FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Evaluating Reliability in Conversational AI Agents for E-Commerce: A Comparative Study of Agent Architectures

José Henrique Nogueira Campos de Morais Pinheiro

Mestrado em Engenharia de Software

Supervisor: Prof. Jorge Augusto Melegati Gonçalves

July 30, 2026

Evaluating Reliability in Conversational AI Agents for E-Commerce: A Comparative Study of Agent Architectures

José Henrique Nogueira Campos de Morais Pinheiro

Mestrado em Engenharia de Software

July 30, 2026

Resumo

Os agentes de Inteligência Artificial (IA) conversacional estão a ser cada vez mais utilizados no contexto do atendimento ao cliente no comércio eletrónico por forma a apoiar a descoberta de produtos, consultas de stock, criação de carrinhos de compras, acompanhamento de encomendas, consulta de políticas e operações pós-venda. Embora os Grandes Modelos de Linguagem (LLMs) tenham melhorado a flexibilidade e a naturalidade destes sistemas, a sua fiabilidade continua a apresentar-se como um desafio crítico quando são chamados a interagir com ferramentas externas e a realizar tarefas com várias etapas. Erros como a seleção incorreta de ferramentas, a construção incompleta de argumentos, referências a produtos inexistentes, uma validação fraca dos resultados ou a interrupção prematura de tarefas podem afetar diretamente a confiança do utilizador e a qualidade do serviço.

Esta dissertação investiga de que forma diferentes arquiteturas de agentes influenciam a fiabilidade de agentes conversacionais de IA com suporte a ferramentas num contexto de e-commerce. Foram concebidas, implementadas e avaliadas três arquiteturas no mesmo ambiente controlado: uma arquitetura Simple Agent, uma arquitetura Planner-Critic e uma arquitetura Context-Specific Multi-Agent. A arquitetura Simple Agent serve como base de comparação, na qual um único agente tem acesso ao catálogo completo de ferramentas. A arquitetura Planner-Critic introduz uma etapa de validação antes da execução, através de um crítico que avalia as ações de ferramenta propostas pelo planner antes da sua execução. Por sua vez, a arquitetura Context-Specific Multi-Agent decompõe o sistema em agentes especializados, coordenados por um encaminhador e supervisionados por um Stop Judge, responsável por avaliar se a tarefa foi concluída.

A avaliação foi realizada através de 89 consultas de teste, compostas por 58 consultas originais e 31 variantes de paráfrase, em dois modelos de base: `gpt-3.5-turbo` e `gpt-4o-mini`. As arquiteturas foram avaliadas com base na correção da resposta final e em métricas ao nível do processo, incluindo o comportamento das chamadas de ferramentas, a estrutura de invocação do LLM, os ciclos de revisão, as decisões do Stop Judge, a latência da resposta, o consumo de tokens e a robustez das paráfrases.

Os resultados mostram que a arquitetura afeta significativamente o perfil de fiabilidade dos agentes baseados em LLM. O Simple Agent melhorou substancialmente ao utilizar o `gpt-4o-mini`, aumentando a correção de 0.872 para 0.956 no conjunto de consultas originais, mas permaneceu vulnerável a falhas de validação e de conclusão de tarefas. A arquitetura Planner-Critic alcançou um forte desempenho com o `gpt-3.5-turbo`, atingindo uma correção de 0.960, mas sofreu uma degradação com o `gpt-4o-mini`, caindo para 0.862. Esta diminuição esteve associada a um comportamento mais restritivo do crítico e a falhas no ciclo de revisão. A arquitetura Context-Specific Multi-Agent alcançou o desempenho mais estável, com pontuações de correção de 0.968 e 0.976 nos dois modelos no conjunto de consultas original, indicando que o encaminhamento para um domínio mais específico, a exposição restrita às ferramentas e o julgamento explícito da conclusão podem reduzir várias classes de falhas.

Em termos gerais, os resultados demonstram que a fiabilidade dos agentes conversacionais

apoiados por ferramentas não é determinada exclusivamente pelo modelo base. Em vez disso, esta é o resultado da interação entre a capacidade do modelo, o acesso às ferramentas, a estrutura de orquestração, os mecanismos de validação e o controlo da conclusão das tarefas. Arquiteturas mais estruturadas podem melhorar a fiabilidade e a robustez, mas também introduzem latência adicional, consumo de tokens e riscos de coordenação. Estes resultados destacam a importância de avaliar os agentes de IA conversacionais não só pela qualidade da resposta final, mas também pelos processos internos através dos quais as respostas e as ações são produzidas.

Palavras-chave: Agentes de IA • Auto-correção • Fiabilidade • Raciocínio • LLMs com ferramentas • Sistemas conversacionais • E-commerce

Abstract

Conversational artificial intelligence (AI) agents are increasingly being used in e-commerce customer service to support product discovery, stock inquiries, shopping cart creation, order tracking, policy inquiries, and after-sales operations. Although Large Language Models (LLMs) have improved the flexibility and naturalness of these systems, their reliability remains a critical challenge when they are called upon to interact with external tools and perform multi-step tasks. Errors such as incorrect tool selection, incomplete argument construction, references to nonexistent products, poor validation of results, or premature termination of tasks can directly affect user trust and service quality.

This dissertation investigates how different agent architectures influence the reliability of tool-supported conversational AI agents in an e-commerce context. Three architectures were designed, implemented, and evaluated in the same controlled environment: a Simple Agent, a Planner-Critic architecture, and a Context-Specific Multi-Agent architecture. The Simple Agent serves as a baseline for comparison, in which a single agent has access to the complete catalogue of tools. The Planner-Critic architecture introduces pre-execution validation through a critic that evaluates the tool actions proposed by the planner before execution. The Context-Specific architecture, on the other hand, breaks the system down into specialised agents, coordinated by a router and supervised by a Stop Judge, which is responsible for evaluating task completion.

The evaluation was conducted using 89 test queries, consisting of 58 original queries and 31 paraphrased variants, on two base models: `gpt-3.5-turbo` and `gpt-4o-mini`. The architectures were evaluated based on the correctness of the final response and on process-level metrics, including tool call behaviour, LLM invocation structure, revision cycles, Stop Judge decisions, response latency, token consumption, and paraphrase robustness.

The results show that the architecture significantly affects the reliability profile of LLM-based agents. The Simple Agent improved substantially when using `gpt-4o-mini`, increasing correctness from 0.872 to 0.956 on the original query set, but remained vulnerable to validation and task completion failures. The Planner-Critic architecture achieved strong performance with `gpt-3.5-turbo`, reaching a correctness score of 0.960, but showed lower correctness with `gpt-4o-mini`, dropping to 0.862 in the original query set. This decrease was associated with more restrictive critic behaviour and failures in the revision cycle. The Context-Specific Multi-Agent architecture achieved the most stable performance, with correctness scores of 0.968 and 0.976 on the two models across the original query set, indicating that targeting a more specific domain, restricted exposure to tools, and explicit completion assessment can reduce various classes of failures.

In general, the results show that the reliability of tool-supported conversational agents is not determined solely by the base model. Rather, it results from the interaction between the model's capabilities, access to tools, the orchestration framework, validation mechanisms, and task completion control. More structured architectures can improve reliability and robustness, but they also introduce additional latency, token consumption, and coordination risks. These results highlight

the importance of evaluating conversational AI agents not only by the quality of the final response but also by the internal processes through which responses and actions are produced.

Keywords: AI Agents • Self-Correction • Reliability • Reasoning • Tool-Augmented LLMs • Conversational Systems • E-commerce

The United Nations Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework for addressing major economic, social, technological, and environmental challenges. Although this dissertation does not directly target environmental or social policy outcomes, it contributes indirectly to the development of more reliable and resilient digital systems. In particular, it relates to the design and evaluation of tool-supported conversational AI agents for e-commerce customer service, where reliability, automation, correctness, and process control are important for digital commerce infrastructure.

This dissertation contributes mainly to SDG 8 and SDG 9. This connection is indirect and should be understood in terms of software engineering research, AI system evaluation, and digital service reliability. The work does not claim to measure direct SDG impact, instead, it provides empirical evidence on how different agent architectures affect the reliability, robustness, latency, and cost of conversational AI systems in e-commerce scenarios.

The specific Sustainable Development Goals considered are the following:

SDG 8 – Promote sustained, inclusive and sustainable economic growth, full and productive employment and decent work for all.
SDG 9 – Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation.

Table 1: Relationship between the dissertation and selected United Nations Sustainable Development Goals

SDG	Target	Contribution	Thesis-related indicators and metrics
8	8.2	This dissertation supports technological upgrading and innovation by studying how conversational AI agents can be made more reliable in e-commerce customer service. By comparing different agent architectures, the work contributes to understanding how automation can support digital service quality and reduce failures in tool-assisted workflows.	Final response correctness; tool-call behaviour; task completion failures; architecture-level comparison across two language models.
9	9.1	This dissertation contributes to the development of more reliable and resilient digital infrastructure by evaluating how tool-supported AI agents behave when connected to product catalogues, shopping cart operations, order tracking, policy retrieval, and after-sales tools. The results highlight how architectural design can reduce or introduce reliability risks in digital commerce systems.	Correctness scores; failure patterns; paraphrase robustness; Stop Judge decisions; revision-cycle behaviour; latency.
	9.5	This dissertation supports applied research and technological innovation in software engineering by designing, implementing, and empirically evaluating three conversational AI agent architectures under a shared controlled environment. The comparison between the Simple Agent, Planner-Critic, and Context-Specific Multi-Agent architectures provides evidence on the relationship between base model capability, tool access, validation, routing, and task completion control.	Comparative evaluation of three architectures; evaluation across `gpt-3.5-turbo` and `gpt-4o-mini`; 89 test queries; process-level metrics; token consumption; latency; robustness to paraphrase variation.

Acknowledgements

I would like to begin by expressing my sincere gratitude to my supervisor, Professor Jorge Melegati, for his guidance, feedback, and support throughout the development of this dissertation. His advice was essential in helping me refine the direction of this work and approach the research problem with greater clarity and rigour.

I would also like to thank Kodly for providing me with the context and opportunity to explore practical challenges in conversational AI within an e-commerce environment. This experience allowed me to connect academic research with real problems encountered in the daily work of a software engineer. In particular, I would like to express my deep appreciation to Carlos Silva and Tiago Gomes, who closely followed this project and supported, encouraged, and motivated me throughout its development.

I am also grateful to my friends from *Coiso*, to Bia, Pedrosa, and Luís, who shared this journey with me as we supported each other through the rollercoaster of writing a master's dissertation. I would also like to thank Bárbara, for always listening to my endless presentations about topics far outside her field over the years. Their presence and support made this process lighter and more meaningful. I would also like to extend my gratitude to all my friends who, in different ways, encouraged me, supported me, and helped me stay motivated throughout this process.

I am deeply grateful to my family for their constant support and encouragement throughout this journey. Their presence gave me the motivation to keep going during the most demanding periods of this work. I would like to thank especially my sister, for patiently listening to my many presentations and reflections about this dissertation, and for supporting me throughout the process.

As this chapter comes to a close, I would like to thank everyone who, directly or indirectly, contributed to this dissertation and to my academic path. This work represents not only the conclusion of a master's degree, but also the result of the support, trust, and encouragement I received along the way.

Finally, and most importantly, I am grateful to God, not only for this journey, but for everything He added along the way and for the way He helped me embrace each new challenge without having to give up anything on the path.

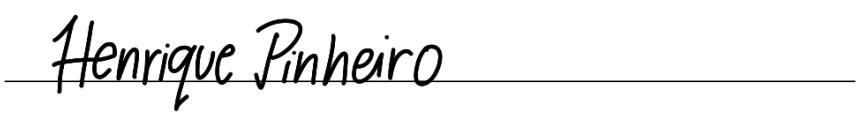
José Henrique Nogueira Campos de Morais Pinheiro

Declaration of honour

I declare that this dissertation is my own work and has not previously been submitted or assessed at this or any other institution. References to other authors (statements, ideas, thoughts), as well as the use of published works, strictly comply with the rules of attribution and are duly identified in the text and in the bibliographic references, in accordance with the applicable referencing standards. I am aware that the practice of plagiarism or self-plagiarism constitutes an academic offence, as established in the U.Porto Code of Ethics.

I also declare that I have used Artificial Intelligence tools to complete part(s) of this dissertation, and that this use and its results are duly noted and have been carefully checked for errors, inaccuracies, unfounded attributions or other forms of bias in content and arguments, as well as determining authorship.

José Henrique Nogueira Campose de Morais Pinheiro



Contents

1 Introduction **1**
- 1.1 Motivation . . . 2
- 1.2 Problem Statement . . . 3
- 1.3 Research Questions . . . 4
- 1.4 Objectives . . . 4
- 1.5 Dissertation Structure . . . 5

2 Literature Review **6**
- 2.1 Introduction . . . 6
- 2.2 Reasoning and Self-correction techniques . . . 6
- 2.3 Frameworks and Architectures . . . 9
  - 2.3.1 From single-agent systems to reusable development frameworks . . . 9
  - 2.3.2 Multi-agent conversation frameworks and collaborative task execution . . . 10
  - 2.3.3 Graph-based orchestration and stateful multi-agent workflows . . . 11
  - 2.3.4 Tool-Augmented and Planning-Centric Architectures . . . 12
- 2.4 E-commerce . . . 13
- 2.5 Summary and Conclusion . . . 15

3 Proposed Solutions **17**
- 3.1 Introduction and Design Philosophy . . . 17
- 3.2 Shared Technical Infrastructure . . . 18
  - 3.2.1 System Stack . . . 18
  - 3.2.2 Knowledge Bases and Semantic Retrieval . . . 18
  - 3.2.3 Tool Catalogue . . . 19
- 3.3 Simple Agent Architecture . . . 20
  - 3.3.1 Motivation and Design Rationale . . . 20
  - 3.3.2 Architectural Overview . . . 20
  - 3.3.3 State Representation . . . 21
  - 3.3.4 Tool Binding and Sequential Invocation . . . 22
  - 3.3.5 Output . . . 22
  - 3.3.6 Limitations . . . 22
- 3.4 Planner-Critic Agents Architecture . . . 23
  - 3.4.1 Motivation and Design Rationale . . . 23
  - 3.4.2 Graph Structure and State . . . 23
  - 3.4.3 Planner Node . . . 24
  - 3.4.4 Critic Node . . . 25
  - 3.4.5 Tool Calling Protocol Compliance . . . 26
  - 3.4.6 Revision Node . . . 26

3.4.7 Tool Execution Node . . . 27
3.4.8 Graph Routing . . . 27
3.4.9 Limitations . . . 27
3.5 Context-Specific Agents Architecture . . . 28
3.5.1 Motivation and Design Rationale . . . 28
3.5.2 Global State . . . 29
3.5.3 Graph Structure . . . 30
3.5.4 Router Node . . . 31
3.5.5 Specialist Agents . . . 31
3.5.6 Domain-Restricted Tool Registry . . . 33
3.5.7 Stop Judge Node . . . 34
3.5.8 Final Answer Agent . . . 35
3.5.9 Product Lookup with Confidence Scoring . . . 35
3.5.10 Limitations . . . 37
3.6 Comparative Overview . . . 37

4 Methodology **39**
4.1 Overview and Evaluation Objectives . . . 39
4.2 Evaluation Dimensions . . . 39
4.3 Test Query Set . . . 40
4.3.1 Composition and Design Approach . . . 40
4.3.2 Testing Techniques and Query Design Criteria . . . 41
4.3.3 Original Query Set (Q1–Q58) . . . 42
4.3.4 Paraphrase Variant Queries (Q1-A to Q57-A) . . . 44
4.3.5 Ground Truth Specification . . . 45
4.4 Evaluation Execution Procedure . . . 45
4.4.1 Evaluation Script . . . 45
4.4.2 State Isolation Between Tests . . . 46
4.4.3 Infrastructure Requirements . . . 46
4.5 Logging and Observability . . . 46
4.5.1 Logging Configuration . . . 46
4.5.2 Logged Events . . . 47
4.5.3 The `executed_tools` Record . . . 47
4.6 Data Extraction Workflow . . . 47
4.6.1 From Logs to Structured Tables . . . 48
4.6.2 Fields Requiring Manual Annotation . . . 48
4.7 Metrics . . . 49
4.7.1 Quantitative Metrics Derivable from Logs . . . 49
4.7.2 Metrics Requiring Manual Annotation . . . 49
4.8 Weaker Model Stress Test . . . 50
4.9 Result Classification Scheme . . . 50
4.9.1 Final Response Correctness . . . 50
4.9.2 Error Type Taxonomy . . . 53
4.9.3 Dual Assessment: Response and Process . . . 53
4.10 Limitations . . . 54
4.10.1 Manual Annotation Requirement . . . 54
4.10.2 Single Execution Per Test . . . 54
4.10.3 Model Version Sensitivity . . . 54
4.10.4 Controlled Catalogue Scope . . . 54

4.10.5 Mocked Transactional Data . . . 55
4.10.6 API Rate Limits . . . 55

5 Results and Discussion **56**
5.1 Overview of Evaluation Results . . . 56
5.2 Response Correctness: Architecture-Level Failure Analysis . . . 57
5.2.1 Quantitative Comparison of All Architectures on Original Queries ($n = 58$) 57
5.2.2 Simple Agent . . . 58
5.2.3 Planner-Critic . . . 60
5.2.4 Context-Specific Multi-Agent . . . 62
5.3 Paraphrase Invariance . . . 63
5.3.1 Simple Agent . . . 64
5.3.2 Planner-Critic . . . 65
5.3.3 Context-Specific Multi-Agent . . . 65
5.4 Process-Level Metrics . . . 66
5.4.1 Number of Tool Calls per Query . . . 66
5.4.2 Number of LLM Invocations per Query . . . 67
5.4.3 Critic Revision Cycles (Planner-Critic Architecture) . . . 67
5.4.4 Stop Judge Decisions (Context-Specific Architecture) . . . 68
5.4.5 Response Latency . . . 68
5.4.6 Token Consumption . . . 70
5.5 Answers to the Research Questions . . . 70

6 Conclusion **75**
6.1 Main Contributions . . . 76
6.2 Limitations . . . 77
6.3 Future Work . . . 77
6.4 Final Remarks . . . 78

References **80**

A Evaluation Queries **82**
A.1 Original Test Suite (Q1–Q58) . . . 83
A.1.1 Presales – Specific Lookup . . . 83
A.1.2 Presales – Brand/Category Browse . . . 83
A.1.3 Presales – Stock Check . . . 83
A.1.4 Presales – Price Filter . . . 84
A.1.5 Cart . . . 84
A.1.6 Information . . . 84
A.1.7 Cross-Cutting . . . 85
A.1.8 Aftersales – Track Order . . . 85
A.1.9 Aftersales – Refund . . . 85
A.1.10 Aftersales – Receipt . . . 86
A.1.11 Campaigns . . . 86
A.1.12 Presales – Close Match . . . 86
A.1.13 Presales – Zero Results . . . 86
A.1.14 Presales – Relevance Sort . . . 87
A.1.15 Cart – Degenerate . . . 87
A.1.16 Cross-Cutting – Combined . . . 87

A.1.17 Out-of-Scope . . . 87
A.2 Paraphrase Variant Queries (Robustness Testing) . . . 88
A.2.1 RT – Specific Lookup . . . 88
A.2.2 RT – Stock Check . . . 88
A.2.3 RT – Close Match . . . 88
A.2.4 RT – Not Found . . . 88
A.2.5 RT – Price Filter . . . 89
A.2.6 RT – Price Range . . . 89
A.2.7 RT – Cart Multi-item . . . 89
A.2.8 RT – Cart Category . . . 89
A.2.9 RT – Information . . . 90
A.2.10 RT – Track Order . . . 90
A.2.11 RT – Refund . . . 90
A.2.12 RT – Campaigns . . . 90
A.2.13 RT – Zero Results . . . 91
A.2.14 RT – Combined . . . 91
A.2.15 RT – Semantic Category . . . 91

List of Figures

3.1 Simple Agent architecture: sequential tool invocation loop. 21
3.2 Planner-Critic Architecture: pre-execution tool-plan validation with conditional routing and revision feedback. 24
3.3 Context-Specific Multi-Agent Architecture: routing-based specialist dispatch with restricted tool access, shared workspace, and Stop Judge completion control. . . . 30

List of Tables

1 Relationship between the dissertation and selected United Nations Sustainable Development Goals v

3.1 Comparative summary of architectural properties across the three proposed solutions 38

4.1 Final response correctness labels and definitions 51
4.2 Error type taxonomy used for manual failure annotation 53

5.1 Correctness scores by architecture and model 56
5.2 Outcome distribution by architecture and model on the original queries 57
5.3 Descriptive correctness statistics by architecture and model on the original queries 57
5.4 Quantitative comparison of Simple Agent correctness on the original queries . . . 58
5.5 Quantitative comparison of Planner-Critic correctness on the original queries . . 61
5.6 Retries and error patterns for Planner-Critic on the original queries 61
5.7 Quantitative comparison of Context-Specific correctness on the original queries . 62
5.8 Error indicators for Context-Specific on the original queries 62
5.9 Paraphrase invariance by configuration 64
5.10 Average tool calls per original query 66
5.11 Response latency by architecture and model on the original queries 69

A.1 Original test suite – Presales – Specific Lookup 83
A.2 Original test suite – Presales – Brand/Category Browse 83
A.3 Original test suite – Presales – Stock Check 83
A.4 Original test suite – Presales – Price Filter 84
A.5 Original test suite – Cart 84
A.6 Original test suite – Information 85
A.7 Original test suite – Cross-Cutting 85
A.8 Original test suite – Aftersales – Track Order 85
A.9 Original test suite – Aftersales – Refund 85
A.10 Original test suite – Aftersales – Receipt 86
A.11 Original test suite – Campaigns 86
A.12 Original test suite – Presales – Close Match 86
A.13 Original test suite – Presales – Zero Results 86
A.14 Original test suite – Presales – Relevance Sort 87
A.15 Original test suite – Cart – Degenerate 87
A.16 Original test suite – Cross-Cutting – Combined 87
A.17 Original test suite – Out-of-Scope 87
A.18 Paraphrase variant queries – RT – Specific Lookup 88
A.19 Paraphrase variant queries – RT – Stock Check 88
A.20 Paraphrase variant queries – RT – Close Match 88

A.21 Paraphrase variant queries – RT – Not Found 88
A.22 Paraphrase variant queries – RT – Price Filter 89
A.23 Paraphrase variant queries – RT – Price Range 89
A.24 Paraphrase variant queries – RT – Cart Multi-item 89
A.25 Paraphrase variant queries – RT – Cart Category 89
A.26 Paraphrase variant queries – RT – Information 90
A.27 Paraphrase variant queries – RT – Track Order 90
A.28 Paraphrase variant queries – RT – Refund 90
A.29 Paraphrase variant queries – RT – Campaigns 90
A.30 Paraphrase variant queries – RT – Zero Results 91
A.31 Paraphrase variant queries – RT – Combined 91
A.32 Paraphrase variant queries – RT – Semantic Category 91

List of Acronyms

AI Artificial Intelligence

LLM Large Language Model

CoT Chain-of-Thought

RAG Retrieval-Augmented Generation

API Application Programming Interface

ABMS Agent-Based Modelling and Simulation

ICL In-Context Learning

SFT Supervised Fine-Tuning

Chapter 1

Introduction

Large Language Models (LLMs) have significantly improved the flexibility and naturalness of conversational systems, thereby enabling more open interaction, reasoning about user requests, and integration with external tools or knowledge sources. This evolution is in line with broader developments in LLM-based systems, including reasoning strategies such as the "Chain-of-Thought" prompt [19], reasoning and action paradigms such as ReAct [21], and tool-enhanced approaches, in which models interact with external functions or APIs to overcome the limitations of purely linguistic generation [12, 17]. However, this flexibility also makes their behaviour less predictable.

The use of conversational artificial intelligence has also been gaining increasing prominence in e-commerce, where users expect fast, accurate, and context-sensitive support throughout the entire customer journey. E-commerce has evolved from a basic channel for online transactions into a complex digital ecosystem, increasingly supported by artificial intelligence for automation, recommendations, customer communication, and operational optimisation [8]. In this context, AI-based chatbots and virtual assistants have become important tools for customer support [15], offering round-the-clock availability, assisting with customer inquiries, supporting product discovery, and contributing to digital customer relationship management [7, 15, 22]. Consequently, conversational agents are no longer limited to answering simple frequently asked questions; they are increasingly expected to compare alternatives, check stock availability, explain policies, create shopping carts, track orders, and provide support for after-sales operations [2, 7, 22].

In real-world customer service scenarios, an agent may select the wrong tool, omit an important restriction when formulating a query, accept an incorrect result provided by a tool, invent unavailable products, or interrupt the process before completing a necessary transactional action. In an e-commerce context, these failures are not mere technical glitches. They can mislead users, reduce trust, compromise service quality, and potentially affect business outcomes. This is especially relevant because the quality of chatbot responses has been linked to customer satisfaction [7], trust [22], and purchasing behaviour [2, 7].

The literature on self-reflection and iterative self-correction suggests that LLM results can sometimes be improved through cycles of feedback, critique, and refinement [13, 16, 18]. At the architectural level, recent work also points to a broader shift from monolithic LLM agents to

modular, stateful, and multi-agent systems [4, 20], supported by orchestration frameworks such as LangGraph [1, 10] and by more comprehensive agent design patterns involving planning, reflection, and tool use [6, 21]. These developments suggest that reliability may depend not only on the base model but also on how reasoning, tool access, validation, and workflow control are structured.

