

Politecnico di Torino

Politecnico di Torino

Computer Engineering

Academic Year 2025/2026

Graduation Session September 2026

Automating the Software Development Lifecycle with Cooperative AI Agents and Human-in-the-Loop Supervision

Supervisor: Prof. Paolo Garza

Candidate: Francesco Alzate

Abstract

The growing capability of Large Language Models to generate and critique code has opened new possibilities for automating software development, yet the gap between generating isolated code snippets and delivering a complete, tested, and documented software artefact remains substantial. Bridging it requires not a single powerful model, but a structured collaboration among specialised components operating under principled human oversight.

This thesis presents MAST (Multi-Agent Software Team), a serverless multi-agent pipeline developed in partnership with Data Reply S.r.l. that orchestrates cooperative AI agents to automate the full Software Development Lifecycle (SDLC) — from natural-language requirements through iterative implementation and testing to documentation generation and version-controlled repository delivery. The pipeline is built on AWS serverless primitives: five agents are implemented as stateless Lambda functions invoking Amazon Bedrock, while two execution-heavy agents (Implementation, Test) run as long-running agentic loops inside sandboxed OpenHands containers on Amazon ECS Fargate. Orchestration is handled by an AWS Step Functions state machine. All inter-agent communication is mediated through structured JSON artefacts stored on a shared Amazon S3 bucket, decoupling agents from one another and providing a complete, auditable trace of every intermediate result.

A central contribution is the formalisation of *Dual-Loop Governance*: a Critic agent evaluates each implementation against a machine-readable JSON specification across up to five autonomous iterations, routing targeted fix directives to either the Implementation or Test agent depending on the nature of the defect. A stagnation-detection algorithm preempts the iteration budget when convergence stalls, escalating to a human checkpoint. Two human-in-the-loop checkpoints are embedded at the points of highest architectural risk — requirement ambiguity and Critic loop exhaustion — and are designed as first-class architectural components rather than ad-hoc fallbacks, implemented through Step Functions Task Tokens that suspend execution with zero compute consumption while waiting for operator input.

An empirical evaluation over 15 tasks spanning four complexity classes and five homogeneous model configurations identifies an optimal *Tiered* configuration that pairs frontier reasoning (Claude 3.5 Sonnet) at the contract-defining and evaluative nodes with cost-efficient execution (Claude 3.5 Haiku) in the high-volume OpenHands containers. This tiered system achieves 100% autonomous pass rate while reducing cost by 59.8% and latency by 12.0% relative to a Sonnet-homogeneous

baseline, establishing the sole non-dominated point on the Cost–Autonomy Pareto frontier across all configurations evaluated.

Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Tristique senectus et netus et malesuada fames ac turpis. Pretium nibh ipsum consequat nisl vel pretium lectus quam. Urna molestie at elementum eu facilisis sed odio morbi quis. Sed egestas egestas fringilla phasellus faucibus scelerisque eleifend. At in tellus integer feugiat. Mauris rhoncus aenean vel elit scelerisque mauris pellentesque pulvinar. Commodo sed egestas egestas fringilla. Nunc lobortis mattis aliquam faucibus purus in massa. Facilisis magna etiam tempor orci eu lobortis elementum nibh. Elementum curabitur vitae nunc sed velit dignissim. Neque volutpat ac tincidunt vitae. Massa id neque aliquam vestibulum morbi blandit cursus risus at. Porta non pulvinar neque laoreet suspendisse interdum consectetur. Turpis in eu mi bibendum. Ut tristique et egestas quis ipsum suspendisse. Integer quis auctor elit sed vulputate mi sit amet. Viverra nam libero justo laoreet sit amet cursus.

Table of Contents

List of Figures x

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Thesis Objectives and Contributions . . . 2
- 1.3 Thesis Organisation . . . 3

2 Background 5
- 2.1 Large Language Models . . . 5
  - 2.1.1 Architecture and Core Capabilities . . . 5
  - 2.1.2 LLMs for Code Generation . . . 6
  - 2.1.3 Prompt Engineering and Structured Output . . . 6
- 2.2 Multi-Agent Systems . . . 7
  - 2.2.1 Principles and Coordination Patterns . . . 7
  - 2.2.2 Agent Specialisation and Decomposition . . . 7
  - 2.2.3 Recent Frameworks for LLM-Based Multi-Agent Systems . . . 8
- 2.3 Tool Integration and the Model Context Protocol . . . 8
- 2.4 The AWS Execution Environment . . . 9
  - 2.4.1 Amazon Bedrock and the Converse API . . . 9
  - 2.4.2 AWS Step Functions . . . 10
  - 2.4.3 AWS Lambda and Amazon S3 . . . 10
- 2.5 Human-in-the-Loop AI Systems . . . 10
- 2.6 Related Work in Multi-Agent Software Engineering . . . 11

3 System Architecture 14
- 3.1 System Overview . . . 14
- 3.2 End-to-End Pipeline Flow . . . 15
- 3.3 The Shared Artefact Store . . . 18
  - 3.3.1 Prefix Structure . . . 18
  - 3.3.2 Why Artefact-Only Communication . . . 20
- 3.4 The Orchestration Layer . . . 20

3.4.1 State Machine Structure 20
3.4.2 Pipeline State Tracking 21
3.5 The Agent Layer 21
3.6 Supporting Components 23
3.6.1 The MCP Tool Integration Server 23
3.6.2 The Operator Dashboard 24
3.7 Architectural Design Principles 24

4 The Multi-Agent Pipeline 26
4.1 Design Philosophy 26
4.2 The Clarification Agent 27
4.2.1 Role and Responsibility 27
4.2.2 Prompt Design 27
4.2.3 Input/Output Schema 28
4.3 The Specification Agent 28
4.3.1 Role and Responsibility 28
4.3.2 Prompt Design 28
4.3.3 Output Schema 29
4.4 The Implementation Agent 30
4.4.1 Execution Model and OpenHands Mechanics 30
4.4.2 Execution Security and Isolation 31
4.4.3 Prompt Design and Tool Configuration 32
4.4.4 Transcript Logging and Observability 32
4.4.5 Output Packaging 32
4.5 The Test Agent 33
4.5.1 Role and Responsibility 33
4.5.2 Test Suite Design 33
4.5.3 Output Schema 33
4.6 The Critic Agent and Feedback Loop 34
4.6.1 Role and Three-Way Validation 34
4.6.2 Output Schema 34
4.6.3 Stagnation Detection 35
4.6.4 The Iteration Gate 36
4.7 The Documentation Agent 37
4.7.1 Role and Responsibility 37
4.7.2 Prompt Design 37
4.7.3 Output 37
4.8 The GitLab Push Agent 38
4.9 Token Accounting and Cost Observability 38

5 Human-in-the-Loop Mechanisms 40
5.1 Design Rationale for Human Checkpoints . . . 40
5.2 The Operator Interface and Security Model . . . 41
5.2.1 Backend Architecture . . . 42
5.2.2 Security: The SigV4 Proxy Pattern . . . 42
5.3 Execution Lifecycle from the Operator's Perspective . . . 43
5.4 Requirement Clarification Checkpoint . . . 43
5.4.1 Structure of the Clarification Output . . . 43
5.4.2 Feedback Propagation . . . 45
5.5 Critic Escalation Checkpoint . . . 45
5.5.1 Trigger Conditions . . . 45
5.5.2 Information Presentation . . . 46
5.5.3 Operator Intervention and Re-injection . . . 46

6 Empirical Evaluation 47
6.1 Evaluation Methodology . . . 47
6.1.1 Benchmark Suite . . . 47
6.1.2 Outcome Definitions . . . 48
6.1.3 Evaluation Protocol . . . 49
6.2 Baseline: Claude 3.5 Sonnet . . . 50
6.3 Cross-Model Homogeneous Comparison . . . 51
6.3.1 Evaluated Configurations . . . 51
6.3.2 Aggregate Results . . . 51
6.3.3 Claude 3.5 Haiku . . . 51
6.3.4 GPT-4o . . . 53
6.3.5 Llama 3.3 70B Instruct . . . 53
6.3.6 Amazon Nova Pro . . . 54
6.4 The Heterogeneous Tiered Configuration . . . 54
6.4.1 Allocation Rationale . . . 54
6.4.2 Empirical Validation . . . 56
6.4.3 The Cost–Autonomy Pareto Frontier . . . 57
6.5 Human-in-the-Loop Dynamics . . . 57
6.5.1 Checkpoint 1 as an Architectural Safety Barrier . . . 57
6.5.2 Checkpoint 2 Recovery Efficacy and the Role of the Critic . 59
6.6 Token Telemetry and Prompt Caching . . . 60
6.7 Discussion . . . 61
6.7.1 The Specification Contract as an LLM Equaliser . . . 61
6.7.2 The Structural Role of Human-in-the-Loop and the Limits of Autonomous Recovery . . . 62
6.7.3 Threats to Validity . . . 62

7 Conclusions and Future Work 64
7.1 Summary of Contributions 64
7.2 Limitations of the Current System 66
7.3 Directions for Future Work 68
7.4 Concluding Remarks 69

Bibliography 70

List of Figures

3.1 Five-layer system architecture 16
3.2 End-to-end multi-agent pipeline flow 17
3.3 Hierarchical S3 artefact store structure 19
3.4 AWS Step Functions state machine execution graph 22

4.1 ReAct Action–Observation cycle 31

5.1 Operator Dashboard composite interface 41
5.2 Task execution timeline in Operator Interface 44

6.1 Outcome distribution across all six evaluated configurations. Each bar represents the 15 benchmark tasks decomposed into autonomous passes (green), human-assisted passes at Checkpoint 2 (blue), and hard failures (red). 52
6.2 Cost vs. autonomous pass rate for all six configurations. Marker area is proportional to average subjective code quality (scale 1–5). The dashed line indicates the Pareto frontier: the Tiered configuration is the only point that achieves 100% autonomous success at minimum cost. All other configurations are strictly dominated or equivalent. . 57
6.3 Per-agent token telemetry and cost breakdown 61

Chapter 1

Introduction

The process of developing software has always been a fundamentally human endeavour: engineers read requirements, debate design choices, write code, craft test suites, review one another's work, and iterate until a solution is both correct and maintainable. Each of these activities demands a different kind of expertise and, taken together, they account for a significant share of the time and cost of any software project. The emergence of Large Language Models (LLMs) has begun to challenge this picture. Systems such as GPT-4 [1], Claude [2], and the models available through Amazon Bedrock [3] have demonstrated a substantially improved ability to generate, explain, and critique code compared to systems available only a few years prior. Yet the gap between generating a snippet of code and reliably delivering a complete, tested, and documented software artefact remains wide. Bridging that gap requires not a single powerful model, but a structured collaboration among multiple specialised agents, each responsible for a well-defined stage of the development process.

This thesis investigates how such a collaboration can be designed, implemented, and evaluated. Specifically, it documents the agentic component of a larger research project carried out in partnership with Data Reply S.r.l., a system that orchestrates a pipeline of cooperative AI agents to automate the full Software Development Lifecycle (SDLC): from the initial clarification of ambiguous requirements, through the generation and iterative refinement of an implementation, to automated testing, code review, and documentation. A central theme throughout the work is the role of human supervision: rather than treating automation as a binary goal, the system is designed so that a human operator can be brought into the loop at precisely those points where autonomous decision-making would be insufficient or risky.

1.1 Motivation

The idea of automating software development is not new. Model-driven engineering, program synthesis, and search-based software engineering have each attempted, in different ways, to reduce the manual effort involved in producing correct software [4]. What distinguishes the current moment is the scale and generality of LLM-based approaches. The Codex model [5], which underpins GitHub Copilot, demonstrated that LLMs can serve as effective code-completion assistants in real-world workflows, accelerating individual developers. However, completion assistance and autonomous end-to-end generation are fundamentally different problems: the former operates at the level of a function or a few lines, while the latter requires maintaining coherent context across requirements, architecture, implementation, and verification.

Multi-agent systems offer a principled way to decompose this harder problem. By assigning each stage of the SDLC to a dedicated agent with a focused prompt and a well-defined input/output contract, it becomes possible to manage context more effectively, to make failures locally diagnosable, and to iterate on individual stages without disturbing the rest of the pipeline [6]. Recent work on frameworks such as AutoGen [7] and OpenHands [8] has shown that multi-agent architectures can produce non-trivial software artefacts, but questions of reliability, quality control, and appropriate human oversight remain open.

The industrial setting of this project adds a further dimension: the pipeline runs on a cloud-native, serverless infrastructure built on AWS, which means that scalability, observability, and cost are real engineering constraints rather than theoretical concerns. Designing agents that respect these constraints — stateless Lambda functions communicating through a shared artefact store, orchestrated by a state machine — forces a level of architectural discipline that has its own research value.

1.2 Thesis Objectives and Contributions

The overarching objective of this thesis is to design, implement, and evaluate a multi-agent pipeline capable of automating the SDLC in a reliable and controllable manner. Within this broader goal, the specific contributions are the following.

- **Design of a cooperative multi-agent pipeline.** The work defines a structured pipeline of six agents — Clarification, Specification, Implementation, Test, Critic, and Documentation, plus a GitLab Push agent for the final delivery step — each with a clearly scoped responsibility and a shared artefact protocol based on JSON documents stored on Amazon S3. The pipeline is orchestrated by an AWS Step Functions state machine that manages control flow, retries, and branching logic without requiring a dedicated orchestration Lambda.

- **Iterative quality control through a Critic feedback loop.** A dedicated Critic agent reviews each implementation against the structured specification produced earlier in the pipeline. When the implementation does not meet the required confidence threshold, the Critic returns structured feedback that is fed directly into the next implementation attempt. This iterative loop continues for up to five autonomous iterations before escalating to a human checkpoint, and is designed to provide a structured mechanism for quality assurance that minimises the need for human involvement in the common case.

- **Human-in-the-Loop integration at design-critical junctures.** The system incorporates two human checkpoints: one triggered when the Clarification agent identifies requirements too ambiguous to resolve autonomously, and one triggered when the Critic loop fails to converge within the allowed iterations or stagnates. At each checkpoint, the operator receives a structured summary of the pipeline state and a task token; their reply resumes execution and can steer the next autonomous phase. This design treats human supervision as a first-class architectural component rather than a fallback.

- **Agent communication via a shared artefact protocol.** Agents do not call one another directly. Instead, each agent reads its inputs from and writes its outputs to a well-defined set of JSON files on a shared S3 bucket, under a per-task prefix. This decoupling makes individual agents independently testable and replaceable, and it provides a natural audit trail of every intermediate artefact produced during a pipeline run.

- **Proof-of-concept evaluation.** The complete pipeline is exercised on a representative software development task in order to validate the end-to-end flow, observe the behaviour of the Critic feedback loop, and identify the conditions under which human checkpoints are triggered. The evaluation surfaces both the capabilities and the current limitations of the approach.

The infrastructure on which the pipeline runs — VPC configuration, Terraform modules, CloudWatch monitoring, the operator dashboard, and the CI/CD pipeline — was developed in parallel as part of the same project by a collaborating student. This thesis focuses exclusively on the agentic layer: the design of the agents, their communication protocol, the orchestration logic, and the human-in-the-loop mechanisms.

1.3 Thesis Organisation

The remainder of this thesis is structured as follows.

- **Chapter 2** reviews the background concepts necessary to understand the system: Large Language Models and their application to code generation, the principles of multi-agent systems, the Model Context Protocol (MCP) used for tool integration, and the AWS services that form the execution environment.
- **Chapter 3** presents the overall system architecture, describing how the agents, the orchestrator, the artefact store, and the operator-facing components fit together and interact.
- **Chapter 4** is the core technical contribution of the thesis. It describes each agent in detail — its prompt design, its input/output artefacts, its security model, and its failure modes — as well as the Critic feedback loop and the artefact communication protocol.
- **Chapter 5** examines the Human-in-the-Loop mechanisms: when checkpoints are triggered, what information is presented to the operator, and how the operator's reply is propagated back into the pipeline.
- **Chapter 6** presents the empirical evaluation of the system, analysing the results of a set of representative software development tasks submitted to the pipeline. It examines iteration efficiency, the evolution of Critic scores, and qualitative observations on the quality of the generated artefacts.
- **Chapter 7** summarises the findings, reflects on the limitations of the current prototype, and outlines the most promising directions for future work.

Chapter 2

Background

This chapter provides the technical foundations required to understand the system described in subsequent chapters. It covers six areas: the architecture and capabilities of Large Language Models, with particular attention to their application to code generation; the principles of multi-agent systems; tool integration through the Model Context Protocol; the AWS services that form the execution environment; the concept of Human-in-the-Loop AI; and an overview of related work in multi-agent software engineering.

2.1 Large Language Models

2.1.1 Architecture and Core Capabilities

Large Language Models are a class of neural networks trained on massive corpora of text with the objective of modelling the probability distribution over sequences of tokens. The architectural foundation shared by virtually all contemporary LLMs is the Transformer [9], introduced by Vaswani et al. in 2017. The Transformer replaces the recurrent connections of earlier sequence models with a mechanism of *self-attention*, which allows each token in a sequence to attend to every other token simultaneously. This property makes the architecture highly parallelisable during training and enables the model to capture long-range dependencies that recurrent networks struggled to maintain.

Scaling such architectures — in terms of parameter count, training data volume, and compute — has produced a series of qualitative improvements in model behaviour. GPT-3 [10], with 175 billion parameters, demonstrated that a single model trained only on next-token prediction could perform a wide variety of tasks without task-specific fine-tuning, simply by conditioning on a natural-language prompt. Subsequent models such as GPT-4 [1] and the Claude family [2] have

extended these capabilities with improved instruction-following, longer context windows, and stronger reasoning across complex multi-step problems.

Of particular relevance to this work is the ability of modern LLMs to produce *structured outputs*: when prompted appropriately, these models reliably generate valid JSON objects, following a schema specified in the prompt. This property is central to the communication protocol used in the pipeline described in Chapter 4, where agents are required to write their results in machine-parseable JSON documents rather than free-form prose.

2.1.2 LLMs for Code Generation

The application of LLMs to software engineering has attracted substantial research attention. Chen et al. introduced Codex [5], a model fine-tuned from GPT-3 on publicly available code from GitHub, and demonstrated its ability to solve a diverse set of programming tasks from natural-language specifications. Codex forms the backbone of GitHub Copilot, a widely adopted code-completion assistant that integrates directly into development environments.

Subsequent work has explored tasks beyond completion: generating entire functions or classes from a docstring, producing test suites for existing code, identifying and explaining bugs, and translating code between programming languages [11]. More recent general-purpose models such as Claude Sonnet and GPT-4 have, in several widely used benchmarks, approached or surpassed specialised code models [12].

