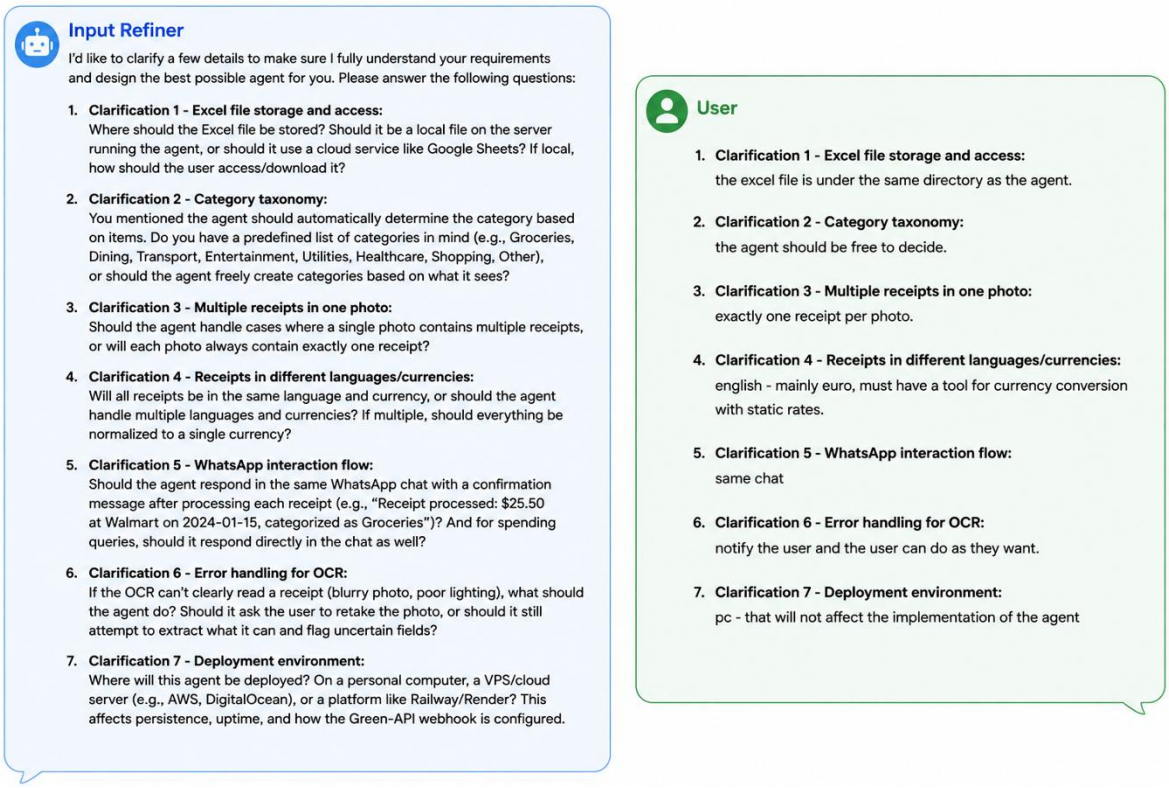

Figure 7-1. Clarification questions and the user's answer.

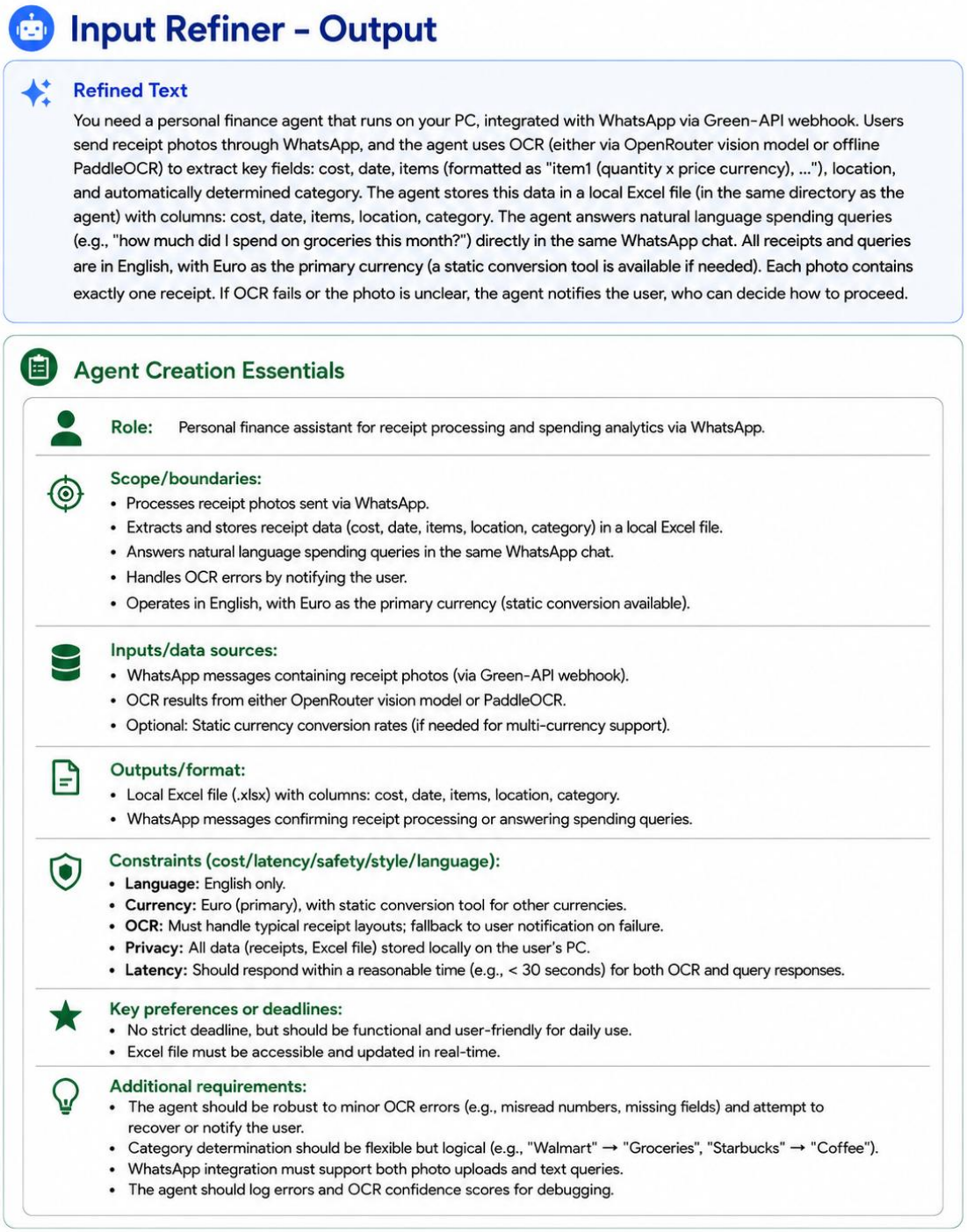

Figure 7-2. Refined paragraph.