This dissertation aims to investigate how different agent architectures influence the reliability of conversational Artificial Intelligence (AI) agents, specifically in the context of e-commerce. The study compares three architectures of increasing structural complexity: a Simple Agent, a Planner-Critic architecture, and a Context-Specific Multi-Agent architecture. The Simple Agent serves as a baseline model, in which a single agent has access to all tools and is responsible for interpreting the user's request, selecting tools, processing the results of the tools, and generating the final response. The Planner-Critic architecture introduces a pre-execution validation phase, in which a critic evaluates the action of the tool proposed by the planner before execution. The Context-Specific Multi-Agent architecture decomposes the system into specialised agents, coordinated by a router and supervised by a Stop Judge, which determines whether the task has been completed.

The main objective of this study is not only to determine which architecture achieves the highest score in terms of correctness, but also to understand how and why each architecture fails. For this reason, the evaluation assigns equal importance to the correctness of the final response and to process-level behaviour, including tool selection, argument construction, frequency of tool calls, revision cycles, Stop Judge decisions, latency, token consumption, and robustness to paraphrase variation. This dual perspective is important because two systems may produce similar final responses, while relying on very different internal processes, costs, and failure modes.

1.1 Motivation

The motivation for this dissertation stems from the discrepancy between the capabilities of LLM-based conversational agents and the reliability requirements of real-world e-commerce systems. Modern LLMs can produce fluent, context-rich responses, but fluency alone is not sufficient for customer service. In e-commerce, responses must be grounded in the actual product catalogue, stock information, business rules, order data, and available service operations. Since customer satisfaction and retention are influenced by service quality, responsiveness, communication style, trust, and perceived usefulness, a convincing but incorrect response can be more harmful than an explicit failure [7, 15, 22].

This challenge becomes even more significant when conversational agents are connected to external tools. Using these tools allows agents to search the catalogue, check inventory, create shopping carts, retrieve policies, or initiate after-sales operations, extending their usefulness beyond generic text generation [12, 17]. However, it also introduces additional points of failure: selecting the appropriate tool, constructing valid arguments, interpreting the returned data, and deciding whether further action is necessary. Errors in any of these steps can compromise the final response.

In light of this, this work aims to explore whether explicit reasoning and self-correction mechanisms could improve the reliability of agents, building on previous research on self-reflection and iterative improvement [13, 16, 18]. During development, the scope evolved into a broader architectural comparison. Instead of evaluating just a single self-correction mechanism, this dissertation compares three approaches to reliability: a simple baseline, a pre-execution validation architecture, and a specialised multi-agent architecture.

This architectural comparison is also supported by previous work on the design of LLM-based agents. Agent-based workflow patterns identify reflection, planning, collaboration among multiple agents, and the use of tools as key mechanisms for improving LLM-based systems [6]. ReAct demonstrates how reasoning and action can be interwoven in tool-using agents [21], while multi-agent frameworks such as AutoGen demonstrate how collaboration among specialised agents can support the execution of complex tasks [20]. Graph-based orchestration frameworks, such as LangGraph, provide mechanisms for stateful and cyclic workflows [1], and verification-oriented tool-using architectures, such as ToolFiVe, highlight the role of tool filtering and plan validation in improving reliability [12]. Taken together, these studies suggest that reliability may depend not only on the underlying model but also on how the surrounding system structures reasoning, tool access, validation, and workflow control.

1.2 Problem Statement

The research problem addressed in this dissertation is the reliability of LLM-based conversational agents when deployed in tool-assisted e-commerce customer service scenarios. Although LLMs can interpret natural language and generate fluent responses, their behaviour remains vulnerable to various recurring failure modes when called upon to interact with external tools and perform multi-step tasks.

In the context of this work, reliability refers to the agent's ability to correctly interpret the user's request, select the appropriate tool or tools, construct semantically valid arguments, use the tool results appropriately, complete the necessary actions, and produce a final response that accurately reflects the system's state. Given the diversity of steps, failures can occur at different stages of the process. An agent may search the wrong product category, ignore a price or brand restriction, treat a semantically similar but incorrect product as a valid match, trigger an unrelated tool, create an incomplete shopping cart, or report task completion when the required transactional step has not been executed.

A further dimension of reliability concerns robustness to paraphrase variation. In real customer service interactions, users may express the same intention through different surface forms, using alternative wording, reordered product names, synonyms, or more implicit descriptions. A reliable conversational agent should therefore preserve the same underlying behaviour when the user intent remains unchanged. This is consistent with the broader view of model consistency as invariance under meaning-preserving changes in the input [5]. For this reason, this dissertation includes

paraphrase variants in the evaluation set and explicitly examines whether each architecture remains stable when equivalent requests are reformulated.

The challenge is not limited, therefore, to improving natural language generation. The central question is how to design and study agent architectures that make the use of tools, reasoning, validation, and task completion more reliable. This raises a broad and rich architectural question: should reliability be achieved primarily through a more robust base model, pre-execution analysis, agent specialisation in specific domains, or explicit workflow control mechanisms?

This dissertation aims to address this question through the design, implementation, and evaluation of three agent architectures in the same e-commerce environment, tool catalogue, query set, and model configurations. By comparing their correctness, failure modes, robustness, latency, and token consumption, the study aims to identify the strengths and limitations of each architectural approach.

1.3 Research Questions

This dissertation is guided by the following research questions:

- **RQ1.** How does agent architecture affect the reliability of tool-augmented conversational AI agents in e-commerce customer service tasks?
- **RQ2.** To what extent do explicit validation and self-correction mechanisms reduce tool-use errors and incomplete task execution?
- **RQ3.** How robust are the evaluated architectures to paraphrase variation?
- **RQ4.** What are the trade-offs between reliability, latency, and token consumption across the evaluated architectures?
- **RQ5.** How does changing the base language model affect each architecture?

1.4 Objectives

The main objective of this dissertation is to evaluate how different agent architectures affect the reliability of conversational AI agents in an e-commerce customer service context.

To achieve this goal, the work defines the following specific objectives:

1. Review relevant literature on LLM reasoning, self-correction, tool use, multi-agent architectures, and conversational AI in e-commerce.
2. Design and implement the architectures under consideration in a controlled e-commerce customer service environment featuring a shared catalogue, knowledge base, order data, and set of tools.

3. Build an evaluation dataset that includes original queries and paraphrased variants covering product discovery, stock checking, cart creation, information retrieval, campaigns, and post-sale operations.

4. Evaluate every architecture using two base language models, `gpt-3.5-turbo` and `gpt-4o-mini`, under comparable conditions.

5. Measure and analyze final response correctness, paraphrase robustness, process-level behaviour, latency, and token consumption.

6. Analyze the primary failure modes of each architecture and identify the conditions under which architectural complexity improves or impairs reliability.

7. Obtain practical insights into the design of reliable, tool-assisted conversational agents for e-commerce customer service.

1.5 Dissertation Structure

This dissertation is organised into six main chapters.

Chapter 1 introduces the research context, motivation, problem statement, research questions, objectives, and overall structure of the dissertation.

Chapter 2 presents a literature review covering LLM reasoning, self-correction techniques, tool-augmented agents, multi-agent architectures, graph-based orchestration, and conversational AI in e-commerce. This chapter establishes the theoretical foundations that underpin the architectural comparison under study.

Chapter 3 describes the proposed solutions in detail, presenting the shared technical infrastructure and providing individual details on the three architectures evaluated: the Simple Agent, the Planner-Critic architecture, and the Context-Specific Multi-Agent architecture.

Chapter 4 defines the methodology used to evaluate the solution, including the set of test queries, paraphrase variants, ground-truth specification, execution procedure, logging process, metrics, classification scheme, and methodological limitations.

Chapter 5 presents and discusses the empirical results obtained by applying the proposed methodology. It compares the architectures in terms of correctness, failure patterns, paraphrase robustness, process-level behaviour, latency, token consumption, and the effect of changing the base model. The chapter concludes by explicitly answering the research questions based on the empirical findings.

Chapter 6 concludes the dissertation by summarising the main findings, highlighting the main contributions and limitations of the work, and identifying directions for future research.

Chapter 2

Literature Review

2.1 Introduction

This chapter reviews the literature relevant to the design and evaluation of reliable conversational AI agents for e-commerce. The review is structured around three complementary perspectives. First, it examines reasoning and self-correction techniques in Large Language Models, focusing on how explicit reasoning, reflection, feedback, and verification mechanisms can improve agent behaviour. Second, it analyzes the evolution of LLM-based architectures, from single-agent systems to tool-augmented, planning-centered, graph-based, and multi-agent frameworks. Finally, it discusses the role of conversational agents in e-commerce, where reliability, response quality, trust, and customer satisfaction are particularly important. Together, these areas provide the theoretical foundation for the architectural development and comparison carried out in this dissertation.

2.2 Reasoning and Self-correction techniques

The advancement of Large Language Models (LLMs) has moved from static prompting to iterative, self-corrective workflows and this path has demonstrated significant improvements in complex problem-solving tasks through explicit reasoning strategies. One of the most influential approaches is the Chain-of-Thought (CoT) prompt, which allows models to generate intermediate steps in the reasoning that leads to the final answer [19]. By decomposing multi-step problems into natural language rationales, CoT prompting improves performance across arithmetic, commonsense, and symbolic reasoning tasks, while also increasing interpretability by exposing the model's reasoning process [19]. However, despite these advantages, CoT reasoning is prone to logical inconsistencies, hallucinations [11, 16], and error propagation, especially when reasoning paths themselves are incorrect [19].

Consequently, recent research has focused on self-reflection, also known as introspection, and the self-correcting mechanisms that have emerged as a vital strategy for identifying and correcting these internal flaws in reasoning without the need for external supervision [16].

In addition to static reflection, iterative self-correction frameworks have emerged to continuously refine reasoning outcomes without the need for additional training data [13, 16, 18], which has proven to be a critical development in improving the problem-solving capabilities of LLM agents [16].

Given this, with the prospect of resolving the aforementioned limitations, LLM agents can be instructed to reflect on their own reasoning traces to identify and explain the causes of errors. The research on these metacognitive capabilities reveals that the depth of the self-correction mechanism directly correlates with the agent's ultimate performance, as it allows the agent to analyze its own reasoning traces, identify errors, and consequently generate corrective guidance, such as instructions, explanations, and step-by-step solutions, which significantly improves re-answering accuracy [16]. Research shows that while some models struggle to identify their own reasoning errors internally, even minimal feedback, such as informing an agent that its previous answer was incorrect, can trigger more diligent second attempts and significantly improve performance, suggesting that awareness of failure alone can alter reasoning behaviour [16].

More informative self-reflection strategies, including error explanations, procedural instructions, and step-by-step solutions, further enhance accuracy compared to shallow retry-based methods. These findings highlight self-reflection as a form of metacognitive capability that enables agents to refine their reasoning processes over time [16].

A closely related line of work formalizes self-correction through iterative refinement. The SELF-REFINE framework models self-correction as a cyclic process in which a single LLM alternates between generating feedback on its own output and refining that output accordingly [13]. Unlike supervised refinement methods or reinforcement learning approaches, SELF-REFINE requires no additional training data, external reward models, or auxiliary refiners. Instead, the same LLM acts as generator, critic, and refiner through carefully designed prompts [13]. Across a wide range of tasks, including mathematical reasoning, constrained generation, dialogue response generation, and code optimization, SELF-REFINE achieves substantial performance improvements, with gains of up to 40% over single-pass generation and consistent preference by human evaluators [13]. These results indicate that even state-of-the-art models such as GPT-4 have latent reasoning potential that can be unlocked through iterative self-feedback.

However, the effectiveness of SELF-REFINE has been shown to depend strongly on the quality of feedback. Actionable and specific feedback, by identifying concrete errors and suggesting targeted improvements, yields significantly better outcomes than generic feedback or unguided iteration [13]. Failure analyses further also revealed that most unsuccessful refinements stem from incorrect or misleading feedback rather than poor execution of valid feedback, highlighting the central role of accurate self-assessment in self-correcting reasoning systems [13]. Importantly, performance generally improves with multiple feedback–refinement iterations, although marginal gains diminish as iterations increase.

Beyond short-horizon iterative refinement, recent work has explored self-correction mechanisms that operate across multiple attempts and trajectories. Reflexion introduces an autonomous agent paradigm that augments reasoning-based agents with dynamic memory and self-reflection,

enabling them to learn from past failures without external supervision [18]. Unlike single-pass or locally iterative approaches, Reflexion allows agents to accumulate reflections over time and reuse them to guide future decision-making, in an attempt to mirror the natural process of human learning by trial and error [18].

Reflexion is founded on existing reasoning paradigms, such as Chain-of-Thought and ReAct, although it addresses their limitations in long-horizon tasks, where agents often suffer from repetitive actions, hallucinations, or inefficient planning sequences [18]. After each failed attempt, the agent carries out a self-reflection step triggered by a binary success signal, generating natural language feedback that summarizes errors and proposes corrective strategies. These reflections are stored in a dynamic memory module and can be consulted in subsequent trials, allowing the agent to adjust both its reasoning and action selection over time [18].

However, Reflexion also exposes important limitations. Its effectiveness depends on the quality of the agent's self-generated reflections and the availability of meaningful feedback signals from the environment. In complex e-commerce-like environments such as WebShop, improvements were limited, suggesting that reasoning self-correction alone may be insufficient when tool quality or external system constraints dominate task outcomes [18]. Nevertheless, Reflexion represents a meaningful step toward autonomous agents capable of learning from experience, bridging the gap between isolated reasoning corrections and sustained behavioural improvement across tasks.

While iterative refinement focuses on improving outputs post hoc, Inverse Prompting addresses self-correction at the level of reasoning plan validation. The InversePrompt approach, although presented in a slightly different context than the methods mentioned above, brings to light a structured selfcorrection mechanism in which an LLM generates inverse actions corresponding to its original plan and verifies whether these inverse actions can restore the system to its initial state [11]. By explicitly reversing operations, such as validating $x+y=z$ through $z-y=x$, the model evaluates the logical coherence of its reasoning steps [11]. This reverse reasoning process produces detailed and interpretable feedback by analyzing discrepancies between the original and reversed states, enabling more reliable error detection than single-step feasibility checks [11].

Empirical evaluations show that inverse prompting yields significantly higher task success rates compared to existing LLM-based task planning methods, with an average improvement of 16.3% across benchmark datasets [11]. In contrast to approaches that rely on external validators or additional models, InversePrompt embeds verification directly into the reasoning process, strengthening internal consistency and reducing reliance on ad hoc correction rules. Furthermore, inverse prompting addresses a critical limitation of naive self-correction approaches, which often fail due to shallow reasoning or lack of explicit justification for detected errors [11].

Collectively, these techniques illustrate a shift from static reasoning prompts to adaptive and self-correcting reasoning paradigms. By integrating self-reflection and iterative refinement, it is hoped that LLMs will be able to increasingly accurately identify, analyze, and progressively correct their own reasoning flaws. However, despite these advances, current methods remain limited in their treatment of long-term and multi-step reasoning and in their execution of real-world tasks,

highlighting the need for further research on scalable and reliable self-correcting reasoning mechanisms.

2.3 Frameworks and Architectures

As agentic systems increase in complexity, linear progression can be observed in LLM-based systems, evolving from linear prompt–response interactions toward agentic workflows and then into specialised orchestration frameworks that support multi-step, stateful, and collaborative behaviour. This change is driven by the growing need for more dynamic and iterative engagement between users and AI, where outcomes improve through repeated cycles rather than a single pass. In particular, agentic workflows have been described as a paradigmatic shift in the deployment of LLMs, where systems adopt iterative engagement to improve performance across tasks [6]. Within this approach, four foundational design patterns are emphasized as core architectural building blocks for increasing productivity and reliability at the system level: reflection, planning, multi-agent collaboration, and tool utilization [6].

One influential and important step in this evolution is ReAct, a prompt-based paradigm that explicitly interleaves reasoning traces with actions. Instead of treating "thinking" as a hidden, static chain-of-thought process, ReAct makes reasoning and acting part of the same trajectory, enabling an LLM to maintain and adjust high-level plans ("reason to act") while also interacting with external environments to obtain information that feeds back into reasoning ("act to reason") [21]. This structure improves interpretability, reliability, and diagnosability because it allows humans to inspect reasoning traces and distinguish the model's internal knowledge from externally retrieved information [21]. ReAct is also presented as highly flexible and effective with very small numbers of in-context examples, consistently outperforming "reasoning-only" or "acting-only" baselines across multiple task types [21]. At the same time, the method highlights an architectural trade-off: ReAct's reasoning relies heavily on retrieving informative external knowledge, and therefore, when search is non-informative, reasoning can get derailed and recovery becomes difficult, motivating hybrid strategies such as combining ReAct with CoT-SC via fallback rules [21].

2.3.1 From single-agent systems to reusable development frameworks

Initial agentic systems treated LLMs as monolithic reasoning engines, enhanced primarily through prompting strategies such as Chain-of-Thought prompting [19]. While this approach improved reasoning transparency and performance, it lacked explicit mechanisms for planning, execution control, or interaction with external environments. As a result, these systems struggled with long-horizon tasks and exhibited brittle behaviour when faced with uncertainty or incomplete information [4].

As LLM capabilities and systems expanded beyond single prompt templates, the ecosystem shifted toward developer-focus orchestration libraries that package these interaction patterns into reusable building blocks, designed to make LLM applications easier to build, integrate, and scale.

Within this direction, LangChain is described as an open-source software library providing "all-inclusive solutions" for multiple stages of LLM application development, with the explicit goal of simplifying integration with external applications and data sources [4]. Architecturally, LangChain emphasizes modularity and reusability through "components" (modular abstractions) and "chains" (custom pipelines built for use-case specific data), while offering prompt-template tools to support repeatable prompting workflows [4]. This marks an important step in the evolution from ad-hoc prompt engineering to structured application assembly, where workflows become composable rather than manually scripted [4].

The same source situates these kinds of frameworks alongside other single-agent systems, including AutoGPT, BabyAGI, and Llama-index, characterized as examples of systems leveraging LLMs for tool usage, function calling, and embodied activities [4]. This stage reflects a transition where LLMs are no longer only text generators, but begin operating as agents that can execute sequences of actions with the ultimate goal of achieving an objective, often through interfacing with external tools and resources [4]. This new dynamic has opened the door for them to now be applied in various fields and contexts, thus increasing their usefulness and impact [4].

2.3.2 Multi-agent conversation frameworks and collaborative task execution

Recent research has shown that scaling the capabilities of LLM-based systems does not necessarily require increasingly complex single-agent designs, but can instead be achieved through the collaboration of multiple agents [20]. As systems move from single-agent pipelines into multi-agent cooperation, frameworks that explicitly formalize agent-to-agent interaction take center stage and become increasingly important. This reflects a broader architectural progression toward systems composed of specialized agents coordinating through structured interaction patterns, rather than a single monolithic model that manages the entire workflow [4, 20].

In the reviewed material, multi-agent frameworks are described as settings where multiple LLMs take on different roles in order to enable cooperative problem-solving and communication in natural language, supporting implementation in different domains [4].

Motivated by this trend, AutoGen is specifically referred to as being intended to enable next-generation LLM applications [20]. It introduces a generalised multi-agent conversation framework that facilitates the construction of complex LLM applications through structured interactions between multiple autonomous agents [20].

Rather than embedding all reasoning, tool usage, and validation within a single monolithic agent, AutoGen decomposes tasks across specialized agents that communicate via natural language conversations[20].

AutoGen agents are designed to be customizable, conversable, and composable, allowing them to be backed by LLMs, tools, human input, or combinations thereof [20]. Each agent maintains its own internal conversational state and can be configured with specific roles, such as planning, execution, validation, or human mediation. This design enables developers to flexibly construct agent ecosystems where reasoning, criticism, and execution emerge through dialogue rather than rigid pipelines [20].

A key architectural contribution of AutoGen is its conversation programming paradigm, which reframes complex workflows as sequences of inter-agent message exchanges governed by conversation-driven control flow [20]. In this paradigm, both computation and coordination are expressed through conversational interactions, allowing developers to define agent behaviours and collaboration patterns using a mixture of natural language and programmatic constructs. This abstraction significantly simplifies the implementation of sophisticated multi-agent behaviours, such as debate, verification, iterative refinement, and human-in-the-loop oversight [20].

Empirical evaluations demonstrate that AutoGen-based systems are effective across a wide range of domains [20]. By facilitating cooperation among agents with complementary capabilities, AutoGen is aligned with previous findings that multi-agent settings can improve factuality, encourage divergent reasoning, and provide internal validation mechanisms. As such, AutoGen represents a significant step toward scalable, modular, and extensible agent architectures, offering a flexible foundation upon which reasoning, self-correction, and tool-augmented behaviours can be systematically integrated.

2.3.3 Graph-based orchestration and stateful multi-agent workflows

A further architectural evolution arises from the need to support multi-actor applications and cyclical operations, especially when tasks are complex and require orchestration across multiple subtasks. In this context, LangGraph is presented as an open-source framework for building stateful LLM applications with multiple actors and cyclic operations [4]. It supports single-agent and multi-agent configurations, allowing developers to decompose complex tasks into modular subcomponents while maintaining shared state across interactions [1]. For this reason, it is often compared in contrast to the frequent use of Retrieval-Augmented Generation (RAG) for single-agent systems, noting that LangGraph can also facilitate multi-agent applications [4].

This entire framework highlights a fundamental evolution: as systems require persistent state and repeated interaction loops, orchestration benefits from graph structures where nodes represent sub-functions and edges represent transitions and dependencies [4]. Unlike sequential chains, graph-based designs allow for cyclical operations, conditional transitions, and parallel execution, significantly enhancing expressiveness and control [1], thereby improving transparency, debuggability, and scalability, particularly in applications that require persistent context and iterative refinement [3, 4].

This graph-based paradigm is concisely illustrated in a multi-agent course syllabus generation system that uses LangGraph to orchestrate collaborative workflows among LLM agents. The workflow is decomposed into explicit stages, including persona analysis, human feedback reception, curriculum retrieval, syllabus creation, and finally expert review and refinement. Each stage is modeled as a modular node in the LangGraph structure [3]. The same study highlights LangGraph's architectural suitability due to its support for statefulness, which allows conversational memory management before each LLM call and thus supports sequential workflows where outputs from one stage are incorporated into prompts for later stages [21]. Furthermore, LangGraph is

described as supporting parallel workflows, and tooling such as LangGraph Studio is used for debugging and visualization, reinforcing the importance of developer-facing observability as workflows grow in complexity [1, 10].

2.3.4 Tool-Augmented and Planning-Centric Architectures

As agentic systems increasingly interact with external environments, the architecture of tool-augmented agents advances further with specialized frameworks that treat planning as a first-class object and embed self-correction directly into the tool-planning lifecycle, emerging as a critical design pattern.

With this in mind, while ReAct primarily formalises how an LLM should act in a closed loop, other work advances towards teaching models to use tools more systematically.

Toolformer, which represents a significant step toward architectural integration of external tools within large language models, incorporates the use of the tool directly into the decision-making process at the model token level [17].

In contrast to prompt-based or task-specific tool invocation methods, ToolFormer enables LLMs to autonomously decide when, how, and which tools to call, using a self-supervised learning objective that assesses whether a tool call reduces prediction loss for future tokens [17]. This design allows the model to overcome some of the intrinsic limitations of language-only reasoning, such as outdated knowledge, numerical inaccuracy, and factual hallucinations, without sacrificing generality or requiring heavy human supervision or task-specific few-shot prompting that pre-specify which tools are needed [17].

Architecturally, ToolFormer introduces a tight coupling between reasoning and execution, where API calls are interspersed directly into the generated text and treated as first-class tokens during fine-tuning. The approach fine-tunes on many sampled API calls that are filtered based on whether they reduce perplexity on future tokens, framing the use of the tool as something the model learns to request when beneficial [17]. By filtering tool calls based on their contribution to language modeling performance, the framework ensures that only functionally useful interactions are retained, resulting in improved zero-shot performance across diverse downstream tasks [17].

Nonetheless, Toolformer's design also exposes important architectural limitations, as it cannot reliably perform tool chaining (using the output of one tool as input to another) since API calls are generated independently, nor does it support interactive use (e.g., iteratively refining search queries and browsing results), which can be crucial for search-type tools. Additionally, it does not take into account the tool-dependent computational cost of calling APIs [17]. These constraints highlight an important trade-off between architectural simplicity and expressive control, motivating subsequent frameworks that incorporate verification and self-correction loops.

Taking this approach even further, ToolFiVe is described as a general, plug-and-play framework for leveraging specialized tools in compositional reasoning tasks, specifically targeting the weaknesses of previous reasoning plan generation approaches [12]. It introduces a structured architecture that explicitly separates reasoning-plan generation, tool filtering, and plan verification [12]. The process is redesigned by integrating tool execution (parsing the plan and invoking tools)

and adding two key modules: a filtering module that removes tools irrelevant to the task in order to form a restricted set of candidate tools, and a verification module that evaluates plan completeness and produces feedback, triggering an iterative self-correction loop when verification fails [12]. Prior methods are summarized as relying mainly on either in-context learning (ICL), which may struggle with accurate tool usage and introduce tool bias, or supervised fine-tuning (SFT), which may fail to adapt quickly to new tasks and toolsets due to the need for ground-truth paired data [12]. By incorporating verification-driven feedback loops within the architectural design, ToolFiVe enables agents to iteratively refine reasoning plans before execution, improving reliability in compositional reasoning tasks. This highlights a broader architectural trend that incorporates verification and validation mechanisms directly into agent's workflows rather than treating them as post-hoc processes, emphasizing the importance of reasoning-plan generation for tool-augmented LLMs [12].