A critical observation for this thesis is that code generation, while impressive, does not on its own constitute a reliable engineering process. A model generating code in a single pass has no mechanism for detecting its own logical errors, for verifying that the output satisfies a formal specification, or for integrating the results of automated tests. Addressing these limitations requires wrapping the generative capability in an iterative, multi-stage process — precisely the design explored in this work.

2.1.3 Prompt Engineering and Structured Output

The behaviour of an LLM is primarily controlled through its input prompt. Prompt engineering [13] is the discipline of constructing prompts that elicit accurate, reliable, and appropriately formatted responses. Several techniques have proven particularly effective in the context of agentic systems.

System prompts and role assignment. The Converse API exposed by Amazon Bedrock distinguishes between a *system prompt*, which sets the model's persona and constraints, and *user turns*, which carry the actual input. By defining a strict persona — for instance, *"You are a senior software engineer. Your output*

must be valid JSON. Do not produce any text outside the JSON object" — the system prompt significantly reduces the probability of the model deviating from the expected output format.

Chain-of-thought reasoning. Wei et al. showed that prompting a model to produce intermediate reasoning steps before a final answer substantially improves accuracy on complex tasks [14]. In the context of code review and specification generation, this technique encourages the model to articulate design constraints, edge cases, and failure modes before producing the final structured document.

ReAct and tool-augmented agents. Yao et al. proposed the ReAct framework [15], in which a model interleaves reasoning traces with actions (tool calls). This paradigm is central to the Implementation and Test agents in the pipeline, which use the OpenHands SDK to run an agentic loop in which the model can invoke file editors, a terminal, and other tools before declaring the task complete.

2.2 Multi-Agent Systems

2.2.1 Principles and Coordination Patterns

A multi-agent system (MAS) is a computational system composed of multiple autonomous agents that perceive their environment, pursue individual or shared goals, and interact with one another [6]. Agents in a MAS can be cooperative, competitive, or a mixture of both; in the context of AI-driven software engineering, the relevant pattern is cooperative: each agent is designed to contribute a specialised capability to a shared goal, and the system as a whole produces an output that no single agent could produce alone.

Coordination among agents in a cooperative MAS can follow different patterns. In a *blackboard architecture* [16], agents read from and write to a shared data structure, which serves both as a communication channel and as a persistent record of intermediate results. In a *pipeline architecture*, agents are arranged in a directed sequence, with the output of each agent becoming the input of the next. In a *hierarchical architecture*, a dedicated orchestrator agent directs the activities of subordinate agents, decomposing tasks and aggregating results.

The system described in this thesis combines elements of all three patterns: a pipeline structure provides the primary control flow, a shared S3 artefact store functions as a blackboard, and a Step Functions state machine acts as an orchestrator that manages branching, iteration, and error handling.

2.2.2 Agent Specialisation and Decomposition

A key motivation for multi-agent architectures in LLM-based systems is *context management*. An LLM has a finite context window; a single agent tasked with

clarifying requirements, writing a specification, implementing code, testing it, reviewing the result, and documenting the output would need to maintain all of this information simultaneously, rapidly exhausting its context budget. By decomposing the task across multiple agents, each of which operates on a focused subset of the information, the system can handle workloads that would be intractable for a single-agent approach.

Specialisation also enables differentiated prompt design: the Clarification agent's prompt is tuned for detecting ambiguity and generating precise questions, while the Critic agent's prompt is tuned for evaluating code against a formal specification. This separation of concerns would be difficult to maintain within a single monolithic agent.

2.2.3 Recent Frameworks for LLM-Based Multi-Agent Systems

Several frameworks have emerged to simplify the construction of multi-agent systems based on LLMs. AutoGen [7] provides a conversation-based abstraction in which agents communicate by exchanging natural-language messages and can be configured with custom reply logic and tool access. LangGraph [17] extends the LangChain ecosystem with a graph-based orchestration model in which nodes represent agent steps and edges represent conditional transitions.

The system in this thesis does not adopt an off-the-shelf multi-agent framework. Instead, agents are implemented as standalone AWS Lambda functions, and the orchestration logic is expressed directly in the Amazon Web Services Step Functions state machine language. This design is intended to provide predictability, auditability, and tight integration with the serverless infrastructure, at the cost of some of the convenience offered by higher-level frameworks. Each transition, retry, and branching decision is an explicit state in a versioned state machine definition, making the system behaviour fully observable through the AWS console and CloudWatch.

2.3 Tool Integration and the Model Context Protocol

As LLM-based agents are applied to increasingly complex tasks, the ability to call external tools — APIs, databases, file systems, execution environments — becomes essential. Several approaches to tool integration have been proposed. The function-calling mechanism offered by the OpenAI and Anthropic APIs allows models to request the invocation of predefined functions by emitting a structured JSON object; the host application executes the function and returns the result

in the next user turn. This pattern is effective but requires each application to implement its own adapter layer.

The **Model Context Protocol** (MCP) [18] is an open standard proposed by Anthropic to provide a uniform interface for connecting LLM agents to external tools and data sources. MCP defines a client-server architecture: an MCP server exposes a set of tools, each described by a name, a human-readable description, and a JSON Schema specifying its parameters; an MCP client (typically the agent or its host application) discovers available tools, selects those relevant to the current task, and invokes them by sending structured requests to the server.

In the system described in this thesis, an MCP server is deployed as an AWS Lambda function fronted by an API Gateway with IAM authentication. The server exposes tools that provide agents with access to enterprise data sources — in the current implementation, a single `fetch_company_data` tool that queries a simulated legacy system. The OpenHands Implementation and Test agents invoke these tools through a SigV4-signing client library baked into the container image. The MCP layer thus allows agent capabilities to be extended without modifying the agents themselves: new tools can be added to the server, and agents discover them automatically at runtime.

2.4 The AWS Execution Environment

This section briefly describes the AWS services that form the execution substrate of the pipeline. A full account of the infrastructure design is outside the scope of this thesis; the description here is limited to the aspects necessary to understand the agentic components.

2.4.1 Amazon Bedrock and the Converse API

Amazon Bedrock [3] is a managed service that provides API access to a curated selection of foundation models from multiple providers, including Anthropic (Claude), Amazon (Nova), Meta (Llama), and others. All reasoning agents in the pipeline access Bedrock through the **Converse API**, a unified interface that accepts messages in a standard format regardless of the underlying model. This abstraction allows the model powering each agent to be changed — by updating a single environment variable — without modifying any agent code.

Each Bedrock request carries a `maxTokens` parameter capping the length of the generated response. In the pipeline, this parameter is set per agent based on expected output size: 4096 tokens for the Clarification, Critic, and Documentation agents, and 8192 tokens for the Specification agent, which must produce a detailed structured document. All agents guard against truncated responses by checking whether the `stopReason` returned by the API is `max_tokens` rather than `end_turn`,

and they return an explicit error rather than allowing the caller to parse an incomplete JSON object.

2.4.2 AWS Step Functions

AWS Step Functions [19] is a serverless orchestration service that allows the definition of workflows as state machines using the Amazon States Language (ASL). A state machine is composed of *states* — Task, Choice, Parallel, Wait, Pass, Succeed, Fail — connected by transitions that can be conditional or unconditional.

Step Functions is well suited to orchestrating a multi-agent pipeline because it externalises all control flow from the agent code. Retry logic, conditional branching (e.g., the Critic gate that decides whether to loop, escalate, or proceed), and error handling are declared in the state machine definition rather than scattered across agent implementations. The service also natively integrates with Lambda, ECS, and several AWS SDK calls, enabling the state machine to invoke Lambda functions, launch ECS Fargate tasks, and write to DynamoDB — all without a dedicated orchestrator Lambda consuming compute.

2.4.3 AWS Lambda and Amazon S3

AWS Lambda [20] is a serverless compute service that executes code in response to events, without requiring the management of underlying servers. In the pipeline, the four reasoning agents (Clarification, Specification, Critic, Documentation) and the GitLab Push agent are each implemented as a Lambda function. Lambda's stateless execution model aligns naturally with the agent design: each invocation reads its inputs from S3, performs a single Bedrock inference, and writes its outputs back to S3, with no state held in memory between invocations.

Amazon S3 [21] serves as the shared artefact store for the entire pipeline. Each pipeline execution is identified by a unique `task_id`, and all artefacts for that execution are stored under the prefix `req-{task_id}/`. Agents read their inputs from and write their outputs to well-defined keys within this prefix, following a convention described in detail in Chapter 4. This design decouples agents from one another: no agent needs to know the network address or invocation ARN of any other agent; it only needs to know the S3 key where its input will appear.

2.5 Human-in-the-Loop AI Systems

The term *Human-in-the-Loop* (HITL) refers to a class of AI system designs in which human judgement is deliberately incorporated into an otherwise automated process [22]. HITL is motivated by two complementary considerations. The first is the recognition that current AI systems, despite their capabilities, are not

uniformly reliable: they can generate plausible-sounding but incorrect outputs, and the conditions under which they fail are not always predictable in advance. The second is the observation that human attention is a scarce resource: a system that requires human review of every intermediate step provides no automation benefit, while a system that never involves a human provides no quality guarantee for cases where the model is unreliable.

Effective HITL design, therefore, requires answering three questions: *when* should human judgement be invoked, *what* information should be presented to the human at that point, and *how* should the human's response be integrated back into the automated process [23, 24].

In the system described in this thesis, these questions are answered as follows. Human intervention is triggered at two specific points: when the Clarification agent identifies requirements too ambiguous for autonomous processing, and when the Critic agent's quality evaluation fails to converge within the allowed autonomous iterations. At each checkpoint, the operator receives a structured summary of the pipeline state — including the questions raised or the evaluation scores produced — alongside a task token that, when submitted, resumes the execution. The operator's reply can include free-text feedback that is injected as an additional context signal into the next autonomous phase, allowing the human's domain knowledge to steer subsequent agent behaviour.

This design treats human supervision as an *architectural* component rather than a patch on a fully autonomous system. The checkpoints are defined in the state machine definition, invoked through a standard AWS Step Functions `waitForTaskToken` mechanism, and their inputs and outputs are typed in the same JSON schema used by the automated agents. The result is a system in which the boundary between autonomous and human-assisted processing is explicit, auditable, and adjustable without modifying agent code.

2.6 Related Work in Multi-Agent Software Engineering

The idea of organising multiple LLM-based agents to collaboratively produce software has attracted growing research interest. ChatDev [25] introduces a chat-based software development framework in which LLM agents assume distinct roles — product manager, programmer, reviewer, tester — and interact through a structured dialogue inspired by software engineering practices. The framework demonstrated that role-specialised LLM agents can produce functional small-scale software products from a natural-language description alone. MetaGPT [26] takes a similar approach but introduces a more formal structure: agents follow standardised operating procedures (SOPs) and communicate through structured documents such

as requirements specifications and design artefacts, rather than through free-form dialogue. Both systems share the pipeline and specialisation principles adopted in this thesis, but neither addresses the problem of deploying agent pipelines in cloud-native, production-grade infrastructure, nor do they provide mechanisms for integrating enterprise tools via a standardised protocol.

A complementary line of work focuses on evaluating the performance of LLM-based agents on realistic software engineering tasks. SWE-bench [12] is a benchmark comprising 2,294 real GitHub issues, each paired with a test suite that verifies a candidate fix. It provides a rigorous, reproducible evaluation framework for assessing whether an agent system can autonomously resolve issues in large, real-world codebases. While this thesis does not evaluate against SWE-bench directly — the scope is a controlled proof-of-concept rather than a competitive benchmark evaluation — SWE-bench represents the natural direction for future empirical work once the pipeline matures.

The system described in this thesis differentiates itself from the above frameworks along four dimensions.

Structured artefact-only communication. Both ChatDev and MetaGPT rely, to varying degrees, on conversational exchanges between agents: agents read and produce natural-language messages that must be interpreted by the next agent in the pipeline. This creates a surface for hallucination propagation — an imprecise description produced by one agent can be silently misinterpreted by the next. The system in this thesis eliminates conversational inter-agent communication entirely. Agents do not exchange messages; instead, each agent reads a strictly typed JSON document from a shared S3 artefact store and writes a strictly typed JSON document in return. The schema of each document is enforced by the agent's system prompt, and any response that fails to parse as valid JSON causes an explicit error rather than a silent downstream failure. This design is intended to ground every agent's input in a machine-verifiable document rather than a reconstructed conversational context, with the aim of reducing the opportunity for accumulated imprecision across pipeline stages.

Separation of reasoning and action. Reasoning agents (Clarification, Specification, Critic, Documentation) run as stateless AWS Lambda functions: they call a Bedrock model, read and write S3 artefacts, and terminate. Code generation and test execution, however, require a fundamentally different execution model: an agent that must write files, run a compiler, interpret test output, and iterate. These steps are handled by two dedicated agents (Implementation and Test) running inside an OpenHands container on AWS ECS Fargate — an isolated, ephemeral sandbox in which the agent can invoke file editors, a terminal, and external tools freely without affecting the rest of the infrastructure. This explicit separation prevents the side effects of code execution from bleeding into the reasoning layer, a boundary that conversation-based frameworks do not enforce.

Quantitative quality gating via the Critic agent. Rather than relying on a binary pass/fail review or a free-form evaluation comment, the Critic agent in this system produces a structured JSON evaluation containing a numeric confidence score, a categorical verdict, and a list of actionable feedback items keyed to specific aspects of the specification (the full schema is described in Chapter 4). The Step Functions gate that follows applies a quantitative threshold to the confidence score and checks for stagnation before deciding whether to loop, escalate, or proceed. This approach is intended to make the pipeline's quality control behaviour more reproducible and auditable than purely qualitative evaluations.

Enterprise tool integration via the Model Context Protocol. Existing academic frameworks assume that agents operate on self-contained tasks with no dependency on external enterprise systems. The system in this thesis integrates legacy data sources through an MCP server — deployed as a Lambda function fronted by an IAM-authenticated API Gateway — that exposes typed, discoverable tools to the agent runtime. Agents call these tools through a standardised client without any knowledge of the underlying infrastructure, and new tools can be added to the server without modifying agent code. This design represents a step toward integrating multi-agent pipelines with enterprise data sources, an area that research-grade frameworks do not currently address.

Chapter 3

System Architecture

This chapter describes the overall architecture of the multi-agent system, providing a map of its components and the relationships among them before the individual agents and human-in-the-loop mechanisms are examined in detail in Chapters 4 and 5. The goal here is to establish a shared vocabulary and a structural reference that subsequent chapters can build on.

3.1 System Overview

At the highest level, the system is a *serverless multi-agent pipeline* that takes a natural-language software requirement as input and, subject to optional human review at designated checkpoints, produces a tested, documented, and version-controlled software artefact as output. The pipeline is fully event-driven: no component polls or busy-waits; every transition is triggered either by the completion of an agent invocation, by a message from the orchestrator, or by an explicit human reply.

The system is composed of five logical layers, illustrated in Figure 3.1, each with a well-defined responsibility:

1. **The Orchestration Layer** — an AWS Step Functions state machine that governs the sequence of agent invocations, manages branching and iteration, enforces termination conditions, and coordinates human checkpoints.

2. **The Agent Layer** — a set of six specialised agents (Clarification, Specification, Implementation, Test, Critic, Documentation) plus a delivery agent (GitLab Push), each responsible for a specific stage of the Software Development Lifecycle.

3. **The Artefact Store** — a shared Amazon S3 bucket that serves as the sole communication medium between agents. Every agent reads its inputs from

and writes its outputs to this store; no agent communicates with any other agent directly.

4. **The Tool Integration Layer** — an MCP server that exposes authenticated, typed tools to agents that require access to external data sources or enterprise systems.

5. **The Operator Interface** — a dashboard and a notification subsystem through which a human operator can monitor pipeline executions, respond to human checkpoints, and inspect intermediate artefacts.

The AWS infrastructure on which these layers run — VPC configuration, IAM roles, Terraform modules, CloudWatch monitoring, and the CI/CD pipeline — was designed and implemented by a collaborating student and is outside the scope of this thesis. The description in this chapter is limited to the aspects of the infrastructure that are necessary to understand the behaviour of the agentic components.

3.2 End-to-End Pipeline Flow

A pipeline execution is initiated by submitting a natural-language requirement to the system, together with a unique `task_id` that identifies the execution. The orchestrator then drives the pipeline through the sequence of stages illustrated in Figure 3.2:

1. **Clarification.** The Clarification agent analyses the requirement and determines whether it is sufficiently precise for autonomous processing. If the requirement contains ambiguities that would propagate downstream — undefined terms, conflicting constraints, missing scope boundaries — the agent formulates a set of targeted questions and the pipeline pauses at the first human checkpoint, awaiting operator clarification before proceeding. If the requirement is unambiguous, the pipeline advances directly to the next stage.

2. **Specification.** The Specification agent transforms the (possibly clarified) requirement into a structured technical specification: a JSON document that captures functional requirements, expected inputs and outputs, edge cases, acceptance criteria, and a suggested implementation approach. This document serves as the authoritative reference for all subsequent agents; its precision at this stage directly affects the quality of the final artefact.

3. **Implementation.** The Implementation agent, running inside an OpenHands container on AWS ECS Fargate, reads the structured specification and produces a working implementation. The agent operates in an agentic loop: it can

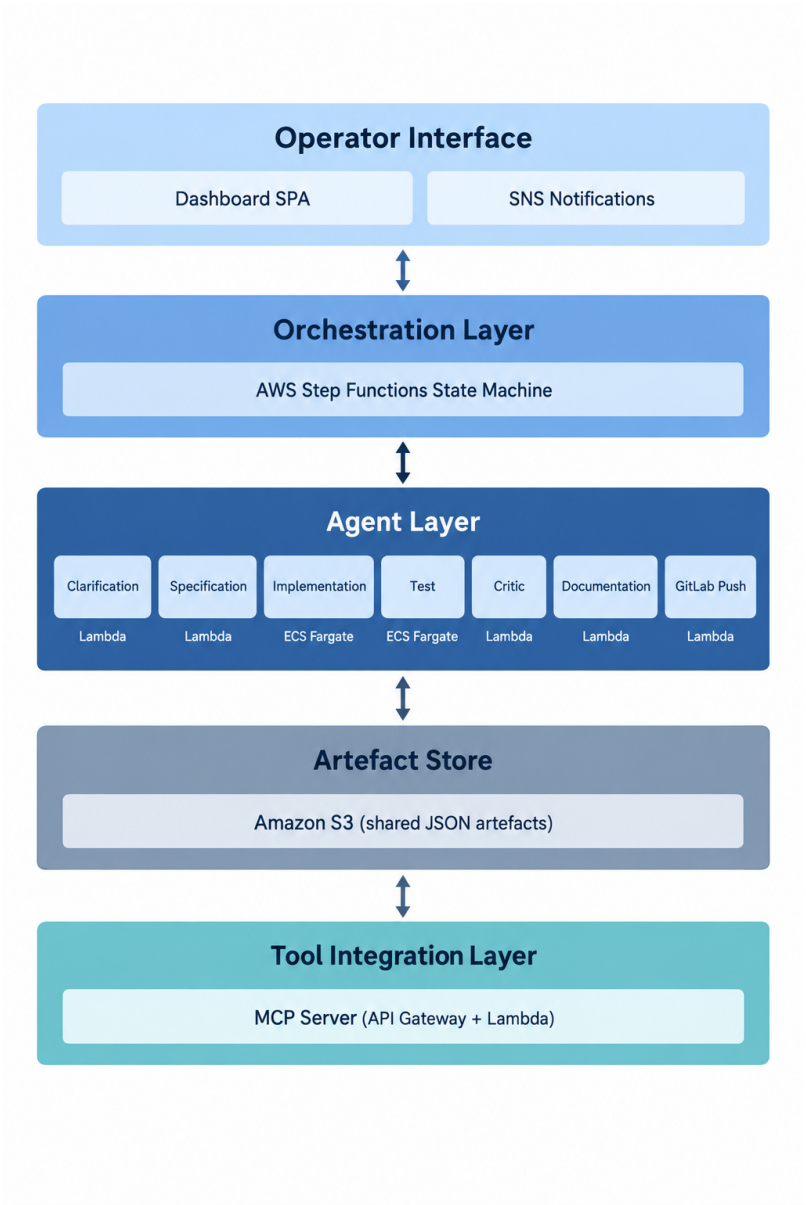


Figure 3.1: Five-layer system architecture of the multi-agent development pipeline, detailing the structural separation between operator interaction, workflow orchestration, agent compute runtimes, shared S3 artefact storage, and tool integration.