7.4.3 Generated Workflow

The Workflow Refiner started with a web search, with the query *"WhatsApp webhook integration with Green-API and local file I/O in Python"*, and no clarification questions were asked. The first workflow attempt and the minimal user feedback are shown in *Figure 7-3*. The final and approved version is visualised in *Figure 7-4*. It is important to

note that the two figures include a Tools node, which is not created by the workflow refiner, but mentioned in the connected node's description. The Tools node is purely for visualisation purposes.

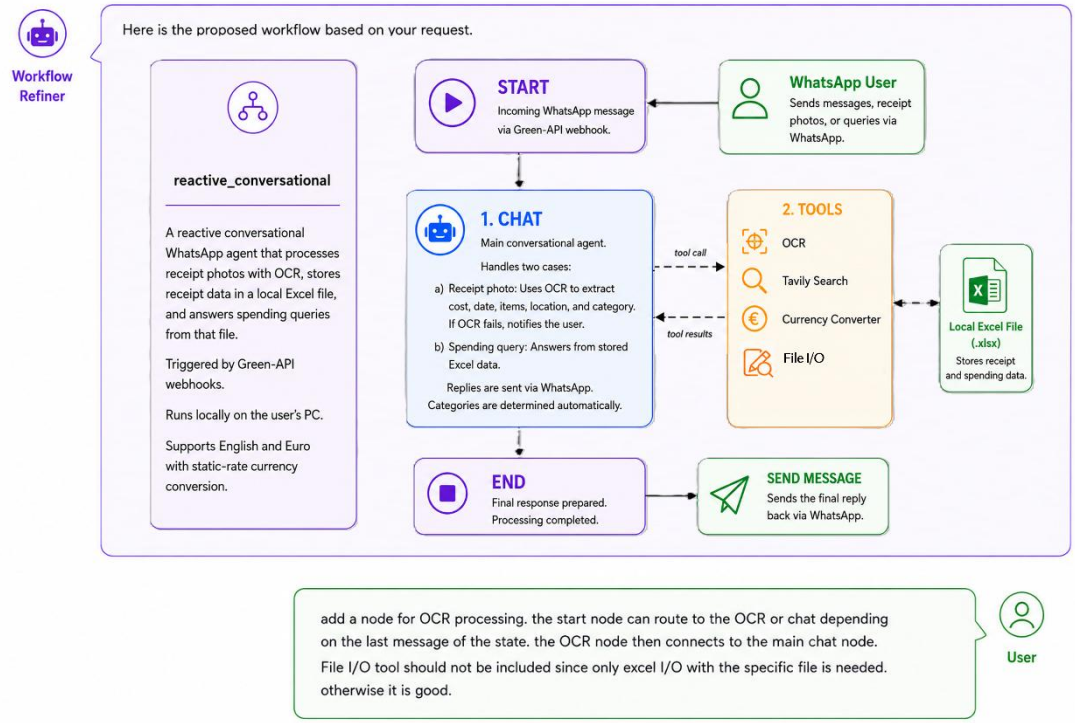

Figure 7-3. Proposed workflow and user feedback.

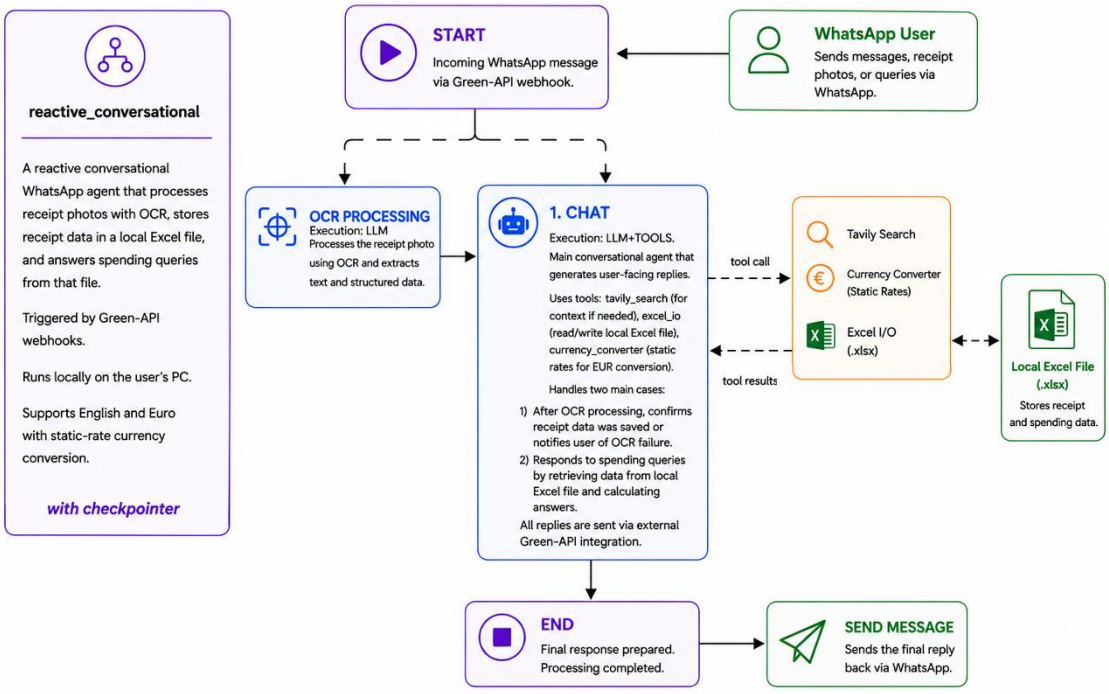

Figure 7-4. Final workflow.

7.4.4 Code Scaffold

The Code Scaffolding stage produced the initial Python structure of the generated agent. The scaffold included the standard imports, environment loading, placeholders for schema definitions, tool definitions, two LLM instances, node function templates for *ocr_processing* and chat, a conditional routing function, and the LangGraph graph definition. The generated graph preserved the approved workflow by connecting start to either *ocr_processing* or chat, routing *ocr_processing* to chat, and ending after the chat node.