Given all these advances, it is possible to observe that the structures researched reveal a consistent shift from linear, prompt-based LLM usage to modular, stateful, agent-oriented architectures that emphasise orchestration, planning, and verification in a logically chained way. By incorporating reasoning, interaction with tools, and self-correction directly into system workflows, these architectures and frameworks aim to address the main limitations of single-pass generation and support more reliable behaviour in complex tasks. Together, these perspectives provide a necessary foundation for understanding how advanced conversational systems can be designed to operate robustly in applied domains.

2.4 E-commerce

E-commerce has evolved significantly since its inception. Initially conceived as a channel for buying and selling products online, it has ultimately transformed into a complex digital ecosystem involving retailers, e-commerce platforms, payment processors, logistics service providers, and customer service infrastructure [8]. With advances in Artificial Intelligence, this ecosystem has increasingly incorporated data-driven automation, recommendation engines, customer communication tools, and operational optimisation processes [8]. Consequently, Artificial Intelligence is no longer a peripheral element in the e-commerce sphere; rather, it is increasingly integrated into how online businesses interact with customers, manage operations, and compete in digital markets.

As part of this transformation, customer communication has become a particularly important area. E-commerce platforms are not just sales channels, but also service environments where customer satisfaction and retention depend on responsiveness, interactivity, the quality of communication, perceived usability, and trust [15]. This helps explain the growing adoption of AI-powered chatbots and virtual assistants in online commerce. These systems are often used to provide ongoing support, answer customer questions, resolve routine issues, assist with product discovery, and contribute to digital customer relationship management [7, 15]. More importantly, the quality of their responses has been linked to customer satisfaction and purchasing behaviour, meaning that conversational performance can have direct implications for the business [7].

For this reason, the use of AI-based conversational systems in e-commerce also raises significant concerns regarding reliability. In customer service, users expect responses that are not only fluent but also accurate, timely, context-sensitive, and grounded in the real business environment. Although chatbots can be effective and cost-efficient for simple or repetitive tasks, they may prove insufficient when requests become more complex, ambiguous, urgent, or dependent on specific information about products, orders, or policies [22]. These limitations are associated with challenges such as linguistic nuances, accuracy, and limited knowledge bases, which are particularly relevant in e-commerce environments where users demand correct, actionable, and context-specific responses [22]. Incorrect or incomplete responses can, therefore, affect user trust, purchasing decisions, and the perceived quality of service [2, 7, 22].

All of this makes it clear that the implementation of chatbots in e-commerce should not be viewed as a binary choice between automation and human support. Previous studies using agent-based modeling and simulation (ABMS) frame online customer service as a complex system in which strategies involving only artificial intelligence, only humans, or collaboration between humans and AI may be more or less appropriate, depending on the volume of requests, the complexity of tasks, urgency, and subjectivity [22]. This methodological perspective is relevant because it treats the automation of customer service as a problem of strategy selection, rather than a simple replacement of human agents. Furthermore, it also reinforces the importance of designing AI agents capable of operating reliably within their scope of action, while recognizing that more complex scenarios may require more robust control mechanisms, better coordination, or escalation to a human agent.

Implementation-oriented studies also show that e-commerce agents are increasingly expected to combine multiple capabilities, rather than functioning merely as systems that answer questions. For example, recent studies describe AI-based shopping agents that integrate natural language processing, recommendation mechanisms, real-time session management, and platform-compatible architectures to provide personalized and agile interactions on a large scale [2]. At the same time, research on multi-agent recommendation systems argues that industrial e-commerce systems are evolving beyond static, one-off recommendation flows toward interactive, context-aware systems capable of maintaining memory, using tools, and orchestrating multi-step workflows [14]. These developments underscore the relevance of this dissertation, as they demonstrate that the reliability of conversational agents in e-commerce depends not only on generating responses but also on the robust use of tools, workflow control, and consistent behaviour in response to complex user requests.

Consequently, reliability becomes a central requirement for conversational Artificial Intelligence in e-commerce, linking technical accuracy to trust, customer experience, and business value. In this context, the study of agent architectures is particularly relevant, since architectural choices influence how user intent is interpreted, how tools are selected, how external information is used, and how multi-step tasks are completed. This link between conversational reliability, tool usage, and service quality in e-commerce motivates the architectural comparison developed in this dissertation.

2.5 Summary and Conclusion

The presented literature review has examined the state of the art in conversational AI systems for e-commerce, with a particular focus on reasoning, self-correction, and architectural design choices that influence reliability.

The evolution of e-commerce from a transactional channel into a complex, AI-driven ecosystem has elevated the role of conversational agents from peripheral support tools to central drivers of business performance and customer retention.

Across the reviewed works, a coherent narrative transpires: as conversational agents move from simple, scripted interactions toward autonomous, multi-step, and tool-augmented behaviours, reliability becomes a central and unresolved challenge rather than a secondary quality attribute.

In terms of reasoning techniques, the evolution from single-pass generation toward explicit reasoning strategies such as Chain-of-Thought, and further toward self-reflection and iterative self-correction mechanisms, has shown clear performance gains. Frameworks such as SELF-REFINE, Reflexion, and inverse prompting demonstrate that Large Language Models possess latent reasoning capacity that can be unlocked through structured feedback, reflection, and verification loops. Nevertheless, the same literature also highlights important limitations: self-correction is highly sensitive to feedback quality, error detection remains imperfect, and gains diminish in long-horizon or real-world tasks where external constraints dominate. Taken together, these findings suggest that reliable reasoning cannot be achieved through prompting strategies alone, but must be supported by system-level design choices.

From an architectural perspective, the current landscape illustrates a decisive shift from monolithic prompt-response systems toward modular, stateful, and agent-oriented frameworks. Paradigms such as ReAct highlight the benefits of interleaving reasoning with action, while orchestration frameworks like LangChain, AutoGen, and LangGraph provide abstractions for planning, memory, tooling, and multi-agent collaboration. These architectures enable explicit control flows, persistent state, and iterative refinement, which prove essential for fostering transparency, debuggability, and robustness in complex applications. However, while serving as one of the foundations for this evolution, they also introduce new points of fragility, including coordination overhead, dependence on external tools, and increased system complexity, reinforcing the equally growing need for reliability mechanisms based on principles embedded directly in agent workflows.

Finally, although already briefly mentioned, an important point in the perceived trend is the integrated use of tools, where structures such as Toolformer and ToolFiVe illustrate a paradigm shift in which the interaction of these tools is incorporated directly into the reasoning lifecycle. By integrating them into verification-driven loops, these systems can more concisely refine reasoning plans prior to execution, aiming to mitigate factual inaccuracies that often affect language-only models.

Within the e-commerce domain, the literature reviewed clearly shows that conversational agents are no longer peripheral support tools, but integral and relevant components in terms of customer experience, electronic customer relationship management, and even direct commercial

interaction. Empirical and simulation-based studies have consistently shown that customer satisfaction, trust, and purchasing behaviour are strongly influenced by the accuracy, timeliness, and contextual relevance of chatbot responses. Although AI-based chatbots offer clear advantages in terms of scalability and cost efficiency, they still struggle in complex, urgent, or ambiguous scenarios, where errors directly translate into a loss of trust and, consequently, reduced adoption. Given this, hybrid strategies between humans and AI and agentive designs are still presented not as transitional solutions, but as valid, reliable, and necessary responses to the limitations inherent in fully automated systems.

Taken together, the literature reviewed highlights a clear gap between advances in reasoning and agent architectures on the one hand, and the stringent reliability requirements of real-world e-commerce applications on the other. Although current self-correcting reasoning techniques and agent frameworks provide promising building blocks for addressing the problems they set out to solve, they are not yet sufficient to guarantee reliable behaviour on a real-world scale. This gap motivates the central focus of future studies: the design and implementation of a framework that systematically integrates reasoning, self-correction, and architectural safeguards to increase the reliability of conversational AI agents in e-commerce contexts, capable of meeting both the needs of businesses and the expectations of users. By basing the system design on recent advances in LLM-based reasoning and the practical constraints of commercial environments, this work aims to contribute to more reliable, transparent, and effective AI-based customer service systems.

Chapter 3

Proposed Solutions

3.1 Introduction and Design Philosophy

This chapter presents three distinct agent architectures proposed to address the reliability challenges identified in the research problem. Each represents a progressive improvement in the design of AI-based customer service agents for e-commerce, driven by the limitations observed in its predecessor. The three architectures, named Simple Agent Architecture, Planner-Critic Agent Architecture and Context-Specific Agent Architecture, share a common operational domain and technical infrastructure, enabling a controlled comparison within a uniform evaluation scenario.

The main challenge driving these solutions is the inherent unpredictability of LLM behaviour when interacting with external tools in complex, multi-step service scenarios. Previous work on LLM systems supplemented by tools and exhibiting autonomous behaviour demonstrates that the use of tools can expand the model's capabilities, but also raises reliability concerns related to factual inaccuracies, dependence on external tools, and increased system complexity [12, 14, 17]. In the context of this dissertation, these concerns manifest as failure modes, such as incorrect tool selection, hallucinated or unsupported product references, incomplete query parameterization, and premature or delayed task completion. Each proposed architecture addresses a subset of these failure modes through different control mechanisms, progressively advancing toward a more reliable system

The three architectures operate within a small-scale yet structurally realistic e-commerce environment. The system has been designed to handle a variety of customer service interactions, including product discovery, stock enquiries, shopping basket management, organisational enquiries and post-purchase operations. The diversity of task types ensures that the architectures are evaluated against a wide range of user intentions, capturing single-step or multi-step interaction patterns representative of the real-world demands of customer service in such environments

3.2 Shared Technical Infrastructure

Before describing the individual architectures, it is necessary to establish the technical foundation common to all three implementations. This shared infrastructure ensures that observed behavioural differences between architectures are attributable to architectural decisions rather than to variations in the underlying models, data representation, or tooling.

3.2.1 System Stack

The system's backend is implemented in Python. The three architectures are evaluated using a dedicated script, `evaluate_agent.py`, which runs each architecture directly from the terminal environment. The script controls the evaluation flow and selects the active architecture via the `TEST_MODE` constant, mapping each configuration to its corresponding entry point. This configuration allows the three architectures to be evaluated based on the same input queries, under comparable execution conditions, without modifying the agent's implementation between runs.

Language model capabilities are provided by the OpenAI API. Each architecture was evaluated under two model configurations: `gpt-4o-mini` and `gpt-3.5-turbo`. Both models were applied identically across all three architectures, enabling a controlled comparison of architectural contribution independent of model capability. All agent components are implemented using LangChain, with the Planner-Critic and Context-Specific architectures additionally employing LangGraph for stateful, graph-based execution control.

3.2.2 Knowledge Bases and Semantic Retrieval

The product knowledge base is stored as a vector collection in Qdrant, a high-performance vector database deployed locally via Docker. The collection, named products, contains approximately thirty mock products spanning nine categories: Laptops, Smartphones, Tablets, Audio, Desktops, Media, Wearables, Accessories, and Monitors. Representative products include the MacBook Air M3, the Dell XPS 13, the iPhone 15, the Samsung Galaxy Tab S9, and the Dell UltraSharp 27" Monitor. This categorical and brand diversity was deliberately designed to exercise semantic disambiguation, brand-specific resolution, and edge cases in retrieval precision.

Each product is indexed as an embedding vector of dimension 1536, corresponding to the output dimensionality of the `text-embedding-3-small` model. It is essential to note that the embedding is deliberately not generated from a simple concatenation of product fields, but rather from a structured textual representation specifically designed to align with the query format used at the time of retrieval:

Listing 3.1: Structured textual representation used for product embedding generation

```
1 def build_embedding_text(p: dict) -> str:
2     return (
3         f"Name: {p['name']}\n"
```

```
        f"Specs: {p['specs']}\n"
        f"Description: {p['description']}"
        f"Category: {p['category']}\n"
    )
```

This structured embedding approach ensures that the semantic space captures product identity, category, technical characteristics, and usage context simultaneously. The use of explicit field labels like Name, Category, Specs and Description establishes a consistent textual schema between indexed documents and runtime queries, with a view to improving retrieval alignment. Numeric attributes, specifically price and stock level, are stored as Qdrant payload fields and accessed through structured payload filters rather than embedded into the vector representation, preserving the semantic signal of the embedding from numeric noise.

A second Qdrant collection, `info_docs`, stores institutional documents, including return policies, shipping conditions, privacy policies, and customer support information, as chunked text segments. These documents were divided into smaller parts prior to indexing, with the aim of improving the alignment between users' short queries and the retrieved text segments. Each chunk is stored with metadata fields for document title, content, and source filename. A third collection, `campaign_docs`, stores promotional content indexed for semantic retrieval using an equivalent structure.

Cosine distance is used as the similarity metric across all collections. This choice aligns with the goal of comparing embedding vectors based on semantic proximity, while the minimum similarity threshold applied after retrieval filters out low-confidence results. The specific value of the threshold was empirically calibrated during development.

3.2.3 Tool Catalogue

The three architectures make use of a common set of tools, although the subset made available to each component of the agent varies depending on the architectural design. Each tool is implemented as a LangChain StructuredTool, defined through the `@tool` decorator, which converts a Python function into a model-callable tool using its function signature and docstring to define the tool schema and description [9].

The shared tool catalogue comprises the following components:

- `filter_products`: Performs hybrid retrieval over the `products` collection. It accepts structured parameters for price range, category, keywords, sort order, and result limit. The semantic query is dynamically constructed to replicate the structured format of indexed product documents, improving recall alignment between query intent and retrieval results.
- `get_product_by_name`: Performs name-specific semantic lookup and returns a structured result comprising a `found` boolean, a confidence score, the matched product if any, and a list of candidate alternatives. This structured output was introduced to prevent the agent from interpreting a semantically proximal but incorrect product as a valid match for a specific user request.

- `check_product_stock`: Accepts a product identifier and returns the current stock level and availability status. It relies on an internal helper function, `get_product_by_id`, which is deliberately not exposed to the LLM to prevent direct invocation with fabricated identifiers.
- `create_cart`: Creates an in-memory shopping cart from a list of product identifiers and returns a cart summary including the products, subtotal, applicable taxes, and total.
- `get_information`: Performs semantic retrieval over the `info_docs` collection, returning the most relevant document chunks with their title, content, and source.
- `available_campaigns`: Retrieves semantically relevant promotional offers from the `campaign_docs` collection.
- `track_order`, `request_refund`, `reissue_receipt`: After-sales tools operating against a mock order database containing representative order records with shipping status, itemised products, and tax calculations.

It is important to note that `get_product_by_id`, used internally by stock and cart tools, is explicitly excluded from the LLM-accessible tool set. This design decision was intended to prevent the model from attempting to call it directly with arbitrary or fabricated identifiers, a failure mode observed in early iterations, as product identifiers are internal database keys unknown to the end user.

3.3 Simple Agent Architecture

3.3.1 Motivation and Design Rationale

The Simple Agent Architecture is presented as the baseline solution against which subsequent architectures are evaluated. Its design follows a common pattern of an agent that invokes tools, inspired by current LLM application frameworks: a single LLM instance has access to the complete catalogue of tools and operates through an iterative cycle of reasoning and action, invoking them as needed until it produces a final response [4, 21].

This architecture represents the most direct alignment between a conversation-based customer service task and a solution powered by a LLM. Its primary design objectives are operational completeness and simplicity of implementation. The agent must be capable of dealing with the full range of customer service requests without domain partitioning, pre-execution validation, or explicit completion judgement. As such, it provides a clear baseline which, by contrast, isolates the contribution of each subsequent architectural mechanism.

3.3.2 Architectural Overview

The Simple Agent operates based on a stateful message-passing cycle, in which a single LLM instance, initialized with a comprehensive system prompt, repeatedly processes the accumulated

message history until no further tool invocations are required. As illustrated in Figure 3.1, the architecture follows a straightforward cycle in which the Simple Agent receives the user's query, accesses the complete tool catalogue, executes tools when necessary, incorporates the returned results into the message history, and finally produces the final response.

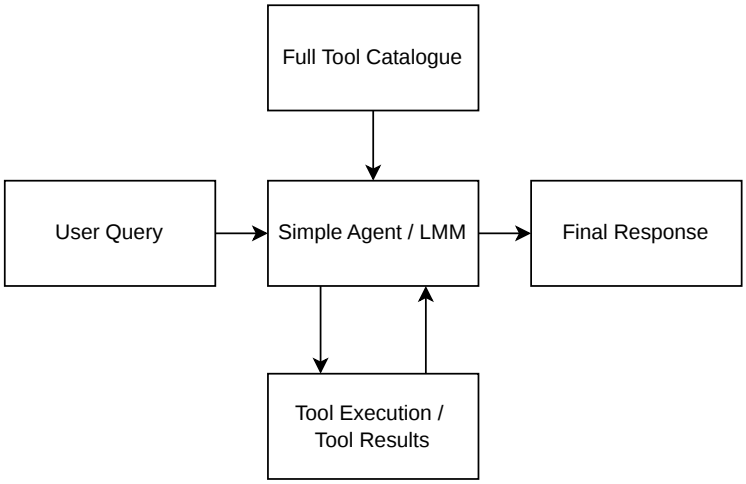


Figure 3.1: Simple Agent architecture: sequential tool invocation loop.

The execution cycle proceeds as follows:

1. The user's input is appended to the message history as a `HumanMessage`.
2. The LLM, bound to the full tool catalogue via `llm.bind_tools(tools)`, processes the complete message history.
3. If the LLM produces no tool calls, the response is interpreted as final and returned.
4. If the LLM proposes tool calls, each proposed tool is executed, and the result is appended to the message history as a `ToolMessage` associated with the corresponding tool call identifier.
5. Control returns to step 2, repeating until a terminal response is produced or the safety loop limit is reached.

The message history is initialised fresh for each user query, thereby preventing cross-query contextual contamination during batch evaluation. This is an explicit methodological decision, whereby each evaluation instance begins solely with the system prompt, ensuring statistical independence between test instances.

3.3.3 State Representation

The agent maintains state through LangChain's message list abstraction. The conversation history comprises four message types: `SystemMessage` (the agent's operational instructions, injected

once at initialisation), `HumanMessage` (the user's request), `AIMessage` (the LLM's response, potentially including tool call specifications), and `ToolMessage` (the structured result of a tool execution, linked to a specific call identifier)

3.3.4 Tool Binding and Sequential Invocation

Tools are made available to the LLM through LangChain's `bind_tools` mechanism, which serialises tool schemas derived from function signatures and docstrings into the model's input context. The LLM selects tools based on the schema descriptions and the semantic content of the conversation.

When the model generates multiple tool calls in a single response iteration, the implementation executes them sequentially within the same execution cycle and appends their results to the message history before invoking the model again. Consequently, in this architecture, the model does not re-evaluate each individual tool result before the remaining tool calls in the same iteration are executed. This lack of re-evaluation can lead to complications in multi-step tasks, since subsequent actions may be executed before the model has explicitly interpreted the outputs of previous calls. The absence of a limit on tool invocations per turn is one of the design limitations addressed in the Planner-Critic architecture.

3.3.5 Output

The agent's `run_agent` function returns the final LLM response as a plain string. No structured wrapping is applied. Then, the evaluation script receives and logs the text response directly.

3.3.6 Limitations

The Simple Agent Architecture, while operationally complete, presents several reliability limitations that directly motivate the design of the subsequent architectures.

The absence of pre-execution validation highlights the possibility that the LLM might select incorrect tools or construct tool arguments that omit semantically critical constraints, without any corrective intervention. For example, when asked to display mobile phones below a certain price threshold, the agent may call `filter_products` using only a price restriction, completely omitting the category filter that would otherwise narrow down the search results.

Furthermore, as tool selection errors are not intercepted prior to execution, their consequences will tend to propagate into the message history as a potentially misleading context for subsequent reasoning steps.

The agent has no explicit mechanism for determining task completion in multi-step scenarios. The cycle only ends when the model produces a response without tool calls, which may occur prematurely or be unnecessarily extended.

Finally, the unrestricted availability of all tools to a single agent may increase the risk of cross-domain tool selection in requests with multiple intents. A request involving, for example, clarification of shipping details, may lead to incorrect selection between general information tools

or campaign search tools (a scenario observed during the solution's development process, resulting in incomplete or incorrect responses).

3.4 Planner-Critic Agents Architecture

3.4.1 Motivation and Design Rationale

The Planner-Critic Architecture approaches the main reliability flaw of the Simple Agent: the absence of pre-execution plan validation. The central design insight is that LLM-proposed tool invocations can be evaluated for semantic and structural adequacy before execution, allowing incorrect or suboptimal plans to be rejected and revised in advance, without thereby consuming tool execution resources or introducing misleading intermediate results into the conversational context.

This architecture introduces a critic module, a separate LLM instance operating without tool access, whose sole purpose is to evaluate the tool plan proposed by the planner before its execution. This is based on the broader literature on self-reflection and iterative self-correction in LLM-based systems, in which the model's results are evaluated and refined through feedback mechanisms [13, 16]. It is also aligned with verification-oriented tool-using architectures, in which reasoning plans are evaluated before or during tool execution in order to improve reliability [12]. The separation of planning and evaluation into distinct components reflects a division of responsibilities designed to reduce error propagation in reactive tool use.

In terms of implementation, this is treated as an explicit directed state graph using LangGraph, allowing for precise control over transitions between nodes, state management, and termination conditions, features that Simple Agent's manual loop cannot provide.

3.4.2 Graph Structure and State

The Planner-Critic Architecture is represented as a directed graph with four main processing nodes: planner, critic, review, and tools, connected via conditional routing functions. As illustrated in Figure 3.2, the planner first proposes a tool call, which is then evaluated by the critic before execution. If the proposed action is approved, the tool is executed and its result is returned to the planner's context and if it is rejected, review feedback is sent back to the planner, initiating a new planning cycle.

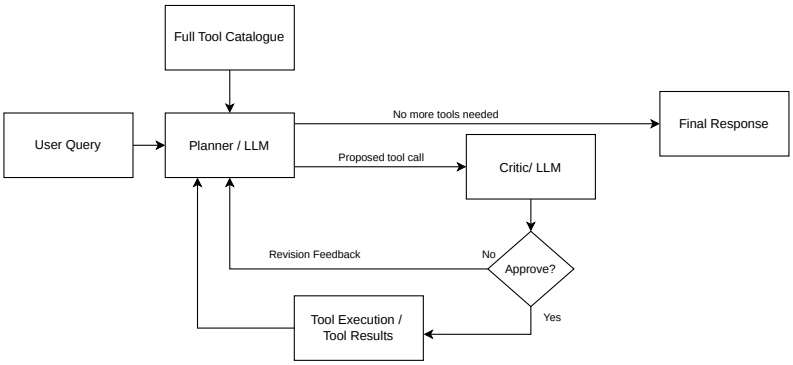


Figure 3.2: Planner-Critic Architecture: pre-execution tool-plan validation with conditional routing and revision feedback.

The shared state is encoded as a `TypedDict` named `AgentState`:

Listing 3.2: State structure used by the Planner-Critic agent label

```
class AgentState(TypedDict):
    messages: List[BaseMessage]
    user_input: str
    executed_tools: list
    tool_calls: list
    pending_assistant_content: str
    critic_decision: Literal["APPROVE", "REVISE", "NONE"]
    critic_feedback: str
    loops: int
    max_loops: int
```

Key fields include `executed_tools`, which maintains a record of all tools invoked during the current query resolution and is passed to the critic at every evaluation cycle, `tool_calls`, which holds the planner's current tool proposal pending critic assessment, and `critic_decision`, which communicates the critic's verdict to the graph's edge routing functions. The loops and `max_loops` fields implement a safety limit that prevents entry into infinite correction loops, thereby avoiding the risk of unlimited recursive execution.

3.4.3 Planner Node

Here, the tool call is a proposal, not a commitment, therefore, the Planner node invokes the LLM, along with the full catalogue of associated tools, and when the model produces a tool call, this is stored in `state["tool_calls"]` without being executed immediately.

A single-invocation policy is introduced here: the planner is limited to proposing a maximum of one tool call per turn. When the model produces multiple tool calls simultaneously, all calls except the first are discarded. This restriction was designed to ensure that each evaluation by the

critic refers precisely to one proposed tool call at a time and that each tool result is incorporated into the conversation history before the next action is decided.

If the model produces a text-only response, i.e. one that does not contain any tool calls, this is interpreted as a final response and the graph transits to the final state.

3.4.4 Critic Node

The Critic Node constitutes the core reliability mechanism of this architecture. It invokes a dedicated LLM instance, operating without any tool bindings, to assess the proposed tool plan through the function `reflect_on_plan`, receiving the original user request, the current message history, the proposed tool call, and the list of already-executed tools:

Listing 3.3: Critic invocation using the current graph state

```
reflect_on_plan(
    state["user_input"],
    state["messages"],
    state["tool_calls"],
    state["executed_tools"]
)
```

The deliberate exclusion of tool bindings from the critic is a design decision driven by several considerations: it prevents the critic from executing actions, strictly contains its role to evaluative reasoning, eliminates the risk of nested tool invocation, and simplifies the analytical isolation of the critic's contribution during evaluation. Although the critic does not have access to the tools, it receives a natural-language description of the available tools, briefly outlining what they are designed to do and their attributes, which is sufficient for evaluating the plan without access to executable code.