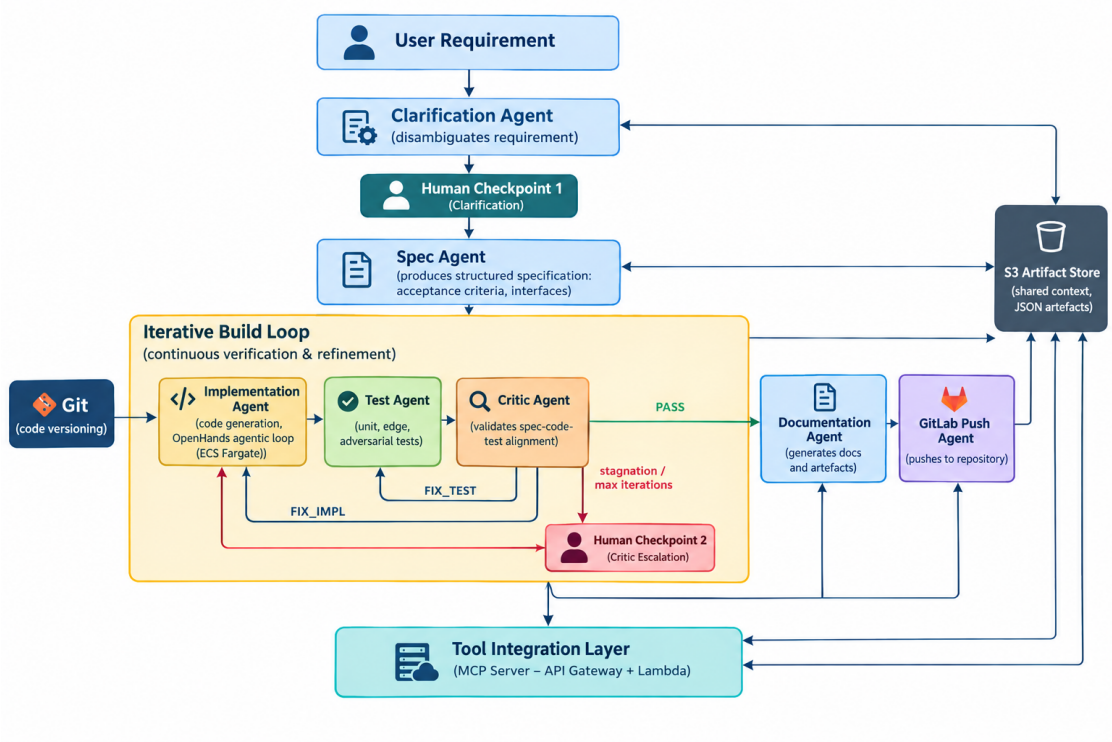


Figure 3.2: End-to-end multi-agent pipeline flow, highlighting the iterative build loop between Implementation, Test, and Critic agents, the two human checkpoint gates, the shared S3 artefact store, and the MCP tool integration layer.

write and edit files, execute shell commands, install dependencies, and inspect outputs before declaring the implementation complete. The result is packaged as a `workspace.zip` archive and written to the artefact store.

4. **Testing.** The Test agent, also running inside an OpenHands container, downloads the implementation artefact and the specification, writes an adversarial test suite covering multiple quality dimensions (happy path, edge cases, boundary conditions, negative tests, idempotency, and others), executes it with `pytest`, and uploads the updated workspace together with a `test_results.json` document summarising the outcomes.

5. **Critic evaluation.** The Critic agent reviews the implementation and test results against the specification and produces a structured evaluation: a confidence score on a 0–100 scale, a categorical verdict, and a list of actionable feedback items. The Step Functions gate that follows the Critic applies the following branching logic:

   - If the confidence score meets or exceeds the threshold of 70 and the verdict

is PASS, the pipeline proceeds to the Documentation stage.

- If the score is below the threshold, the pipeline loops back to the Implementation stage, injecting the Critic's feedback as additional context for the next attempt. This loop continues for up to five iterations.
- If the Critic detects stagnation — successive iterations that produce negligible improvement — or if the maximum iteration count is reached without convergence, the pipeline escalates to the second human checkpoint.

6. **Documentation.** The Documentation agent reads the final specification, the approved implementation, and the complete Critic history, and produces a `README.md` and a `CHANGELOG.md` for the artefact. These documents are written to the artefact store alongside the implementation.

7. **Delivery.** The GitLab Push agent reads the approved artefacts from the store, retrieves the repository credentials from AWS Secrets Manager, and pushes the deliverables to the designated GitLab repository. In the current implementation, this step logs the intended push rather than executing the actual API call, matching the scope of the original system design.

3.3 The Shared Artefact Store

The shared artefact store is an Amazon S3 bucket that functions as the single source of truth for all pipeline state. Its role is central to the system's architecture: rather than passing data between agents through function calls, API responses, or conversational messages, the pipeline uses the artefact store as a *typed, persistent, shared memory* that any agent can read from and write to independently.

3.3.1 Prefix Structure

All artefacts for a given execution are stored under the prefix `req-{task_id}/`, providing natural isolation between concurrent executions. Within this prefix, artefacts are organised hierarchically by stage and iteration, as shown in the directory tree below and illustrated in Figure 3.3:

```
req-{task_id}/
  00_requirement.txt          (input requirement)
  01_clarification.json       (Clarification agent output)
  02_structured_spec.json     (Specification agent output)
  phase-{n}/
    iter-{k}/
      workspace.zip           (Implementation agent output)
```

```
    test_results.json        (Test agent output)
    critic_evaluation.json   (Critic agent output)
99_documentation/
  README.md
  CHANGELOG.md
```

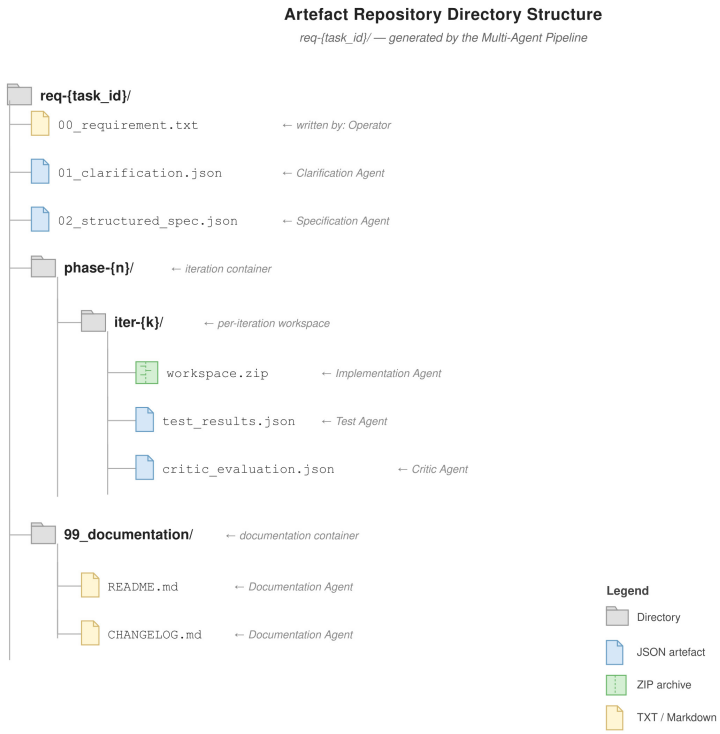


Figure 3.3: Hierarchical directory structure of the Amazon S3 artefact store under the `req-{task_id}/` prefix, detailing input, per-phase, per-iteration, and final documentation deliverables with their producing agents and file formats.

The `phase` counter increments each time a human checkpoint is triggered and the operator approves a new autonomous phase; the `iter` counter increments with each Critic feedback loop within a phase. This naming scheme means that the full history of every autonomous iteration, across every human-approved phase, is preserved in the artefact store and remains available for inspection after the pipeline completes.

3.3.2 Why Artefact-Only Communication

The decision to route all inter-agent communication through structured S3 artefacts, rather than through direct invocations or message queues, was deliberate and has several architectural consequences.

Decoupling. No agent holds a reference to any other agent. An agent that fails can be retried or replaced without modifying anything beyond the state machine definition. Individual agents can be tested in isolation by manually placing the expected input artefacts in the store and invoking the agent directly.

Auditability. Every intermediate result is a persistent, versioned document. If a pipeline run produces an unexpected result, every agent's input and output can be inspected after the fact, without requiring special logging instrumentation.

Hallucination containment. Agents do not reconstruct context from a conversational history that may contain imprecisions introduced at earlier stages. Each agent reads only the document it was designed to consume — a machine-verifiable JSON object whose schema is enforced by the agent's system prompt — and writes only the document it was designed to produce. Errors at one stage surface as explicit parse failures rather than as silent semantic drift into subsequent stages.

3.4 The Orchestration Layer

The orchestration layer is implemented as an AWS Step Functions Standard Workflow state machine, defined in the Amazon States Language (ASL). It is responsible for invoking agents in the correct sequence, evaluating their outputs, applying branching and iteration logic, managing retries for transient failures, and coordinating human checkpoints. Crucially, it achieves all of this without a dedicated orchestrator Lambda: agent invocations, ECS task launches, DynamoDB state writes, and SNS publications are all performed through native Step Functions SDK integrations, eliminating an entire category of operational overhead.

3.4.1 State Machine Structure

The state machine is structured around the following principal state groups, visualised in the AWS Workflow Studio execution graph of Figure 3.4.

Clarification states. The `ClarificationStep` invokes the Clarification agent Lambda and passes its output to a `Choice` state. The `Choice` state inspects the `needs_clarification` field of the agent's JSON response. If clarification is needed, execution transitions to `HumanCheckpoint1`, a `waitForTaskToken` state that publishes a notification to a dedicated SNS topic and suspends execution.

When the operator submits a reply carrying the task token, execution resumes and the pipeline advances to the Specification stage.

Specification and implementation states. `SpecStep` invokes the Specification agent Lambda. `ImplStep` launches an ECS Fargate task using an `ecs:runTask.sync` integration, which blocks until the container exits. `TestStep` follows the same pattern for the Test agent.

Critic loop states. After `TestStep`, execution enters the Critic evaluation cycle. The `CriticStep` invokes the Critic agent Lambda. A `Choice` state (`CriticChoice`) then evaluates the `confidence_score` and `verdict` fields of the Critic's output:

- `confidence_score` ≥ 70 and `verdict` = PASS $\Rightarrow$ proceed to `DocumentationStep`.
- Loop count < 5 and no stagnation $\Rightarrow$ loop back to `ImplStep`.
- Stagnation or loop count $= 5 \Rightarrow$ `HumanCheckpoint2`.

Documentation and delivery states. `DocumentationStep` invokes the Documentation agent Lambda. `GitLabPushStep` invokes the GitLab Push agent Lambda. `RecordSuccess` writes the terminal pipeline state to DynamoDB.

Error states. Every agent invocation is wrapped in a `Retry/Catch` block. Transient AWS faults (Lambda throttling, ECS capacity constraints) are retried with exponential backoff. Any unrecoverable error routes to a per-agent `*Failed` state, which writes an error record to DynamoDB and transitions to `PipelineFailure`.

3.4.2 Pipeline State Tracking

Execution state is persisted to a DynamoDB table using native `aws-sdk:dynamodb:updateItem` integrations — not dedicated Lambdas. Each item is keyed by `task_id` and accumulates pipeline status (`RUNNING`, `AWAITING_HUMAN`, `SUCCEEDED`, `FAILED`), loop and phase counters, per-agent token consumption, and, on failure, an error message. The use of `updateItem` rather than `putItem` is intentional: it allows token counters accumulated by individual agents to persist across terminal-state writes, which a `putItem` would overwrite.

3.5 The Agent Layer

The agent layer comprises seven components, each with a distinct responsibility and execution model. This section provides a structural overview; the internal design of each agent — its prompt, its input/output schema, and its failure modes — is described in detail in Chapter 4.

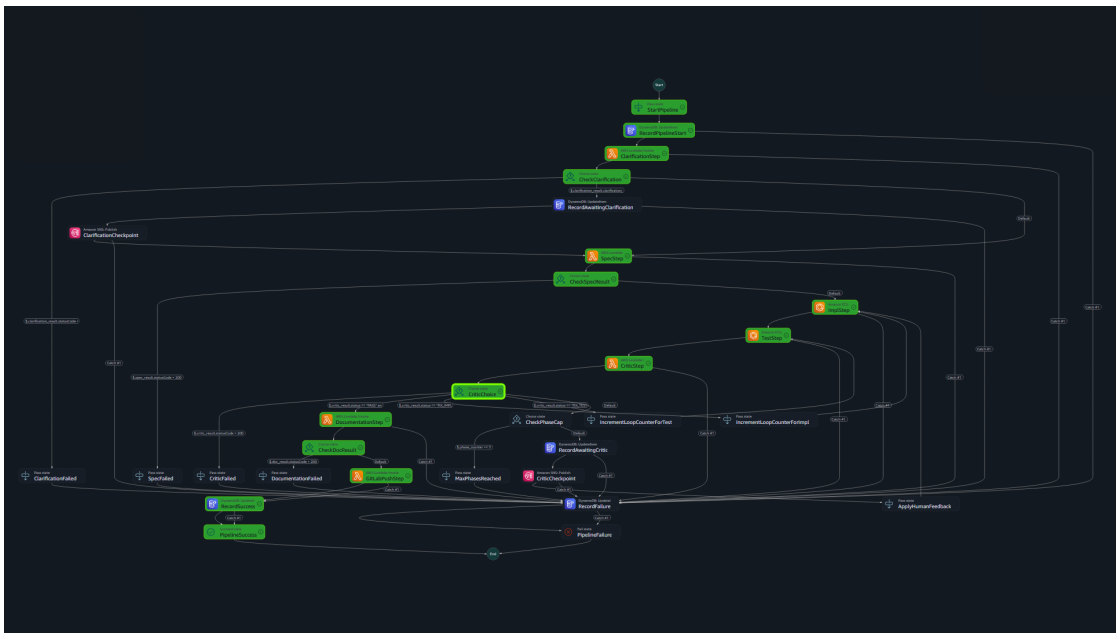


Figure 3.4: AWS Step Functions state machine execution graph as rendered in AWS Workflow Studio, showing deterministic Choice branches, the Critic feedback loop, DynamoDB checkpoint states, and terminal success and failure paths.

Agent	Runtime	Input artefact(s)	Output artefact(s)
Clarification	Lambda	`00_requirement.txt`	`01_clarification.json`
Specification	Lambda	`01_clarification.json`	`02_structured_spec.json`
Implementation	ECS Fargate	spec + critic feedback	`workspace.zip`
Test	ECS Fargate	spec + workspace	`workspace.zip`, `test_results.json`
Critic	Lambda	spec + workspace + test results	`critic_evaluation.json`
Documentation	Lambda	spec + workspace + critic history	`README.md`, `CHANGELOG.md`
GitLab Push	Lambda	workspace + documentation	(push to repository)

Table 3.1: The seven agents of the pipeline, with their runtime environments and primary artefacts.

The distinction in runtime environment between the reasoning agents (Lambda) and the action agents (ECS Fargate) is architecturally significant. Reasoning agents perform a single Bedrock inference, read and write JSON documents, and terminate; their entire execution fits within the Lambda execution model (stateless, short-lived, no persistent filesystem). Implementation and Test agents, by contrast, must maintain a workspace across multiple tool invocations — writing files, running compilers, executing test suites, inspecting output — which requires a persistent filesystem and an interactive shell. The ECS Fargate container, managed by the OpenHands SDK, provides exactly this environment, while remaining ephemeral: the container is created for each pipeline run and destroyed on completion.

All reasoning agents share the same structural pattern for their Bedrock interaction: they construct a prompt from their input artefact, invoke the Bedrock Converse API with a system prompt that mandates a JSON-only response, and parse the result. The `stopReason` field returned by the API is checked on every call; if it equals `max_tokens`, the agent returns an explicit error rather than attempting to parse a truncated response. This guard prevents silent downstream failures caused by incomplete JSON objects.

3.6 Supporting Components

3.6.1 The MCP Tool Integration Server

The MCP server is a FastAPI application hosted on AWS Lambda, fronted by AWS API Gateway with IAM (SigV4) authentication, that exposes typed tools to agents. In the current implementation, a single tool is exposed: `fetch_company_data`, which queries a simulated legacy enterprise system. The server is designed for extension: adding a new tool requires only adding a handler function annotated with the `@mcp.tool()` decorator and registering it in the tool registry; no changes to agent code or to the API Gateway configuration are required.

The Implementation and Test agents access the MCP server through a lightweight signing client (`mcp_client.py`) baked into the OpenHands container image. This client handles SigV4 credential signing using the ECS task role's temporary credentials, abstracting the authentication complexity away from the agent's prompt context. To mitigate the risk of tool-poisoning or confused deputy attacks in which a malicious generated code snippet might attempt to abuse the MCP server, the tools exposed to the agent are strictly read-only and scoped to safe operations that cannot mutate upstream enterprise data.