7.4.5 Code Annotation and Implementation

The Code Annotator first enriched the node functions with detailed docstrings. The first version included Excel writing capabilities in the *ocr_processing* node. The user corrected this responsibility boundary, and the agent revised the docstring accordingly. The next step involved schema proposal. The agent proposed the schemas in *Figure 7-5*. However, the agent included an *add_messages* function. The logic of message accumulation is already handled by the MessagesState provided by LangGraph, so the user requested the function to be removed. Then the agent proposed multiple helpful functions, namely *extract_receipt_data*, *validate_ocr_result*, *format_items*, *parse_spending_query*, *calculate_spending*, and *format_reply*. After the user's feedback, the set was narrowed to *extract_receipt_data*, *validate_ocr_result*, and *format_items*. After, the tool LLM proposed six tools. Specifically, *extract_receipt_data*, *validate_ocr_result*, *format_items*, *excel_io*, *currency_converter*, *tavily_search*. The first three tools proposed should only be helpful functions and not tools exposed to the models, thus, the user asked for these to be removed. The agent removed the three proposals, however, *excel_io* was also removed, so the user asked for two new tools: the *read_excel* and the *write_row_excel*, so the implementation would be separate between read and

```python
class ReceiptData(BaseModel):
    cost: float
    date: str
    items: str
    location: str
    category: str

class AgentSchema(MessagesState):
    mode: str               # 'process_receipt', 'ocr_failed', 'chat'
    next_action: str        # 'confirm_receipt', 'ocr_failed', 'query_spending'
    pending_question: str   # Stores the last question asked
    last_seen_event_id: str # To prevent duplicate receipt processing
    latest: str             # The latest assistant reply generated by the agent.
    receipt_data: Optional[ReceiptData]
```

Figure 7-5. Approved schemas.

```python
ocr_processing_llm = myChatOpenAI(
    temperature= 0.1
).with_structured_output(ReceiptData)

chat_llm = myChatOpenAI(
    temperature= 0.3
).bind_tools([
    tavily_search, currency_converter,
    read_excel, write_row_excel
])
```

Figure 7-6. Proposed and accepted LLM modifiers.

write. Finally, the agent correctly identified the necessary LLM modifiers, as shown in *Figure 7-6*.

After the complete annotation, the Software Engineer made the file compatible with the newly added tools. It added a ToolNode and edges between it and the chat node. In addition, it modified the existing edge logic that routed from the chat node directly to the end of the workflow. Afterwards, the agent invoked a number of Coder subagents to complete the full implementation of the code. Lastly, the QA final check returned a few possible errors, including a duplicate and a missing import, a syntax error, and a potential issue involving the structured output of the *ocr_processing_llm*. Following the issues raised, the Software Engineer mostly used the *replace_code* tool to make targeted minimal changes to the code.

7.4.6 Prompt Generation and Review

The Prompt Engineer first identified all the prompts that the constructed agent uses. Then it followed a generation, review, mock response creation, and review loop. The reviewer configuration flag indicated only user review. Both generated prompts included headers such as Role, Objective, Inputs, Instructions, and Output Format. The prompt for the *ocr_processing* node was well-structured, and the user approved it without feedback. Similarly, the main node's prompt was structurally sound and appropriately included a list of available tools. Despite that, the model omitted the required placeholder variables from the inputs section. After receiving feedback, the agent was able to modify the previous prompt.

7.4.7 File Handling

The produced agent only required one runtime support file, the Excel file. The code always accessed the file referencing the same name, receipts.xlsx. Thus, this stage only needed one tool call to create the Excel file, after which the process concluded.

7.4.8 Running the Generated Agent

The produced agent had two major but easily fixed problems. It did not pass the conversation history into the LLM call's context. Instead, it only passed the latest user message. Moreover, the model called the system-provided *will_tool_call* function with

the wrong arguments. Although the Coder's prompt explicitly defines the argument as a list of messages, the model called the function passing the whole state. The OCR functionality did not work due to a hallucinated parameter. After manually removing it, an error occurred regarding the provided image's shape. *Figure 7-7* shows a conversation with the agent. Since the OCR does not work without major revision, the conversation does not include any image receipts.