A crucial improvement to the critic's evaluation framework concerns the scope of the evaluation. The critic assesses whether the proposed tool call represents the best next step towards fulfilling the user's request, taking into account the tool's previous executions, rather than whether the proposed call, on its own, resolves the task in its entirety. This reformulation was necessary to resolve a systematic failure mode in earlier iterations, in which the critic incorrectly rejected valid incremental steps on the basis that they did not, individually, solve the task in its entirety. Few-shot examples incorporated into the critic's prompt were a key element in correcting this type of problem, explicitly illustrating this distinction, covering cases such as:

- A price-filtered product query without a category constraint — `REVISE`, because the semantic constraint is expressible and omitted.
- A product name lookup as a first step before a stock check — `APPROVE`, because it is a valid prerequisite step.

- A category-filtered retrieval for the first item in a multi-item cart request — `APPROVE`, because partial resolution of a multi-step task is a valid incremental action.

The critic returns a structured JSON verdict:

Listing 3.4: Structured JSON verdict returned by the critic

```
{
  "decision": "APPROVE" | "REVISE",
  "feedback": "Short explanation..."
}
```

3.4.5 Tool Calling Protocol Compliance

A technically significant design constraint concerns the tool calling protocol enforced by the OpenAI API. The protocol requires that any `AIMessage` containing `tool_calls` in the message history be immediately followed by `ToolMessage` responses for each tool call identifier. Inserting an `AIMessage` with `tool_calls` without the corresponding `ToolMessage` responses produces a protocol violation error.

In the Planner-Critic Architecture, when the critic issues a `REVISE` verdict, the proposed tool calls must not be committed to the message history as an `AIMessage`, since the corresponding tools will not be executed and no `ToolMessage` will follow. Instead, rejected plans are treated as internal deliberative hypotheses, and only the critic's feedback is injected into the message history as a `HumanMessage`, maintaining protocol compliance while communicating the evaluative rationale to the planner:

Listing 3.5: Revision feedback injection while preserving tool-calling protocol compliance

```
if review["decision"] == "REVISE":
    messages.append(
        HumanMessage( content=(
            "Your previous tool plan was rejected by an internal reviewer.\n\n"
            f"Feedback: {review['feedback']}\n\n"
            "Propose a new tool plan that better matches the user's request."
        ))
    )
```

3.4.6 Revision Node

When the critic issues a `REVISE` verdict, control passes to the Revision Node, which appends the critic's structured feedback to the message history and returns control to the Planner Node. The

planner then generates a revised tool proposal that takes the feedback into account, initiating a new planning-critique cycle.

3.4.7 Tool Execution Node

Upon critic approval, the Tools Node executes the validated tool call and appends both the `ToolMessage` to the message history and a structured record to `executed_tools`:

Listing 3.6: Structured record of executed tool calls in the Tools Node

```
state["executed_tools"].append({
    "tool_name": tool_name,
    "tool_arguments": tool_args,
    "result": result
})
```

The `executed_tools` list is initialised empty at the beginning of each query and discarded upon completion, functioning as an episodic working memory scoped to the current interaction. This scoping decision ensures that tool execution history from one query does not influence the critic's evaluation of another during batch evaluation, preserving the statistical independence of test cases.

3.4.8 Graph Routing

State transitions are governed by edge functions that inspect specific state fields:

Listing 3.7: Conditional routing functions in the Planner-Critic graph

```
def route_from_planner(state: AgentState) -> str:
    if state["tool_calls"]:
        return "critic"
    return "end"

def route_from_critic(state: AgentState) -> str:
    if state["critic_decision"] == "REVISE":
        return "revision"
    return "tools"
```

3.4.9 Limitations

The Planner-Critic architecture represents an improvement in terms of action control compared to the Simple Agent by introducing structured pre-execution validation, but it retains several unresolved reliability limitations.

Most fundamentally, the critic evaluates the structural and semantic adequacy of tool plans but cannot assess the quality of tool outputs. A plan may be correctly structured, semantically coherent, and duly approved by the critic, yet the tool's execution may return semantically noisy, miscategorised, or incomplete results. In such cases, the evaluator's approval offers no protection against incorrect final responses. This distinction between the correctness of the plan and the correctness of the results represents a key limitation of the scope of pre-execution validation mechanisms.

One particularly sensitive issue that emerged throughout the development of this solution was the fact that the evaluator's reliability is inherently dependent on the prompt. Its behaviour is a function of the quality and comprehensiveness of its instructions and the few examples provided. Subtle changes to the critic's prompt can shift its decision threshold from overly permissive to overly restrictive, with corresponding effects on task completion rates, iteration counts and token consumption.

The architecture also does not tackle the multi-domain tool confusion problem of the Simple Agent. All tools remain available to the single planner instance, without domain-based partitioning, and the critic is not designed to detect or prevent cross-domain tool selection errors.

Finally, the architecture lacks an explicit mechanism for assessing task completion in multi-step scenarios. Completion is inferred from the absence of tool calls in the planner's response, which constitutes an indirect and potentially unreliable signal for complex multi-step workflows. Thus, there may be cases that are resolved, even though the task is not yet marked as completed due to the continued proposal of tools by the planner.

3.5 Context-Specific Agents Architecture

3.5.1 Motivation and Design Rationale

The Context-Specific Multi-Agent Architecture represents the most sophisticated of the three proposed solutions, addressing the remaining reliability limitations of the Planner-Critic Architecture through two complementary mechanisms: domain-specific agent specialisation and explicit task completion judgement.

The rationale behind this approach lies in the fact that a general-purpose agent, which simultaneously handles all domains of e-commerce, faces a broader decision space than a specialized agent operating within a defined functional scope. By distributing the system's tool catalog and prompt space among domain-specific agents, the architecture aims to reduce decision complexity per agent and mitigate the risk of errors in cross-domain tool selection. This design aligns with the broader multi-agent paradigm in LLM application architecture, in which complex tasks can be decomposed among specialized agents with distinct roles [20]. It is also compatible with graph-based orchestration approaches, such as LangGraph, which support stateful workflows, conditional routing, and cyclic execution patterns [1]. The architecture also reflects the organizational

structure of real-world e-commerce customer service operations, in which distinct functional areas, such as pre-sales, post-sales, technical support, and order fulfillment, handle specialized types of requests while simultaneously contributing to a shared customer service process.

Beyond specialisation, the architecture introduces a dedicated task completion evaluation component (stop judge), responsible for explicitly assessing task completion, decoupling the question of whether a task is done from the question of which agent should act next.

3.5.2 Global State

The shared state of the multi-agent graph is richer than in the preceding architectures, reflecting the additional coordination requirements of a multi-agent system:

Listing 3.8: State structure used by the Context-Specific Multi-Agent graph

```
class AgentState(TypedDict):
    messages: List[BaseMessage]
    user_input: str
    route: Route
    done: bool
    supervisor_note: str
    executed_tools: list
    tool_calls: list
    pending_assistant_content: str
    final_answer: str
    workspace: Dict[str, Any]
    loops: int
    max_loops: int
```

Two state fields merit particular attention. The `route` field carries the active agent assignment, typed as a literal union:

Listing 3.9: Literal route values used for specialist agent selection

```
Route = Literal["presales", "cart", "aftersales", "information"]
```

The `workspace` field provides a structured shared memory accessible to all agents, carrying resolved intent, candidate products, confirmed product selections, the active cart identifier, and pending clarification flags:

Listing 3.10: Shared workspace structure used for cross-agent coordination

```
"workspace": {
    "intent": None,
```

```
    "requested_items": [],
    "candidates": [],
    "selected_products": [],
    "cart_id": None,
    "needs_clarification": False,
    "clarification_question": None,
}
```

This structured workspace complements the `messages` list, which carries the conversational history in natural language, by providing explicit, machine-readable representations of resolution state that agents can inspect and update programmatically, without relying on natural language parsing of prior messages.

3.5.3 Graph Structure

This graph consists of eight main processing nodes: `router`, `pre_sales`, `cart`, `after_sales`, `information`, `tools`, `stop_judge`, and `final_responder`. As illustrated in Figure 3.3, the architecture begins with a routing phase that selects a specialist agent based on the user's intent. Each specialist operates with restricted access to their corresponding set of tools, while the results of the tools are incorporated into a shared workspace. The Stop Judge then evaluates whether the task has been completed. If completion is confirmed, the workflow proceeds to the development of the final response, otherwise, control returns to the router to dispatch a new specialist.

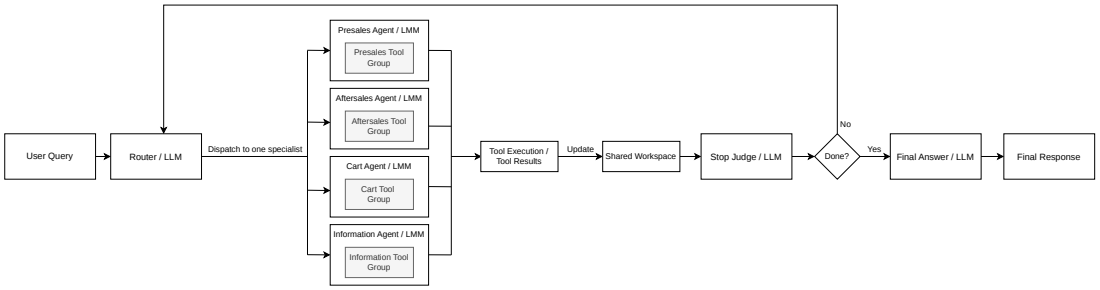


Figure 3.3: Context-Specific Multi-Agent Architecture: routing-based specialist dispatch with restricted tool access, shared workspace, and Stop Judge completion control.

A representative execution workflow, consisting of several steps, for a request to create a shopping cart may involve more than one specialist. For example, first, the request is forwarded to the pre-sales agent so that they can identify the relevant products. Next, the Stop Judge determines that the task is not yet complete and returns control to the forwarder. The forwarder can then forward the request to the agent responsible for the shopping cart, who performs the shopping cart creation step. Once the Stop Judge confirms completion, the workflow proceeds to the final responder and ends.

3.5.4 Router Node

The Router Node acts as the system's supervisor, forwarding each processing cycle to the appropriate specialist agent. At each invocation, it receives the original user request and a compact view of the current workspace state, constructs a structured prompt, and invokes a dedicated instance of the LLM to determine the next agent assignment:

Listing 3.11: Router prompt construction and LLM invocation

```
prompt = (
    f"User request:\n{state['user_input']}\n\n"
    f"Workspace (structured):\n{json.dumps(ws, ensure_ascii=False, indent=2)}\n"
)

resp = router_llm.invoke([
    ROUTER_SYSTEM_PROMPT,
    HumanMessage(content=prompt)
])
```

The router outputs a structured JSON decision:

Listing 3.12: Structured routing decision returned by the Router Node

```
{
  "next_agent": "presales",
  "done": false,
  "note": "short reason"
}
```

The router's system prompt encodes domain assignment rules derived from the operational semantics of the e-commerce scenario. For cart-related requests in which no product identifiers have been resolved, the router directs execution to the presales agent for product resolution before any cart operation can proceed. Once `selected_products` are available in the workspace, subsequent routing to the cart agent is enabled. This conditional routing logic supports multi-agent workflows, in which one specialist agent acts as a required prerequisite for another.

3.5.5 Specialist Agents

Each specialist agent is implemented in a dedicated module, encapsulating its own system prompt, tool bindings, and tool name registry. This modular structure supports independent development, testing, and logging per agent.

3.5.5.1 Presales Agent

The presales agent handles product discovery, recommendation, availability queries, campaign retrieval, and, critically, product name-to-identifier resolution for cart-bound requests. Its tool set comprises: `filter_products`, `check_product_stock`, `get_product_by_name`, and `available_campaigns`.

There is a fundamental behavioural rule governing the role of presales agent in cart-oriented workflows. When the routing context implies a cart action, the presales agent's exclusive responsibility is to resolve product names to valid internal identifiers. Upon completing this resolution, rather than producing a natural language response, it outputs a structured handoff artefact:

Listing 3.13: Structured handoff artefact produced by the Presales Agent for cart-bound requests

```
{
  "handoff": {
    "selected_products": [
      {"id": 1, "name": "MacBook Air M3", "qty": 1},
      {"id": 6, "name": "iPhone 15", "qty": 1}
    ]
  }
}
```

The system extracts this handoff using the `try_extract_handoff` function and stores the identified products in the `workspace["selected_products"]` variable, signaling to the router that product identification is complete and that the shopping cart's execution can proceed. This structured handoff protocol is preferable to natural language communication between agents, as it eliminates ambiguity in product identification, ensuring that the cart agent receives exact identifiers rather than names subject to re-identification. Furthermore, it was found that this approach helped prevent the cart agent from hallucinating identifiers for products that did not exist.

3.5.5.2 Cart Agent

The cart agent is designed to simulate a simplified version of shopping cart-related operations using the `create_cart tool`. It is important to note that this agent does not have access to product search or retrieval tools. Its input data consists of product identifiers that must have been resolved by the pre-sales agent before the cart agent is called. This strict functional constraint prevents the agent from attempting to resolve products independently, which would duplicate work and carry the risk of inconsistencies and data hallucinations.

3.5.5.3 Aftersales Agent

The after-sales agent handles post-purchase operations, such as tracking orders, initiating refund requests, and reissuing receipts, using the `track_order`, `request_refund`, and `reissue_`

`receipt` tools. These tools operate based on a simulated order database containing representative order records, including product lists, shipping status, subtotal, tax calculations, and currency fields.

3.5.5.4 Information Agent

The information agent answers institutional inquiries regarding return and exchange policies, shipping conditions, warranty terms, and general customer support. All of this is done through semantic search in the `info_docs` collection, using the `get_information` tool. Its responses are based on excerpts from retrieved documents, rather than on parametric model knowledge, ensuring consistency with the policies defined by the store.

3.5.6 Domain-Restricted Tool Registry

A central reliability mechanism in this architecture is the domain-restricted tool registry. Each specialist agent exports a dictionary mapping tool names to their corresponding tool instances:

Listing 3.14: Specialist-specific tool dictionary for the Presales Agent

```
PRESALES_TOOLS = {t.name: t for t in tools}
```

These per-agent dictionaries are consolidated into a routing-keyed registry:

Listing 3.15: Routing-keyed registry of domain-restricted tool sets

```
TOOLS_BY_ROUTE = {
    "presales": PRESALES_TOOLS,
    "cart": CART_TOOLS,
    "aftersales": AFTERSALES_TOOLS,
    "information": INFO_TOOLS,
}
```

The shared Tools Node resolves all tool calls through this registry based on the active route:

Listing 3.16: Runtime tool resolution based on the active route

```
registry = TOOLS_BY_ROUTE.get(route, {})
selected_tool = registry.get(tool_name)
```

If a requested tool is not registered for the currently active agent, execution is blocked and a structured error message is appended to both the message history and `executed_tools`, making the violation visible for logging and evaluation:

Listing 3.17: Structured error returned when a tool is not allowed for the active agent

```
err = {
    "error": f"Tool '{tool_name}' not allowed for agent '{route}'. Allowed: {
        allowed}"
}
```

This enforcement mechanism prevents cross-domain tool calls, such as the shopping cart agent attempting to call the `filter_products` function, and ensures that each agent's scope of operation is strictly limited to its designated domain, thereby improving both the reliability and interpretability of the execution log.

3.5.7 Stop Judge Node

The Stop Judge node addresses one of the most persistent reliability challenges in multi-stage agent execution: determining when a complex task has been satisfactorily completed. Before its introduction, the system was prone to post-completion execution loops, in which the router continued to assign agents after the user's objective had been met. In some cases, this resulted in unwanted secondary actions, such as creating a shopping cart after an informational query or even hitting LangGraph's recursion limit.

The stop judge receives the original user request, the last agent reply and the structured list of recently executed tools, reasoning over the relationship between the stated objective and the actions performed to determine whether the task is complete:

Listing 3.18: Stop Judge prompt construction using the user request, last agent reply, and recent tool executions

```
prompt = (
    f"Original user request:\n{state['user_input']}\n\n"
    f"Last agent reply [{route}]:\n{last_agent_reply}\n\n"
    f"Executed tools (recent):\n{json.dumps(executed, ensure_ascii=False, indent=2)
        }\n\n"
)
```

It returns a structured binary verdict with justification:

Listing 3.19: Structured binary verdict returned by the Stop Judge

```
{
  "done": true,
  "reason": "short reason"
}
```

A particularly important rule built into this agent's prompt addresses the case of unsuccessful product searches. When `get_product_by_name` returns `Found=False` for a specific product request, the stop judge agent is instructed to classify the task as completed, with the implicit conclusion that the request cannot be fulfilled as specified. This prevents the system from entering endless recovery loops for products missing from the catalogue.

Graph routing following the stop judge directs execution to the final answer node on task completion, or back to the router for continued execution:

Listing 3.20: Conditional routing after the Stop Judge decision

```
def route_from_stop_judge(state: AgentState) -> str:
    if state.get("done"):
        return "final_answer"
    return "router"
```

3.5.8 Final Answer Agent

The Final Response Agent constitutes the final stage of processing in the multi-agent workflow. Its function is to synthesize the accumulated resolution history, gathered throughout the conversation via messages and the log of executed tools, into a coherent natural-language response tailored to the user. This separation between response generation and task execution ensures that specialised agents remain focused on selecting and invoking tools, while the quality of communication with the user is the exclusive and consistent responsibility of a single, separate component.

Before the introduction of this agent, the final response was simply the last AIMessage in the message history at the time the task was completed. This produced unreliable output: the final message could mistakenly be a technical handoff artefact (e.g., '"handoff": ...'), a stop log entry, or an intermediate tool selection statement—none of which constitute appropriate responses for the user. The Final Response Agent eliminates this risk by always generating a specific, context-sensitive response based on the completed execution:

`stop_judge(done=True)` → `final_answer_agent` → `END`

3.5.9 Product Lookup with Confidence Scoring

A cross-architectural refinement that is especially relevant to the reliability goals of this work concerns the `get_product_by_name` tool. In its initial implementation, the tool performed an unrestricted semantic search and returned the top-ranked results regardless of their similarity score relative to the query. This behaviour led to a particularly problematic failure mode: when a user requested a product absent from the catalogue, the tool returned the most semantically proximal available product, and the agent treated it as a valid match.

A representative instance of this failure was a user request for a Nokia smartphone. The vector search returned a Xiaomi device as the highest-scoring candidate, and the agent produced a response identifying the Xiaomi as the requested Nokia product. This constitutes a factual hallucination at the tool level rather than the model level, highlighting that reliability failures in tool-augmented agents are not exclusively attributable to LLM reasoning errors.

The tool was redesigned to classify each lookup result into one of three outcome categories, determined by comparing the best candidate's confidence score, a weighted combination of lexical similarity and token overlap, against two fixed thresholds:

Listing 3.21: Confidence-based outcome categories returned by the product lookup tool

```
# Confirmed match (confidence >= 0.90)
{
    "found": True,
    "product": {...},
    "confidence": 0.97
}

# Close but not confident enough (0.60 <= confidence < 0.90)
{
    "found": False,
    "reason": "close_match",
    "candidates": [...]
}

# No meaningful match (confidence < 0.60)
{
    "found": False,
    "reason": "no_match",
    "candidates": [...]
}
```

When `found=True`, the confidence score is returned alongside the matched product, providing the agent with an explicit signal of retrieval certainty. When `found=False`, the reason field distinguishes between a close but insufficiently confident match (`close_match`) and a retrieval that produced no semantically meaningful candidate (`no_match`). In both cases where no match is found, a list of the top candidate products is returned, allowing the agent to suggest alternatives to the user, if applicable.

This three-way classification allows agents to reason about search results with greater precision than a binary found/not found indicator or even just a numerical score would allow. A `close_match` result may warrant an explanatory response to the user, while a `no_match` result is a stronger indication that the requested product is absent from the catalogue. The stop judge is instructed to treat any `found=False` result for a specific product request as a terminal condition, preventing the system from entering into unresolvable retrieval loops.

3.5.10 Limitations

Despite its architectural sophistication, the Context-Specific Multi-Agent Architecture retains several limitations that require acknowledgement.

The reliability of the stop judge depends on its ability to infer the completion of a task based solely on the record of executed tools. Ambiguous requests, particularly those whose intent spans both informational and transactional domains, may cause it to terminate prematurely or persist unnecessarily on a task. An availability query such as *"Are there any Asus laptops available?"* may be classified as incomplete if the stop judge implicitly requires an explicit stock check, rather than accepting a retrieval response as sufficient proof of task completion.

The router's routing decisions are sensitive to the state of the `workspace["selected_products"]`. In cases where the pre-sales agent generates a transfer artefact for an informational request in which there is no purchase intent, the presence of `selected_products` in the workspace may cause the router to direct the execution to the shopping cart agent. This represents a confusion between product identification (a cognitive act) and purchase intent (a user directive).

The depth of orchestration in the architecture introduces a substantially higher operational cost compared to simpler architectures. A multi-step interaction can invoke five or more distinct LLM calls (router, expert agent, tools, stop judge, final response) per user query cycle, increasing both latency and cumulative token consumption. This trade-off between cost and reliability is a central dimension of the architectural comparison presented in the Results chapter.

3.6 Comparative Overview

The three architectures form a progressive design trajectory in which each solution introduces targeted control mechanisms in response to failure modes identified in its predecessor. Table 3.1 provides a structured comparison of the key architectural properties of the three solutions.

Table 3.1: Comparative summary of architectural properties across the three proposed solutions

Property	**Simple Agent**	**Planner-Critic Agents**	**Context-Specific Agents**
Pre-execution plan validation	No	Yes (Critic)	No
Domain-specific specialisation	No	No	Yes
Tool domain enforcement	No	No	Yes (`TOOLS_BY_ROUTE`)
Explicit task completion judgement	No	No	Yes (Stop Judge)
Structured inter-agent handoff	N/A	N/A	Yes (`workspace`)
Number of LLM instances	1	2	5+
Stateful graph execution	No	Yes	Yes
Dedicated response synthesis	No	No	Yes (Final Answer Agent)

The evolution from the Simple Agent to the Planner-Critic Architecture introduces pre-execution validation as the primary reliability mechanism, accepting an increase in LLM invocations per query. The further progression to the Context-Specific Agents Architecture replaces this per-call validation with a more comprehensive structural control: domain partitioning, tool access restriction, explicit completion assessment, and separation of response generation. Each control layer is motivated by a specific class of failure mode and contributes independently to the system's overall reliability profile, a relationship that is examined quantitatively in the Results and Discussion chapters.

Chapter 4

Methodology

4.1 Overview and Evaluation Objectives

This chapter describes the methodology used to evaluate and compare the three agent architectures proposed. The evaluation was designed to generate empirical evidence about how architectural design choices (pre-execution plan validation, domain specialisation, and explicit task completion control) affect the behaviour of a conversational AI agent in a structured e-commerce customer service scenario.

The evaluation approach focuses on a fixed set of test queries executed on the three architectures under controlled conditions. This design allows for intra-query comparison, examining how each architecture handles the same query, as well as inter-architecture aggregation across the entire test set. Fundamentally, the scope of the evaluation goes beyond assessing final responses, also looking closely at the internal execution trace of each test, which includes tool selection, argument construction, intermediate tool results, routing decisions, and termination conditions. This is captured in structured logs and treated as primary analytical data. This process-level instrumentation reflects the view that a correct final response does not, in itself, constitute proof of reliable agent behaviour. The process through which a response is generated must also be examined to distinguish genuine reliability from coincidental correctness.

The chapter covers the evaluation dimensions, the design and composition of the test query set, the execution procedure, the logging and data extraction infrastructure, the metrics collected, the stress testing variants, and the result classification scheme, concluding with a discussion of the principal methodological limitations.

4.2 Evaluation Dimensions

The evaluation is structured around five dimensions, each corresponding to a class of reliability failure motivating the design of the proposed architectures.

Tool selection correctness: concerns whether the agent selects the appropriate tool for a given request and constructs its arguments with sufficient specificity. This dimension captures both

outright tool misselection and the more subtle failure of argument underspecification, for example, not applying the product category in a discovery task and applying only the price constraint.

Multi-step task resolution: concerns whether the agent correctly sequences tool invocations across multiple steps in requests that require results from one operation to inform the next.

Domain specificity: concerns whether the agent correctly routes requests to domain-appropriate tools and avoids cross-domain tool invocations, for example, invoking product retrieval tools in response to institutional policy queries.

Task completion control: concerns whether the agent terminates execution at the appropriate point, neither prematurely before all required steps have been completed nor belatedly, by continuing to invoke tools after the user's request has been satisfied.

Operational cost: concerns the number of LLM invocations and the total token consumption required per query, as these determine the latency and financial cost of deployment at scale.

4.3 Test Query Set

4.3.1 Composition and Design Approach

To ensure that the evaluation provides meaningful and reproducible evidence of agent reliability, a systematic black-box testing methodology was followed for the construction of the test dataset. Rather than selecting queries ad hoc, each test case was designed with a defined expected outcome, a primary testing technique, and verifiable assertions against the known ground truth of the mock catalogue. The design process was structured to achieve comprehensive coverage across all tool execution paths, equivalence classes, boundary conditions, error cases, and dimensions of paraphrase invariance, reflecting the diversity of reliability challenges that an e-commerce AI agent is likely to encounter in production.

The scope of the dataset was determined by the system's tool interface and consists of a set of nine tools available across the three architectures: `get_product_by_name`, `filter_products`, `check_product_stock`, `create_cart`, `track_order`, `request_refund`, `reissue_receipt`, `get_information`, and `available_campaigns`. The aim is to ensure that no functional path is left unevaluated.