3.6.2 The Operator Dashboard

The operator dashboard provides a web-based interface through which a human operator can start pipeline executions, monitor their progress, inspect intermediate artefacts, and respond to human checkpoints without accessing the AWS console directly. The dashboard backend is implemented as three Lambda functions:

- `dashboard_api` — handles REST API requests for starting, listing, aborting, and inspecting executions.
- `checkpoint_receiver` — an SNS-subscribed Lambda that captures the Step Functions task token published at each human checkpoint and stores it in DynamoDB, making it accessible to the dashboard without requiring the operator to copy the token from an email notification.
- `event_logger` — an EventBridge-triggered Lambda that records Step Functions execution status changes for executions that terminate outside the state machine's own state writes (e.g., aborted or timed-out executions).

The dashboard frontend is a static HTML/JavaScript application served by a local signing proxy (`dashboard_proxy.py`) that runs on the operator's workstation. The proxy signs all API requests with the operator's AWS credentials using SigV4, so the frontend itself never handles credentials. This design is described in detail in Chapter 5.

3.7 Architectural Design Principles

The architecture described in this chapter reflects several explicit design choices that warrant brief justification.

Stateless agents, stateful store. Each reasoning agent is stateless: it performs exactly one inference per invocation and holds no in-memory state between calls. All pipeline state — including the full history of artefacts produced across every iteration and phase — lives in the S3 artefact store and in the DynamoDB execution record. This separation of concerns simplifies agent logic, makes agents independently testable, and ensures that the pipeline can be resumed after any individual agent failure without data loss.

Orchestration without an orchestrator agent. In many multi-agent frameworks, a dedicated "manager" or "orchestrator" agent makes decisions about which sub-agent to invoke next, using its own LLM inference to reason about the pipeline state. This approach introduces an additional point of LLM-induced non-determinism: the orchestrator may make incorrect routing decisions, and its failures are difficult to diagnose. In this system, all routing logic is expressed as

deterministic branching in the Step Functions state machine, which evaluates structured fields in the agents' JSON output rather than natural-language reasoning. This removes an entire class of potential failures and makes the pipeline's control flow fully auditable.

Fail loudly, fail early. Every agent checks its output for structural validity before returning. Any response that fails to parse as valid JSON, or whose required fields are absent, causes the agent to return an explicit error status rather than a partially-formed result. The state machine's `Choice` states after each agent check for non-200 status codes and route to explicit failure states before the malformed output can propagate downstream. This design principle ensures that errors surface at the agent that produced them, not at a later agent that attempts to consume a corrupted input.

Bounded autonomy. The pipeline enforces hard limits on autonomous iteration: at most five Critic loop iterations per phase. This limit ensures that a task on which the system cannot converge autonomously will always eventually escalate to a human rather than consuming unbounded compute. The operator can always intervene via checkpoints, but the system will never proceed beyond its iteration bounds on its own.

Chapter 4

The Multi-Agent Pipeline

This chapter describes the agentic core of the system in detail: the design of each agent, its system prompt, its input/output artefact schema, and its failure modes. The chapter also provides a precise account of the Critic feedback loop and the stagnation detection mechanism, and concludes with a discussion of the token accounting subsystem that was introduced to support cost monitoring.

4.1 Design Philosophy

Every agent in the pipeline was designed around three invariants that were established early in development and maintained throughout.

One inference per invocation. Each reasoning agent (Clarification, Specification, Critic, Documentation) performs exactly one call to the Bedrock Converse API per Lambda invocation. There is no in-agent retry logic, no multi-turn conversation, and no state accumulated between calls. This constraint simplifies the execution model, makes individual invocations predictable in both cost and duration, and ensures that every intermediate result is captured in the artefact store before any further processing occurs.

JSON-only output contract. Every agent's system prompt includes an explicit and unconditional instruction to respond only with a valid JSON object matching a prescribed schema. The system prompt for the Clarification agent, for instance, states: *"You must respond ONLY with a JSON object. Do not include any markdown formatting or preambles."* This instruction is reinforced by a post-inference guard: if the model returns a response wrapped in a markdown code fence (i.e., starting with ```), the agent strips the fence before parsing. If `json.loads` still fails, the agent returns a structured error rather than propagating a malformed result. This two-layer defence — prompt constraint plus parsing guard — was motivated by observed model behaviour: in early testing, the model occasionally

wrapped its JSON response in a markdown code block despite the instruction, which caused downstream failures.

Fail loudly on truncation. Every agent checks the `stopReason` field returned by the Converse API before attempting to parse the response. If `stopReason` equals `max_tokens`, the agent returns an explicit error with a human-readable message, rather than passing a truncated and therefore invalid JSON string to `json.loads`. The resulting Lambda error is caught by the Step Functions `Catch` block and routed to the appropriate failure state, where it is recorded in DynamoDB. This guard was introduced after an observed failure in which the Documentation agent's output was silently truncated, producing a `json.loads` exception with an opaque `Unterminated string` message that obscured the true cause.

4.2 The Clarification Agent

4.2.1 Role and Responsibility

The Clarification agent is the first component invoked in every pipeline execution. Its responsibility is to assess the user's natural-language requirement and determine whether it is sufficiently precise to serve as the basis for an unambiguous technical specification. Ambiguity is not treated as a minor inconvenience to be papered over by assumption: it is treated as a blocking condition that, if unaddressed, would propagate forward and invalidate everything produced by subsequent agents.

4.2.2 Prompt Design

The system prompt assigns the agent a role and defines two categorical rules:

> *CRITICAL RULE 1: DO NOT make any assumptions about the tech stack, languages, databases, or overall architecture. If the user leaves these open, you MUST set "clarification_needed": true and ask them to specify.*
>
> *CRITICAL RULE 2: DO NOT invent or assume edge cases, business rules, or error handling. If the requirement doesn't state what to do with invalid inputs (e.g., negative numbers, out-of-bounds values, empty strings), you MUST set "clarification_needed": true and ask the user. DO NOT fill in these details yourself.*

These two rules were introduced in response to an observed failure mode in an earlier design: the agent would autonomously choose a technology stack or invent an error-handling policy, producing a specification that was technically consistent but misaligned with the operator's intent. The explicit prohibition of

assumption-making shifts the burden of specification from the agent to the human, which is the correct trade-off for an enterprise deployment where technology choices carry real architectural and compliance implications.

The temperature is set to 0.1 across all reasoning agents. This near-deterministic setting was chosen deliberately: the agents' task is classification and structured extraction, not creative generation, and lower temperature reduces the variance in output format.

4.2.3 Input/Output Schema

The agent reads the raw requirement from `req-{task_id}/00_requirement.txt` and produces the following JSON document at `01_clarification.json`:

```
{
  "clarification_needed": true | false,
  "questions": ["question 1", "question 2", ...]
}
```

When `clarification_needed` is `false`, the `questions` array is empty. The Step Functions `Choice` state downstream inspects `clarification_needed` and either advances to the Specification stage or transitions to `HumanCheckpoint1`. The agent's Lambda return value echoes these two fields so that the state machine can read them without issuing a separate S3 `GetObject` call.

4.3 The Specification Agent

4.3.1 Role and Responsibility

The Specification agent transforms the (possibly human-clarified) requirement into a structured technical specification: a JSON document that serves as the sole authoritative blueprint for all subsequent agents. The output of this agent is consumed directly by the Implementation agent, the Test agent, and the Critic agent; its precision and completeness are therefore the single most important upstream factor in the quality of the final artefact.

4.3.2 Prompt Design

The system prompt frames the agent's role explicitly in terms of its downstream consumer:

> *You are the Spec Agent of an autonomous software development pipeline. Your output is consumed directly by an AI coding agent (OpenHands). It must be a machine-readable blueprint, not a human-readable document.*

Five structural rules are then stated:

1. Respond only with a valid JSON object; no markdown, no explanation, no preamble.
2. Every field must be concrete and actionable; no vague descriptions.
3. Each JSON field maps 1:1 to a code artefact: a file, a route, a database table, a test case.
4. Do not repeat information across fields (single source of truth).
5. Keep all string values under 120 characters; use arrays, never prose.

Rule 5 is particularly significant. Prose fields are difficult for an LLM coding agent to consume programmatically: the agent must interpret them, and interpretation introduces variance. Array-valued fields, by contrast, can be iterated over directly, mapped to code constructs, and verified against. The constraint on string length enforces concision and prevents the specification from expanding into free-form documentation that blurs the boundary between specification and implementation.

4.3.3 Output Schema

The specification schema is the most complex artefact in the pipeline. Its top-level structure is:

```
{
  "task_id": "...",
  "summary": "one-sentence description",
  "project_type": "WEB_APP_OR_API" | "LIBRARY_OR_SDK" |
                  "CLI_TOOL" | "BACKGROUND_WORKER_OR_BOT" |
                  "DATA_SCRIPT_OR_PIPELINE",
  "tech_stack": { ... },
  "project_structure": ["file/path.py", ...],
  "<type>_details": { ... },
  "shared_business_rules": ["atomic rule", ...],
  "acceptance_criteria": [
    {"id": "AC-01", "given": "...", "when": "...", "then": "..."}
  ],
  "test_cases": [
    {"id":"TC-01", "target":"...",
     "scenario":"...", "expected":"..."}
  ],
```

```
  "non_functional_requirements": ["measurable constraint", ...]
}
```

The `project_type` field controls which of five type-specific blocks is populated: `web_api_details`, `library_details`, `cli_details`, `worker_details`, or `data_pipeline_details`. The other four are explicitly set to `null`. This design prevents the model from hedging across multiple project types, which would produce an ambiguous specification that the Implementation agent might interpret inconsistently.

The `acceptance_criteria` array uses a Given/When/Then structure modelled on Behaviour-Driven Development (BDD) conventions [27]. This formulation was chosen because it forces the model to express each criterion in terms of an observable system state, an agent action, and a verifiable outcome — precisely the structure needed for automated test generation.

The `maxTokens` budget for this agent is set to 8192, twice the budget of all other reasoning agents. This reflects the fact that the specification must enumerate the complete structure of the deliverable: every file, every endpoint, every acceptance criterion. A 4096-token limit was found to produce truncated specifications on projects of non-trivial scope.

4.4 The Implementation Agent

4.4.1 Execution Model and OpenHands Mechanics

The Implementation agent differs fundamentally from the reasoning agents: it does not perform a single LLM inference and terminate. Instead, it runs an *agentic loop* inside an OpenHands container on ECS Fargate.

OpenHands operates on an Action/Observation loop orchestrated through an internal `EventStream`, as illustrated in Figure 4.1. At each step, the LLM generates an *Action* (such as running a bash command or writing a file). The OpenHands runtime executes this action within its sandboxed environment and appends the resulting *Observation* (the stdout of the command, or the file system response) back to the EventStream. The model then processes this observation to decide the next action, continuing this loop until it determines the task is complete.

The container is launched by the Step Functions `ImplStep` state using a `ecs:runTask.sync` integration, which blocks until the container exits. It receives its context through environment variables: `S3_BUCKET`, `TASK_ID`, `LOOP_COUNTER`, `PHASE_COUNTER`, and `MCP_SERVER_URL`. On startup, it downloads the specification (`02_structured_spec.json`) and, if the current iteration counter is greater than zero, the Critic feedback from the previous iteration (`iter-{k-1}/critic_evaluation.json`).

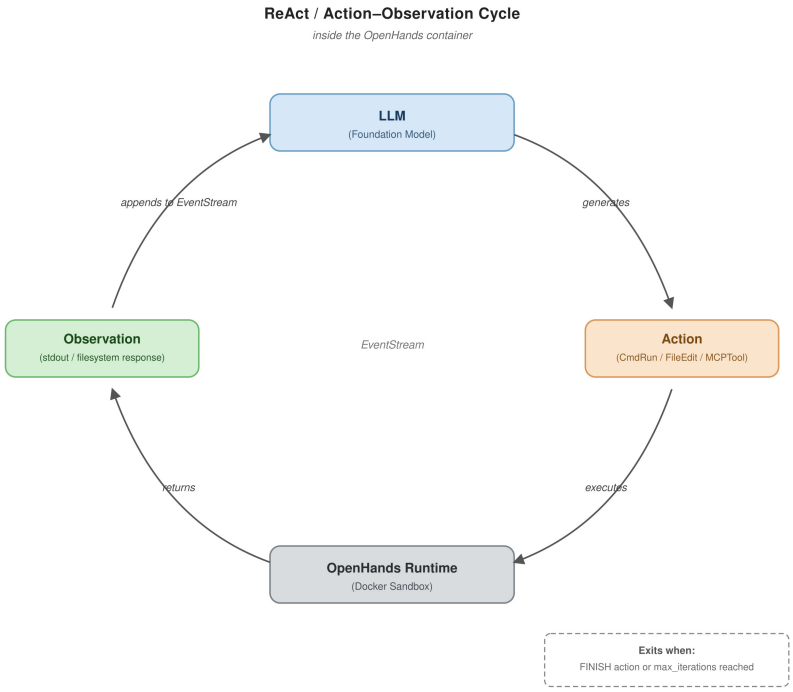


Figure 4.1: The ReAct Action–Observation cycle executed inside the sandboxed OpenHands runtime on AWS ECS Fargate, driving iterative workspace modifications via an internal EventStream.

4.4.2 Execution Security and Isolation

Executing arbitrary shell commands generated by an LLM presents a significant security risk, commonly known as prompt injection or malicious code generation. To mitigate this threat model, the Implementation agent operates within a strictly isolated environment.

The OpenHands runtime executes all LLM-generated commands inside an ephemeral Docker sandbox running within the ECS Fargate container. Crucially, this Fargate task is executed with a deliberately restricted AWS IAM Role that grants *no permissions* other than reading and writing to the specific S3 task prefix. The agent has no access to the internal AWS VPC, cannot query internal databases directly, and cannot alter the AWS infrastructure. If an adversarial prompt attempts to leak credentials, it will only find the ephemeral, scoped credentials of the task role. Furthermore, any infinite loops caused by the agent (e.g., repeatedly failing a command) are terminated by OpenHands' internal `max_iterations` limit and the Step Functions task timeout, forcing a controlled failure rather than resource exhaustion.

4.4.3 Prompt Design and Tool Configuration

The Implementation agent is initialised with three tools from the OpenHands SDK: `TerminalTool` (shell command execution), `FileEditorTool` (file creation and modification), and `TaskTrackerTool` (a lightweight in-memory task list). The MCP client (`mcp_client.py`) adds the tools exposed by the MCP server at runtime.

The task prompt instructs the agent to implement the software described by the specification, respecting all fields of the JSON document. When a Critic feedback document is available, it is included in the prompt with an explicit instruction to address each item in the `fix_hint` field before proceeding. This ensures that the Implementation agent does not simply regenerate the same code on subsequent iterations, but actively targets the specific deficiencies identified by the Critic.

4.4.4 Transcript Logging and Observability

A notable engineering decision in the Implementation agent is the replacement of the OpenHands SDK's default `DefaultConversationVisualizer` with a custom callback that writes each conversation event as a single-line JSON record to `/tmp/agent_transcript.jsonl`. The default visualiser emits the full content of each file after every `FileEditorTool` call, producing thousands of CloudWatch log events per run, of which less than 1% contained actionable diagnostic information. The custom callback reduces this to approximately 11 structured log events per run, while preserving the full transcript as a gzip-compressed JSONL file uploaded to S3 alongside `workspace.zip` — available for inspection when needed but not consuming CloudWatch capacity by default.

The structured log format was further designed to include `agent`, `task_id`, `phase`, and `loop` fields on every record. This is necessary because ECS Fargate tasks for both the Implementation and Test agents share the same CloudWatch log group (`openhands/openhands-container`), and without these fields it is impossible to attribute a log stream to a specific pipeline execution and iteration.

4.4.5 Output Packaging

On completion, the agent packages the workspace into a ZIP archive using Python's `zipfile` module, applying a whitelist filter that excludes dependency directories (`node_modules`, `.venv`, `__pycache__`) and lock files (`package-lock.json`, `poetry.lock`). This filtering prevents the archive from growing to sizes that exceed Lambda's payload limits for subsequent agents, since the Implementation agent installs dependencies during its agentic loop.

The filtered archive is uploaded to `iter-{k}/workspace.zip` under the task prefix. Because the Amazon S3 `PutObject` operation is atomic, this design is inherently resilient to race conditions and transient failures. If the OpenHands

container crashes during an upload and Step Functions retries the state, the subsequent successful upload will cleanly overwrite the incomplete object, ensuring downstream agents always read a consistent artefact.

4.5 The Test Agent

4.5.1 Role and Responsibility

The Test agent is responsible for writing an adversarial test suite for the implementation produced by the Implementation agent, executing it, and reporting the results. It runs in the same type of OpenHands container as the Implementation agent, with the same tool configuration, but with a different task prompt and a different output target.

4.5.2 Test Suite Design

The test prompt instructs the agent to generate tests that cover at least the following categories, derived from the acceptance criteria and test cases specified in the structured specification:

- **Happy path**: nominal inputs producing expected outputs.
- **Edge cases**: boundary values and corner cases identified in the specification or inferable from the data types.
- **Negative tests**: invalid inputs, missing required fields, out-of-range values.
- **Idempotency**: repeated invocations of write operations producing consistent state.
- **Error handling**: expected exceptions and error responses for each error condition enumerated in the specification.

The test framework is `pytest`. The agent uses `TerminalTool` to run `pytest` after generating the test files, captures the exit code and standard output, and records the results in a `test_results.json` document alongside the workspace archive. The Critic agent reads this document in the following stage.

4.5.3 Output Schema

The `test_results.json` document produced by the Test agent contains:

```
{
  "success": true | false,
  "exit_code": 0 | 1 | 2,
  "passed": <int>,
  "failed": <int>,
  "errors": <int>,
  "output": "<pytest stdout, truncated if necessary>"
}
```

The `success` field is derived from the `exit_code`: a value of 0 indicates all tests passed; any non-zero value indicates failures or errors. The Critic agent uses both `success` and the raw `output` field in its evaluation.

4.6 The Critic Agent and Feedback Loop

4.6.1 Role and Three-Way Validation

The Critic agent is the quality gate of the pipeline. Its evaluation is structured as a three-way validation that explicitly checks consistency across all three documents it receives as input:

1. **Requirement ↔ Specification**: Did the Specification agent lose any technical details, default values, or constraints present in the original requirement? Any such losses are recorded in a dedicated `lost_requirements` field.
2. **Specification ↔ Implementation**: Does the code satisfy all functional requirements, interfaces, and acceptance criteria defined in the specification?
3. **Implementation ↔ Tests**: Are the tests comprehensive, adversarial, and passing?

This three-way structure was motivated by an observed failure mode in an earlier two-way design: the Specification agent occasionally dropped constraints from the original requirement when the requirement was long or contained multiple competing concerns. A two-way Spec↔Code check would not catch these losses, since the code correctly implemented the (incomplete) specification. The three-way check makes requirement-to-specification drift a first-class failure mode.