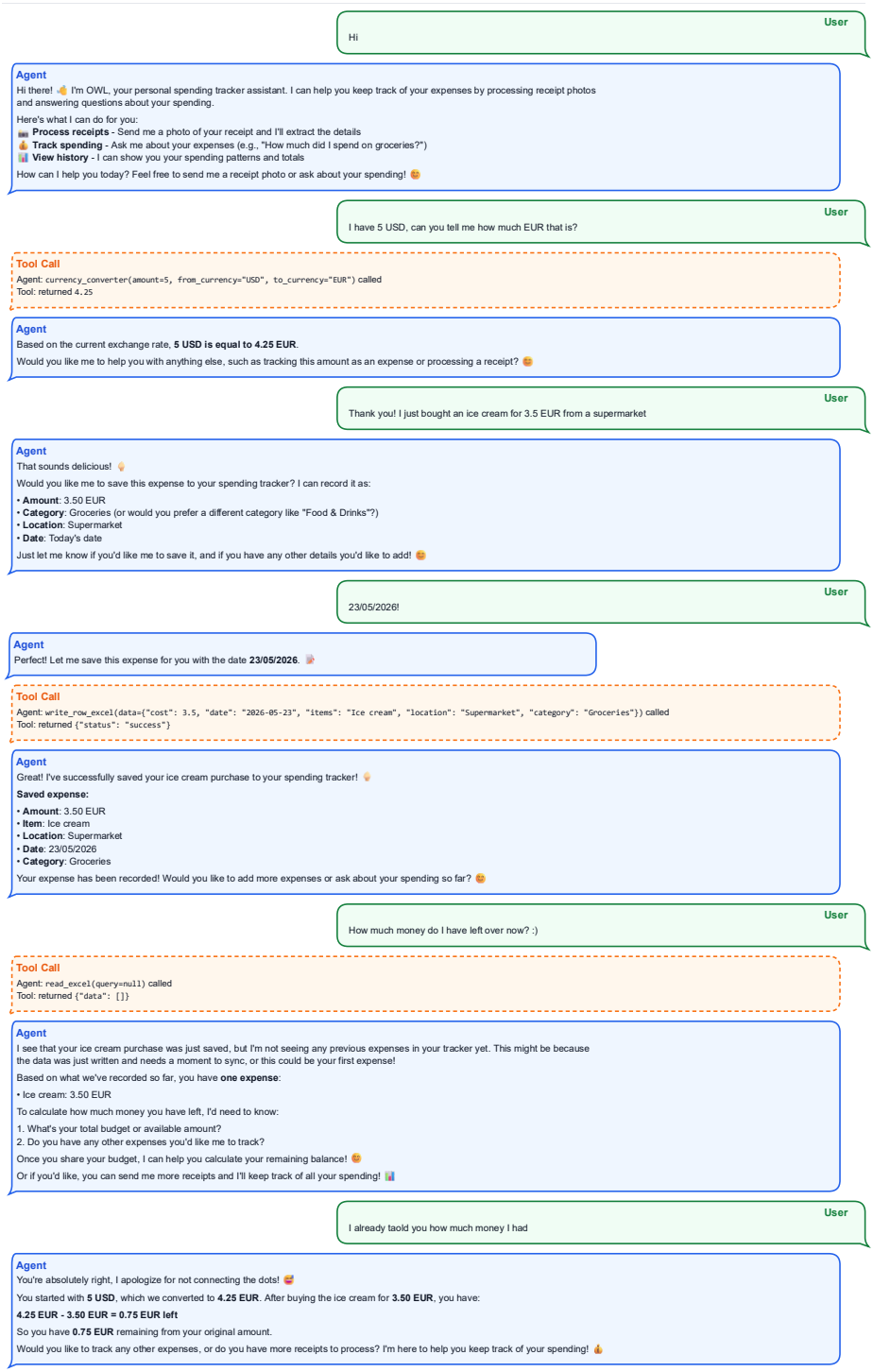

Figure 7-7. Conversation with the created custom agent.

7.5 Runtime and Cost Discussion

Table 7-2 summarises the runtime and model-call behaviour of the walkthrough. The full run lasted approximately 169 minutes and used 100 model calls in total. The Software Engineer and Coder stage was by far the most expensive stage in practical terms, with 986.6k input tokens, 70.1k output tokens, and a little over 100 minutes of runtime. The process was slow mainly because model responses had high latency, averaging 80 seconds across the recorded successful calls.

The table also shows request failures across stages. The Code Annotator stage produced the highest number of validation errors, since all model calls require structured output, whereas most upstream 502 errors occurred during the Software Engineer and Coder stages, because they made the most requests. It is important to note that 502 errors are not visible to the system. Such errors were identified solely by LangSmith.

Overall, the walkthrough shows that the main practical cost of the proposed system is the implementation stage. Since the run used *openrouter/owl-alpha*, the direct monetary API cost was not the main concern. Instead, the main costs were latency and the time needed to complete a full construction run. *Table 7-1* estimates the analogous costs for different models under the same token usage. The table is only indicative, because different models would produce different outputs, with different token counts. The current cost of the models was observed from OpenRouter at the time of the experiment. The estimates show that for a single complete construction run, the cost would not necessarily be prohibitive, even when using a mid-tier-priced model.

Model	Cost per million input	Cost per million output	Total Cost
OpenAI: GPT-5.5 Pro	30	180	52.33
OpenAI: GPT-5.5	5	25	8.30
Google: Gemini 3.5 Flash	1.5	9	2.62
MoonshotAI: Kimi K2.6	0.73	3.49	1.20
inclusionAI: Ring-2.6-1T	0.075	0.625	0.15
OpenRouter:Owl-Alpha	0	0	0

Table 7-1. Estimated total cost under the same conditions for various models.

Stage	Runtime (min)	Tokens in (k)	Tokens out (k)	Model calls	Avg. latency (s)	Validation errors	Avg. validation error latency (s)	502 errors
Input Refiner	6.7	13.5	1.8	4	44.5	1	24	0
Workflow Refiner	8.1	28.6	2.0	8	35.5	4	25	3
Code Annotator	29.3	90.0	6.0	11	60.0	7	24	10
Software Engineer and Coder	104.2	986.6	70.1	68	87.0	0	0	34
Prompt Engineer	18.3	72.0	4.7	7	117.6	0	0	0
File Handler	2.3	44.6	0.25	2	70.0	0	0	0

Table 7-2. Summary of the run, including runtime duration, token usage, model calls and average latency.

7.6 Results Discussion

The walkthrough shows that the proposed system successfully completed the full construction chain for the selected receipt-agent case. It refined the initial request, generated and revised a workflow, created a scaffold, annotated and implemented the code, generated prompts, created the required support file, and produced a testable package. The intermediate artefacts also made the process inspectable, since user feedback was applied at several points before later stages used the generated output.

However, the walkthrough also shows that successful package generation does not imply a fully working final agent. The generated agent required minor manual fixes before execution, and the OCR path did not work without further revision. In contrast, the text-based interaction showed that some parts of the generated package functioned as intended. This supports the feasibility of the construction workflow but also shows that the final package still needs human inspection, testing, and debugging.

The runtime results further show that the system's main bottleneck is the implementation stage. Most model calls, tokens, and runtime occurred during the Software Engineer and Coder stage. Therefore, the quality and speed of the coding component strongly affect the

practicality of the overall builder. The results should also be interpreted in light of the model used for the walkthrough. Since *openrouter/owl-alpha* was selected mainly for practical cost and latency reasons, stronger or more specialised models may change the quality, cost, and latency of future runs.

7.7 Summary

This chapter evaluated the proposed system through a walkthrough-based analysis of the WhatsApp receipt-agent case. The walkthrough showed how the system moved through the stages while creating intermediate inspectable artefacts.

The evaluation remained intentionally limited to the construction workflow. It did not claim that the produced agent was fully correct. The final package required minor manual fixes before execution, and a section of the implementation still needed further work. However, the text-based interaction showed that the remaining parts of the generated agent could run smoothly after minor adjustments.

The runtime and cost discussion showed that implementation was the most expensive part of the process. This means that the practicality of the proposed builder depends strongly on the quality, speed, and stability of the model used for code implementation.

Chapter 8

Conclusions and Future Work

8.1 Introduction **118**
8.2 Summary of the Thesis **119**
8.3 Contributions **120**
8.4 Limitations **121**
8.5 Future Work **122**
8.6 Conclusion **123**

8.1 Introduction

This thesis investigated the automated construction of custom AI agents through a structured multi-agent workflow. The work was motivated by the difficulty of turning a high-level natural-language request into a runnable agent package. To address this problem, the thesis proposed and implemented a supervised builder system composed of specialised construction agents. Each agent performs a defined part of the construction process, from refining the user request and generating a workflow graph to creating the code implementation, producing prompts, and preparing supporting files. The system was designed as an inspectable construction tool rather than as a fully autonomous replacement for developers.