This process resulted in a test set consisting of eighty-nine queries in natural language, of which fifty-eight comprise the set of original queries, covering the system's entire functional scope, and thirty-one queries with paraphrased variants, which constitute a layer dedicated to assessing robustness. The complete set of original and paraphrased evaluation queries is presented in Appendix A.

4.3.2 Testing Techniques and Query Design Criteria

Eight black-box testing techniques were applied to guide the design process for the test cases, determining which expected inputs and outputs to include in the test suite. The techniques operate from the user's perspective, defining test cases in terms of what a customer would ask and the result they expected and should receive, without any reference to internal implementation choices. Although the design of the queries is thus based on black-box principles, the evaluation itself operates on two levels. The first black-box layer aims to evaluate the correctness of the final responses against fundamental truth. The second layer, however, pertains to the process level, which examines the internal execution trail, including tool selection, argument construction, and routing decisions, in order to distinguish genuine reliability from fortuitous correctness. Each query is annotated with the primary technique it employs, reflecting the diversity of reliability challenges targeted by the evaluation.

Equivalence Partitioning (EP) divides the input space into classes within which all members are expected to produce the same category of agent behaviour. One representative query is selected per class, covering the main functional paths: exact product match, brand browsing, stock check, price filtering, cart creation, institutional information retrieval, order tracking, refund initiation, and campaign lookup. The majority of the original queries belong to this category.

Boundary Value Analysis (BVA) targets the edges of price ranges, where retrieval defects are most likely to manifest. The price boundaries tested include \$499/\$500, \$699/\$700, \$999/\$1000, \$1499/\$1500, and \$2000, as well as zero-result ranges such as the sub-\$500 laptop filter, where no catalogue product exists below the boundary. BVA queries expose filter implementation errors that would not be detected by mid-range price queries.

Decision Table Testing (DT) tests combinations of conditions that trigger different system responses. The primary DT case is the duplicate product identifier in a cart request, Q55, *"two iPhone 15 units and one MacBook Air M3"*, which tests whether the cart tool handles repeated product identifiers correctly and whether the final total reflects the expected multiplication of unit prices.

Error and Negative Testing (ET) exercises system behaviour with invalid or absent inputs: non-existent order identifiers, e.g., `ORDER456`, products with no catalogue match, e.g., *"Nokia 3310"*, out-of-scope product categories, e.g., *"gaming chairs"*, and empty cart scenarios resulting from failed product resolution. These queries verify that the system fails gracefully rather than hallucinating, fabricating order data, or adding incorrect products to the cart.

Semantic and Exploratory Testing (ST) targets queries with indirect or ambiguous intent, where the user's goal must be inferred from vocabulary not directly reflected in the catalogue schema. The representative case is Q36, *"Find a cheap way to watch Netflix"*, which requires the agent to interpret streaming intent as a product category signal, mapping the query to streaming devices such as the Chromecast or Apple TV, without any explicit product name or category in the input.

Similarly, Q35, *"Show me available gaming products"*, requires keyword-based inference across multiple product categories. These queries probe the semantic generalisation capability of the retrieval layer.

Multi-condition and Integration Testing (MT) tests sequences of tool calls and cross-agent chaining. The canonical MT case is a cart-bound request requiring filter-then-create sequences across two product categories, Q26, *"one laptop and one monitor"*, and combined presales-plus-campaign queries such as Q56. These queries are the primary evaluation cases for the presales-to-cart handoff in the Context-Specific Multi-Agent Architecture.

Robustness and Paraphrase Invariance Testing (RT) verifies whether the system produces semantically equivalent results for queries that express the same intent in different surface linguistic forms. This technique is described in a subsequent section.

Robustness combined with Boundary Value Analysis (RT+BVA) applies paraphrase variation to a boundary value query, simultaneously testing linguistic robustness and filter boundary behaviour. The representative case is Q52-B, *"Show me the cheapest laptop you have under 400 dollars"*, which combines a zero-result price constraint with a superlative phrasing that additionally tests sort-and-filter interaction.

4.3.3 Original Query Set (Q1–Q58)

The fifty-eight original queries are grouped into functional categories aligned with the tool and domain structure of the system.

Presales – Specific Lookup (Q1, Q2, Q4, Q5, Q11–Q14, Q27): Requests targeting named products, stock confirmation, or a specific close-match case. The primary tool path is `get_product_by_name`, optionally chained with `check_product_stock`.

Presales – Brand and Category Browse (Q6–Q10): Requests over a brand or product family using `filter_products` with keyword signals. Representative ground truth: Q10, *"Any Asus laptops available?"*, is expected to return the Asus ROG Zephyrus G14 ($1799) as the sole matching result.

Presales – Price Filtering (Q16–Q20): Price-bounded product queries targeting BVA cases. Representative ground truth: Q16, *"Show me phones under $1000"*, should return the Google Pixel 8 ($999) and the Xiaomi Redmi Note 13 Pro ($399), with the iPhone 15 ($1199) correctly excluded.

Presales – Not Found and Close Match (Q3, Q15, Q50, Q51): Products absent from the catalogue or available only in a generationally adjacent version. Q15, *"Is the Nokia 3310 available?"*, has a defined negative outcome: no Nokia product exists in the catalogue and the confidence score must fall below the acceptance threshold (0.60), triggering a `reason=no_match` response. Q3 and Q50 target the Samsung Galaxy S25 / S25 Ultra, for which the closest catalogue entry is the Galaxy S24, producing a `reason=close_match` response.

Presales – Relevance Sort and Zero Results (Q37, Q52, Q53): Q37, *"List laptops sorted by price descending"*, exercises the `sort_by=price_desc` path explicitly. Q52, *"Show me laptops under $500"*, is a zero-result BVA case: the cheapest laptop in the catalogue is $1199, so the filter must return an empty list without hallucinating any products. Q53 tests the relevance sort path with a semantic query, *"Show me the most relevant laptops for video editing"*.

Cart Operations (Q21–Q26, Q38, Q54, Q55): Transactional requests requiring product resolution followed by cart creation. Representative ground truth: Q21, *"Add the MacBook Air M3 and iPhone 15 to my cart"*, resolves to `ids=[1, 6]` with a total of $2598.00 before tax. Q24, *"Add all tablets to my cart"*, resolves to `ids=[10, 11, 12]` with a total of $2397.00. Q54 is a degenerate cart case: a non-existent product yields no valid identifiers, so `create_cart` receives an empty list and must produce an empty or error response without fabricating a transaction. Q55, *"Add two iPhone 15 units and one MacBook Air M3"*, tests duplicate product identifier handling: `ids=[6, 6, 1]`, expected total $3797.00.

Aftersales – Order Tracking, Refund, and Receipt (Q40–Q45): Requests operating against the mock order database. Valid order identifiers, `ORDER123` and `ORDER999`, produce defined responses, while invalid identifiers, `ORDER456`, must produce graceful error responses.

Campaigns (Q46–Q49): Semantic retrieval over the `campaign_docs` collection. Q46 targets the apple week deals campaign, with discount, additional 5% above €1200, free iPhone case. Q47 targets the free shipping weekend campaign, with free standard shipping on orders above €49, Friday to Sunday. Q48 targets the student discount campaign, with 12% off laptops and tablets with valid student ID. Q49, *"Do you have any ongoing promotions?"*, is a broad query expected to return all three campaigns.

Institutional Information (Q28–Q33): Policy and support queries resolved through `get_information` over the `info_docs` collection. Representative ground truth: Q28, *"What's your return policy?"*, should retrieve the returns text chunk specifying a 30-day return window with 7–10 day refund processing. Q29 should retrieve the shipping schedule, Monday to Friday, with weekend orders processed on the next business day. Q30 should retrieve international shipping coverage for the EU and UK.

Cross-Cutting and Combined Queries (Q34–Q36, Q39, Q53, Q56, Q57, Q58): Queries requiring multi-tool chaining across domains, semantic inference, or combined aftersales and information retrieval. Q56, *"I want to buy a MacBook Air M3. Is there any student discount for it?"*, requires `get_product_by_name` and `available_campaigns` in the same interaction, producing a product result alongside a campaign result. Q58, *"Do you sell gaming chairs?"*, is an out-of-scope query: no furniture category exists in the catalogue and the semantic score filter should eliminate all results, requiring the agent to respond that the product category is not available rather than hallucinating an entry.

4.3.4 Paraphrase Variant Queries (Q1-A to Q57-A)

The thirty-one paraphrasing variants constitute a layer dedicated to assessing robustness, systematically testing whether each architecture produces semantically equivalent results when the same user intent is expressed in a different linguistic form. Each variant is derived from a base query selected for its high discriminatory power, specifically in cases where the domain classification of the Context-Specific router or the argument extraction of the Planner-Critic are more susceptible to discrepancies in the event of rephrasing.

The variants exercise the following paraphrase dimensions:

Synonym substitution: Lexical replacements that preserve intent while changing surface vocabulary. Examples include "basket" in place of "cart" (Q21-A) and "currently available" in place of "in stock" (Q2-A). These variants test whether tool selection and argument extraction are robust to vocabulary variation within the same semantic field.

Word order and name reversal: Reorderings that preserve all content words but alter their sequence. Q1-B, *"I'm looking for the M3 MacBook Air"*, reverses the product name, testing whether the name-matching logic of `get_product_by_name` is order-sensitive.

Implicit intent: Queries in which the user's goal must be inferred from context without explicit vocabulary. Q28-B, *"Can I send back a product if I change my mind?"*, contains no direct reference to a return policy, requiring the router to classify the query as information-domain and the agent to invoke `get_information` without keyword anchoring.

Abbreviated and partial product names: Queries using shortened model identifiers. Q3-B, *"What's the price of the Samsung S25?"*, omits the "Galaxy" prefix, and Q15-A, *"Do you carry Nokia phones?"*, omits the model number. These variants test the robustness of partial name matching in `get_product_by_name`, as well as the threshold between specific name searches and category filtering.

Spelled-out numeric values: Queries in which price constraints are expressed in natural language rather than numerals. Q16-A, *"What smartphones do you have for less than a thousand dollars?"*, tests whether the agent correctly extracts numeric constraints from textual representations.

Register variation: Shifts between formal and colloquial phrasing. Q2-B, *"Do you still carry the Pixel 8?"*, uses an informal commercial register, and Q21-B, *"Put the MacBook Air M3 and the iPhone 15 in my cart please"*, adds a politeness marker. These test robustness against superficial variation in register.

Implicit price constraint: Q16-B, *"Show me budget smartphones"*, contains no explicit price bound, requiring the agent to interpret "budget" as a price constraint and apply a modest price filter or sort results by price in ascending order. This variant highlights differences in how each architecture handles semantically implicit constraints.

A total of thirty-one variants was set as a pragmatic constraint, reflecting a balance between robustness coverage and the operational cost of executing each query across six evaluation configurations (three architectures x two base models). Within this constraint, seventeen query families were selected based on discriminatory power. Two variants were assigned to fourteen families, covering cases where two distinct dimensions of paraphrase could be tested independently within the same base query, and one variant was assigned to three families, in which only a single significant dimension of paraphrase was identified. This resulted in thirty-one variants across seventeen families.

A paraphrase invariance failure is recorded when a variant query produces a result substantially different from its base query or within the same family, for example, a different tool being selected, a different product being returned, or the query being routed to the wrong domain agent, even though the queries express the same underlying intent.

4.3.5 Ground Truth Specification

Because the product catalogue, order database, and institutional document collection are fully controlled and known, it is possible to define verifiable ground truth assertions for the majority of test queries. These assertions serve as the reference standard against which agent responses are evaluated during the manual annotation stage of the data extraction workflow.

4.4 Evaluation Execution Procedure

All evaluations were conducted using fixed model configuration settings for each architecture. The main tool-using agents, namely the Simple Agent, the Planner in the Planner-Critic architecture, and the expert agents in the Context-Specific architecture, were run with a temperature value of 0.9. In contrast, the validation and control components with deterministic decision functions, namely the Critic, the Router, and the Stop Judge, were run with a temperature value of 0.0. These settings were kept consistent across all corresponding components throughout the evaluation to ensure comparability between the architectures, while also reflecting the different roles of the generative agents and the decision-oriented control components. The higher temperature used for tool-using agents was intended to allow more flexible natural-language interpretation and response generation, whereas the deterministic setting used for validation and routing components was intended to reduce variability in control decisions.

4.4.1 Evaluation Script

The evaluation is conducted through a dedicated Python script, `evaluate_agent.py`, which iterates over the fixed query set and dispatches each query to the selected architecture via a `TEST_MODE` parameter. This parameterised runner allows all three architectures to be evaluated against identical inputs without modification to the query set or the evaluation logic.

In this script, tests are run sequentially, with the start and end of each test being logged and a configurable pause applied between tests (`PAUSE_SECONDS = 3`) to avoid API rate-limiting errors during prolonged evaluation runs, a topic covered in a section below.

4.4.2 State Isolation Between Tests

Each test is run with a clean initial state. In the case of the Simple Agent, this requires resetting the message list at the start of each call to the `run_agent` function, ensuring that the conversation history from previous test cases does not contaminate subsequent cases. For the Planner-Critic and Multi-Agent architectures, state isolation is enforced structurally, through initialisation, on each invocation of `run_agent_graph` or `run_multi_agent_graph`, of a new instance of `AgentState`, with `messages`, `executed_tools`, `tool_calls` and all control fields reset to their initial values. This design aims to preserve the statistical independence of individual test cases, a requirement for valid aggregation across queries.

4.4.3 Infrastructure Requirements

Prior to executing the evaluation, the following infrastructure components must be active and correctly populated:

- `Qdrant` must be running locally, and the collections that store product data, informational documents, and campaign documents must be properly populated, with no duplicate entries. The product collection is recreated with each pop-up run to prevent duplicate indexing resulting from multiple runs.
- OpenAI API access must be available, with valid credentials configured as environment variables.
- The evaluation script must be executed within the project's virtual environment, with all Python dependencies installed.

4.5 Logging and Observability

4.5.1 Logging Configuration

Each evaluation session writes to a dedicated log file, configured using Python's logging module.

The `force=True` parameter is required to ensure that the logging configuration takes effect even when other modules have previously initialised the root logger, a situation that otherwise causes log output to be directed to an unintended file.

The log files are separated and named according to the architecture and model being tested, so as to allow each evaluation run to be analysed separately.

4.5.2 Logged Events

The logging infrastructure captures the following events for each test case:

Test boundary markers: delimit the start and end of each test, including the test index and query text.

Tool plan: records the proposed tool call prior to execution. In the Planner-Critic Architecture, this corresponds to the proposed tool call before critic evaluation.

Critic events: record the prompt sent to the critic, its decision, and its feedback. These events apply only to the Planner-Critic Architecture.

Tool execution events: record the tool invoked and its arguments.

Tool results: record the structured output returned by each tool.

Routing and Stop Judge decisions: record the router's agent assignment and the Stop Judge's completion assessment. These events apply only to the Context-Specific Multi-Agent Architecture.

Final response: records the agent's natural language output to the user.

Test metrics: record operational measurements at the end of each test.

These structured log entries provide the primary data source for subsequent extraction, tabulation, and comparative analysis.

4.5.3 The `executed_tools` Record

In the Planner-Critic and Multi-Agent architectures, each tool execution is additionally logged in the `executed_tools` status field. This structured log serves a dual purpose, being passed to the critic (Planner-Critic Architecture) and the stop judge (Multi-Agent Architecture) in each evaluation cycle to guide their decisions, while also providing a machine-readable audit trail of the complete tool execution sequence for each test, enabling the precise reconstruction of the agent's decision-making process.

4.6 Data Extraction Workflow

Data collection combined two complementary activities: the automatic extraction of operational and execution-tracking data from structured log files, and the manual annotation of response quality. The automatic layer covers all fields that can be obtained without human intervention, such as timings, token counts, tool call sequences, transactional results, and routing decisions. The manual layer is limited to fields that require a qualitative assessment, such as correctness scores and error type classification. The following sections describe each layer individually.

4.6.1 From Logs to Structured Tables

The structured log files generated by the evaluation script were fed into a Large Language Model, along with the desired output structure for each sheet and for each architecture. The model then extracted and populated the fields from the log content, producing two Excel sheets per architecture, one per analysis unit. Since the manual annotation phase required individual review of each row for correctness scoring and error classification, this review also served as an implicit validation step, as the values of the extracted fields were verified against the corresponding log entries during the same read. Although this process reduces the risk of extraction errors, it does not formally guarantee their absence, particularly for operational fields, such as token counts or elapsed time, which were visible during the review but were not the primary focus of the annotation task.

The first sheet contains one row per test case, recording for all three architectures: the time spent on reasoning (overall or per agent in multi-agent architectures), the time spent executing the tools, the total time to respond to a query, the final response given to the user, as well as operational metrics (different token's metrics). In addition to the fields extracted from the logs, it includes a 'correctness score' column, which is applied following a manual assessment of the test.

The second sheet contains one row per tool call, recording each step taken in resolving the query. Each row contains the number of the step in question and the agent called in that step (applied only in multi-agent architectures). If it is a tool call step, it is possible to check the chosen tool and the arguments used. If it is a verification step (as is the case with the critic and stop judge in the Planner-Critic and Context-Specific Agents architectures, respectively), it is possible to see the agent's verdict and the explanation for its decision.

4.6.2 Fields Requiring Manual Annotation

The following fields require a manual review of the final response and the execution log, as they have not been automatically derived from the log content:

Correctness assessment: Each test is assigned a categorical correctness label and a numerical correctness score based on manual comparison between the system's final response, the execution trace, and the expected reference behaviour for the query. This assessment distinguishes between fully correct responses, partially correct responses, and incorrect responses. The detailed scoring rules used to assign these labels and numerical values are described later in Section 4.9.1.

Error classification: Tests classified as incorrect or partially correct are additionally annotated with an error type. These error types identify the main failure mechanism observed in the execution trace, such as tool selection errors, argument errors, critic decision errors, Stop Judge decision errors, or loop-related failures. The complete taxonomy of error types is presented later in Section 4.9.2.

Process correctness: Regardless of the final response, the execution trace is evaluated for process correctness, verifying whether the invoked tools, routing decisions, critic decisions, Stop Judge decisions, and completion criteria were appropriate for the request. This distinction is necessary

because a correct final response can mask an incorrect process, for example when post-tool reasoning filters out incorrect retrieval results before producing the response. Conversely, an incorrect final response can reflect a process error at a specific, identifiable stage.

4.7 Metrics

4.7.1 Quantitative Metrics Derivable from Logs

The following metrics can be computed directly from the extracted log data without manual response assessment:

Total number of tests and completion rate: the proportion of tests that produced a final response without a runtime error.

Total token consumption: per test, per architecture, and per query category, decomposed into prompt tokens and completion tokens.

Elapsed time: per test and mean elapsed time per architecture.

Number of tool calls per test: total invocations of any tool across the full execution trace.

Number of LLM invocations per test: relevant to comparing the per-query overhead of each architecture.

Number of critic revisions per test: the count of `REVISE` decisions issued per query in the Planner-Critic Architecture, indicating how frequently the critic intercepted and corrected the planner's initial proposal.

Stop Judge decision count: the number of Stop Judge evaluations per test in the Context-Specific Multi-Agent Architecture and the distribution of `done=True` versus `done=False` verdicts.

Paraphrase invariance rate: the proportion of paraphrase variant queries that produce an outcome consistent with their base query, per architecture. A divergent outcome, such as a different tool selected, different domain routed, or materially different response, constitutes a paraphrase invariance failure.

4.7.2 Metrics Requiring Manual Annotation

Correctness rate per architecture and per query category: the proportion of tests classified as correct.

Error type distribution: the frequency of each error type from the taxonomy defined, enabling attribution of failures to specific pipeline stages.

Process correctness rate: the proportion of tests in which the execution trace was assessed as process-correct, independently of final response correctness.

4.8 Weaker Model Stress Test

In addition to the primary evaluation conducted with `gpt-4o-mini`, an additional stress test was performed by replacing the evaluation model with a weaker model, `gpt-3.5-turbo`. No separate stress test set was created for this purpose. Instead, the same evaluation queries, tool catalogue, and agent configurations were preserved, allowing the impact of reduced model capability to be isolated as the main experimental variable.

The motivation behind this was the idea that more robust language models could potentially compensate for architectural weaknesses. When the underlying model possesses stronger capabilities for reasoning, following instructions, and contextual retention, certain architectural limitations might remain less visible. As a result, the reliability benefits of more structured architectures, such as the Planner-Critic and Context-Specific Multi-Agent approaches, could be partially masked. By evaluating the same architectures with a weaker model, the experiment aims to observe whether architectural control mechanisms become more relevant when the base model is less capable.

The weakest-model condition is particularly relevant for assessing whether architectures offer reliability advantages that are independent of the model's raw capacity. For example, the Planner-Critic Architecture may benefit from the additional validation step prior to tool execution, but this mechanism still depends on the quality of the reasoning performed by the model used by the planner and the critic. Similarly, the Context-Specific Multi-Agent Architecture can reduce the probability of incorrect tool selection by restricting each agent to a smaller, more domain-specific set. This type of structural constraint can be especially valuable when the model is weaker, as it reduces the space of possible actions available to the agent.

4.9 Result Classification Scheme

4.9.1 Final Response Correctness

Each test is assigned a numerical correctness score in the continuous range [0, 1] and a corresponding categorical rating, based on a manual analysis of the final response compared to the known reference answer. The numerical score allows the partial score to be expressed more precisely than a three-point categorical scale and enables direct aggregation into average correctness rates at the architecture level.

In all cases, the results of the tool execution recorded in the step sheet were consulted alongside the final response to verify that the score reflected the actual execution result, rather than merely the agent's self-reported response.

Table 4.1: Final response correctness labels and definitions

Label	**Definition**
Correct	The response fully and accurately addresses the user's request in accordance with catalogue ground truth.
Partially correct	The response addresses part of the request accurately but omits or incorrectly handles a component.
Incorrect	The response contains factual errors, substitutes an incorrect product, or performs an unrequested action.

The numeric score was assigned according to the following rules, applied depending on query type:

- **Specific product lookup:** The score is binary. A score of 1.0 is assigned when the agent correctly identifies and returns the requested product. A score of 0.0 is assigned when a wrong product is returned without an explicit disclaimer that the requested item was unavailable.
- **Category and filter browse:** When the agent is expected to return a list of products matching a constraint, such as category, price range, or brand, the score is computed as the ratio of correctly matching items to total items returned:

$$\text{score} = \frac{\#\text{correct\_items}}{\#\text{total\_items\_returned}}$$

- **Non-existent product – availability query:** A score of 1.0 is assigned when the agent explicitly informs the user that the product is not in the catalogue. A score of 0.0 is assigned if the agent incorrectly claims that the product is available.
- **Non-existent product – transactional query:** A score of 0.0 is assigned when the agent silently substitutes a different product without disclosing the unavailability of the requested item. If the agent correctly discloses unavailability and offers alternatives, the score is computed as:

$$\text{score} = \frac{\#\text{correct\_domain\_alternatives}}{\#\text{total\_alternatives\_offered}}$$

  where a correct alternative is one belonging to the same product category as the requested item.
- **Multi-step transactional queries:** The score reflects the percentage of required steps completed. A score of 1.0 assumes that all steps have been completed, while a score of 0.5 is assigned when only intermediate steps are completed, for example, products are selected but the cart has not been created. Consequently, a score of 0.0 is assigned when no significant steps have been completed.

- **Order management queries:** The score is binary. A score of 1.0 is assigned when the correct action is performed or a graceful error is returned for an invalid order identifier. A score of 0.0 is assigned when the wrong action is taken or a required step is omitted.
- **Information and campaign retrieval:** The score is binary. A score of 1.0 is assigned when the retrieved content is relevant to the user's query. A score of 0.0 is assigned when the response contains unrelated content or fails to retrieve any relevant information.

4.9.2 Error Type Taxonomy

Tests classified as *incorrect* or *partially correct* are additionally annotated with an error type, as shown in Table 4.2.

Table 4.2: Error type taxonomy used for manual failure annotation

Error Type	**Description**
`wrong_tool_selection`	The agent invoked a tool inappropriate to the request domain.
`wrong_tool_arguments`	The correct tool was invoked but with incomplete or semantically incorrect arguments, for example, missing category constraint in a price filter.
`wrong_revise_decision`	The critic issued a `REVISE` decision for a plan that was correct and would have led to a successful outcome, triggering unnecessary revision cycles in the Planner-Critic Architecture.
`wrong_approve_decision`	The critic approved a plan containing a fundamental error, for example, an incorrect tool selection or underspecified arguments, allowing a defective plan to proceed to execution unchallenged in the Planner-Critic Architecture.
`wrong_done_false_ decision`	The Stop Judge incorrectly assessed the specialist agent's output as incomplete, triggering an unnecessary additional specialist dispatch when the task was already adequately resolved in the Context-Specific Multi-Agent Architecture.
`wrong_done_true_ decision`	The Stop Judge incorrectly assessed the specialist agent's output as complete, allowing an insufficient response to proceed to final answer generation in the Context-Specific Multi-Agent Architecture.
`limit_of_tries`	The configured maximum number of revision attempts was exhausted after repeated `REVISE` or `REJECT` decisions, terminating the execution to prevent an unlimited correction loop.
`loop_or_recursion`	The execution reached the maximum loop count or LangGraph recursion limit without completing.