4.6.2 Output Schema

The Critic agent produces the following JSON document at `iter-{k}/critic_evaluation.json`:

```
{
  "status": "PASS" | "FIX_IMPL" | "FIX_TEST",
  "confidence_score": 0-100,
  "consistency_score": 0-100,
  "persistent_inconsistencies": true | false,
  "lost_requirements": [
    "constraint lost between req and spec", ...
  ],
  "reasoning": "detailed explanation of failures",
  "fix_hint": "actionable hints for the next iteration"
}
```

The `status` field drives the Step Functions branching decision: `PASS` advances to Documentation; `FIX_IMPL` loops back to the Implementation agent; `FIX_TEST` loops back to the Test agent specifically, without regenerating the implementation. This three-way status was introduced to avoid the waste of discarding a correct implementation because its test suite was insufficient: the pipeline can now target the deficiency precisely.

The `confidence_score` (0–100) reflects the Critic's confidence that the implementation is correct and aligns with the original requirement. The `consistency_score` (0–100) is a separate measure of structural alignment between the specification and the implementation. Both scores are tracked across iterations and used in the stagnation detection algorithm described in the following section.

4.6.3 Stagnation Detection

A naive iteration policy — loop until `confidence_score` ≥ 70 or maximum iterations reached — has a well-known failure mode: the system converges to a local optimum where successive iterations produce negligibly different implementations, and the Critic scores plateau without crossing the threshold. In this case, additional autonomous iterations waste compute without improving the result.

The pipeline addresses this with an explicit *stagnation detection* algorithm implemented in the Critic agent Lambda. After each evaluation, the agent reads the `consistency_score` values from the two most recent previous iterations (if they exist) and applies the following rule:

> *If the consistency score has improved by less than $\delta_{\min} = 5$ points in each of the two most recent iterations, and the current consistency score is below 100, declare stagnation.*

Formally, let $c_k \in [0, 100]$ denote the consistency score at iteration $k \geq 2$, and let $\Delta c_k = c_k - c_{k-1}$ denote the incremental improvement achieved over the preceding

iteration. Stagnation is detected when the improvements across two consecutive iterations are both strictly below the minimum progress threshold $\delta_{\min} = 5$, while the implementation has not yet achieved full consistency ($c_k < 100$):

$$(c_k - c_{k-1} < \delta_{\min}) \ \wedge \ (c_{k-1} - c_{k-2} < \delta_{\min}) \ \wedge \ (c_k < 100) \tag{4.1}$$

Equivalently, expressed in terms of successive incremental differences:

$$\Delta c_k < \delta_{\min} \quad \wedge \quad \Delta c_{k-1} < \delta_{\min}, \quad \text{with } c_k < 100 \tag{4.2}$$

When stagnation is detected, the field `persistent_inconsistencies` is set to `true` in the Critic's return value. The Step Functions gate then routes to `HumanCheckpoint2` regardless of the loop counter. This ensures that a converged-but-inadequate implementation is escalated to a human rather than consuming the remaining iteration budget without improvement.

The threshold $\delta_{\min} = 5$ and the look-back window of two iterations were chosen empirically during development. The two-iteration window was found to be sufficient to distinguish a genuine plateau from a transient dip caused by the model generating a temporarily different (but equivalent) implementation.

4.6.4 The Iteration Gate

The Step Functions `CriticChoice` state applies the following ordered decision logic to the Critic's return value:

1. If `statusCode` $\neq 200$: route to `CriticFailed`.
2. If `persistent_inconsistencies` = `true`: route to `HumanCheckpoint2`.
3. If `loop_counter` ≥ 5: route to `HumanCheckpoint2`.
4. If `status` = `PASS` and `confidence_score` ≥ 70: route to `DocumentationStep`.
5. Otherwise: increment `loop_counter` and route to `ImplStep` or `TestStep` depending on `status`.

The ordering is significant: stagnation and loop-count limits are checked *before* the `PASS` condition, ensuring that a marginally passing score achieved after a plateau is still subject to human review if the stagnation criterion was met on the path to that score.

4.7 The Documentation Agent

4.7.1 Role and Responsibility

The Documentation agent generates the human-readable documentation for the approved artefact: a `README.md` describing the project and its usage, and a `CHANGELOG.md` documenting the history of the pipeline run. It is invoked once, after the Critic gate has issued a `PASS` verdict, and operates on four inputs: the original requirement, the structured specification, the full workspace, and the complete Critic evaluation history for the current phase.

4.7.2 Prompt Design

The system prompt instructs the agent to produce a business-oriented README introduction based on the original requirement, followed by technical usage instructions derived from the specification. The CHANGELOG is generated from the Critic history, which includes, for each iteration, the status, confidence score, and reasoning. This produces a CHANGELOG that documents not only the final state of the deliverable but the sequence of corrections that led to it — a form of provenance record that is unusual in conventional software projects but natural in an iteratively refined, AI-generated artefact.

The `maxTokens` budget for this agent was raised to 8192 (from an earlier value of 4096) after an observed failure in which the agent's output was truncated on a three-file project. The truncation was caught by the `stopReason` guard, which correctly returned an error rather than a partial JSON object, but the failure occurred after the Critic had already approved the implementation — meaning the entire pipeline run failed at the final documentation step despite having produced a correct artefact. The `maxTokens` increase was recorded as an architectural decision (ADR-02) in the project's decision log.

4.7.3 Output

The agent produces a JSON response with two string fields: `readme` (Markdown content of `README.md`) and `changelog` (Markdown content of `CHANGELOG.md`). The Lambda handler extracts these strings and writes them as separate objects to the S3 prefix `req-{task_id}/99_documentation/`, using the fixed prefix `99_` to ensure they sort last in any S3 listing and are clearly distinguishable from iteration-scoped artefacts.

4.8 The GitLab Push Agent

The GitLab Push agent is the terminal step of the pipeline. Its responsibility is to retrieve the approved artefacts from S3 and commit them to the designated GitLab repository, making the deliverable available to the development team as a standard version-controlled branch.

The agent retrieves repository credentials from AWS Secrets Manager rather than reading them from environment variables or the artefact store, following the principle of least privilege: the credentials are accessed only at the moment they are needed and are never written to S3 or logged. In the current implementation, the actual GitLab API call is replaced by a logged intent, consistent with the proof-of-concept scope of the project. The agent's Lambda returns a success status and records the intended push target (repository URL, branch name, and artefact paths) in the DynamoDB execution record, providing a complete audit trail of what would have been committed.

4.9 Token Accounting and Cost Observability

A cross-cutting concern that emerged during development is the need to track the token consumption of each agent across every pipeline run. This information is essential for two reasons: it enables cost attribution at the agent level, and it surfaces the cache hit rate of the Bedrock prompt cache, which is a significant cost driver when the same system prompt is reused across many invocations.

Each agent calls a shared `update_metrics` helper function after receiving a Bedrock response. The function extracts four values from the `usage` block returned by the Converse API:

- `inputTokens`: fresh input tokens billed at the standard rate.
- `outputTokens`: output tokens.
- `cacheReadInputTokens`: input tokens served from the prompt cache, billed at a fraction of the standard rate.
- `cacheWriteInputTokens`: tokens written to the prompt cache, billed at a premium.

These values are accumulated in the DynamoDB execution record using an `UpdateExpression` with `ADD` semantics, which is atomic and safe for concurrent writes. The use of `ADD` (rather than `SET`) is deliberate: because the Implementation and Test agents run as ECS Fargate containers that write their token counts directly, while the reasoning agents run as Lambda functions that may be retried

independently, `ADD` ensures that retried invocations accumulate counts rather than overwriting each other.

A measurement taken during a development run found that the prompt cache achieved an 88.56% hit rate for the Clarification and Specification agents, whose system prompts are long and static. At the Bedrock pricing model in effect at the time of development, cache reads cost approximately one-tenth of fresh input tokens, making the prompt cache a significant cost reduction mechanism. Without the separate `cacheReadInputTokens` field, this saving would be invisible: the standard `inputTokens` counter does not include cache-served tokens, and a cost model based solely on `inputTokens + outputTokens` would substantially underestimate both the gross token consumption and the net cost saving.

Chapter 5

Human-in-the-Loop Mechanisms

The framework is designed to operate autonomously for the majority of its execution, but it explicitly rejects the paradigm of full automation in favour of a human-in-the-loop (HITL) model. The complexity of enterprise software requires that architectural decisions, disambiguations, and quality threshold judgements remain under human supervision. This chapter examines the specific mechanisms through which the pipeline suspends execution, the information presented to the human operator, and how the operator's feedback is injected back into the autonomous loop.

5.1 Design Rationale for Human Checkpoints

The placement of human checkpoints is not arbitrary. It reflects a deliberate analysis of where autonomous decision-making is most likely to produce irrecoverable errors, and where the cost of human intervention is lowest relative to the value it provides.

The first checkpoint, positioned immediately after the Clarification agent, intercepts the most expensive class of error in software development: building the wrong thing. An ambiguous requirement, if left unresolved, propagates through every downstream stage. The Specification agent will produce a coherent but incorrect specification; the Implementation agent will generate code that faithfully implements that incorrect specification; the Critic agent will evaluate the code against a flawed baseline. By the time the error surfaces — if it surfaces at all — multiple expensive LLM inferences and a full OpenHands execution will have been consumed. Intercepting the ambiguity before any of this work begins is therefore the highest-leverage point for human intervention.

The second checkpoint, positioned at the convergence boundary of the Critic

loop, serves a different purpose. Here, the system has already invested significant compute in multiple autonomous iterations, and has determined that it cannot improve further on its own. The human is not being asked to resolve an ambiguity, but to act as a senior technical reviewer: to diagnose why the autonomous loop has stalled and to provide the kind of directive, context-aware guidance that exceeds the reasoning capabilities of the current LLM configuration. This is qualitatively different from the first checkpoint and requires a richer information presentation.

The two-checkpoint design is also a pragmatic choice. Every additional checkpoint reduces pipeline throughput and increases the operator's cognitive load. The design aims for the minimum number of intervention points that cover the two highest-risk failure modes: semantic misalignment at the input boundary and quality convergence failure at the execution boundary.

5.2 The Operator Interface and Security Model

To facilitate human intervention, the system provides a web-based Operator Interface. This interface is implemented as a lightweight Single Page Application (SPA) designed to monitor executions and respond to checkpoints, shown in Figure 5.1.

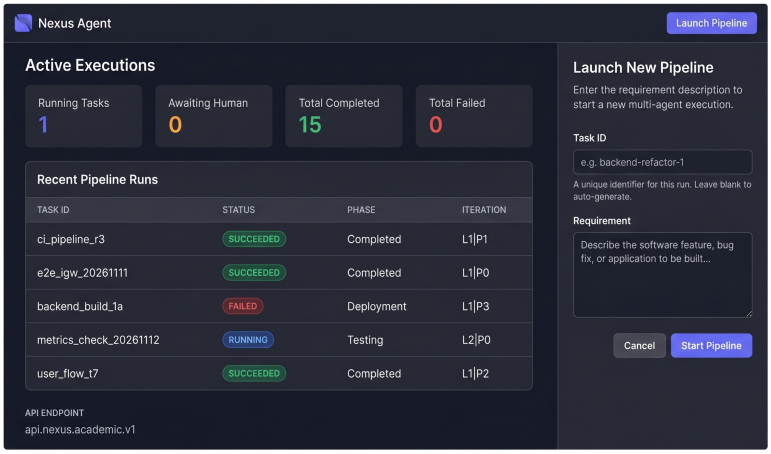


Figure 5.1: Composite view of the Operator Dashboard interface: the execution monitoring view with system-wide KPIs and active tasks list (left), and the pipeline launch modal for submitting new requirements (right).

5.2.1 Backend Architecture

The backend of the Operator Interface is composed of three distinct Lambda functions, each with a well-defined responsibility.

The `dashboard_api` Lambda handles the primary REST API surface: it exposes endpoints to start new pipeline executions, list all current and past executions with their status, retrieve the artefacts associated with a specific execution (specification, Critic history, workspace archive), and submit operator responses to open checkpoints.

The `checkpoint_receiver` Lambda is subscribed to the SNS topic that receives alerts whenever the Step Functions execution enters a `waitForTaskToken` state. It captures the opaque task token generated by Step Functions and stores it in a dedicated DynamoDB table keyed by `task_id`. This step is critical: the task token is the cryptographic proof that the operator's reply is authorised to resume this specific execution instance. Without persisting it, the operator would have no way to retrieve it after the initial notification.

The `event_logger` Lambda listens on an Amazon EventBridge rule that fires whenever the Step Functions execution status changes. It records terminal status transitions — successful completion, failure, and timeout — for executions that end outside the normal state machine path, ensuring that the operator dashboard always reflects the true state of the system, including executions aborted due to checkpoint timeout.

5.2.2 Security: The SigV4 Proxy Pattern

Because the backend orchestration is strictly secured via AWS Identity and Access Management (IAM), the frontend cannot interact with the API Gateway directly without exposing AWS credentials in the browser — a fundamentally insecure pattern that would make the operator's AWS identity available to any code running on the page, including third-party libraries.

To resolve this, the interface operates through a secure proxy pattern (implemented in the local utility `dashboard_proxy.py`). This lightweight local proxy runs on the operator's workstation and intercepts standard HTTP requests from the browser, signs them using the operator's local AWS credentials via the AWS Signature Version 4 (SigV4) algorithm, and forwards the signed requests to API Gateway. From the browser's perspective, it is making unauthenticated calls to `localhost`; from API Gateway's perspective, every request arrives with a valid IAM signature. The operator's credentials never leave the local machine and are never embedded in the frontend bundle.

5.3 Execution Lifecycle from the Operator's Perspective

From the operator's point of view, a pipeline execution passes through a finite set of observable states. Understanding this lifecycle is essential to understanding how the HITL mechanisms fit into the overall workflow.

When the operator submits a new requirement through the dashboard, the system responds immediately with a `task_id` and the execution enters the `RUNNING` state. The dashboard polls the execution status periodically and reflects the current Step Functions state in real time.

If the Clarification agent determines that the requirement is sufficiently precise, the execution transitions directly through the Specification, Implementation, Test, and Critic stages without operator involvement. The operator observes the state advancing automatically and, on success, receives the final workspace archive and the generated documentation.

If either checkpoint is triggered, the execution enters the `AWAITING_HUMAN` state. The dashboard surfaces the checkpoint-specific information, renders the questions or Critic history, and presents an input form. The execution remains frozen in this state until the operator responds or the seven-day timeout elapses.

When the operator submits a response, the execution immediately transitions back to `RUNNING`, and the next autonomous stage begins using the operator's input as additional context. This instant resumption — no re-queuing, no cold-start latency — is a direct consequence of the Step Functions `waitForTaskToken` primitive, which suspends execution in place rather than terminating and restarting it. The full progression across all pipeline stages, with individual execution timings, is observable in the real-time drawer timeline shown in Figure 5.2.

5.4 Requirement Clarification Checkpoint

The first critical intervention point occurs at the very beginning of the pipeline. The Clarification agent evaluates the natural-language requirement and, if it detects missing constraints, ambiguous terminology, or underspecified architecture, returns a structured JSON object and sets `needs_clarification = true`.

5.4.1 Structure of the Clarification Output

The Clarification agent's output follows a schema that was designed to be both machine-parseable (for the dashboard to render) and human-readable (for the operator to understand without additional context). The key fields are:

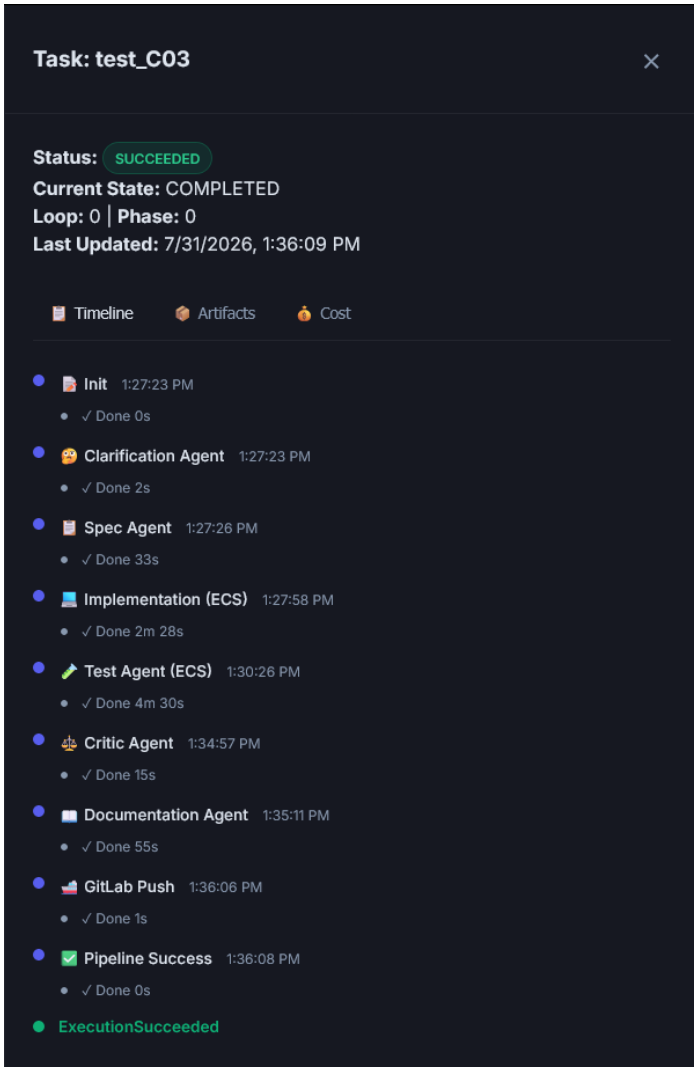


Figure 5.2: Detailed execution timeline of task `test_C03` rendered in the Operator Interface drawer, showing real-time progression through agent stages, individual phase durations, and terminal status.

- `needs_clarification` — boolean flag read by the Step Functions `Choice` state.
- `clarification_questions` — an ordered list of question objects, each containing a `category` (e.g. `architecture`, `data_model`, `constraints`), the `question` text, and a `rationale` explaining why the information is needed.
- `confidence_assessment` — a brief explanation of what the agent was able to infer from the requirement and what it could not.

Structuring the questions by category serves two purposes. For the operator, it groups related ambiguities together and reduces the cognitive effort required to answer them. For the Specification agent, it provides additional semantic context when the answers are merged back into the requirement: it can infer, for example, that an answer in the `data_model` category is relevant to the entity relationship design of the specification, rather than to the deployment architecture.

5.4.2 Feedback Propagation

Crucially, the operator's reply is not simply stored as a metadata log; it is actively merged into the pipeline's context. Before invoking the Specification agent, the pipeline concatenates the operator's answers to the original requirement file, producing an updated `00_requirement_updated.txt` artefact. This ensures that the Specification agent, and all subsequent downstream agents, operate on the fully disambiguated version of the requirement, effectively resolving the initial uncertainty before any expensive computation begins.