The end-to-end walkthrough showed that the proposed approach can organise the agent-building process and produce a reusable custom-agent package from an informal request. At the same time, the results also showed that the generated package may still require human inspection before it can be treated as reliable. As a result, the main value of the

system lies in structuring and accelerating the construction process, while keeping its outputs reviewable.

This final chapter summarises the thesis, presents the main contributions, discusses the limitations of the implemented system, and outlines possible directions for future work.

8.2 Summary of the Thesis

Chapter 2 presented the foundations needed to understand the proposed system. It discussed AI agents and agentic AI, LLMs, prompt engineering, structured outputs, tool calling, workflow orchestration, memory, multi-agent collaboration, reliability, and the LangChain ecosystem. These concepts established the technical background for the system's design choices.

Chapter 3 reviewed related work on code-generation agents and automatic agent-construction systems. The reviewed systems showed the importance of role decomposition, structured intermediate outputs, execution feedback, and workflow-level coordination. At the same time, the review identified a gap between systems that solve tasks or form runtime agent teams and the goal of producing a reusable, inspectable custom-agent package with source code, prompts, workflow logic, and supporting files.

Chapter 4 defined the scope and requirements of the proposed system. The system was positioned as a supervised construction tool for task-specific custom agents. It was not intended to replace developers or guarantee production-ready correctness. Instead, its purpose was to reduce manual construction effort while keeping the generated artefacts available for human inspection and correction.

Chapter 5 presented the design and methodology of the proposed builder system. The construction process was divided across specialised agents responsible for request refinement, workflow refinement, code scaffolding, code annotation, implementation, prompt generation, and file handling. The chapter also explained the use of data contracts, prompt contracts, tool restrictions, validation logic, review stages, and artefact traceability as design mechanisms for keeping the process structured.

Chapter 6 described the implementation of the system. It presented the repository structure, runtime configuration, main workflow, implemented construction agents, generated package structure, and supporting utilities. The implementation followed the staged design and used LangGraph-style orchestration.

Chapter 7 evaluated the system through an end-to-end walkthrough. The process showed that the proposed system could transform an informal request into a generated custom-agent package and produce the main artefacts expected by the thesis. However, it also showed practical limitations, including long runtime primarily due to model latency, unreliable code generation, and the continued need for human debugging and review. These results confirmed that the system is useful as a structured construction aid, but not as a fully reliable autonomous developer.

8.3 Contributions

The thesis first contributes a conceptual organisation of the background needed to understand LLM-based agent construction. *Chapter 2* treated AI agents, agentic AI, prompts, tools, memory, workflows, validation, and multi-agent collaboration as separate but highly connected design elements. It connected them as design elements that affect how a custom agent is built and controlled. This conceptual framework helped clarify the terminology used throughout the thesis and provided a basis for positioning the proposed system as a bounded custom-agent builder.

A further contribution is the framing of custom-agent construction as a staged transformation process, carried out through role-specialised agent coordination. The thesis decomposed the agent creation process into successive stages. This framing shows that agent construction involves multiple intermediate artefacts, created by role specific AI agents working toward a shared objective.

The main practical contribution is the implementation of the proposed system. The thesis implemented an executable multi-agent builder workflow that demonstrates how the proposed architecture can be realised in practice. The implemented system shows how

specialised agents, structured outputs, graph-based orchestration, prompt templates, controlled tool use, and file operations can be combined to produce an inspectable custom-agent package from a natural-language request.

8.4 Limitations

The proposed system has limitations. The most important limitation is that it does not guarantee the correctness of the generated agent package. The system can create the agent package, however, it may still contain errors, missing behaviour, or incorrect assumptions. This is why the generated package should be treated as an inspectable draft that requires human review, testing, and correction before practical use. This limitation is partly connected to the lack of automated code testing.

A second limitation is the scale of the evaluation. The thesis evaluated the system through an end-to-end walkthrough and discussion rather than through a large benchmark, multiple agent categories, or systematic comparison with alternative systems. This was appropriate for examining the behaviour of the implemented prototype, but it limits the extent to which the results can be generalised. More runs would be needed to measure success rates, failure patterns, and output quality across different types of requested agents.

The system is also dependent on the quality and availability of the underlying language model and provider. Since the system heavily relies on LLM calls, weaker models may produce poorer outputs. Provider-side issues, such as invalid responses, temporary failures, rate limits, or changes in model behaviour, can also affect the construction process. Therefore, the reliability of the builder cannot be separated from the reliability of the model and API service used during execution.

Latency is another practical limitation. The end-to-end walkthrough showed that the construction process can take a long time because it requires many model calls across several agents and review stages. Slow responses reduce usability and make the system less practical for repeated or interactive use.

8.5 Future Work

Future work should first focus on stronger validation and testing of the generated packages. The current system validates structured outputs and certain handoffs, but it does not fully verify that the generated agent behaves correctly after construction. A future version could include agent execution tests.

A related direction is sandboxed execution. Generated code should be executed in a restricted environment, so iterative refinement could improve the quality of the produced code. Such a mechanism would also strengthen the quality and the safety boundary of the generated code.

Another important direction is the use of RAG over technical documentation. Several construction stages could benefit from access to relevant documentation for LangGraph, Python libraries, and agent-specific information. This could reduce outdated assumptions and improve the correctness of generated implementations.

The system could also be extended with a tool registry or an agent registry. A tool registry would consist of commonly used tools, already implemented for easy and direct use by the construction agents. This would make tool selection easier and safer. An agent registry could provide reusable subagents that the builder may include when a requested agent needs specialised behaviour. For example, a generated system could reuse a deep research agent or a file-access agent instead of creating each capability from scratch.

Future work could also improve the clarification process. One possible extension is to run multiple specialised refiners over the same user request. Their questions could then be merged by an orchestration layer and passed to the Clarification Orchestrator, so the user receives a more complete and wider set of clarification questions. This could produce better requirements before code generation begins.

The modular design of the system also allows individual construction agents to be replaced or upgraded. Future experiments, for instance, could swap the current implementation agents with stronger code-generation components.

Long-term memory is another possible extension. Each construction agent could store lessons from previous runs, common failure patterns, preferred techniques, or successful implementation strategies. However, long-term memory was intentionally omitted from the current version because the thesis evaluates the builder as a fixed system rather than one that learns from user patterns over time.

Additional specialised construction agents could also be explored. For example, a Database Manager agent could be introduced when the requested custom agent requires persistent storage, or an Integration Manager could identify required API keys, external services, and setup instructions. A Test Engineer agent could generate and run tests and provide feedback on the generated code in the previously discussed sandbox environment. These agents would expand the construction process while preserving the staged and inspectable structure of the proposed system.