4.9.3 Dual Assessment: Response and Process

Each test is assessed independently along two dimensions:

1. **Response correctness:** whether the user received an appropriate final response.
2. **Process correctness:** whether the internal execution trace, including tool selection, argument construction, routing, and termination, was appropriate for the request.

This dual evaluation is necessary because these two dimensions do not always align. An answer may appear correct even though the underlying process is incorrect, for example due to insufficiently specified tool arguments or noisy retrieval results that are later filtered out during response generation. Conversely, the execution trace may be correct, while the final response

is incorrect due to a failure in the final response generation phase. Reporting both dimensions prevents apparently successful answers from hiding systematic process errors that would pose reliability risks in production implementations.

4.10 Limitations

4.10.1 Manual Annotation Requirement

Response correctness and process correctness assessments require manual review. This introduces a dependence on human judgement and the risk of annotation inconsistency. An annotation guideline should be established and applied consistently across all tests and architectures to minimise inter-rater variability.

4.10.2 Single Execution Per Test

The primary evaluation executes each query once per architecture. At elevated temperature settings, the model's output is stochastic and results may vary across runs. The extent to which single-execution results are representative of each architecture's typical behaviour under those settings cannot be established without multiple runs. Where elevated temperature is used in stress testing, this limitation should be acknowledged in the interpretation of results.

4.10.3 Model Version Sensitivity

The behaviour of any individual architecture is partially determined by the specific model version used at evaluation time. Differences in model capability, instruction following, and tool selection behaviour across model versions mean that findings are specific to the model configuration used and may not generalise to other models.

4.10.4 Controlled Catalogue Scope

The product catalogue used in the evaluation contains about thirty fictional products spread across nine categories. Because the catalogue is small, semantic search has few competing products to choose from, which makes retrieval relatively easy. In a real e-commerce store with thousands of products, many of which are similar in name, category, or description, the agent would face a more difficult disambiguation problem and would likely produce more retrieval errors. The results should therefore be interpreted with the understanding that this evaluation measures the architecture's behaviour under controlled conditions and not necessarily the performance that would be observed in a production implementation, though it can still serve as a predictive measure for that purpose.

4.10.5 Mocked Transactional Data

The cart and after-sales tools operate against in-memory mock data rather than a live order management or inventory system. This is appropriate for the purposes of architectural comparison in a research prototype, but limits the validity of any conclusions about production deployment behaviour, particularly with respect to concurrency, data consistency, and failure modes arising from external system integration.

4.10.6 API Rate Limits

During extended evaluation runs, the OpenAI API may return "429 Too Many Requests" errors, particularly for architectures with high per-query LLM invocation counts. Such errors must be classified as technical faults rather than agent failures and must not be counted in correctness rate calculations. The inter-test pause and retry mechanisms implemented in the evaluation script mitigate but do not eliminate this risk.

Chapter 5

Results and Discussion

5.1 Overview of Evaluation Results

This chapter presents and interprets the empirical results obtained by running a test set comprising 89 queries (58 original + 31 paraphrase variants, spanning 17 query families) across six configurations: three architectures (Simple Agent, Planner-Critic, Context-Specific Multi-Agent) paired with two base language models (`gpt-3.5-turbo` and `gpt-4o-mini`).

The main evaluation metric is the correctness score, a continuous value in the range [0, 1], assigned by the evaluator. However, process-level indicators (tool call counts, LLM invocations, review cycles, judge interruption decisions) and robustness to paraphrase variation are treated as first-order dimensions throughout the entire process. Table 5.1 consolidates the correctness scores measured across all configurations.

Table 5.1: Correctness scores by architecture and model

Architecture	**Model**	**Original queries** ($n = 58$)	**Full test set** ($n = 89$)
Simple Agent	`gpt-3.5-turbo`	0.872	0.891
Simple Agent	`gpt-4o-mini`	0.956	0.972
Planner-Critic	`gpt-3.5-turbo`	0.960	0.959
Planner-Critic	`gpt-4o-mini`	0.862	0.888
Context-Specific	`gpt-3.5-turbo`	0.968	0.974
Context-Specific	`gpt-4o-mini`	0.976	0.973

When analysing this table, two main findings immediately stand out. Firstly, an increase in model capacity does not necessarily imply a proportional improvement in correctness. The relationship between model choice and architectural performance is not monotonic. The upgrade from `gpt-3.5-turbo` to `gpt-4o-mini` clearly improves the Simple Agent ($+8.4$ percentage points on the original queries), produces only a small change in the Context-Specific architecture ($+0.8$ percentage points on the original queries, but -0.1 percentage points on the full test set),

and reverses sharply for the Planner-Critic architecture (−9.8 percentage points on the original queries), where the more capable model appears to destabilise the planner-critic feedback loop. Secondly, even the worst-performing configuration, Planner-Critic `gpt-4o-mini` with 0.862 on the original queries, achieved a substantial absolute correctness score, reflecting the tractability of the evaluated domain. Thus, the differentiation between architectures lies primarily in the type and distribution of errors, rather than merely in raw correctness.

5.2 Response Correctness: Architecture-Level Failure Analysis

5.2.1 Quantitative Comparison of All Architectures on Original Queries ($n = 58$)

Table 5.2 summarises the distribution of correctness labels across all six configurations for the 58 original queries. Table 5.3 then reports the corresponding descriptive statistics.

Table 5.2: Outcome distribution by architecture and model on the original queries

Architecture	**Model**	**Correct rate**	**Partially correct rate**	**Incorrect rate**
Simple Agent	`gpt-3.5-turbo`	0.828	0.069	0.103
Simple Agent	`gpt-4o-mini`	0.931	0.034	0.069
Planner-Critic	`gpt-3.5-turbo`	0.931	0.052	0.017
Planner-Critic	`gpt-4o-mini`	0.862	0.000	0.138
Context-Specific	`gpt-3.5-turbo`	0.931	0.069	0.000
Context-Specific	`gpt-4o-mini`	0.931	0.069	0.000

Table 5.3: Descriptive correctness statistics by architecture and model on the original queries

Architecture	**Model**	**Mean correctness**	**Median**	**Std. deviation**
Simple Agent	`gpt-3.5-turbo`	0.872	1.000	0.312
Simple Agent	`gpt-4o-mini`	0.956	1.000	0.189
Planner-Critic	`gpt-3.5-turbo`	0.960	1.000	0.163
Planner-Critic	`gpt-4o-mini`	0.862	1.000	0.348
Context-Specific	`gpt-3.5-turbo`	0.968	1.000	0.119
Context-Specific	`gpt-4o-mini`	0.976	1.000	0.101

Based on the results presented in Tables 5.2 and 5.3, several patterns become apparent. First, all six configurations share a median correctness of 1.000, confirming that most queries are handled perfectly regardless of architecture or model, with the difference lying in the tail of failures rather than in the central trend. Second, the Simple Agent with `gpt-3.5-turbo` is the weakest and most variable `gpt-3.5-turbo` configuration, with a mean correctness of 0.872,

a standard deviation of 0.312, and the highest incorrect rate among the `gpt-3.5-turbo` configurations (0.103). Third, the Planner-Critic architecture with `gpt-4o-mini` exhibits the most severe failure pattern: its incorrect rate (0.138) is the highest among all configurations, and its partially correct rate drops to zero, meaning that all its failures are complete failures. This reflects execution-cycle interruptions that produce no usable response. Finally, the Context-Specific architecture is the most consistent across both models, achieving the lowest standard deviation in both configurations: 0.119 with `gpt-3.5-turbo` and 0.101 with `gpt-4o-mini`. It is also the only architecture that produced no fully incorrect responses in either model configuration, with its failures limited to partially correct cases. This suggests that domain-specific routing and reduced tool exposure helped contain failures, even when the system did not fully complete a task. The following subsections examine the specific failure modes that determine these distributions for each architecture.

5.2.2 Simple Agent

This section analyses the correctness of the Simple Agent architecture across the two evaluated models. The objective is not only to compare aggregate correctness scores, but also to identify the recurring failure patterns that explain the remaining errors. Since the Simple Agent relies on a single agent loop with access to all tools, its correctness depends heavily on the model's ability to select the appropriate tool, provide valid arguments, interpret tool results, and decide when the user request has been fully satisfied.

Table 5.4 presents the quantitative comparison of Simple Agent correctness on the original queries.

Table 5.4: Quantitative comparison of Simple Agent correctness on the original queries

Model	**Correct rate**	**Partially correct rate**	**Incorrect rate**	**Mean correctness**	**Median**	**Std. deviation**
`gpt-3.5-turbo`	0.828	0.069	0.103	0.872	1.000	0.312
`gpt-4o-mini`	0.931	0.034	0.069	0.956	1.000	0.189

Table 5.4 summarises the correctness results of the Simple Agent across the 58 original queries. `gpt-3.5-turbo` achieved 48 fully correct answers, 4 partially correct answers, and 6 incorrect answers, corresponding to a mean correctness score of 0.872. `gpt-4o-mini` improved this result, reaching 54 fully correct answers, 2 partially correct answers, and 2 incorrect answers, with a mean correctness score of 0.956. Both configurations have a median correctness score of 1.000, indicating that the typical query was answered correctly in both cases. However, the lower standard deviation of `gpt-4o-mini`, 0.189 compared with 0.312 for `gpt-3.5-turbo`, shows that its correctness was more stable across the evaluation set. Therefore, `gpt-4o-mini` did not merely improve the average score, it also reduced the frequency and severity of failure cases.

The tool-level annotations provide further insight into the correctness differences between the two Simple Agent configurations. `gpt-3.5-turbo` produced more failures related to tool argument construction, with 9 wrong-tool-argument annotations and 2 wrong-tool-selection annotations across the original queries. `gpt-4o-mini` reduced these errors, with 6 wrong-tool-argument annotations and 1 wrong-tool-selection annotation, but produced more retry-related behaviour, with 9 retry annotations. This suggests that `gpt-4o-mini` was generally more successful at reaching correct answers, but often did so through more exploratory or corrective tool use. In contrast, `gpt-3.5-turbo` more frequently failed because the initial tool call or its arguments did not correctly represent the user's intent.

Despite the higher correctness of `gpt-4o-mini`, the error analysis shows that the Simple Agent remains vulnerable to four main failure patterns: category filter misapplication, tool-result validation failure, hallucination or unauthorized substitution, and incomplete task execution.

The first failure pattern concerns category filter misapplication, where the model failed to translate the user's implicit product type into the correct tool arguments. In Q16, *"phones under $1000"*, `gpt-3.5-turbo` omitted the Smartphone category constraint and returned products such as the HomePod Mini, Lenovo Tab P11 Pro, and Dell Inspiron Desktop alongside actual smartphones. In Q53, the same behaviour can be observed. These cases show that the model's failures were often caused by incorrect argument construction rather than by missing catalogue data.

The second pattern is tool-result validation failure. Q52, *"laptops under $500"*, is particularly relevant because `gpt-3.5-turbo` called `filter_products` with the correct high-level constraints, including `category=Laptop` and `max_price=500`. However, the tool returned the Lenovo Tab P11 Pro, which is a tablet, and the agent accepted the result instead of detecting the category mismatch. A related boundary issue appears in Q18, where GPT-3.5-turbo returned the HP Omen 45L Gaming Desktop together with the MacBook Pro 14" M3 Pro in response to *"which laptops are over $2000?"*. `gpt-4o-mini` handled both situations correctly, identifying and filtering the products received against the order placed, generating a correct final response. This shows that correctness depends not only on tool selection and argument construction, but also on validating tool outputs before producing the final answer.

The third pattern concerns hallucination or unauthorized substitution in non-existent product queries. In Q23, the user asked to add a Nokia phone to the cart, but no Nokia phone existed in the catalogue. `gpt-3.5-turbo` substituted a Samsung Galaxy S24 and reported that it had been added, without clearly informing the user that the requested product was unavailable. Q54 shows a similar issue. `gpt-4o-mini` improved this behaviour by refusing Q54 and avoiding product fabrication in Q23.

The fourth pattern is incomplete task execution. In Q25, *"add five random products"*, `gpt-3.5-turbo` retrieved and listed five products but did not execute the cart-creation step. This indicates that the Simple Agent could sometimes gather the information required to satisfy the request but fail to complete the final transactional action.

Although `gpt-4o-mini` corrected most of the flaws from `gpt-3.5-turbo`, a smaller set

of correction issues still persisted. In Q9, *"show me Dell computers"*, it included a Dell monitor alongside Dell laptops and a Dell desktop computer, thus maintaining the flaw caused by the model's failure to analyse the tool's results. In Q22, *"create a shopping basket with three cheap products"*, it failed despite `gpt-3.5-turbo` having resolved the same query correctly, because it issued several excessively restrictive price range filters that returned empty results and concluded that no products were available. In Q36, *"find a cheap way to watch Netflix"*, it selected the information tool and generated generic advice on saving on the subscription, rather than consulting the catalogue to find a relevant product, such as a streaming device.

Overall, the Simple Agent achieves a high level of correctness when the query directly corresponds to a known product, stock check, or simple order management task. However, its errors reveal the limitations of relying on a single agent cycle without explicit validation. `gpt-3.5-turbo` is primarily affected by incorrect tool arguments, misinterpretation of categories, lack of validation of tool results, unauthorised substitutions, and incomplete transactional execution. `gpt-4o-mini` improves both average correctness and consistency, but its remaining shortcomings show that greater model capacity alone does not fully resolve issues with tool routing, category boundaries, or multi-step selection.

5.2.3 Planner-Critic

This section analyses the correctness of the Planner-Critic architecture across the two evaluated models. The objective is not only to compare aggregate correctness scores, but also to examine whether the addition of an explicit critic improved the reliability of the agent's decisions. In this architecture, correctness depends on two stages: first, the planner must select an appropriate tool and construct valid arguments, and second, the critic must evaluate whether that plan is sufficient before execution. Therefore, the analysis focuses not only on final-answer correctness, but also on planner errors, critic misjudgements, retries, and cases where the critic either failed to detect an incomplete plan or rejected a valid one.

The Planner-Critic architecture achieved a correctness score of 0.960 with `gpt-3.5-turbo` on the original queries. According to the comparative results, this placed it above the Simple Agent configuration, but slightly below the Context-Specific Agents architecture. Its failures were limited to Q5, Q9, Q18, and Q34, and were mainly related to constraint preservation rather than complete task misunderstanding. For example, in Q34, the user asked for *"Apple laptops under $1500"*, but the system retrieved Apple products under $1500 without preserving the laptop constraint. This suggests that the critic did not consistently detect semantically incomplete tool arguments.

To complement the qualitative analysis, the Planner-Critic runs were also compared using correctness distribution and execution-level error metrics. Table 5.5 summarises the correctness results, while Table 5.6 reports the retries and error annotations observed on the original queries.

Table 5.5: Quantitative comparison of Planner-Critic correctness on the original queries

Model	Correct rate	Partially correct rate	Incorrect rate	Mean correctness	Median	Std. deviation
`gpt-3.5-turbo`	0.931	0.052	0.017	0.960	1.000	0.163
`gpt-4o-mini`	0.862	0.000	0.138	0.862	1.000	0.348

Table 5.6: Retries and error patterns for Planner-Critic on the original queries

Model	Retries	Wrong tool selection	Wrong argument selection	Wrong revise decisions
`gpt-3.5-turbo`	0	2	5	0
`gpt-4o-mini`	2	4	8	34

Both tables show that both Planner-Critic configurations had a median correctness of 1.000, meaning that the typical query was answered correctly in both cases. However, `gpt-4o-mini` produced fewer fully correct answers and more incorrect executions than `gpt-3.5-turbo`. Its higher standard deviation also indicates a more polarised distribution, where executions tended to be either fully correct or entirely unsuccessful. The diagnostic metrics further explain this difference. `gpt-3.5-turbo` recorded no retries, two wrong-tool-selection cases, five wrong-argument-selection cases, and no wrong revise decisions. `gpt-4o-mini`, by contrast, recorded two retries, four wrong-tool-selection cases, eight wrong-argument-selection cases, and 34 wrong revise decisions. Therefore, the main instability of `gpt-4o-mini` was not caused by retries, but by critic-level misjudgements. The critic was substantially more likely to reject executable or recoverable plans, which contributed to failed executions even when the planner had selected a plausible direction.

The `gpt-4o-mini` configuration produced a less stable result, with a correctness score of 0.862 on the original queries. Its failed queries were Q22, Q23, Q25, Q35, Q36, Q42, Q52, and Q58. Six of these failures produced no final answer, mainly because the planner and critic entered repeated revision cycles or reached the attempt limit before executing a successful plan. Q25 illustrates this behaviour: the user requested five random products, but the critic repeatedly rejected `filter_products` with `limit=5` because no category or keyword was provided, even though the request was intentionally broad.

Two `gpt-4o-mini` failures produced final answers but were incorrect. In Q35, the system searched using category *"Gaming"* and the critic approved the plan, although *"Gaming"* was not a valid category in the dataset. The correct strategy would have been to search by gaming-related keywords, since relevant products existed under other categories. In Q36, the system interpreted *"a cheap way to watch Netflix"* as a campaign query and selected `available_campaigns`, rather than searching for low-cost streaming-related products.

Overall, these results show that the Planner-Critic architecture can improve reliability when the critic correctly detects genuinely incomplete plans, as seen in the stronger `gpt-3.5-turbo` result. However, the architecture also introduces a new failure mode: when the critic's validation criteria are overly strict, inconsistent, or misaligned with the actual tool interface, the planner-critic loop can prevent execution altogether. The comparison between `gpt-3.5-turbo` and `gpt-4o-mini` therefore suggests that adding a critic does not automatically improve robustness. Its benefit depends on the critic's ability to distinguish between genuinely unsafe or incomplete plans and plans that are already executable and sufficient.

5.2.4 Context-Specific Multi-Agent

This section analyses the performance of the Context-Specific Agent architecture in the two models evaluated, using the original set of 58 queries. Unlike the Simple Agent, this architecture does not assign all tools to a single agent. Instead, the router assigns the request to a specialised context agent, and the Stop Judge checks whether the workflow followed up to that point should continue or be terminated. Consequently, correctness depends not only on tool selection and argument construction, but also on the quality of routing, handover between agents, and the Stop Judge's ability to determine whether the user's request has been effectively completed.

Table 5.7 summarises the correctness scores for the original 58-query set.

Table 5.7: Quantitative comparison of Context-Specific correctness on the original queries

Model	**Correct rate**	**Partially correct rate**	**Incorrect rate**	**Mean correctness**	**Median**	**Std. deviation**
`gpt-3.5-turbo`	0.931	0.069	0.000	0.968	1.000	0.119
`gpt-4o-mini`	0.931	0.069	0.000	0.976	1.000	0.101

Table 5.8 summarises the main error indicators for the original 58-query set.

Table 5.8: Error indicators for Context-Specific on the original queries

Model	**Wrong tool selection**	**Wrong argument selection**	**Retries**	**Wrong `done=False` decisions**	**Wrong `done=True` decisions**
`gpt-3.5-turbo`	3	8	1	5	2
`gpt-4o-mini`	3	8	2	4	2

The results show that both models achieved a high level of correctness, with a median score of 1.000 in both cases. `gpt-4o-mini` achieved a slightly higher mean score, increasing from 0.968 to 0.976, and also exhibited a lower standard deviation, decreasing from 0.119 to 0.101. This indicates that `gpt-4o-mini` was marginally more stable across the original query set. However, the error metrics show that this improvement was not caused by the elimination of tool-related issues.

Both models recorded the same number of wrong tool selections and wrong argument selections, and both had two incorrect `done=True` decisions. Therefore, the improvement is better interpreted as a small increase in scoring quality rather than a major reduction in architectural failure modes.

The most significant remaining failure pattern is premature termination in transactional requests. In Q24, *"Add all tablets to my cart"*, both models correctly identified the relevant tablet products, including the Lenovo Tab P11 Pro, the Samsung Galaxy Tab S9, and the iPad Pro 11. However, the workflow stopped after product selection and did not execute the required `create_cart` tool call. A similar issue occurred in both models in Q26, *"Create a cart with one laptop and one monitor"*, where the system searched for and selected the requested products but again terminated before executing `create_cart`. These cases show that the architecture can correctly handle the product-selection part of an order while still failing to verify whether the required transactional action was actually performed.

A second recurring pattern is the failure to distinguish between semantic categories. In Q9, *"Show me Dell computers"*, both models included the Dell UltraSharp monitor in their response, even though the user had asked for computers. `gpt-3.5-turbo` also exhibited a similar issue in Q18. These failures suggest that the specific structure of the context reduces the available tool space for each agent, but does not entirely prevent errors arising from the incorrect use of tool arguments or when the results of those tools are not sufficiently constrained by the category.

The architecture also showed clear strengths. In Q22, *"Create a cart with three cheap products"*, `gpt-3.5-turbo` correctly decomposed the task into product discovery followed by cart creation. It selected the three cheapest products returned by the catalogue and then executed `create_cart`. This case illustrates the benefit of separating presales reasoning from cart execution: the system first identified suitable products and then transferred the selected items to the cart agent to complete the transaction.

Overall, the Context-Specific Agents architecture achieved solid results in terms of correctness and benefited from domain-specific routing and reduced exposure to tools. Its remaining weaknesses are concentrated in three areas: semantic filtering of tool results, verification of the transactional execution of tools, and Stop Judge termination decisions. Consequently, the architecture improves reliability compared to less structured agent designs, but still requires more rigorous completion checks for the patterns explained above.

5.3 Paraphrase Invariance

Paraphrase invariance captures whether the system produces consistent outcomes when the same query intent is expressed through different surface forms. Across the 17 query families that include at least one variant, a family is classified as inconsistent when the correctness scores of its members differ meaningfully, that is, when at least one member is answered correctly and at least one related variant partially or fully fails.

Table 5.9: Paraphrase invariance by configuration

	`gpt-3.5-turbo`		`gpt-4o-mini`	
Architecture	**Inconsistent families**	**Inconsistency rate**	**Inconsistent families**	**Inconsistency rate**
Simple Agent	5 / 17	29.4%	0 / 17	0.0%
Planner-Critic	3 / 17	17.6%	3 / 17	17.6%
Context-Specific	2 / 17	11.8%	1 / 17	5.9%

5.3.1 Simple Agent

For the Simple Agent, `gpt-3.5-turbo` showed the highest paraphrase sensitivity. The inconsistent families were Q1, Q2, Q16, Q46, and Q52. These cases show that small changes in wording could alter product resolution, category interpretation, or tool selection.

The Q1 and Q2 families reveal sensitivity to product-name phrasing. In Q1, the original query and Q1-A correctly retrieved the MacBook Air M3, while Q1-B failed to resolve the reordered product name directly and returned multiple close matches instead. This pattern was also seen in the Q2 family. In both cases, the product existed in the catalogue, but the paraphrased wording reduced retrieval precision.

Q16 shows category-level paraphrase sensitivity. The original query, *"Show me phones under $1000"*, was only partially correct because `gpt-3.5-turbo` omitted the Smartphone category constraint and returned non-phone products alongside smartphones. However, Q16-A and Q16-B used the term *"smartphones"* and were answered correctly. This suggests that the more explicit category wording helped the model preserve the intended product type.

Q46 produced a clear tool-selection inconsistency. The original query and Q46-B correctly used `available_campaigns` to answer questions about Apple discounts or deals. However, Q46-A was routed to product filtering instead of campaign retrieval.

Q52 highlights sensitivity around category and price boundaries. The original query, *"Show me laptops under $500"*, failed because the tool returned the Lenovo Tab P11 Pro, a tablet, and the agent accepted it as a laptop result. In contrast, Q52-A and Q52-B correctly reported that no laptops were available under the specified price thresholds, suggesting that the alternative phrasings led to more reliable category validation.

In comparison, the Simple Agent using `gpt-4o-mini` produced 0 inconsistent families out of 17. According to the score-based definition used in this analysis, all families of paraphrases produced equivalent correction results. This result is significant because the architecture remained unchanged and only the model differed. Therefore, within this experimental setup, the stronger linguistic understanding and tool-using behaviour of `gpt-4o-mini` were sufficient to eliminate paraphrase inconsistency at the scoring level for the Simple Agent. However, this result should not be interpreted as absolute proof that the model's capability alone solved sensitivity to paraphrases. Rather, it shows that, on this specific dataset, the more robust model compensated for the observed

weaknesses, particularly in the normalisation of product names, the interpretation of categories, and the selection of campaign-related tools.

5.3.2 Planner-Critic

The Planner-Critic architecture appears to offer partial protection against paraphrase invariance, but does not eliminate sensitivity to query formulation. With `gpt-3.5-turbo`, three of the 17 paraphrase families showed inconsistencies: Q1, Q2, and Q46. The Q1-B and Q2-B variants reproduced the degradation in similarity search observed in the Simple Agent, where paraphrased product names led to broader or less accurate retrieval. Q46-A, however, raised a different issue, as the system treated a query for information about iPhone campaigns as a product search. In these cases, the evaluator approved plausible plans but failed to detect that the selected strategy or arguments were semantically misaligned with the user's intent.

With `gpt-4o-mini`, three families were also inconsistent: Q15, Q42, and Q52. These inconsistencies were mainly caused by critic-loop behaviour rather than by ordinary retrieval errors. In Q15, the original Nokia availability query completed correctly, while the paraphrased variants entered repeated revision cycles and produced no final answer. Similarly, Q42 and Q52 show that small wording changes could determine whether the planner-critic loop reached a valid execution or failed through over-revision.