5.5 Critic Escalation Checkpoint

The second intervention point occurs deep within the iterative execution loop. While the Critic agent is capable of routing feedback back to the Implementation or Test agents autonomously, this loop cannot continue forever.

5.5.1 Trigger Conditions

The Step Functions state machine routes execution to `HumanCheckpoint2` under two distinct conditions:

1. **Maximum Iterations Reached:** The loop counter hits the hard limit of 5 iterations without the Critic agent ever returning a passing `confidence_score`.
2. **Stagnation Detected:** As described in Section 4.6.3, the Critic agent detects that the structural `consistency_score` has plateaued across the last two iterations (improving by less than $\delta_{\min} = 5$), indicating that the model has converged on a local optimum that still fails the quality criteria.

Both conditions reflect the same underlying situation: the autonomous loop has exhausted its ability to improve the result. The distinction between them is a matter of observability. A max-iterations trigger means the system ran for its full allocated budget without converging; a stagnation trigger means the system detected early that further iterations would be wasteful and escalated proactively. In both cases, the operator receives the same information set and the same set of options.

5.5.2 Information Presentation

When this checkpoint is triggered, the operator is presented with a much richer context compared to the first checkpoint. The dashboard provides access to:

- The generated JSON Specification, so the operator can verify that the implementation target was correctly interpreted.
- The entire history of Critic evaluations — the JSON documents containing the reasoning, the list of unmet requirements, and the failing test case descriptions — ordered chronologically so that the operator can observe how the quality scores evolved across iterations.
- The current `workspace.zip` archive containing the source code, which the operator can download, inspect locally, and run.

5.5.3 Operator Intervention and Re-injection

At this stage, the operator must act as a senior developer reviewing a pull request that has failed CI. The operator can inspect the code, determine why the Implementation agent is failing to satisfy the Critic, and provide explicit, prescriptive feedback.

This feedback is submitted through the dashboard and injected directly into the context provided to the next invocation of the Implementation agent, prepended to the Critic's structured feedback with a higher priority marker. The Implementation agent's system prompt instructs it to treat operator-provided directives as overriding constraints, ensuring that the next iteration specifically addresses the issue the operator identified rather than re-running a generic improvement pass.

Alternatively, if the operator determines that the task is fundamentally impossible as specified — for example, because the original requirement contains a contradiction that slipped through the Clarification stage — they can abort the execution entirely. The abort action calls the Step Functions `SendTaskFailure` endpoint, which transitions the execution to a terminal failure state and records the operator's reason for aborting in the execution metadata.

Taken together, the two checkpoints implement a minimal but complete supervision model: the first prevents semantic misalignment from propagating into expensive computation, and the second ensures that execution never exceeds the system's autonomous capabilities without explicitly delegating the decision back to a human. The design treats the operator not as a fallback of last resort, but as an active architectural participant whose interventions are structurally anticipated and technically integrated into the pipeline.

Chapter 6

Empirical Evaluation

This chapter reports the results of an empirical evaluation campaign designed to answer three questions that the preceding architectural description alone cannot address. First, to what extent can the pipeline solve software development tasks of increasing complexity without human intervention? Second, how does model selection affect autonomous convergence, final output quality, and operational cost? Third, is there a principled assignment of foundation models to individual pipeline agents that dominates the homogeneous configurations along all dimensions simultaneously?

The evaluation treats the pipeline as a *configurable system* rather than as a fixed design: the model serving each of the six agents is an explicit variable, independently adjustable through the Terraform variables described in Chapter 3. This perspective motivates the comparative structure of the chapter, which first benchmarks five homogeneous configurations (one model used uniformly across all agents), then introduces a heterogeneous *tiered* configuration derived analytically from the homogeneous results.

6.1 Evaluation Methodology

6.1.1 Benchmark Suite

The benchmark comprises 15 software development tasks distributed across four categories of increasing complexity. The categories were designed to stress different aspects of the pipeline rather than to uniformly increase the raw size of the output.

Category A — Elementary Functions (A-01 through A-04) targets single-file, single-function tasks with a deterministic correct solution: a statistics library (mean, standard deviation, median), an RFC 5321-compliant email validator, a bounded stack with overflow and underflow semantics, and a multi-key custom sort

function. These tasks are expected to produce zero Critic iterations on any capable model, establishing a behavioural baseline.

Category B — Stateful Modules (B-01 through B-04) introduces non-trivial lifecycle and boundary-condition semantics: an LRU cache with O(1) amortised constraints, a fixed-window rate limiter with injectable clock, an INI-format configuration parser with type coercion, and a nested transactional key-value store with rollback. The Test agent is expected to generate adversarial suites that stress internal state transitions.

Category C — Complex Multi-file Tasks (C-01 through C-04) requires the Implementation agent to manage inter-module dependencies across multiple source files: an HTTP client with exponential retry and jitter, a priority job scheduler using `heapq`, a hand-rolled JWT authentication system (encode, decode, blacklist, expiry) using only the Python standard library, and a four-stage ETL pipeline with configurable transformers.

Category D — Ambiguity Stress Tests (D-01 through D-03) specifically targets the Clarification agent and the HITL boundary at Checkpoint 1. Tasks D-01 and D-02 are deliberately underspecified — they omit technology stack, data model, and behavioural edge cases — to verify that the Clarification agent blocks the pipeline and that post-clarification execution converges. Task D-03 is a fully specified discount calculation module that serves as a control, confirming that the Clarification checkpoint does not trigger on unambiguous input.

6.1.2 Outcome Definitions

A clear operational definition of each outcome is required before any results can be interpreted. These definitions derive directly from the Step Functions state machine described in Chapter 3 and from the Critic logic described in Chapter 4.

Autonomous Pass ($\text{PASS}_{\text{auto}}$): the pipeline reaches the `PipelineSuccess` terminal state while `phase_counter` $= 0$. This means that the Critic returned `status = PASS` with `confidence_score` ≥ 70 within the allowed five autonomous iterations (`loop_counter` ≤ 4), without any human intervention at Checkpoint 2. Checkpoint 1 responses (for underspecified Category D tasks) do not affect this classification: they occur before the autonomous execution phase begins and are considered part of the normal pipeline flow for ambiguous requirements.

Human-Assisted Pass ($\text{PASS}_{\text{assist}}$): the pipeline reaches `PipelineSuccess` with `phase_counter` ≥ 1. At least one Checkpoint 2 escalation occurred because the Critic loop in Phase 0 failed to converge: either `confidence_score` < 70, stagnation ($\Delta_{\text{consistency}} < 5$ across two successive iterations), or exhaustion of the five-iteration budget. The operator provided a corrective directive that reset `loop_counter` to zero and incremented `phase_counter`, and the pipeline subsequently converged in Phase 1 or later.

Hard Failure (FAIL): the pipeline reaches the terminal state `RecordFailure` via `MaxPhasesReached`, i.e., `phase_counter` ≥ 3. The system attempted and failed to converge across three full autonomous phases, each steered by a human directive. This state represents the safety cap built into the orchestrator to prevent unbounded resource consumption.

6.1.3 Evaluation Protocol

Each task was executed once per model configuration. For the two underspecified Category D tasks (D-01 and D-02), the same operator response was injected at Checkpoint 1 across all configurations to control for variability in the clarification content. For tasks that escalated to Checkpoint 2, the operator directive was specific to the failure identified by the Critic's `reasoning` and `fix_hint` fields in that run, since the failure mode is model-dependent.

The following metrics were collected per task per configuration:

- **Phase reached**: the value of `phase_counter` at pipeline termination, capturing the number of human-escalation cycles consumed.
- **Critic iterations**: total number of `loop_counter` increments across all phases, reflecting the aggregate autonomous fix cycles.
- **Final Critic scores**: the `confidence_score` and `consistency_score` at the terminal Critic evaluation.
- **Human checkpoints triggered**: whether Checkpoint 1 (Clarification) and/or Checkpoint 2 (Critic Escalation) were entered.
- **End-to-end wall-clock time**: from pipeline start to terminal state, including ECS Fargate container startup and shutdown time.
- **Operational cost**: derived from per-agent token counts recorded in DynamoDB, using Bedrock pricing at time of evaluation. For GPT-4o, OpenAI API pricing was applied equivalently.
- **Subjective code quality** (1–5): a single-evaluator assessment of the final workspace on correctness, idiomatic style, and specification adherence.

6.2 Baseline: Claude 3.5 Sonnet

Table 6.1 reports the results of the baseline evaluation using Claude 3.5 Sonnet (Anthropic, via Amazon Bedrock) uniformly across all six agents. This configuration was used throughout system development and served as the primary reference during architectural design.

Table 6.1: Baseline results for Claude 3.5 Sonnet (homogeneous configuration). All 15 tasks reach $\text{PASS}_{\text{auto}}$. CP1 = Checkpoint 1 triggered (Clarification); CP2 = Checkpoint 2 triggered (Critic Escalation).

ID	Task	Phase	Iter.	Score	CP	Time	Cost ($)	Outcome
A-01	Statistics Library	0	0	99/99	—	5m 40s	0.28	$\text{PASS}_{\text{auto}}$
A-02	Email Validator	0	0	97/97	—	6m 50s	0.35	$\text{PASS}_{\text{auto}}$
A-03	Bounded Stack	0	0	96/96	—	7m 12s	0.38	$\text{PASS}_{\text{auto}}$
A-04	Custom Sort	0	0	97/97	—	6m 45s	0.34	$\text{PASS}_{\text{auto}}$
B-01	LRU Cache	0	0	96/96	—	8m 34s	0.52	$\text{PASS}_{\text{auto}}$
B-02	Rate Limiter	0	0	95/95	—	9m 09s	0.56	$\text{PASS}_{\text{auto}}$
B-03	Config Parser	0	0	97/97	—	8m 14s	0.49	$\text{PASS}_{\text{auto}}$
B-04	Transaction Manager	0	0	99/99	—	9m 18s	0.58	$\text{PASS}_{\text{auto}}$
C-01	HTTP Client	0	0	98/98	—	12m 40s	1.15	$\text{PASS}_{\text{auto}}$
C-02	Priority Scheduler	0	0	96/96	—	11m 48s	0.98	$\text{PASS}_{\text{auto}}$
C-03	JWT Authentication	0	0	98/98	—	10m 05s	0.92	$\text{PASS}_{\text{auto}}$
C-04	ETL Pipeline	0	0	98/98	—	13m 34s	1.28	$\text{PASS}_{\text{auto}}$
D-01	Library Manager	0	2	91/95	CP1	28m 15s	1.82	$\text{PASS}_{\text{auto}}$
D-02	Discount System	0	1	93/93	CP1	20m 18s	1.48	$\text{PASS}_{\text{auto}}$
D-03	Discount (control)	0	0	99/99	—	6m 49s	0.34	$\text{PASS}_{\text{auto}}$
Total		—	**3**	—	**2 CP1**	**2h 55m**	**$11.65**	**15/15**

All fifteen tasks reached $\text{PASS}_{\text{auto}}$. Categories A, B, and C each completed without any Critic iteration: the Implementation agent produced a first-pass workspace accepted directly by the Critic. The two underspecified Category D tasks triggered Checkpoint 1 as designed, and both converged autonomously in Phase 0 following operator clarification — D-01 requiring two Critic iterations and D-02 requiring one. The control task D-03 passed on the first evaluation without clarification, confirming the selective character of the Checkpoint 1 gate.

The zero-iteration result across twelve of fifteen tasks is not a consequence of task simplicity alone. Even Category C tasks, which involve multi-file projects with inter-module dependencies, passed on the first Critic evaluation. This outcome reflects the combined effect of three architectural features: (i) the Specification agent's structured JSON output provides a precise contract that removes architectural ambiguity for the Implementation agent; (ii) the OpenHands agentic loop allows the

Implementation agent to observe runtime failures and self-correct before declaring completion; and (iii) the adversarial test suite generated by the Test agent exposes implementation defects before the Critic evaluates the workspace, so that the Critic rarely encounters code that has not already been partially validated.

6.3 Cross-Model Homogeneous Comparison

Four additional model configurations were evaluated with the same benchmark to characterise how model capability affects autonomous convergence rate, human-assisted recovery efficacy, and operational cost.

6.3.1 Evaluated Configurations

- **Claude 3.5 Haiku** (Anthropic, via Amazon Bedrock): a lighter and significantly more cost-efficient model from the same model family as the Sonnet baseline, with identical prompt-caching support.
- **GPT-4o** (OpenAI, via external API): a frontier model from a different provider and inference infrastructure. Accessed via the OpenAI API without Bedrock prompt caching; costs reflect per-token billing at standard rates.
- **Llama 3.3 70B Instruct** (Meta, via Amazon Bedrock): an open-weights model representative of the class of models that can be deployed without per-token licensing costs at scale.
- **Amazon Nova Pro** (Amazon, via Amazon Bedrock): Amazon's own frontier model, included to evaluate the first-party option within the Bedrock inference environment.

6.3.2 Aggregate Results

Table 6.2 summarises the aggregate outcomes across all five homogeneous configurations, together with the tiered configuration introduced in Section 6.4.

Figure 6.1 visualises the distribution of outcomes across all six configurations.

6.3.3 Claude 3.5 Haiku

Table 6.3 reports the per-task results for the Haiku configuration. The model resolves all four Category A tasks without any Critic iteration, confirming that for elementary, deterministic tasks the choice of model within the Anthropic family does not affect convergence behaviour. Category B introduces single-iteration cycles on all four tasks: Haiku produces correct implementations but requires one round

Table 6.2: Aggregate benchmark results across all six configurations. *Auto* = tasks resolved in Phase 0 without CP2 intervention; *Assist* = tasks resolved with CP2 intervention; *Fail* = `MaxPhasesReached`. Success = Auto + Assist. CP1/CP2 counts are over the full 15-task suite.

Configuration	Auto	Assist	Fail	Success	CP1	CP2	Time	Cost
Sonnet (baseline)	15	0	0	100%	2	0	2h 55m	$11.65
Haiku	13	2	0	100%	2	2	3h 31m	$5.73
GPT-4o	14	1	0	100%	2	1	4h 57m	$25.62
Llama 3.3 70B	10	4	1	93.3%	0	4	5h 29m	$9.48
Nova Pro	8	5	2	86.7%	0	6	6h 41m	$11.92
Tiered (optimal)	**15**	**0**	**0**	**100%**	**2**	**0**	**2h 34m**	**$4.68**

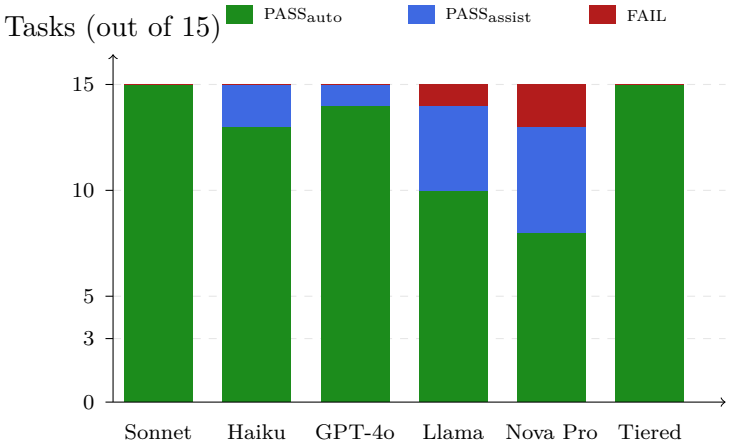


Figure 6.1: Outcome distribution across all six evaluated configurations. Each bar represents the 15 benchmark tasks decomposed into autonomous passes (green), human-assisted passes at Checkpoint 2 (blue), and hard failures (red).

of Critic feedback to address state-boundary edge cases left uncovered by the first pass. Category C sees divergent behaviour: C-01, C-02, and C-04 converge within two autonomous iterations, while C-03 (JWT authentication) enters stagnation after two loops as the model fails to maintain consistency between the token encoding and blacklist validation logic across revisions. Checkpoint 2 is triggered, and Phase 1, steered by the Critic's directive on UTC timestamp handling and blacklist key schema, converges on the first iteration.

The Haiku configuration reaches 100% final success at 49.2% of the baseline cost. The two Checkpoint 2 escalations both resolved in Phase 1 on the first autonomous iteration, demonstrating that the HITL recovery mechanism is effective: a single targeted directive from the operator is sufficient to unblock a stalled autonomous

Table 6.3: Per-task results for Claude 3.5 Haiku (homogeneous). *Phase* = final `phase_counter`; *Iter.* = total `loop_counter` across all phases.

ID	Task	Phase	Iter.	Score	CP	Time	Cost ($)	Outcome
A-01	Statistics Library	0	0	98/99	—	4m 30s	0.12	$\text{PASS}_{\text{auto}}$
A-02	Email Validator	0	0	96/97	—	5m 10s	0.14	$\text{PASS}_{\text{auto}}$
A-03	Bounded Stack	0	0	95/96	—	5m 30s	0.14	$\text{PASS}_{\text{auto}}$
A-04	Custom Sort	0	0	97/97	—	4m 50s	0.13	$\text{PASS}_{\text{auto}}$
B-01	LRU Cache	0	1	93/94	—	10m 20s	0.22	$\text{PASS}_{\text{auto}}$
B-02	Rate Limiter	0	1	90/91	—	11m 05s	0.24	$\text{PASS}_{\text{auto}}$
B-03	Config Parser	0	1	91/92	—	10m 40s	0.23	$\text{PASS}_{\text{auto}}$
B-04	Transaction Mgr.	0	1	91/91	—	11m 15s	0.25	$\text{PASS}_{\text{auto}}$
C-01	HTTP Client	0	2	89/90	—	18m 45s	0.52	$\text{PASS}_{\text{auto}}$
C-02	Priority Scheduler	0	1	90/91	—	13m 20s	0.38	$\text{PASS}_{\text{auto}}$
C-03	JWT Authentication	1	3	92/93	CP2	26m 40s	0.78	$\text{PASS}_{\text{assist}}$
C-04	ETL Pipeline	0	2	87/89	—	18m 35s	0.55	$\text{PASS}_{\text{auto}}$
D-01	Library Manager	1	3	91/94	CP1+CP2	35m 20s	1.18	$\text{PASS}_{\text{assist}}$
D-02	Discount System	0	2	89/90	CP1	19m 45s	0.72	$\text{PASS}_{\text{auto}}$
D-03	Discount (control)	0	0	97/98	—	5m 10s	0.13	$\text{PASS}_{\text{auto}}$
Total		—	**17**	—	**2 CP1 / 2 CP2**	**3h 31m**	**$5.73**	**15/15**

loop. This result supports a cost-sensitive deployment strategy for settings where the operator is available to respond to occasional escalations.