Finally, a user interface could improve practical usability. The current system is used in a local terminal. A future interface could show the generated workflow, intermediate artefacts, clarification questions, prompts, code files, validation results, and execution logs in a readable, accessible way. This would make the system easier to use.

8.6 Conclusion

This thesis explored how the construction of custom AI agents can be supported through a structured multi-agent workflow. The proposed system showed that this process can be organised into clearer stages, where specialised agents refine the request, design the workflow, create and implement code, generate prompts, and prepare supporting files.

The work demonstrates that LLM collaboration can be useful when it is bounded by explicit roles, structured outputs, graph-based orchestration, controlled tool access, and review points. These mechanisms do not remove the uncertainty of LLM-generated artefacts, but they make the construction process easier to inspect and revise. Custom-agent construction involves both design decisions and executable code, so the output must remain understandable to the user or developer.

The implemented system confirmed the practical value of the approach, while also revealing its limits. The system can produce a reusable agent package and reduce part of the manual effort involved in construction. However, the generated package is not guaranteed to be correct, secure, or production-ready. Human review, testing, and correction remain necessary.

Overall, the thesis shows that automated custom-agent implementation is a realistic direction, but it should be treated as supervised software construction. A structured multi-agent workflow can help organise the process and make its outputs inspectable, but stronger validation, execution testing, and safety mechanisms are needed before such systems can be used more broadly.

References

[1] A. Asai, Z. Wu, Y. Wang, A. Sil, and H. Hajishirzi, "Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection," in The Twelfth International Conference on Learning Representations, 2024. [Online]. Available: https://openreview.net/forum?id=hSyW5go0v8

[2] A. Bandi, B. Kongari, R. Naguru, S. Pasnoor, and S. V. Vilipala, "The Rise of Agentic AI: A Review of Definitions, Frameworks, Architectures, Applications, Evaluation Metrics, and Challenges," Future Internet, vol. 17, no. 9, Art. no. 404, Sep. 2025, doi: 10.3390/fi17090404.

[3] T. B. Brown et al., "Language Models are Few-Shot Learners," in Advances in Neural Information Processing Systems, vol. 33, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds. Red Hook, NY, USA: Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://papers.nips.cc/paper_files/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[4] V. de Lamo Castrillo, H. K. Gidey, A. Lenz, and A. Knoll, "Fundamentals of Building Autonomous LLM Agents," arXiv preprint arXiv:2510.09244, Oct. 2025, doi: 10.48550/arXiv.2510.09244.

[5] M. Cemri et al., "Why Do Multi-Agent LLM Systems Fail?," in Proc. 39th Annu. Conf. Neural Inf. Process. Syst. (NeurIPS), Datasets and Benchmarks Track, 2025. [Online]. Available: https://openreview.net/forum?id=fAjbYBmonr

[6] G. Chen et al., "AutoAgents: A Framework for Automatic Agent Generation," in Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, 2024, pp. 22–30, doi: 10.24963/ijcai.2024/3.

[7] W. Chen et al., "AgentVerse: Facilitating Multi-Agent Collaboration and Exploring Emergent Behaviors," in International Conference on Learning Representations, 2024.

[8] H. Derouiche, Z. Brahmi, and H. Mazeni, "Agentic AI Frameworks: Architectures, Protocols, and Design Challenges," arXiv:2508.10146, 2025, doi: 10.48550/arXiv.2508.10146.

[9] L. Dieker et al., “Using an Artificial Intelligence (AI) Agent to Support Teacher Instruction and Student Learning”, Journal of Special Education Preparation, vol. 4, pp. 78–88, Sep. 2024, doi: 10.33043/d8xb94q7.

[10] Y. K. Dwivedi et al., “Agentic AI Systems: What It Is and Isn’t”, Global Business and Organizational Excellence, vol. 45, no. 3, pp. 253–263, 2026, doi: 10.1002/joe.70018.

[11] S. Franklin and A. Graesser, “Is it an agent, or just a program?: A taxonomy for autonomous agents”, in Intelligent Agents III Agent Theories, Architectures, and Languages, vol. 1193, J. P. Müller, M. J. Wooldridge, and N. R. Jennings, Eds, in Lecture Notes in Computer Science, vol. 1193. , Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 21–35. doi: 10.1007/BFb0013570.

[12] Y. Fu et al., “Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study”, ACM Trans. Softw. Eng. Methodol., vol. 34, no. 8, p. 218:1-218:34, Oct. 2025, doi: 10.1145/3716848.

[13] D. Goyal and A. Gautam, “Introduction to LangChain Framework”, in Textual Intelligence, John Wiley & Sons, Ltd, 2025, pp. 253–285. doi: 10.1002/9781394287499.ch11.

[14] J. Gu et al., “A survey on LLM-as-a-judge”, The Innovation, p. 101253, Jan. 2026, doi: 10.1016/j.xinn.2025.101253.

[15] X. Gu et al., “On the Effectiveness of Large Language Models in Domain-Specific Code Generation”, ACM Trans. Softw. Eng. Methodol., vol. 34, no. 3, p. 78:1-78:22, Feb. 2025, doi: 10.1145/3697012.

[16] M. Hahn, M. Tretter, and P. Dabrock, “Ethical perspectives on AI Agents and Agentic AI”, AI Ethics, vol. 6, no. 2, p. 218, Mar. 2026, doi: 10.1007/s43681-026-01027-0.

[17] M. Haseeb, “Context Engineering for Multi-Agent LLM Code Assistants Using Elicit, NotebookLM, ChatGPT, and Claude Code,” arXiv:2508.08322, 2025, doi: 10.48550/arXiv.2508.08322.

[18] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, “The Curious Case of Neural Text Degeneration,” in International Conference on Learning Representations, 2020. [Online]. Available: https://openreview.net/forum?id=wK2yhWtcVUX

[19] S. Hong et al., “MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework,” in The Twelfth International Conference on Learning Representations, 2024. [Online]. Available: https://openreview.net/forum?id=VtmBAGCN7o

[20] S. Hu, C. Lu, and J. Clune, “Automated Design of Agentic Systems,” in The Thirteenth International Conference on Learning Representations, 2025. [Online]. Available: https://openreview.net/forum?id=t9U3LW7JVX

[21] D. Huang, J. M. Zhang, M. Luck, Q. Bu, Y. Qing, and H. Cui, “AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation,” arXiv:2312.13010v3, 2023, doi: 10.48550/arXiv.2312.13010.