Overall, the Planner-Critic architecture improved the stability of paraphrasing compared to the `gpt-3.5-turbo` Simple Agent configuration, but introduced a distinct risk of non-invariance: incorrect calibration of the critic. When the critic is too permissive, it approves semantically incomplete plans, however, when it is too strict, it may reject executable plans and cause the cycle to fail. Consequently, the results suggest that the robustness of the architecture's paraphrasing depends not only on the planner's ability to interpret equivalent user intentions, but also on the critic's ability to evaluate plans consistently across variations in surface form.

5.3.3 Context-Specific Multi-Agent

The Context-Specific architecture showed strong paraphrase robustness, with only two inconsistent families for `gpt-3.5-turbo` and one for `gpt-4o-mini`. In `gpt-3.5-turbo`, the Q3 family exposed a near-match substitution problem: while Q3 and Q3-A correctly reported that the Samsung Galaxy S25 was unavailable, Q3-B returned the price of the Samsung Galaxy S24 as if it answered the S25 price request. The second inconsistency was Q24, where the original query identified the tablets but failed to execute the required cart-creation step, while its variants completed the task correctly.

For `gpt-4o-mini`, the only inconsistent family was Q42. The original query and Q42-A completed the refund process successfully, whereas Q42-B produced a loop or timeout instead of reaching a valid completion state. This indicates that the remaining paraphrase failure was caused by workflow-control instability rather than by product-search or tool-selection confusion.

In general, context-specific design appears to improve invariance to paraphrases by routing queries to specialised agents with more restricted tool sets and clearer responsibilities. This seemingly reduces the likelihood that minor changes in phrasing will trigger a different tool strategy. However, the remaining failures show that specialisation does not entirely eliminate sensitivity to paraphrases, as inconsistencies may still arise from the substitution of near-matches, incomplete transaction execution, or incorrect workflow termination.

5.4 Process-Level Metrics

5.4.1 Number of Tool Calls per Query

The number of tool calls per query reflects the depth of information retrieval required to satisfy each request. As shown in Table 5.10, the averages are relatively close across architectures and models, ranging from 1.22 to 1.52 calls per query. This suggests that the architectures relied on a broadly similar level of tool usage for the original query set.

Table 5.10: Average tool calls per original query

Architecture	**Model**	**Mean tool calls**	**Max**
Simple Agent	`gpt-3.5-turbo`	1.31	5 (Q55)
Simple Agent	`gpt-4o-mini`	1.52	5 (Q24, Q38, Q55)
Planner-Critic	`gpt-3.5-turbo`	1.36	5 (Q55)
Planner-Critic	`gpt-4o-mini`	1.22	3 (Q21, Q26, Q38, Q55)
Context-Specific	`gpt-3.5-turbo`	1.40	5 (Q38)
Context-Specific	`gpt-4o-mini`	1.50	6 (Q55)

Most queries required only one tool call, especially direct product searches, stock checks, and information requests. Higher values were mainly caused by multi-step tasks, such as selecting several products before creating a cart or combining product search with stock verification.

The Simple Agent increased from 1.31 tool calls with `gpt-3.5-turbo` to 1.52 with `gpt-4o-mini`. The logs suggest that `gpt-4o-mini` more often performed additional verification steps, such as checking stock before making a recommendation, whereas `gpt-3.5-turbo` tended to answer after fewer interactions.

The Planner-Critic architecture shows a different pattern. While `gpt-3.5-turbo` remains close to the Simple Agent baseline at 1.36 calls, `gpt-4o-mini` drops to 1.22, the lowest value in the table. However, this should not be interpreted simply as higher efficiency, since some failed or interrupted reasoning loops prevented the system from reaching the expected tool execution stage.

The Context-Specific architecture remains within the same range, with 1.40 calls for `gpt-3.5-turbo` and 1.50 for `gpt-4o-mini`. Its higher values are mainly linked to specialist agents performing several search or filtering steps before completing actions such as cart creation, reflecting a more deliberate but also more complex retrieval process.

Overall, tool-call frequency is useful but insufficient on its own. Fewer calls may indicate efficiency, but also premature termination. More calls may indicate overhead, but also useful verification. Therefore, this metric must be interpreted together with correctness, failure modes, and execution traces.

5.4.2 Number of LLM Invocations per Query

LLM invocation counts complement tool-call metrics because they capture the internal orchestration cost of each architecture. While tool-call averages are relatively similar, the architectures differ substantially in how many model calls are required to process a query.

The Simple Agent is the lightest design, typically requiring one LLM call to decide the tool action and another to generate the final answer. The Planner-Critic adds a fixed overhead by requiring both planning and critic validation before execution. The Context-Specific architecture has the highest structural cost, since each query normally passes through routing, a specialist agent, stop-judge validation, and final response generation.

Therefore, this metric is useful mainly as an explanation for later cost results: similar tool-call counts do not imply similar computational cost. More complex architectures introduce additional LLM invocations to support validation, routing, and answer synthesis, which helps explain differences in token consumption and latency.

5.4.3 Critic Revision Cycles (Planner-Critic Architecture)

A distinctive feature of the Planner-Critic architecture is its capacity for iterative plan refinement: the critic may reject a proposed plan, requiring the planner to revise and resubmit it until approval or until the maximum iteration limit is reached.

In the case of `gpt-3.5-turbo`, the review process was nearly non-existent. Of the 89 queries evaluated, only Q16 triggered a review cycle, with two rejections by the reviewer before approval. This indicates that the reviewer accepted the planner's initial strategy in almost all cases. Although this prevented loop failures, it also suggests that the critic was relatively lenient. Upon observing this behaviour, it is evident that the critic failed to consistently detect subtle planning errors, such as the incomplete or semantically misaligned plans that led to failures in queries Q5, Q9, Q18, and Q34.

With `gpt-4o-mini`, however, this architecture exhibited the opposite pattern. The critic proved to be far more interventionist, and several failures were caused by unresolved revision cycles between the planner and the critic. Six original queries failed without producing a final response: Q22, Q23, Q25, Q42, Q52, and Q58, and the same occurred in variants Q15-A and Q15-B. These failures typically involved non-existent products, empty result sets, or multi-step transactional tasks. For example, in Q25, the user requested five random products, but the critic repeatedly rejected a broad call to `filter_products` because a category or keyword was missing, even though the request itself was intentionally broad.

This behaviour highlights a fundamental trade-off in the Planner-Critic architecture. A permissive critic, such as in `gpt-3.5-turbo`, may fail to catch subtle errors in the tools or arguments, while an overly aggressive critic, such as in `gpt-4o-mini`, may reject feasible plans and prevent the system from reaching a final answer. Given this, the value of the critic should not be assessed solely by the idea of having an extra validation step, it actually depends not only on its ability to identify weak plans, but also on its ability to avoid unnecessary or inappropriate revisions.

5.4.4 Stop Judge Decisions (Context-Specific Architecture)

In the Context-Specific architecture, the Stop Judge evaluates whether the current specialist output is sufficient for final answer generation or whether the workflow should continue. In the original 58-query set, both model configurations received a positive Stop Judge decision on the first evaluation in 53 out of 58 queries, corresponding to 91.4%. This indicates that, in most cases, a single specialist response was judged sufficient to answer the user request.

However, Stop Judge decisions should be distinguished from the number of specialist agents dispatched. In `gpt-3.5-turbo`, five queries involved two specialist agents: Q21, Q22, Q25, Q38, and Q55. In the `gpt-4o-mini` configuration, this occurred in four queries: Q21, Q25, Q38, and Q55. These cases were mainly multi-step transactional requests, where product identification had to be followed by cart creation. Therefore, the additional specialist dispatches generally reflect intentional task decomposition rather than error correction.

The queries that initially received a `done=False` decision were Q5, Q12, Q52, Q53, and Q57 in both configurations. These cases differed from multi-agent transactional flows, as they reflected situations in which the expert's first response was incomplete, ambiguous, or insufficiently aligned with the request. Overall, the results show that the Stop Judge typically terminated the workflow after an evaluation. When it continued, the continuation either supported a legitimate decomposition of the task into multiple steps or reflected an insufficient initial response from the expert. However, cases Q24 and Q26 show that the Stop Judge could still accept partially completed transactional workflows when product selection occurred but cart creation was not executed. A related limitation also emerged in the set of paraphrases, where Q42-B caused a loop until the timeout, showing that the Stop Judge could also fail in the opposite direction by continuing the workflow for too long.

5.4.5 Response Latency

Response latency was measured as the overall execution time required to complete each query, including model invocations, tool execution, routing or critic steps where applicable, and final response generation. Table 5.11 summarises the mean, median, standard deviation, and maximum latency observed for each architecture and model configuration on the original query set.

Table 5.11: Response latency by architecture and model on the original queries

Architecture	**Model**	**Mean overall time**	**Median**	**Std. deviation**	**Max**
Simple Agent	`gpt-3.5-turbo`	3.75s	3.36s	1.40s	8.89s (Q1)
Simple Agent	`gpt-4o-mini`	6.16s	5.74s	3.39s	23.26s (Q6)
Planner-Critic	`gpt-3.5-turbo`	4.57s	3.98s	1.92s	11.50s (Q55)
Planner-Critic	`gpt-4o-mini`	8.66s	7.74s	5.93s	38.69s (Q22)
Context-Specific	`gpt-3.5-turbo`	6.08s	5.35s	2.13s	13.10s (Q5)
Context-Specific	`gpt-4o-mini`	10.30s	8.25s	5.80s	36.74s (Q54)

The latency results show a clear trade-off between speed and model capability in the Simple Agent architecture. The configuration using `gpt-3.5-turbo` was the fastest, with an average total time of 3.75s, a median of 3.36s, and a standard deviation of 1.40s across the 58 original queries. In contrast, `gpt-4o-mini` was consistently slower and more variable, with a mean of 6.16s, a median of 5.74s, and a standard deviation of 3.39s. This suggests that `gpt-4o-mini` did not merely produce occasional outliers. Its typical response time was also higher, as demonstrated by the median difference exceeding two seconds. Interestingly, the cases of higher latency were not necessarily caused by the number of tool calls. For `gpt-3.5-turbo`, the slowest query was Q1, which took 8.89s despite requiring only one tool call, while the query that used the most tools, Q55, required five tool calls but remained slightly faster at 7.67s. The same pattern is most evident for `gpt-4o-mini` in Q6, which took 23.26s with just a single tool call. This suggests that the main cost stemmed from generating a long, detailed response covering multiple catalogue items, rather than from tool execution itself. Therefore, although `gpt-4o-mini` improved correctness in the Simple Agent configuration, this improvement was accompanied by higher and less predictable latency. The results suggest that, in this architecture, response time is influenced not only by the depth of tool usage but also by the model's tendency to produce richer and more elaborate results.

The Planner-Critic architecture's iterative plan refinement mechanism behaved very differently in the two models. In the case of `gpt-3.5-turbo`, revision behaviour was practically nonexistent, as analysed in the previous section, and this behaviour, mostly single-pass, was reflected in the latency results for the original queries, where this configuration achieved an overall average time of 4.57s, a median of 3.98s, a standard deviation of 1.92s, and a maximum of 11.50s in Q55. In contrast, the `gpt-4o-mini` Planner-Critic exhibited the opposite pattern: the critic was more interventionist, and this directly increased latency, with an overall mean time of 8.66s, a median of 7.74s, a standard deviation of 5.93s, and a maximum of 38.69s on Q22. Overall, these results highlight a key trade-off: a permissive evaluator may fail to detect subtle planning errors, rushing the time needed to correctly respond to the request, while an overactive evaluator may reject executable plans, increasing latency.

The Context-Specific architecture incurs a significant latency cost, since a typical query requires sequential calls to the router, one or more specialised agents, the Stop Judge, and the final

response generator. In the original set of 58 queries, `gpt-3.5-turbo` recorded an average of 6.08s per query, with a median of 5.35s and a standard deviation of 2.13s. In contrast, the configuration with `gpt-4o-mini` was slower and more variable, with an average of 10.30s, a median of 8.25s, and a standard deviation of 5.80s. The maximum latency was 13.10s for `gpt-3.5-turbo` in Q5 and 36.74s for `gpt-4o-mini` in Q54. Multi-step transactional queries also produced high response times, especially when product identification had to be followed by cart creation, as seen in questions Q38 and Q55. These results show that the architecture's additional coordination and verification mechanisms can improve task structuring, but their cost ultimately translates significantly into response time as well, which constitutes an important limitation to consider for customer service scenarios where the user experience depends on rapid interactions.

5.4.6 Token Consumption

Token consumption differs substantially among the three architectures, as each project structures interaction with the LLM differently. The Simple Agent architecture is the most token-efficient, as it uses a single agent cycle in which the system prompt, conversation context, tool calls, and tool results are handled within the same interaction flow. In contrast, the Planner-Critic architecture introduces an additional prompt overhead by separating planning and validation into distinct calls to the LLM. The Context-Specific architecture, on the other hand, consequently increases token usage because each query typically passes through multiple components of the LLM, each with its own prompt context.

This difference is visible in the average token consumption across the original query set. With `gpt-3.5-turbo`, the Simple Agent consumed an average of 2,068 tokens per query, compared with 5,047 tokens for Planner-Critic and 4,609 tokens for Context-Specific. This corresponds to approximately 2.44× and 2.23× the Simple Agent baseline, respectively. With `gpt-4o-mini`, the Simple Agent consumed an average of 1,989 tokens per query, while Planner-Critic reached 7,828 tokens and Context-Specific reached 5,337 tokens. In this case, the overhead increased to approximately 3.94× for Planner-Critic and 2.68× for Context-Specific.

Overall, these results show that orchestrated architectures require a substantially higher number of tokens than the Simple Agent, although the magnitude of the overhead varies depending on the model and architecture. This overhead is not necessarily a drawback, as it enables additional reasoning, validation, routing, and response synthesis. However, it has direct implications for production deployment, where token consumption affects both API cost and scalability. Consequently, the additional cost of the Planner-Critic and Context-Specific architectures must be interpreted in conjunction with their benefits in terms of correctness, robustness, and error reduction.

5.5 Answers to the Research Questions

RQ1. How does agent architecture affect the reliability of tool-augmented conversational AI agents in e-commerce customer service tasks?

The results show that architecture has a clear impact on reliability. Although all configurations achieved a median correctness of 1.000 on the original query set, their average scores, error distributions, and failure patterns differed substantially. This indicates that most queries were handled correctly by all architectures, but they differed in the frequency and severity of the errors made.

Simple Agent was the one that demonstrated the greatest dependence on the base model in terms of correctness. With `gpt-3.5-turbo`, it produced the lowest average correctness among the model configurations, primarily due to incorrect tool arguments, poor category filtering, insufficient validation of tool results, hallucinated substitutions, and incomplete task execution. With `gpt-4o-mini`, its performance improved substantially, but the architecture remained structurally vulnerable, as it lacked explicit validation, domain partitioning, and evaluation of task completion.

The Planner-Critic architecture improved reliability with `gpt-3.5-turbo` compared to a simple architecture, suggesting that pre-execution validation can reduce some planning and tool-usage errors. However, its performance with `gpt-4o-mini` demonstrated that this architecture is highly sensitive to the critic's behaviour. When the critic became too strict or misaligned with the tool interface, it rejected viable plans and triggered revision cycles that prevented execution.

The Context-Specific architecture achieved the most consistent reliability profile. By routing requests to specialised agents and restricting each agent's access to tools, it reduced the decision space available for each model invocation. This was shown to help limit confusion between tools from different domains and supported more structured multi-step workflows. However, it did not eliminate all failures, particularly in semantic filtering, cart creation verification, and Stop Judge termination decisions.

Therefore, we can argue that architecture affects reliability not only by altering the final correctness score but also by changing the type of errors that occur. Simpler architectures are faster and easier to implement, but are more prone to uncontrolled tool selection and validation failures. More structured architectures can improve reliability by constraining behaviour, but introduce additional coordination and control risks.

RQ2. To what extent do explicit validation and self-correction mechanisms reduce tool-use errors and incomplete task execution?

The results show that explicit validation and self-correction mechanisms can improve reliability, but only when they are properly calibrated and aligned with the task structure.

In the Planner-Critic architecture, the critic was designed to validate tool plans prior to execution. With `gpt-3.5-turbo`, this mechanism contributed to a high correctness score of 0.960 on the original query set. However, the revision process was virtually absent, with only one query triggering a revision cycle throughout the entire evaluation. This suggests that the critic was relatively lenient. While it avoided excessive loops, it also meant that the critic failed to detect some semantically incomplete plans.

With `gpt-4o-mini`, the same architecture revealed the opposite problem. The critic became more interventionist, producing many incorrect revision decisions. Several failures occurred not because the planner selected an impossible direction, but because the critic repeatedly rejected

executable plans. As a result, some queries failed before producing a final response. This demonstrates that self-correction can become detrimental when the correction mechanism is too strict or poorly aligned with the available tools.

In the Context-Specific architecture, explicit completion evaluation was implemented via the Stop Judge. This mechanism generally worked as intended, with most original queries receiving a positive decision from the Stop Judge on the first evaluation. However, it also produced significant failure cases. In transactional tasks, such as adding products to a cart, the Stop Judge sometimes accepted workflows where product selection had occurred, but the creation of the cart with those products had not actually been executed. This shows that completion validation must verify not only whether the agent identified the correct information, but also whether the required action was actually performed.

It was therefore verified that explicit validation and self-correction mechanisms do indeed reduce certain reliability risks, but their implementation is not automatically beneficial. Their effectiveness depends on precise evaluation criteria, an understanding of the tool's semantics, and careful calibration of the points at which to approve, review, continue, or stop.

RQ3. How robust are the evaluated architectures to paraphrase variation?

The results regarding paraphrasing invariance reveal that robustness to variations in phrasing differs across architectures and models. The Simple Agent with `gpt-3.5-turbo` was the most sensitive to paraphrasing, with five inconsistent families out of seventeen. These inconsistencies were associated with product name resolution, category interpretation, and tool selection. In contrast, the Simple Agent with `gpt-4o-mini` did not produce inconsistent families according to the score-based definition used in the evaluation, indicating that the more robust model compensated for several linguistic and semantic weaknesses in the baseline architecture.

The Planner-Critic architecture demonstrated partial robustness to paraphrasing, however, it did not eliminate sensitivity to query formulation. With `gpt-3.5-turbo`, inconsistencies were primarily related to product resolution and tool selection errors. With `gpt-4o-mini`, inconsistencies were more strongly associated with the behaviour of the criticism cycle. In this case, minor changes in the formulation could determine whether the Planner-Critic workflow achieved a valid execution or failed due to excessive revision. This implies that robustness to paraphrases in the Planner-Critic architecture depends not only on the planner's interpretation of the query but also on the critic's consistency in evaluating equivalent intentions expressed in different ways.

The Context-Specific architecture generally demonstrated greater robustness against paraphrases, with two inconsistent families for `gpt-3.5-turbo` and one for `gpt-4o-mini`. This suggests that routing queries to specialised agents with restricted tool sets helps stabilise behaviour in the face of superficial changes in phrasing. However, the remaining inconsistencies show that specialisation does not completely eliminate paraphrase-related failures. These failures may still arise from approximate match substitutions, incomplete transactional execution, or instability in workflow control.

Overall, the results indicate that both more robust models and more structured architectures

can improve robustness against paraphrases. However, invariance against paraphrases is not guaranteed by any of these factors alone. It depends on the harmonious interaction between language understanding, retrieval behaviour, tool constraints, and workflow control.

RQ4. What are the trade-offs between reliability, latency, and token consumption across the evaluated architectures?

The results reveal a clear trade-off between architectural control and operational cost. The Simple Agent was the lightest architecture in terms of LLM calls, latency, and token consumption. Typically, it required fewer internal reasoning steps, since a single agent handled interpretation, tool usage, and the generation of the final response. This made it the most efficient design, but also the least structurally protected against tool usage and validation errors.

The Planner-Critic architecture introduced additional costs due to the separation between planning and criticism. Even when no revision cycle occurred, the architecture required additional LLM calls compared to the Simple Agent. With `gpt-4o-mini`, this cost became more significant because the critic's repeated interventions increased latency and, in some cases, prevented the successful completion of the task being executed. This demonstrates that validation mechanisms can increase reliability in some cases, but they can also increase execution time and the risk of failure when they generate unnecessary review cycles.

The Context-Specific architecture incurred the highest structural orchestration cost. A typical query passed through a router, one or more specialised agents, the Stop Judge, and the final response generator. This increased both latency and token consumption. However, this cost enabled a clearer decomposition of tasks, reduced tool exposure per agent, domain-specific reasoning, and more interpretable execution traces. The architecture thus illustrates a trade-off between reliability and cost, providing stronger structural control at the expense of slower response times and higher token usage.

Token consumption reinforced this conclusion. With `gpt-3.5-turbo`, Simple Agent consumed an average of 2,068 tokens per query, compared to 5,047 for Planner-Critic and 4,609 for Context-Specific. With `gpt-4o-mini`, Simple Agent consumed an average of 1,989 tokens per query, while Planner-Critic reached 7,828 and Context-Specific reached 5,337. These results show that orchestrated architectures can require two to four times more tokens than the baseline, depending on the model and architecture.

Therefore, the reliability benefits of more structured architectures must be weighed against their inherent operational costs. In production customer service systems, this trade-off is particularly important, as API response time and cost directly impact scalability and the user experience.

RQ5. How does changing the base language model affect each architecture?

The change in the base model had diverse and specific effects on each architecture. The transition from `gpt-3.5-turbo` to `gpt-4o-mini` did not result in a uniform improvement across the three architectures.

Since each query was executed once per architecture and model configuration, these results should be interpreted as architectural sensitivity observed within the scope of this controlled evaluation, rather than as a statistical estimate of performance variability at the model level.

In Simple Agent, `gpt-4o-mini` clearly improved correctness, increasing the average correctness score from 0.872 to 0.956 on the original query set. This improvement was associated with better handling of non-existent products, better interpretation of categories and limits, and more accurate tool selection in campaign-related queries. Since Simple Agent has few architectural constraints, improvements in the base model translated more directly into better performance.

In the Planner-Critic architecture, `gpt-4o-mini` produced a lower correctness score in the observed execution. Correctness dropped from 0.960 to 0.862 on the original query set. The main issue was not simply weaker planning, but rather the interaction between the planner and the critic. The critic behaved more interventionistically, leading to excessive revision cycles and execution failures in several cases. This result suggests that a more capable model does not necessarily improve a given architecture when its validation behaviour is not properly calibrated.

The change in model, however, had only a limited effect on the Context-Specific architecture. Correctness increased slightly from 0.968 to 0.976 on the original query set, while the full test set remained virtually unchanged. This suggests that the architecture itself already constrained many behaviours that a more robust model could otherwise improve. The routing structure, specialised agents, reduced exposure to tools, and Stop Judge mechanism limited the model's decision space, making the architecture less dependent on the model's capabilities than the Simple Agent.

Given all this, the observed results suggest that the impact of changing the base model depends largely on the architecture. A more robust model may improve a simple architecture, have limited marginal benefits in a highly constrained architecture, or amplify failure modes in an architecture with poorly calibrated feedback loops.

Chapter 6

Conclusion

This dissertation investigated how different agent architectures affect the reliability of conversational AI agents in the context of e-commerce customer service. The work was motivated by the observation that, although Large Language Models can produce fluent and context-rich responses, their behaviour becomes less predictable when they are called upon to interact with external tools, retrieve structured information, and complete multi-step transactional tasks. In these scenarios, reliability depends not only on the quality of language generation but also on the correct selection of tools, the construction of valid arguments, the interpretation of tool results, the control of task completion, and the ability to avoid hallucinatory or unfounded actions.

To study this problem, three architectures were designed, implemented, and evaluated in the same controlled e-commerce environment: a Simple Agent, a Planner-Critic architecture, and a Context-Specific Multi-Agent architecture. The Simple Agent served as a baseline, where a single agent had access to all tools and was responsible for the entire interaction cycle. The Planner-Critic architecture introduced a pre-execution validation mechanism, in which a critic evaluated the action of the tool proposed by the planner before execution. The Context-Specific Multi-Agent architecture, on the other hand, decomposed the workflow into specialised agents, coordinated by a router and supervised by a Stop Judge responsible for evaluating task completion.

The evaluation was conducted using 89 queries, consisting of 58 original queries and 31 paraphrased variants, on two base models: `gpt-3.5-turbo` and `gpt-4o-mini`. The results were subsequently structured and analyzed not only in terms of the correctness of the final response, but also with a strong focus on process-level indicators, such as tool-calling behaviour, the LLM invocation structure, the critic's review cycles, Stop Judge decisions, latency, token consumption, and paraphrase robustness. This evaluation across different fronts was important because final correctness alone does not fully explain how reliable an agent is internally. A system may produce a correct response despite an imperfect process, or it may fail due to a specific internal decision, even when the overall strategy was plausible.

Overall, the results show that architectural design has a significant impact on the reliability profile of tool-supported conversational agents. The Simple Agent benefited greatly from the switch to the more capable `gpt-4o-mini` model, improving from an average correctness of 0.872 to

0.956 on the original query set. However, its remaining shortcomings demonstrate the limitations of relying on a single unconstrained agent cycle, especially in cases involving category boundaries, validation of retrieved results, non-existent products, and incomplete transactional execution. The Planner-Critic architecture achieved solid results with `gpt-3.5-turbo`, reaching an average correctness of 0.960, but suffered a substantial degradation with `gpt-4o-mini`, dropping to 0.862. This revealed that adding a critic does not automatically improve reliability, as the critic must be properly calibrated to distinguish between genuinely incomplete plans and executable intermediate steps. The Context-Specific architecture, on the other hand, achieved the most stable performance overall among all architectures, with average correctness scores of 0.968 and 0.976 on the two models, respectively, on the original query set. Their results indicate that domain-specific routing, restricted exposure to tools, structured transfer, and explicit completion judgement can mitigate various risks associated with a general-purpose agent.