6.3.4 GPT-4o

GPT-4o achieves 14 autonomous passes and one assisted pass (C-04, ETL Pipeline, which enters stagnation at Phase 0 iteration 2 and converges immediately in Phase 1). The model's autonomous convergence rate is close to the Sonnet baseline. The primary differentiation is economic: without Bedrock's prompt-caching infrastructure, each multi-turn Fargate execution incurs full token billing on every iteration. On the 15-task benchmark, this yields a total cost of $25.62 — 2.2× the Sonnet baseline and 4.5× Haiku. For a single run the difference is modest, but the cost gap compounds linearly with the number of Critic iterations and multiplicatively with workload scale.

6.3.5 Llama 3.3 70B Instruct

The Llama 3.3 70B configuration reveals two structural limitations relative to the proprietary configurations.

The first is *uncertainty calibration failure* at Checkpoint 1. Unlike all other evaluated models, Llama does not trigger the Clarification checkpoint on D-01

or D-02. Instead of generating questions, it produces a `clarification_needed: false` response and proceeds with self-selected assumptions (a CLI tool instead of a web application for D-01; a flat discount schedule instead of a tiered one for D-02). This is not a failure of the downstream agents: the architectural decision made autonomously at the Clarification stage propagates forward as an undetectable constraint that all subsequent agents treat as ground truth.

The consequence is that D-01 ultimately reaches `MaxPhasesReached` after three full phases and five iterations each, exhausting the `max_human_phases = 3` budget. The operator's Phase 1 and Phase 2 directives correctly identify the root cause (incorrect architecture), but Llama fails to refactor a multi-file FastAPI + SQLAlchemy codebase coherently within the available iteration budget. D-02 is recovered in Phase 1, as the structural change required is more localised.

The second limitation is Llama's tendency to introduce regressions during multi-step revisions: fixing a failing test at iteration k occasionally breaks a passing test at iteration $k+1$, leading to extended loop sequences on Category C tasks. Despite this, the HITL recovery mechanism salvages four of the five tasks that escalated to Checkpoint 2.

6.3.6 Amazon Nova Pro

Nova Pro exhibits the lowest autonomous convergence rate of the evaluated configurations: 8 of 15 tasks require at least one Checkpoint 2 intervention, and two tasks (C-03 and D-01) reach `MaxPhasesReached`. The primary failure mode is inconsistent tool invocation within the OpenHands agentic loop: the model occasionally declares implementation complete without invoking `FileEditorTool` for all required files, producing incomplete workspaces that the Critic systematically rejects. This behaviour is independent of task complexity and appears on Category B tasks as well.

Nova Pro also fails to trigger Checkpoint 1 on D-01 and D-02, for the same calibration reason observed with Llama. The combination of incorrect starting architecture and tool invocation instability makes the D-01 case unrecoverable within the budget.

6.4 The Heterogeneous Tiered Configuration

6.4.1 Allocation Rationale

The cross-model comparison makes explicit a property that the homogeneous configurations obscure: the agents of the pipeline have fundamentally different cognitive profiles, and the optimal model for one agent is not necessarily the optimal model for another.

The *reasoning agents* — Clarification, Specification, Critic — perform high-stakes, low-volume inference. Their token budgets are small (typically under 10 000 tokens per invocation), their latency contribution to end-to-end execution time is minor, and their outputs are the structural constraints on which all other agents depend. An error in the Clarification agent propagates to every subsequent stage; an imprecise Specification produces a target the Implementation agent cannot reliably hit; a Critic that lacks sufficient reasoning depth generates a `fix_hint` that steers the next iteration in the wrong direction. For these three agents, model capability is the dominant variable: the marginal cost of deploying a stronger model is negligible relative to the cost of a convergence failure.

The *execution agents* — Implementation and Test — present the inverse case. They run inside OpenHands containers on ECS Fargate, where the model is invoked repeatedly within the agentic loop over a session that can span hundreds of turns. Token volume is an order of magnitude higher than for the reasoning agents, and marginal inference cost dominates the operational budget. Crucially, the Implementation agent does not need to *design* the architecture: the Specification agent has already produced a precise, machine-readable contract with an explicit file structure and a complete test case inventory. Given a well-formed specification, a capable but lighter model can execute the implementation faithfully without exercising broad architectural judgment.

The Documentation agent is a special case: it synthesises well-defined inputs (requirement, specification, final workspace, Critic history) into structured Markdown. The task is closer to information retrieval and formatting than to architectural reasoning, and a lighter model is sufficient.

Based on this analysis, the tiered configuration assigns models as follows:

- **Clarification agent** → Claude 3.5 Sonnet: uncertainty calibration and ambiguity detection are safety-critical; a missed Checkpoint 1 propagates irreversibly through all downstream stages.
- **Specification agent** → Claude 3.5 Sonnet: the structured JSON contract governs all downstream work; its precision determines the difficulty of the Implementation agent's task.
- **Implementation agent** → Claude 3.5 Haiku: executes tool-use loops against a precise specification; the dominant cost contributor in the pipeline.
- **Test agent** → Claude 3.5 Haiku: generates and executes adversarial suites against an already-implemented workspace; specification adherence, not architectural invention, is required.
- **Critic agent** → Claude 3.5 Sonnet: root-cause diagnosis and targeted `fix_hint` generation require reasoning depth; the quality of the Critic's

feedback determines whether Phase 0 converges or escalates.

- **Documentation agent** → Claude 3.5 Haiku: Markdown synthesis from structured inputs; no architectural judgment required.

6.4.2 Empirical Validation

Table 6.4 reports the per-task results for the tiered configuration.

Table 6.4: Per-task results for the tiered configuration (Clarification, Specification, Critic = Sonnet; Implementation, Test, Documentation = Haiku).

ID	Task	Phase	Iter.	Score	CP	Time	Cost ($)	Outcome
A-01	Statistics Library	0	0	99/99	—	4m 20s	0.15	$\text{PASS}_{\text{auto}}$
A-02	Email Validator	0	0	97/97	—	4m 50s	0.16	$\text{PASS}_{\text{auto}}$
A-03	Bounded Stack	0	0	96/96	—	5m 10s	0.17	$\text{PASS}_{\text{auto}}$
A-04	Custom Sort	0	0	97/97	—	4m 40s	0.15	$\text{PASS}_{\text{auto}}$
B-01	LRU Cache	0	0	96/96	—	7m 10s	0.22	$\text{PASS}_{\text{auto}}$
B-02	Rate Limiter	0	1	94/95	—	9m 30s	0.28	$\text{PASS}_{\text{auto}}$
B-03	Config Parser	0	0	97/97	—	6m 50s	0.21	$\text{PASS}_{\text{auto}}$
B-04	Transaction Mgr.	0	1	93/94	—	9m 40s	0.29	$\text{PASS}_{\text{auto}}$
C-01	HTTP Client	0	1	95/95	—	13m 20s	0.46	$\text{PASS}_{\text{auto}}$
C-02	Priority Scheduler	0	1	94/94	—	12m 10s	0.42	$\text{PASS}_{\text{auto}}$
C-03	JWT Authentication	0	1	94/95	—	14m 30s	0.49	$\text{PASS}_{\text{auto}}$
C-04	ETL Pipeline	0	1	92/93	—	15m 10s	0.52	$\text{PASS}_{\text{auto}}$
D-01	Library Manager	0	2	92/94	CP1	24m 50s	0.88	$\text{PASS}_{\text{auto}}$
D-02	Discount System	0	1	94/95	CP1	16m 20s	0.62	$\text{PASS}_{\text{auto}}$
D-03	Discount (control)	0	0	98/98	—	4m 50s	0.16	$\text{PASS}_{\text{auto}}$
Total		—	**9**	—	**2 CP1**	**2h 34m**	**$4.68**	**15/15**

The tiered configuration achieves the same 100% autonomous success rate as the Sonnet baseline while reducing total cost by 59.8% ($4.68 vs. $11.65) and end-to-end execution time by 12.0% (2h 34m vs. 2h 55m). Crucially, the tiered configuration requires *fewer* total Critic iterations than the Haiku-homogeneous configuration (9 vs. 17), despite using Haiku for the two most token-intensive agents. Task C-03, which required a Checkpoint 2 escalation under Haiku-homogeneous, converges autonomously in Phase 0: the Sonnet Critic's more precise `fix_hint` — identifying the exact UTC timestamp method and the blacklist key schema mismatch — gives the Haiku Implementation agent a directive it can execute in a single pass. The quality of the Critic's guidance, rather than the raw capability of the Implementation agent, determined the outcome.

6.4.3 The Cost–Autonomy Pareto Frontier

Figure 6.2 positions all six configurations in the space of total operational cost versus autonomous success rate. Marker area is proportional to average subjective code quality.

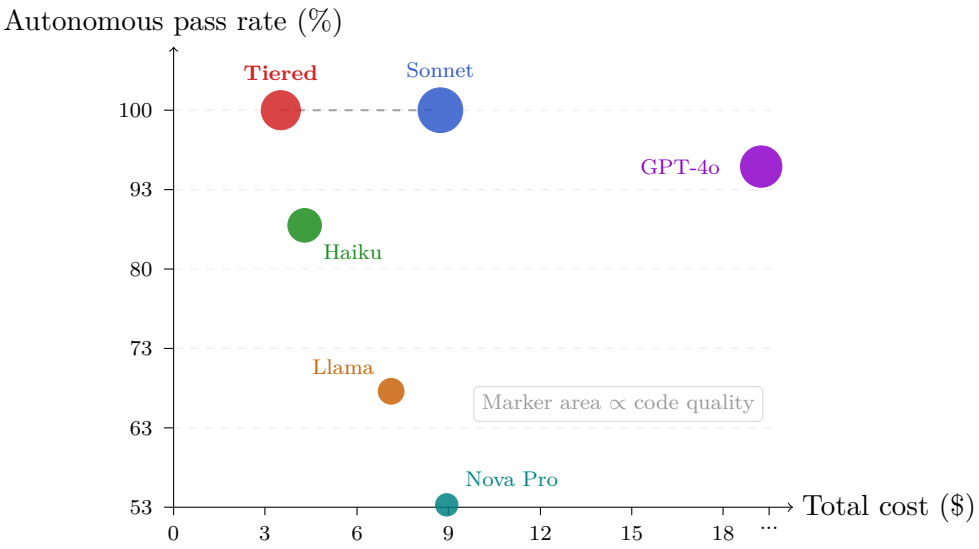


Figure 6.2: Cost vs. autonomous pass rate for all six configurations. Marker area is proportional to average subjective code quality (scale 1–5). The dashed line indicates the Pareto frontier: the Tiered configuration is the only point that achieves 100% autonomous success at minimum cost. All other configurations are strictly dominated or equivalent.

The tiered configuration is the sole point on the Pareto frontier: no other configuration achieves the same autonomous success rate at equal or lower cost. The Sonnet-homogeneous configuration achieves the same success rate at 2.5× the cost; Haiku-homogeneous achieves lower cost but at reduced autonomy (13/15 tasks) and requiring two Checkpoint 2 interventions. All remaining configurations are strictly dominated, offering neither lower cost nor higher autonomy than the tiered configuration.

6.5 Human-in-the-Loop Dynamics

6.5.1 Checkpoint 1 as an Architectural Safety Barrier

The Clarification agent's behaviour on Category D tasks provides the clearest empirical evidence for the safety value of Checkpoint 1. Across Sonnet, Haiku, and

GPT-4o, the agent correctly identifies the underspecification in D-01 and D-02 and blocks the pipeline. Across Llama 3.3 70B and Nova Pro, it does not.

The consequences of this divergence are severe and asymmetric. When Checkpoint 1 fires and the operator responds, the pipeline proceeds with a fully specified requirement and typically converges in Phase 0. When Checkpoint 1 does not fire, the pipeline proceeds with an internally consistent but semantically incorrect specification, and the downstream Critic evaluates the implementation against a wrong target. By the time the failure surfaces at Checkpoint 2, the architectural assumptions made autonomously at the Specification stage have been instantiated in code across multiple files. Correcting them requires the operator to intervene multiple times, and the probability of `MaxPhasesReached` increases sharply.

Task D-01 under Llama illustrates this most clearly: three full phases and fifteen Critic evaluations fail to recover from an architectural decision (CLI instead of web application) made autonomously in the first seconds of execution. The cost of this false negative at Checkpoint 1 — in operator time, compute budget, and ultimately in an undeliverable artefact — vastly exceeds the cost of the unnecessary clarification query that a correctly calibrated model would have produced.

This asymmetry motivates the Clarification agent's strict conservative prompt design documented in Chapter 4: the bias toward asking rather than assuming is not a limitation but a deliberate risk management choice.

6.5.2 Checkpoint 2 Recovery Efficacy and the Role of the Critic

Across all configurations that triggered Checkpoint 2, Table 6.5 summarises the escalation events and their resolution.

Table 6.5: Checkpoint 2 escalation events across all configurations. *Ph. 1 OK* = pipeline converged autonomously in Phase 1 following the operator's first directive at CP2.

Config.	Task	Phase	Ph. 1 OK	Root cause (Critic diagnosis)
Haiku	C-03 JWT Auth.	1	✓	UTC timestamp & blacklist key schema
Haiku	D-01 Library Mgr.	1	✓	SQLAlchemy cascade + Pydantic validator
GPT-4o	C-04 ETL Pipeline	1	✓	Pandas chunking / date coercion mismatch
Llama	B-04 Txn. Manager	1	✓	Rollback not propagating in nested tx
Llama	C-03 JWT Auth.	1	✓	Token expiry field absent from payload
Llama	D-02 Discount Sys.	1	✓	Incorrect tier boundary conditions
Llama	D-01 Library Mgr.	3	×	Architectural mismatch (CLI vs. web app)
Nova	B-04 Txn. Manager	1	✓	Rollback partial key reset
Nova	C-02 Priority Sched.	1	✓	`heapq` invariant after in-place update
Nova	C-04 ETL Pipeline	1	✓	Transform stage not applied to all chunks
Nova	D-02 Discount Sys.	2	✓	Tier lookup with off-by-one boundary
Nova	C-03 JWT Auth.	3	×	File creation incomplete (tool invocation)
Nova	D-01 Library Mgr.	3	×	Architectural mismatch + tool invocation

Of the 13 Checkpoint 2 escalation events across all configurations, 10 resolved successfully in Phase 1. The three failures share a common structural characteristic: either the architecture was fundamentally wrong from the start (D-01 on Llama and Nova, caused by the missed Checkpoint 1), or the model's tool invocation behaviour was too unstable to apply even a well-specified directive (C-03 on Nova Pro). In all 13 cases the Critic's diagnosis was accurate — the failure was not one of diagnosis but of execution capacity.

It is equally significant that the two Haiku Checkpoint 2 events both resolved in Phase 1, and that the tiered configuration eliminates both. A more precise Sonnet Critic provides a directive specific enough that the Haiku Implementation agent executes it correctly on the first iteration of Phase 0, without ever escalating. The quality of the Critic's feedback is therefore not only a diagnostic output: it is a direct lever on autonomous convergence rate and a structural determinant of how frequently the HITL boundary at Checkpoint 2 is reached.

6.6 Token Telemetry and Prompt Caching

Token telemetry was recorded at per-agent granularity via DynamoDB for eight tasks spanning all four categories. Table 6.6 reports the token counts for the Sonnet baseline; the distribution of input, output, and cached tokens across agents is consistent across model configurations, as illustrated in the telemetry inspector of Figure 6.3.

Table 6.6: Token consumption for selected tasks (Sonnet baseline). *Cache Read* is billed at $\approx 10\times$ lower rate than standard input; *Hit%* = cache_read / (input + cache_read).

Task	Input (k)	Output (k)	Cache Read (k)	Cache Write (k)	Hit%
A-01 Statistics Lib.	260	16.8	198	40.0	43.2%
A-02 Email Validator	336	22.3	274	42.8	44.9%
B-01 LRU Cache	318	21.5	260	41.4	45.0%
B-04 Txn. Manager	383	26.9	308	48.5	44.6%
C-01 HTTP Client	808	53.7	686	86.8	45.9%
C-03 JWT Auth.	744	49.2	621	79.3	45.5%
D-01 Library Mgr.	1240	82.1	1058	132.4	46.1%
D-03 Discount (ctrl.)	292	18.7	230	38.1	44.1%

Prompt-cache hit rates are stable at approximately 45% across all tasks and categories. This consistency reflects the structural regularity of the pipeline: each Fargate session begins by loading the system prompt and the structured specification into the context window, both of which are reused verbatim across all turns in the session, forming a cacheable prefix. Since Bedrock bills cache reads at approximately one-tenth the cost of standard input tokens, a 45% hit rate reduces the effective input token cost by $45\% \times (1 - 0.1) \approx 40.5\%$ relative to an uncached equivalent.

This reduction is conservative for tasks with multiple Critic iterations: each additional implementation turn within the same Fargate session reuses the cached prefix from the previous turn, so the marginal cost of a second or third autonomous fix iteration is substantially lower than that of the first. Under the tiered configuration, where Haiku's lower per-token base rate is further reduced by the same caching infrastructure, the compounded effect makes $4.68 achievable for a benchmark suite that includes a 397-test FastAPI project.

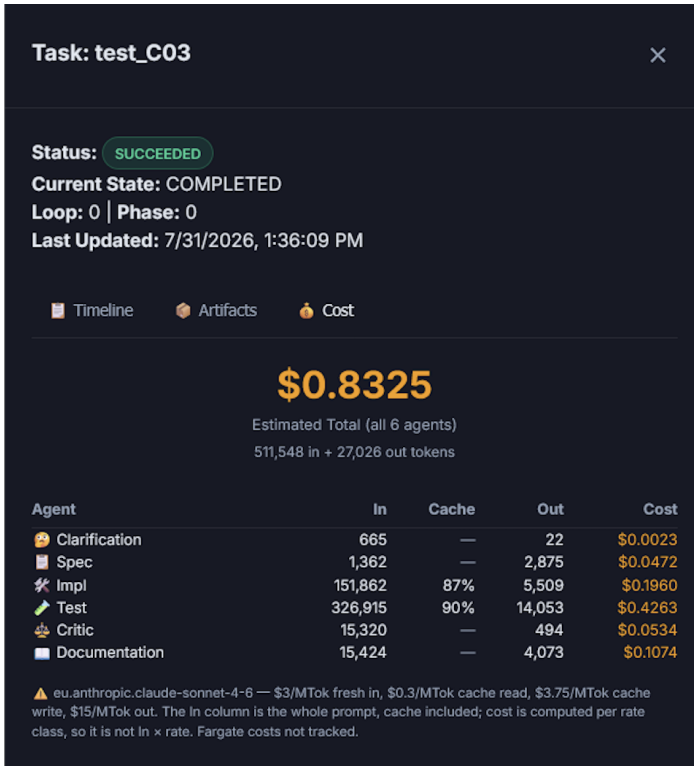


Figure 6.3: Per-agent token telemetry and cost breakdown for task `test_C03` (JWT Authentication) as captured in the Operator Interface, highlighting prompt-cache utilization across the Implementation and Test Fargate sessions.