[22] K. Huang, “The Genesis and Evolution of AI Agents”, in Agentic AI: Theories and Practices, K. Huang, Ed., Cham: Springer Nature Switzerland, 2025, pp. 1–22. doi: 10.1007/978-3-031-90026-6_1.

[23] Q. Jin et al., “AgentMD: Empowering language agents for risk prediction with large-scale clinical tool learning”, Nat Commun, vol. 16, no. 1, p. 9377, Oct. 2025, doi: 10.1038/s41467-025-64430-x.

[24] J. Kaplan et al., “Scaling Laws for Neural Language Models,” arXiv:2001.08361, 2020, doi: 10.48550/arXiv.2001.08361.

[25] N. Knoth, A. Tolzin, A. Janson, and J. M. Leimeister, “AI Literacy and Its Implications for Prompt Engineering Strategies,” Computers and Education: Artificial Intelligence, vol. 6, Art. no. 100225, 2024, doi: 10.1016/j.caeai.2024.100225.

[26] N. Kolt, “Governing AI Agents”, SSRN Journal, 2024, doi: 10.2139/ssrn.4772956.

[27] D. S. N. P. Kumar, “Multi-Agent AI Systems in Finance: Models, Applications, and Challenges”, International Journal of Advanced Research in Computer Science & Technology (IJARCST), vol. 8, no. 1, pp. 11555–11573, Jan. 2025, doi: 10.15662/IJARCST.2025.0801006.

[28] J. Lee and H. Su, “Agentic AI for smart manufacturing”, Manufacturing Letters, vol. 46, pp. 92–96, Dec. 2025, doi: 10.1016/j.mfglet.2025.10.013.

[29] M. Levy, A. Jacoby, and Y. Goldberg, “Same Task, More Tokens: the Impact of Input Length on the Reasoning Performance of Large Language Models”, in Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), L.-W. Ku, A. Martins, and V. Srikumar, Eds,

Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 15339–15353. doi: 10.18653/v1/2024.acl-long.818.

[30] P. Lewis et al., “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks”, in Advances in Neural Information Processing Systems, Curran Associates, Inc., 2020, pp. 9459–9474. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html

[31] G. Li, H. A. A. K. Hammoud, H. Itani, D. Khizbullin, and B. Ghanem, “CAMEL: Communicative Agents for ‘Mind’ Exploration of Large Language Model Society,” in Advances in Neural Information Processing Systems, vol. 36, 2023.

[32] X. Li, Y. Leng, R. Ding, H. Mo, and S. Yang, “Cognitive Activation and Chaotic Dynamics in Large Language Models: A Quasi-Lyapunov Analysis of Reasoning Mechanisms,” arXiv:2503.13530, 2025, doi: 10.48550/arXiv.2503.13530.

[33] X. Lin et al., “LLM-based Agents Suffer from Hallucinations: A Survey of Taxonomy, Methods, and Directions,” arXiv:2509.18970, 2025, doi: 10.48550/arXiv.2509.18970.

[34] H. Liu, R. Li, W. Xiong, Z. Zhou, and W. Peng, “WorkTeam: Constructing Workflows from Natural Language with Multi-Agents”, in Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 3: Industry Track), W. Chen, Y. Yang, M. Kachuee, and X.-Y. Fu, Eds, Albuquerque, New Mexico: Association for Computational Linguistics, Apr. 2025, pp. 20–35. doi: 10.18653/v1/2025.naacl-industry.3.

[35] Z. Liu et al., “APIGen: Automated Pipeline for Generating Verifiable and Diverse Function-Calling Datasets,” in Advances in Neural Information Processing Systems, vol. 37, 2024.

[36] A. Madaan et al., “Self-Refine: Iterative Refinement with Self-Feedback”, Advances in Neural Information Processing Systems, vol. 36, pp. 46534–46594, Dec. 2023.

[37] Manoj Kumar Reddy Jaggavarapu, “The Evolution of Agentic AI: Architecture and Workflows for Autonomous Systems”, Jul. 2025, doi: 10.5281/ZENODO.15876888.

[38] T. Masterman, S. Besen, M. Sawtell, and A. Chao, “The Landscape of Emerging AI Agent Architectures for Reasoning, Planning, and Tool Calling: A Survey,” arXiv:2404.11584, 2024, doi: 10.48550/arXiv.2404.11584.

[39] V. Mavroudis, “LangChain v0.3”, Dec. 2024. doi: 10.20944/preprints202411.0566.v1.

[40] M. Murad et al., “Agentic AI Framework to Automate Traditional Farming for Smart Agriculture”, AgriEngineering, vol. 8, no. 1, p. 8, Jan. 2026, doi: 10.3390/agriengineering8010008.

[41] S. Nowaczyk, “Architectures for Building Agentic AI,” arXiv:2512.09458, 2025, doi: 10.48550/arXiv.2512.09458.

[42] OpenAI, “GPT-4 Technical Report,” arXiv:2303.08774, 2023, doi: 10.48550/arXiv.2303.08774.

[43] P. Patel et al., “Splitwise: Efficient Generative LLM Inference Using Phase Splitting”, in 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA), Jun. 2024, pp. 118–132. doi: 10.1109/ISCA59077.2024.00019.

[44] A. Plaat, M. van Duijn, N. van Stein, M. Preuss, P. van der Putten, and K. J. Batenburg, “Agentic Large Language Models, a survey”, jair, vol. 84, Dec. 2025, doi: 10.1613/jair.1.18675.

[45] M. E. Pollack and M. Ringuette, “Introducing the Tileworld: Experimentally Evaluating Agent Architectures,” in Proceedings of the Eighth AAAI Conference on Artificial Intelligence, 1990.

[46] S. Qiao et al., “Benchmarking Agentic Workflow Generation,” in International Conference on Learning Representations, 2025. [Online]. Available: https://openreview.net/forum?id=vunPXOFmoi.

[47] S. Russell and P. Norvig, “Artificial Intelligence: A Modern Approach.” Englewood Cliffs, NJ, USA: Prentice Hall, 1995.

[48] P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, “A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications,” arXiv:2402.07927, 2024, doi: 10.48550/arXiv.2402.07927.

[49] R. Sapkota, K. I. Roumeliotis, and M. Karkee, “AI Agents vs. Agentic AI: A Conceptual Taxonomy, Applications and Challenges”, Information Fusion, vol. 126, p. 103599, Feb. 2026, doi: 10.1016/j.inffus.2025.103599.