The main conclusion of this dissertation is, therefore, that the reliability of LLM-based agents in the e-commerce context is not determined solely by the base model. The model's capabilities are important, but they interact strongly with the architectural design. A more robust model, which significantly improved the Simple Agent, had only a marginal effect on the Context-Specific architecture and destabilized the Planner-Critic architecture. This demonstrates that architectural mechanisms can mitigate model weaknesses, reduce the model's decision space, or amplify model-specific failure modes. Consequently, the design of reliable conversational agents should not rely solely on replacing weaker models with more robust ones. Instead, reliability should be approached as a system-level property, shaped by the interaction between model capability, access to tools, validation mechanisms, orchestration structure, and task completion control.

6.1 Main Contributions

This dissertation makes three main contributions.

First, it presents a controlled comparative implementation of three LLM-based agent architectures for customer service in e-commerce. By evaluating a Simple Agent, a Planner-Critic architecture, and a context-specific multi-agent architecture using the same tool catalogue, query set, and model configurations, the work isolates the effect of architectural design on reliability.

Second, it provides a multidimensional evaluation methodology that goes beyond the correctness of the final response. The evaluation considers process-level behaviour, including tool selection, argument construction, revision cycles, Stop Judge decisions, latency, token consumption, and paraphrase invariance. This is relevant because reliability in tool-augmented agents cannot be fully understood based solely on the final response.

Finally, the dissertation identifies practical architectural trade-offs for reliable conversational AI in e-commerce. The results show that domain-specific specialisation and restricted exposure to tools can improve stability but also increase the cost of orchestration. They also show that critic-based validation can improve reliability when properly calibrated, but can become a source of failure when too strict or misaligned with the tool interface.

6.2 Limitations

The findings of this dissertation should be interpreted in light of several limitations.

First, the evaluation relies on manual annotation to determine the correctness of the final response and the correctness of the process. While this allows for a detailed qualitative assessment, it introduces a reliance on human interpretation. Future work would benefit from using multiple annotators or a more formal process of inter-rater agreement.

Second, each query was executed only once per model architecture and configuration. Given that LLM results can vary across runs, especially with non-zero temperature settings, repeated runs, possibly with different temperature settings, would be necessary to estimate the stability of each architecture more rigorously.

In the third place, the evaluation used a controlled catalogue of approximately thirty fictional products. While this was sufficient to isolate architectural behaviour, real-world e-commerce systems, the context in which this study was conducted, contain much larger catalogues with more similar products, more ambiguous categories, and more complex availability constraints. Retrieval and disambiguation errors may therefore be more frequent in production environments.

Furthermore, the results are specific to the evaluated model versions, `gpt-3.5-turbo` and `gpt-4o-mini`. Different models may exhibit distinct behaviours regarding instruction compliance, reliability in tool usage, latency, or critique calibration. The conclusions should therefore be understood as evidence specific to the tested configurations, and not as universal statements about all LLM-based agents.

6.3 Future Work

Several directions can extend this work.

A first line of research involves evaluating the same architectures using additional, more recent language models. The results of this dissertation show that changes in the models can indeed interact strongly with the architectural design. Testing more recent models would help determine whether the observed patterns persist across different model families and capability levels.

A second line of research involves increasing the robustness of the evaluation methodology through repeated runs. Executing each query multiple times would allow us to measure variability, confidence intervals, and the probability of failure in the context of the model's stochastic behaviour. This would provide a statistically more robust view of reliability. In addition, queries could also be executed at different temperature values for a more comprehensive evaluation.

A third direction, more focused on the context of this study, involves expanding the e-commerce environment. A broader product catalogue, more complex product attributes, realistic stock changes, customer profiles, authentication, and order data similar to real-world scenarios would make the test environment more closely resemble production conditions. This would help assess whether the benefits of domain-specific specialisation persist under more realistic retrieval and integration complexity.

In addition, it would also be worthwhile to seek ways to improve the completion verification mechanisms of the Context-Specific architecture. The Stop Judge should be expanded to verify not only whether the agent has produced sufficient information, but also whether the necessary transactional tools have actually been executed. This is particularly important for creating shopping carts, refunds, and other action-oriented tasks, in which product identification alone is insufficient.

A related line of research concerns the systematic study of prompt engineering for evaluative components, namely the Critic in the Planner-Critic architecture and the Stop Judge in the Context-Specific architecture. Both components perform critical control functions: the Critic determines whether a proposed tool plan should be executed or revised, while the Stop Judge decides whether a multi-step workflow has been completed. The results of this dissertation suggest that these decisions are highly dependent on prompt design, including instruction clarity, decision criteria, and few-shot examples. Given this, future work could focus on comparing different prompt formulations, few-shot examples, decision criteria, and structured evaluation rubrics to analyze how these affect approval thresholds, revision behaviour, premature termination, and unnecessary continuation. This could help make validation and completion control mechanisms more stable and transferable across different models and e-commerce scenarios.

Similarly, for the Planner-Critic architecture, it would be advantageous to improve it through better calibration of the critic. The results show that the critic's behaviour can range from excessively permissive to excessively restrictive. Future work could explore adaptive thresholds for the critic, separate models for the planner and critic functions, tool-sensitive validation rules, or structured plan schemas that make the critic's evaluation less dependent on natural language interpretation.

A final direction for future work could explore hybrid human-AI escalation mechanisms. Given that the results have demonstrated an inability to completely eliminate all types of errors, it is worth noting that, in real-world customer service environments, full automation may not always be appropriate, particularly for ambiguous, high-risk, or commercially sensitive requests. Thus, a practical extension of this work could involve designing architectures that detect uncertainty or repeated failures and escalate the interaction to a human operator, while simultaneously preserving the execution log for auditability purposes.

6.4 Final Remarks

In summary, this dissertation has demonstrated that improving the reliability of conversational artificial intelligence agents for e-commerce requires more than simply selecting a more robust base language model. Reliability results from the interaction between the model, the available tools, the orchestration framework, and the mechanisms used to validate decisions and determine task completion. The results indicate that structured architectures can reduce significant classes of failures, particularly those related to tool selection, domain confusion, and uncontrolled task execution. However, they also introduce additional coordination costs, latency, token consumption, and new failure modes when validation or completion control mechanisms are poorly calibrated.

Consequently, the design of reliable conversational agents for e-commerce must be approached as a system-level problem. The most effective designs are not necessarily the most complex, but rather those that balance model capability with architectural constraints, explicit validation, clear task decomposition, robust tool usage, and practical implementation considerations.

References

[1] LangChain Academy. *Introduction to LangGraph*. 2024.

[2] Tina Babu et al. "AI-Powered Chat Agent: Revolutionizing Online Shopping". In: 2024. ISBN: 9798331506452. DOI: 10.1109/SCOPES64467.2024.10991141.

[3] Peerawat Chomphooyod et al. "Multi-agentic AI for Automatic Course Syllabus Generation using LangGraph". In: *Proceedings - 2025 10th International STEM Education Conference, iSTEM-Ed 2025*. Institute of Electrical and Electronics Engineers Inc., 2025. ISBN: 9798331542764. DOI: 10.1109/iSTEM-Ed65612.2025.11129303.

[4] Arafat Md Easin, Saha Sourav, and Orosz Tamas. "An Intelligent LLM-Powered Personalized Assistant for Digital Banking Using LangGraph and Chain of Thoughts". In: *SISY 2024 - IEEE 22nd International Symposium on Intelligent Systems and Informatics, Proceedings*. Institute of Electrical and Electronics Engineers Inc., 2024, pp. 625–630. ISBN: 9798350385601. DOI: 10.1109/SISY62279.2024.10737601.

[5] Yanai Elazar et al. "Measuring and Improving Consistency in Pretrained Language Models". In: *Transactions of the Association for Computational Linguistics* 9 (2021), pp. 1012–1031. DOI: 10.1162/tacl_a_00410.

[6] Sorouralsadat Fatemi and Yuheng Hu. "Enhancing Financial Question Answering with a Multi-Agent Reflection Framework". In: *ICAIF 2024 - 5th ACM International Conference on AI in Finance*. 2024. DOI: 10.1145/3677052.3698686.

[7] M. Jayakumar et al. "The Implication of Artificial Intelligence-Powered Chatbots at e-Commerce Platform for Customer Satisfaction". In: Institute of Electrical and Electronics Engineers (IEEE), Nov. 2025, pp. 1–6. ISBN: 9798331596682. DOI: 10.1109/iccds64403.2025.11209174.

[8] I. Mohana Krishna et al. "The Impact of Artificial Intelligence Effect on e-Commerce: A Framework for Key Research Areas". In: *3rd IEEE International Conference on ICT in Business Industry and Government, ICTBIG 2023*. Institute of Electrical and Electronics Engineers Inc., 2023. ISBN: 9798350343274. DOI: 10.1109/ICTBIG59752.2023.10455980.

[9] LangChain. *Tools*. https://docs.langchain.com/oss/python/langchain/tools. LangChain Documentation. Accessed: 2026-06-19. 2024.

[10] LangChain-ai. *LangGraph [GitHub repository]*. 2024.

[11] Jiho Lee et al. "Self-Corrective Task Planning by Inverse Prompting with Large Language Models". In: *Proceedings - IEEE International Conference on Robotics and Automation*. Institute of Electrical and Electronics Engineers Inc., 2025, pp. 11017–11023. ISBN: 9798331541392. DOI: 10.1109/ICRA55743.2025.11128028.

[12] Hailun Lu et al. “ToolFiVe: Enhancing Tool-Augmented LLMs via Tool Filtering and Verification”. In: *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*. Institute of Electrical and Electronics Engineers Inc., 2025. ISBN: 9798350368741. DOI: 10.1109/ICASSP49660.2025.10887544.

[13] Aman Madaan et al. “SELF-REFINE: Iterative Refinement with Self-Feedback”. In: *Advances in Neural Information Processing Systems*. Vol. 36. Neural information processing systems foundation, 2023. ISBN: 9781713899921.

[14] Reza Yousefi Maragheh et al. “Multi-Agentic Recommender Systems: Foundations, Design Patterns, and E-Commerce Applications - An Industrial Tutorial”. In: *RecSys2025 - Proceedings of the 19th ACM Conference on Recommender Systems*. Association for Computing Machinery, Inc, Aug. 2025, pp. 1427–1429. ISBN: 9798400713644. DOI: 10.1145/3705328.3748008.

[15] K. V.S. Prasad et al. “AI-Driven Chatbots for E-Commerce Customer Support”. In: *Proceedings - 3rd International Conference on Advances in Computing, Communication and Applied Informatics, ACCAI 2024*. Institute of Electrical and Electronics Engineers Inc., 2024. ISBN: 9798350389432. DOI: 10.1109/ACCAI61061.2024.10602261.

[16] Matthew Renze and Erhan Guven. “Self-Reflection in LLM Agents: Effects on Problem-Solving Performance”. In: (Oct. 2024). DOI: 10.1109/FLLM63129.2024.10852493. URL: http://arxiv.org/abs/2405.06682%20http://dx.doi.org/10.1109/FLLM63129.2024.10852493.

[17] Timo Schick et al. “Toolformer: Language Models Can Teach Themselves to Use Tools”. In: (Feb. 2023). URL: http://arxiv.org/abs/2302.04761.

[18] Noah Shinn et al. “Reflexion: Language Agents with Verbal Reinforcement Learning”. In: (Oct. 2023). URL: http://arxiv.org/abs/2303.11366.

[19] Jason Wei et al. “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models”. In: (Jan. 2023). URL: http://arxiv.org/abs/2201.11903.

[20] Qingyun Wu et al. “AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation”. In: (Oct. 2023). URL: http://arxiv.org/abs/2308.08155.

[21] Shunyu Yao et al. “ReAct: Synergizing Reasoning and Acting in Language Models”. In: (Mar. 2023). URL: http://arxiv.org/abs/2210.03629.

[22] Yali Zhang et al. “Optimizing AI strategies in e-commerce customer service: An agent-based simulation”. In: *Electronic Markets* 35 (1 2025). DOI: 10.1007/s12525-025-00821-8.

Appendix A

Evaluation Queries

This appendix presents the complete set of evaluation queries used in the experimental evaluation. The query set is divided into the original test suite and the paraphrase variant queries used for robustness testing.

A.1 Original Test Suite (Q1–Q58)

The original test suite contains 58 queries covering the main tool paths and equivalence classes. Each query is annotated with the expected tool path and primary testing technique.

A.1.1 Presales – Specific Lookup

Table A.1: Original test suite – Presales – Specific Lookup

#	Query	Expected Tool(s) Called	Technique
Q1	*"Show me the MacBook Air M3."*	get_product_by_name	EP
Q2	*"Do you have the Google Pixel 8 in stock?"*	get_product_by_name → check_product_stock	EP
Q3	*"Can you find the Samsung Galaxy S25?"*	get_product_by_name	EP
Q4	*"Tell me about the Lenovo Tab P11 Pro."*	get_product_by_name	EP
Q5	*"Is the iMac 24-inch available?"*	get_product_by_name → check_product_stock	EP
Q15	*"Is the Nokia 3310 available?"*	get_product_by_name	ET
Q27	*"Add the MacBook Pro 16-inch."*	get_product_by_name → create_cart	EP

A.1.2 Presales – Brand/Category Browse

Table A.2: Original test suite – Presales – Brand/Category Browse

#	Query	Expected Tool(s) Called	Technique
Q6	*"Show me Apple products."*	filter_products	EP
Q7	*"List Samsung devices."*	filter_products	EP
Q8	*"Do you sell Bose headphones?"*	filter_products	EP
Q9	*"Show me Dell computers."*	filter_products	EP
Q10	*"Any Asus laptops available?"*	filter_products	EP

A.1.3 Presales – Stock Check

Table A.3: Original test suite – Presales – Stock Check

#	Query	Expected Tool(s) Called	Technique
Q11	*"How many iPhone 15 units are in stock?"*	get_product_by_name → check_product_stock	EP

#	Query	Expected Tool(s) Called	Technique
Q12	*"Check stock for the HP Omen 45L."*	get_product_by_name → check_product_stock	EP
Q13	*"Do you still have AirPods Pro 2?"*	get_product_by_name → check_product_stock	EP
Q14	*"Can I still buy the LG Ultragear monitor?"*	get_product_by_name → check_product_stock	EP

A.1.4 Presales – Price Filter

Table A.4: Original test suite – Presales – Price Filter

#	Query	Expected Tool(s) Called	Technique
Q16	*"Show me phones under $1000."*	filter_products	BVA
Q17	*"Find tablets cheaper than 600."*	filter_products	BVA
Q18	*"Which laptops are over 2000?"*	filter_products	BVA
Q19	*"Show me accessories below $150."*	filter_products	BVA
Q20	*"List monitors between $500 and $700."*	filter_products	BVA

A.1.5 Cart

Table A.5: Original test suite – Cart

#	Query	Expected Tool(s) Called	Technique
Q21	*"Add the MacBook Air M3 and iPhone 15 to my cart."*	get_product_by_name ×2 → create_cart	MT
Q22	*"Create a cart with three cheap products."*	filter_products → create_cart	EP
Q23	*"Add a Nokia phone to my cart."*	get_product_by_name → create_cart	ET
Q24	*"Add all tablets to my cart."*	filter_products → create_cart	EP
Q25	*"Add five random products."*	filter_products → create_cart	ST
Q26	*"Create a cart with one laptop and one monitor."*	filter_products ×2 → create_cart	MT
Q38	*"Add one gaming desktop and one monitor to my cart."*	filter_products ×2 → create_cart	MT

A.1.6 Information

Table A.6: Original test suite – Information

#	Query	Expected Tool(s) Called	Technique
Q28	*"What's your return policy?"*	get_information	EP
Q29	*"When do you ship orders?"*	get_information	EP
Q30	*"Do you have international shipping?"*	get_information	EP
Q31	*"How can I contact customer support?"*	get_information	EP
Q32	*"Tell me about warranty coverage."*	get_information	EP
Q33	*"When are you open?"*	get_information	EP

A.1.7 Cross-Cutting

Table A.7: Original test suite – Cross-Cutting

#	Query	Expected Tool(s) Called	Technique
Q34	*"Find Apple laptops under $1500."*	filter_products	MT
Q35	*"Show me available gaming products."*	filter_products	ST
Q36	*"Find a cheap way to watch Netflix."*	filter_products	ST
Q37	*"List laptops sorted by price descending."*	filter_products	EP
Q39	*"Do you sell smartwatches?"*	filter_products	ST

A.1.8 Aftersales – Track Order

Table A.8: Original test suite – Aftersales – Track Order

#	Query	Expected Tool(s) Called	Technique
Q40	*"Where is my order? My order ID is ORDER123."*	track_order('ORDER123')	EP
Q41	*"Can you track my order ORDER456?"*	track_order('ORDER456')	ET

A.1.9 Aftersales – Refund

Table A.9: Original test suite – Aftersales – Refund

#	Query	Expected Tool(s) Called	Technique
Q42	*"I want to return my order ORDER123. The product arrived damaged."*	request_refund('ORDER123', 'product arrived damaged')	EP
Q43	*"I'd like a refund for order ORDER456."*	request_refund('ORDER456', ...)	ET

A.1.10 Aftersales – Receipt

Table A.10: Original test suite – Aftersales – Receipt

#	Query	Expected Tool(s) Called	Technique
Q44	*"Can you send me a new receipt for order ORDER999?"*	reissue_receipt('ORDER999')	EP
Q45	*"I need a duplicate receipt for order ORDER456."*	reissue_receipt('ORDER456')	ET

A.1.11 Campaigns

Table A.11: Original test suite – Campaigns

#	Query	Expected Tool(s) Called	Technique
Q46	*"Are there any discounts on Apple products?"*	available_campaigns('Apple discounts')	EP
Q47	*"Do you have free shipping available?"*	available_campaigns('free shipping')	EP
Q48	*"Is there a student discount for laptops?"*	available_campaigns('student discount laptops')	EP
Q49	*"Do you have any ongoing promotions?"*	available_ campaigns('promotions')	EP

A.1.12 Presales – Close Match

Table A.12: Original test suite – Presales – Close Match

#	Query	Expected Tool(s) Called	Technique
Q50	*"Do you have the Samsung Galaxy S25 Ultra?"*	get_product_by_name	EP
Q51	*"Show me the MacBook Air M2."*	get_product_by_name	EP

A.1.13 Presales – Zero Results

Table A.13: Original test suite – Presales – Zero Results

#	Query	Expected Tool(s) Called	Technique
Q52	*"Show me laptops under $500."*	filter_products	BVA

A.1.14 Presales – Relevance Sort

Table A.14: Original test suite – Presales – Relevance Sort

#	Query	Expected Tool(s) Called	Technique
Q53	*"Show me the most relevant laptops for video editing."*	filter_products	EP

A.1.15 Cart – Degenerate

Table A.15: Original test suite – Cart – Degenerate

#	Query	Expected Tool(s) Called	Technique
Q54	*"Add a product that doesn't exist to my cart."*	get_product_by_name $\rightarrow$ create_cart([])	ET
Q55	*"Add two iPhone 15 units and one MacBook Air M3 to my cart."*	get_product_by_name $\times 2 \rightarrow$ create_cart([6,6,1])	DT

A.1.16 Cross-Cutting – Combined

Table A.16: Original test suite – Cross-Cutting – Combined

#	Query	Expected Tool(s) Called	Technique
Q56	*"I want to buy a MacBook Air M3. Is there any student discount for it?"*	get_product_by_name + available_campaigns	MT
Q57	*"I received a wrong item in order ORDER123. What is your return policy and can I request a refund?"*	get_information + request_ refund	MT

A.1.17 Out-of-Scope

Table A.17: Original test suite – Out-of-Scope

#	Query	Expected Tool(s) Called	Technique
Q58	*"Do you sell gaming chairs?"*	filter_products	ET

A.2 Paraphrase Variant Queries (Robustness Testing)

The paraphrase variant queries rephrase selected base queries to test paraphrase invariance, assessing whether semantically equivalent requests produce equivalent behaviour under different surface forms.

A.2.1 RT – Specific Lookup

Table A.18: Paraphrase variant queries – RT – Specific Lookup

#	Query	Expected Tool(s) Called	Technique
Q1-A	*"Can you show me details about the MacBook Air M3?"*	get_product_by_name	RT
Q1-B	*"I'm looking for the M3 MacBook Air."*	get_product_by_name	RT

A.2.2 RT – Stock Check

Table A.19: Paraphrase variant queries – RT – Stock Check

#	Query	Expected Tool(s) Called	Technique
Q2-A	*"Is the Google Pixel 8 currently available?"*	get_product_by_name → check_product_stock	RT
Q2-B	*"Do you still carry the Pixel 8?"*	get_product_by_name → check_product_stock	RT

A.2.3 RT – Close Match

Table A.20: Paraphrase variant queries – RT – Close Match

#	Query	Expected Tool(s) Called	Technique
Q3-A	*"Do you have the Galaxy S25 in your store?"*	get_product_by_name	RT
Q3-B	*"What's the price of the Samsung S25?"*	get_product_by_name	RT

A.2.4 RT – Not Found

Table A.21: Paraphrase variant queries – RT – Not Found

#	Query	Expected Tool(s) Called	Technique
Q15-A	*"Do you carry Nokia phones?"*	get_product_by_name / filter_products	RT
Q15-B	*"I'm looking for a Nokia 3310, do you have it?"*	get_product_by_name	RT

A.2.5 RT – Price Filter

Table A.22: Paraphrase variant queries – RT – Price Filter

#	Query	Expected Tool(s) Called	Technique
Q16-A	*"What smartphones do you have for less than a thousand dollars?"*	filter_products	RT
Q16-B	*"Show me budget smartphones."*	filter_products	RT

A.2.6 RT – Price Range

Table A.23: Paraphrase variant queries – RT – Price Range

#	Query	Expected Tool(s) Called	Technique
Q20-A	*"Find me monitors that cost between five hundred and seven hundred dollars."*	filter_products	RT
Q20-B	*"Which monitors are priced from $500 up to $700?"*	filter_products	RT

A.2.7 RT – Cart Multi-item

Table A.24: Paraphrase variant queries – RT – Cart Multi-item

#	Query	Expected Tool(s) Called	Technique
Q21-A	*"I'd like to add a MacBook Air M3 and an iPhone 15 to my basket."*	get_product_by_name $\times 2 \rightarrow$ create_cart	RT
Q21-B	*"Put the MacBook Air M3 and the iPhone 15 in my cart please."*	get_product_by_name $\times 2 \rightarrow$ create_cart	RT

A.2.8 RT – Cart Category

Table A.25: Paraphrase variant queries – RT – Cart Category

#	Query	Expected Tool(s) Called	Technique
Q24-A	*"I want every tablet you have in my cart."*	filter_products $\rightarrow$ create_cart	RT
Q24-B	*"Add the full tablet range to my cart."*	filter_products $\rightarrow$ create_cart	RT

A.2.9 RT – Information

Table A.26: Paraphrase variant queries – RT – Information

#	Query	Expected Tool(s) Called	Technique
Q28-A	*"How do returns work?"*	get_information	RT
Q28-B	*"Can I send back a product if I change my mind?"*	get_information	RT

A.2.10 RT – Track Order

Table A.27: Paraphrase variant queries – RT – Track Order

#	Query	Expected Tool(s) Called	Technique
Q40-A	*"What's the status of ORDER123?"*	track_order('ORDER123')	RT
Q40-B	*"I placed an order with ID ORDER123, can you tell me where it is?"*	track_order('ORDER123')	RT

A.2.11 RT – Refund

Table A.28: Paraphrase variant queries – RT – Refund

#	Query	Expected Tool(s) Called	Technique
Q42-A	*"I need to return something from ORDER123, it came broken."*	request_refund('ORDER123', ...)	RT
Q42-B	*"Can you start a refund process for my order ORDER123?"*	request_refund('ORDER123', ...)	RT

A.2.12 RT – Campaigns

Table A.29: Paraphrase variant queries – RT – Campaigns

#	Query	Expected Tool(s) Called	Technique
Q46-A	*"Are iPhones on sale right now?"*	available_campaigns	RT
Q46-B	*"Do you have any Apple deals this week?"*	available_campaigns	RT
Q47-A	*"How much does shipping cost?"*	get_information / available_campaigns	RT

A.2.13 RT – Zero Results

Table A.30: Paraphrase variant queries – RT – Zero Results

#	Query	Expected Tool(s) Called	Technique
Q52-A	*"Do you sell any laptops for 300 euros?"*	filter_products	RT
Q52-B	*"Show me the cheapest laptop you have under 400 dollars."*	filter_products	RT+BVA

A.2.14 RT – Combined

Table A.31: Paraphrase variant queries – RT – Combined

#	Query	Expected Tool(s) Called	Technique
Q56-A	*"Is there a discount if I buy a MacBook Air M3 as a student?"*	get_product_by_name + available_campaigns	RT
Q57-A	*"My ORDER123 had the wrong product. How do I get a refund?"*	get_information + request_ refund	RT

A.2.15 RT – Semantic Category

Table A.32: Paraphrase variant queries – RT – Semantic Category

#	Query	Expected Tool(s) Called	Technique
Q39-A	*"Do you have any wearables?"*	filter_products	RT
Q39-B	*"I'm looking for a smart watch."*	filter_products	RT