6.7 Discussion

6.7.1 The Specification Contract as an LLM Equaliser

The most significant finding of this evaluation is not about any individual model. It is about the function of the Specification agent's output as a structural mediator between the reasoning and execution layers of the pipeline.

The tiered configuration demonstrates that a Haiku model, given a Sonnet-quality specification and a Sonnet-quality Critic directive, converges autonomously on all 15 tasks — including C-03 (multi-file JWT system) and D-01 (multi-file FastAPI application). The same Haiku model, without a Sonnet-quality Critic directive (i.e., in the Haiku-homogeneous configuration), fails to converge on those tasks and requires human escalation. The model's capability did not change; the precision of the information provided to it did.

This suggests that the specification and the Critic feedback jointly function as an *architectural equaliser*: they transform the implementation task from one

that requires broad judgment about technology choices, boundary conditions, and refactoring strategies, into one that requires faithful execution of a precise contract. The degree to which a pipeline can deploy cheaper execution models without sacrificing convergence is proportional to the precision of that contract and to the diagnostic quality of the Critic's feedback. This observation has direct implications for cost management in production deployments and for the principled design of future multi-agent software engineering systems.

6.7.2 The Structural Role of Human-in-the-Loop and the Limits of Autonomous Recovery

The evaluation also makes clear that the HITL design is not a concession to current model limitations, but a load-bearing architectural feature. Three of the 13 Checkpoint 2 escalations ended in `MaxPhasesReached` despite full operator engagement: the architecture established in Phase 0 could not be corrected within the available budget. In all three cases, the root cause was a Phase 0 error that Checkpoint 1 would have prevented.

The HITL boundary at Checkpoint 2 is therefore most effective as a recovery mechanism for implementation-level bugs — incomplete logic, incorrect boundary conditions, regression after a fix — and not as a repair path for architectural misalignments introduced before the first line of code was written. The two-checkpoint design described in Chapter 5 reflects this explicitly: Checkpoint 1 is a structural safety barrier that prevents the most expensive class of errors, while Checkpoint 2 handles the residual implementation failures that autonomous convergence cannot resolve.

6.7.3 Threats to Validity

Several limitations of the evaluation design should be acknowledged.

Benchmark scope. All 15 tasks are Python-only, self-contained modules or applications with no live external service dependencies. Production software development frequently involves cross-language interoperability, asynchronous distributed components, and real database or network interactions under partial failure conditions. Generalisation to such tasks would require a separate evaluation campaign.

Single-evaluator quality scores. The subjective code quality ratings were assigned by a single evaluator without a formal rubric beyond the stated criteria of correctness, readability, and idiomatic style. Inter-rater reliability was not assessed.

Absence of component ablation. The evaluation compares complete pipeline configurations but does not include ablation conditions (e.g., without the Critic loop, or without the Specification agent). Isolating the marginal contribution of

individual components would require a controlled ablation study, which is outside the scope of the current work but represents a direct direction for future research.

Operator response quality. The Checkpoint 2 directives used in this evaluation were specific and technically accurate. In a deployment with less experienced operators, recovery efficacy from Checkpoint 2 escalations would likely be lower. The present results therefore represent an upper bound on HITL recovery efficacy for the evaluated model configurations.

Chapter 7

Conclusions and Future Work

This thesis has investigated how the Software Development Lifecycle (SDLC) can be automated in an enterprise-grade cloud environment by orchestrating a cooperative team of specialised, foundation-model-powered agents under disciplined human oversight. Motivated by the gap between single-prompt code generation assistants and the end-to-end delivery of verified, tested, and documented software systems, the work developed, deployed, and empirically evaluated a serverless multi-agent pipeline on Amazon Web Services in collaboration with Data Reply S.r.l.

This concluding chapter synthesises the principal contributions of the research, reflects critically on the limitations of the current prototype, outlines strategic avenues for future work, and offers final considerations on the evolving role of human software engineers in an agentic development paradigm.

7.1 Summary of Contributions

The research presented across Chapters 3 through 6 advances both the theoretical framing and the practical engineering of multi-agent software engineering systems. Its contributions can be synthesised across four interrelated dimensions:

1. Decoupled Serverless Architecture and the Shared Artefact Protocol

Existing multi-agent frameworks often rely on persistent runtime state, direct inter-agent RPCs, or in-memory message brokers that introduce severe coupling, non-deterministic failure modes, and unbounded operational costs. This work

demonstrates that an enterprise-grade agentic pipeline can be constructed entirely on serverless primitives (Chapter 3).

By standardising all agent interactions around an immutable, versioned artefact protocol hosted on Amazon S3 and coordinated via AWS Step Functions, the pipeline achieves complete decoupling between the reasoning agents (stateless Lambda functions) and the heavy execution environments (sandboxed OpenHands containers on ECS Fargate). Each stage consumes validated JSON contracts from preceding phases and writes discrete output artefacts under deterministic S3 keys. This architecture guarantees full auditability, simplifies isolated unit testing of individual agent prompts, and enforces strict least-privilege security boundaries across the entire lifecycle.

2. Dual-Loop Governance and Human-in-the-Loop Integration

A central conceptual contribution of this thesis is the formalisation of *Dual-Loop Governance* for autonomous software development (Chapters 4 and 5). Rather than treating human intervention as an ad-hoc fallback or pursuing full autonomy as an unconstrained binary objective, the pipeline nests a rapid, autonomous self-correction loop within a slower, authoritative supervisory loop:

- **Autonomous Convergence Loop**: The Critic agent evaluates the workspace against the formal specification across up to five iterative cycles, generating targeted, structured diagnostic directives (`fix_hint`) that guide the Implementation agent to resolve defects without human involvement.
- **Supervisory Escalation Loop**: When autonomous convergence fails due to stagnation ($\Delta_{\text{consistency}} < 5$) or iteration exhaustion, the orchestrator pauses execution via Step Functions Task Tokens and triggers Checkpoint 2. The operator injects high-level corrective guidance that resets the loop budget and steers subsequent autonomous phases, preventing runaway inference while systematically rescuing stalled tasks.
- **Architectural Safety Barrier**: Upfront at Checkpoint 1, the Clarification agent detects requirement ambiguity before code generation begins. The empirical campaign demonstrated that this barrier is safety-critical: preventing false assumptions at inception avoids catastrophic architectural debt that cannot be repaired downstream.

3. The Specification Contract as an LLM Equaliser

Chapter 6 surfaced a major methodological finding: the structured specification generated by the Specification agent acts as an *architectural equaliser* between

the cognitive tiers of the pipeline. By translating natural-language intent into an unambiguous, machine-readable JSON schema defining file hierarchies, public interfaces, and adversarial test scenarios, the specification reduces the cognitive load of the implementation phase from open-ended software design to deterministic contract adherence.

Consequently, the pipeline does not require costly frontier models uniformly across all agents. Given a high-reasoning specification and high-precision Critic feedback, a lighter model can execute complex multi-file implementations faithfully, establishing that architectural precision at the contract boundary compensates for lower parameter scale at the execution layer.

4. Empirical Discovery of the Heterogeneous Tiered Configuration

Through a rigorous 15-task benchmark spanning four complexity classes and five homogeneous model configurations (Claude 3.5 Sonnet, Claude 3.5 Haiku, GPT-4o, Llama 3.3 70B, and Amazon Nova Pro), this work analytically derived and experimentally validated an optimal *Tiered* configuration. By pairing frontier reasoning (Claude 3.5 Sonnet) at the contract-defining and evaluative nodes (Clarification, Specification, Critic) with cost-efficient execution (Claude 3.5 Haiku) in the high-volume OpenHands containers (Implementation, Test), the tiered system achieved:

- **100% Autonomous Pass Rate** across all 15 tasks, matching the convergence reliability of the Sonnet-homogeneous baseline;
- **59.8% Cost Reduction** ($4.68 vs. $11.65 for the complete suite) and a **12.0% Latency Improvement** (2h 34m vs. 2h 55m);
- **Strict Pareto Dominance**: The tiered configuration established the sole non-dominated point on the Cost–Autonomy Pareto frontier, outperforming homogeneous deployments of GPT-4o ($25.62, 93.3% autonomy), Llama 3.3 70B ($9.48, 66.7% autonomy), and Nova Pro ($11.92, 53.3% autonomy).

7.2 Limitations of the Current System

While the prototype validates the core thesis and demonstrates industrial viability, several inherent limitations must be acknowledged:

Greenfield Scope and Isolated Application Boundaries

The evaluation suite was intentionally focused on greenfield Python development: standalone modules, algorithms, and multi-file micro-applications with self-contained dependencies. In enterprise software engineering, however, the overwhelming majority of engineering hours is dedicated to *brownfield* maintenance: modifying existing, multi-million-line codebases, refactoring legacy architectures, navigating implicit domain conventions, and managing subtle backward-compatibility constraints. The current pipeline lacks repository-scale static indexing or abstract syntax tree (AST) retrieval mechanisms necessary to ingest an established code repository without exceeding LLM context windows or inflating token costs.

Sandbox Environment and Live Service Mocking

The execution agents run inside ephemeral Docker containers on AWS Fargate with restricted egress and local filesystem tools. In the evaluation tasks, external dependencies (such as external HTTP endpoints, identity providers, or relational databases) were handled either in-memory (e.g., SQLite, `unittest.mock`) or via local standard-library implementations. Real-world systems frequently depend on distributed cloud infrastructure, live database schemas, streaming message brokers, and third-party APIs. Testing such interactions requires dynamic provisioning of isolated staging environments or testcontainers, capabilities that are not currently integrated into the orchestrator.

Single-Tenant and Single-Operator Governance

The human-in-the-loop checkpoint mechanisms are architected around a single operator receiving an email or dashboard notification with an AWS Step Functions task token. While effective for validating the state-machine dynamics, enterprise adoption requires multi-tenant governance: role-based access control (RBAC), multi-party approval chains for mission-critical code changes, cryptographically signed audit logs, and integration with organizational identity providers (e.g., via OAuth2/OIDC).

Frontier-Model Dependency for Uncertainty Calibration

The cross-model evaluation revealed that open-weights models (specifically Llama 3.3 70B) exhibited severe uncertainty calibration failure at Checkpoint 1, declaring ambiguous requirements as fully specified and propagating invalid architectural assumptions forward. Consequently, the pipeline's autonomous safety guarantees currently depend on proprietary frontier models for the Clarification and Specification nodes. Until open-weights models achieve comparable calibration and

instruction-following fidelity in complex structured extraction, fully self-hosted, air-gapped enterprise deployments will encounter higher failure rates.

7.3 Directions for Future Work

The findings and architectural abstractions established in this thesis provide a foundation for several promising research and engineering directions:

1. Brownfield Evolution and Repository-Scale Context Engineering

Extending the pipeline to brownfield development requires replacing the single-file requirement ingestion with an intelligent codebase comprehension layer. Future work should investigate integrating hierarchical AST indexing, graph databases of module call graphs, and semantic vector embeddings to enable agents to retrieve only the relevant slices of a large codebase. Furthermore, the Implementation and Test agents should be augmented with automated git-diff and impact-analysis tools, allowing the Critic to assess not only specification adherence but also regression risk against existing regression suites.

2. Dynamic and Adaptive Cognitive Tiering

The tiered configuration evaluated in Chapter 6 assigns foundation models statically per agent role. A natural evolution is *Dynamic Cognitive Tiering*: an adaptive orchestration policy where the Implementation agent begins execution using a highly cost-efficient model (such as Claude 3.5 Haiku), but the Step Functions state machine dynamically promotes subsequent iterations to a frontier model (such as Claude 3.5 Sonnet) if the Critic detects semantic stagnation or boundary-condition oscillation. Such an elastic allocation strategy could further compress operational costs while guaranteeing convergence on adversarial edge cases.

3. Interactive Streaming HITL via the Model Context Protocol (MCP)

The current checkpoint mechanism operates as an asynchronous, discrete stop-and-go gate. Future iterations should bridge the pipeline directly into the developer's integrated development environment (IDE) using the Model Context Protocol (MCP). By exposing agent telemetry, live Critic deliberations, and partial test results as an interactive streaming feed within tools like Visual Studio Code or JetBrains IDEs, the developer could provide fine-grained steering, accept or reject

intermediate diffs, and interact with the agents conversationally without leaving their primary workspace.

4. Polyglot and Multi-Container Staging Environments

Expanding the pipeline beyond Python requires container templates equipped with multilingual toolchains (Node.js, Go, Rust, Java) and build orchestrators. Integrating dynamic container orchestration (such as ephemeral Docker-in-Docker or Kubernetes test pods) would permit the Test agent to spin up live backing services (PostgreSQL, Redis, Kafka) during the verification loop, enabling true end-to-end integration testing of distributed microservices.

7.4 Concluding Remarks

The emergence of foundation models has provoked intense debate regarding the future of software engineering, ranging from naive claims of total automation to dismissive scepticism regarding model hallucinations and unreliability. The research presented in this thesis supports a more nuanced and disciplined conclusion.

Autonomous software engineering cannot be achieved merely by scaling the parameter count of a single monolithic language model and prompting it to write entire systems. Reliability, maintainability, and correctness emerge when classic software engineering principles — modular separation of concerns, explicit interface contracts, adversarial automated testing, and hierarchical decomposition — are applied to the orchestration of artificial intelligence. By decomposing the SDLC into discrete cognitive responsibilities, establishing immutable artefact contracts, and instituting formal human checkpoints at safety-critical boundaries, multi-agent systems can achieve remarkable levels of autonomy without sacrificing human control or economic viability.

In this paradigm, the role of the human software engineer is not eliminated; rather, it is elevated. Relieved from repetitive syntactic boilerplate and routine implementation cycles, the engineer transitions from a manual programmer into a system architect, specification designer, and supervisory arbiter. The collaborative framework developed in this thesis demonstrates that when autonomous multi-agent pipelines and human expertise are joined through principled system design, software development can become faster, more rigorous, and profoundly more accessible.

Bibliography

[1] OpenAI. *GPT-4 Technical Report*. Tech. rep. OpenAI, 2023. URL: https://arxiv.org/abs/2303.08774 (cit. on pp. 1, 5).

[2] Anthropic. *The Claude 3 Model Family: Opus, Sonnet, Haiku*. Technical report, Anthropic. 2024. URL: https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf (cit. on pp. 1, 5).

[3] Amazon Web Services. *Amazon Bedrock — Fully Managed Foundation Models*. AWS Documentation. 2024. URL: https://aws.amazon.com/bedrock/ (cit. on pp. 1, 9).

[4] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer, 2012 (cit. on p. 2).

[5] Mark Chen et al. «Evaluating Large Language Models Trained on Code». In: *arXiv preprint arXiv:2107.03374* (2021) (cit. on pp. 2, 6).

[6] Michael Wooldridge. *An Introduction to MultiAgent Systems*. Second. John Wiley & Sons, 2009 (cit. on pp. 2, 7).

[7] Qingyun Wu et al. «AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation». In: *Proceedings of the ICLR 2024 Workshop on Large Language Model (LLM) Agents*. 2024. URL: https://arxiv.org/abs/2308.08155 (cit. on pp. 2, 8).

[8] Xingyao Wang et al. «OpenHands: An Open Platform for AI Software Developers as Generalist Agents». In: *arXiv preprint arXiv:2407.16741* (2024). URL: https://arxiv.org/abs/2407.16741 (cit. on p. 2).

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. «Attention Is All You Need». In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 30. 2017 (cit. on p. 5).

[10] Tom B. Brown et al. «Language Models are Few-Shot Learners». In: *arXiv preprint arXiv:2005.14165* (2020) (cit. on p. 5).

[11] Daniel Fried et al. «InCoder: A Generative Model for Code Infilling and Synthesis». In: *arXiv preprint arXiv:2204.05999* (2022) (cit. on p. 6).

[12] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. «SWE-bench: Can Language Models Resolve Real-World GitHub Issues?» In: *International Conference on Learning Representations (ICLR)*. 2024. URL: https://arxiv.org/abs/2310.06770 (cit. on pp. 6, 12).

[13] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. *Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing*. Vol. 55. 9. 2023, pp. 1–35 (cit. on p. 6).

[14] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. «Chain-of-Thought Prompting Elicits Reasoning in Large Language Models». In: *arXiv preprint arXiv:2201.11903* (2022) (cit. on p. 7).

[15] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. «ReAct: Synergizing Reasoning and Acting in Language Models». In: *International Conference on Learning Representations (ICLR)*. 2023. URL: https://arxiv.org/abs/2210.03629 (cit. on p. 7).

[16] Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy. «The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty». In: *ACM Computing Surveys* 12.2 (1980), pp. 213–253 (cit. on p. 7).

[17] LangChain AI. *LangGraph: Build Resilient Language Agents as Graphs*. GitHub repository. 2024. URL: https://github.com/langchain-ai/langgraph (cit. on p. 8).

[18] Anthropic. *Model Context Protocol — Specification*. Online documentation. 2024. URL: https://modelcontextprotocol.io/specification (cit. on p. 9).

[19] Amazon Web Services. *AWS Step Functions — Developer Guide*. AWS Documentation. 2024. URL: https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html (cit. on p. 10).

[20] Amazon Web Services. *AWS Lambda — Developer Guide*. AWS Documentation. 2024. URL: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html (cit. on p. 10).

[21] Amazon Web Services. *Amazon S3 — User Guide*. AWS Documentation. 2024. URL: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html (cit. on p. 10).

[22] Robert Munro Monarch. *Human-in-the-Loop Machine Learning*. Shelter Island, NY: Manning Publications, 2021 (cit. on p. 10).

[23] Tongshuang Wu, Marco Tulio Ribeiro, Jeffrey Heer, and Daniel S. Weld. «Promptchainer: Chaining Large Language Model Prompts through Visual Programming». In: *arXiv preprint arXiv:2203.06566* (2022) (cit. on p. 11).

[24] Saleema Amershi et al. «Guidelines for Human-AI Interaction». In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI 2019)*. ACM, 2019. URL: https://dl.acm.org/doi/10.1145/3290605.3300233 (cit. on p. 11).

[25] Chen Qian et al. «ChatDev: Communicative Agents for Software Development». In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL 2024)*. 2024. URL: https://arxiv.org/abs/2307.07924 (cit. on p. 11).

[26] Sirui Hong et al. «MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework». In: *arXiv preprint arXiv:2308.00352* (2023). URL: https://arxiv.org/abs/2308.00352 (cit. on p. 11).

[27] John Ferguson Smart. *BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle*. Second. Shelter Island, NY: Manning Publications, 2023 (cit. on p. 30).