[50] R. Sapkota, R. Shrestha, M. Rijal, and M. Karkee, “LangChain vs. LangGraph vs. LangSmith: Taxonomies of Agentic AI Toolchains for End-to-End Orchestration,” TechRxiv, 2025, doi: 10.36227/techrxiv.175695645.52670060/v1.

[51] P. Schmidtová et al., “How Important is ‘Perfect’ English for Machine Translation Prompts?”, in Findings of the Association for Computational Linguistics: EACL 2026, Morocco: Association for Computational Linguistics, Mar. 2026, pp. 760–777. doi: 10.18653/v1/2026.findings-eacl.38.

[52] S. Schulhoff et al., “The Prompt Report: A Systematic Survey of Prompt Engineering Techniques,” arXiv:2406.06608, 2024, doi: 10.48550/arXiv.2406.06608.

[53] M. Sclar, Y. Choi, Y. Tsvetkov, and A. Suhr, “Quantifying Language Models’ Sensitivity to Spurious Features in Prompt Design or: How I Learned to Start Worrying About Prompt Formatting,” in The Twelfth International Conference on Learning Representations, 2024.

[54] Y. Shi et al., “Aime: Towards Fully-Autonomous Multi-Agent Framework,” arXiv:2507.11988, 2025, doi: 10.48550/arXiv.2507.11988.

[55] W. Tang et al., “AgentBuilder: Automating agent creation via large language model-driven systems”, Neurocomputing, vol. 646, p. 130476, Sep. 2025, doi: 10.1016/j.neucom.2025.130476.

[56] E. Toledo et al., “AI Research Agents for Machine Learning: Search, Exploration, and Generalization in MLE-bench”, Advances in Neural Information Processing Systems, vol. 38, pp. 35309–35348, Apr. 2026.

[57] O. Topsakal and T. C. Akinci, “Creating Large Language Model Applications Utilizing LangChain: A Primer on Developing LLM Apps Fast”, International Conference on Applied Engineering and Natural Sciences, vol. 1, pp. 1050–1056, Jul. 2023, doi: 10.59287/icaens.1127.

[58] A. Vaswani et al., “Attention is All you Need”, in Advances in Neural Information Processing Systems, Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[59] Y. Wan and K.-W. Chang, “White Men Lead, Black Women Help? Benchmarking and Mitigating Language Agency Social Biases in LLMs”, in Proceedings of the 63rd Annual Meeting of the Association for Computational

Linguistics (Volume 1: Long Papers), W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, Eds, Vienna, Austria: Association for Computational Linguistics, Jul. 2025, pp. 9082–9108. doi: 10.18653/v1/2025.acl-long.445.

[60] D. Y.-B. Wang, Z. Shen, S. S. Mishra, Z. Xu, Y. Teng, and H. Ding, "SLOT: Structuring the Output of Large Language Models", in Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track, S. Potdar, L. Rojas-Barahona, and S. Montella, Eds, Suzhou (China): Association for Computational Linguistics, Nov. 2025, pp. 472–491. doi: 10.18653/v1/2025.emnlp-industry.32.

[61] J. Wei et al., "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," in Advances in Neural Information Processing Systems, vol. 35, 2022.

[62] J. White et al., "A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT", in Proceedings of the 30th Conference on Pattern Languages of Programs, in PLoP '23. USA: The Hillside Group, Oct. 2023, pp. 1–31. [Online]. Available: https://dl.acm.org/doi/10.5555/3721041.3721046

[63] B. Xu et al., "ExpertPrompting: Instructing Large Language Models to be Distinguished Experts," arXiv:2305.14688, 2023, doi: 10.48550/arXiv.2305.14688.

[64] S.-Q. Yan, J.-C. Gu, Y. Zhu, and Z.-H. Ling, "Corrective Retrieval Augmented Generation," arXiv:2401.15884, 2024, doi: 10.48550/arXiv.2401.15884.

[65] C. Yang et al., "Graph-based Agent Memory: Taxonomy, Techniques, and Applications," arXiv:2602.05665, 2026, doi: 10.48550/arXiv.2602.05665.

[66] S. Yao et al., "ReAct: Synergizing Reasoning and Acting in Language Models," in International Conference on Learning Representations, 2023. [Online]. Available: https://openreview.net/forum?id=WE_vluYUL-X.

[67] R. Ye et al., "MAS-GPT: Training LLMs to Build LLM-based Multi-Agent Systems," in International Conference on Machine Learning, 2025. [Online]. Available: https://openreview.net/forum?id=3CiSpY3QdZ

[68] B. Yetiştiren, I. Özsoy, M. Ayerdem, and E. Tüzün, "Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT," arXiv:2304.10778, 2023, doi: 10.48550/arXiv.2304.10778.

[69] C. Yu et al., “A Survey on Agent Workflow – Status and Future”, in 2025 8th International Conference on Artificial Intelligence and Big Data (ICAIBD), May 2025, pp. 770–781. doi: 10.1109/ICAIBD64986.2025.11082076.

[70] L. Yue et al., “From Static Templates to Dynamic Runtime Graphs: A Survey of Workflow Optimization for LLM Agents,” arXiv:2603.22386, 2026, doi: 10.48550/arXiv.2603.22386.

[71] K. Zhang, J. Li, G. Li, X. Shi, and Z. Jin, “CodeAgent: Enhancing Code Generation with Tool-Integrated Agent Systems for Real-World Repo-level Coding Challenges”, in Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), L.-W. Ku, A. Martins, and V. Srikumar, Eds, Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 13643–13658. doi: 10.18653/v1/2024.acl-long.737.

[72] M. Zhang, O. Press, W. Merrill, A. Liu, and N. A. Smith, “How Language Model Hallucinations Can Snowball,” in International Conference on Machine Learning, 2024. [Online]. Available: https://openreview.net/forum?id=FPlaQyAGHu

[73] X. Zhang, X. Dong, Y. Wang, D. Zhang, and F. Cao, “A Survey of Multi-AI Agent Collaboration: Theories, Technologies and Applications”, in Proceedings of the 2nd Guangdong-Hong Kong-Macao Greater Bay Area International Conference on Digital Economy and Artificial Intelligence, in DEAI ‘25. New York, NY, USA: Association for Computing Machinery, Jul. 2025, pp. 1875–1881. doi: 10.1145/3745238.3745531.

[74] L. Zheng et al., “Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena,” in Advances in Neural Information Processing Systems, Datasets and Benchmarks Track, 2023. [Online]. Available: https://openreview.net/forum?id=uccHPGDlao

[75] J. Zhou et al., “Instruction-Following Evaluation for Large Language Models,” arXiv:2311.07911, 2023, doi: 10.48550/arXiv.2311.07911.