THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

Foundations of Agent-to-Agent Communication in a Software-Mediated Research Agent

A Tool-Augmented, Multi-Stage LLM Agent for Open-Domain Research Question Answering

by

Praneet Dhoolia

School of Electrical Engineering and Computer Science,
The University of Queensland.

Submitted for the degree of Bachelor of Engineering (Honours) as part of the course REIT4842.

Supervisor: Dr. Shakes Chandra
October 2026.

Letter to the Head of School

School of Electrical Engineering and Computer Science
The University of Queensland
St Lucia, QLD 4072

August 18, 2026

Professor Michael Brünig
Head of School
School of Electrical Engineering and Computer Science
The University of Queensland
St Lucia, QLD 4072

Dear Professor Brünig,

In accordance with the requirements of the Degree of Bachelor of Engineering (Honours) in the School of Electrical Engineering and Computer Science, I submit the following thesis entitled

"Foundations of Agent-to-Agent Communication in a Software-Mediated Research Agent"

The thesis was performed under the supervision of Dr. Shakes Chandra. I declare that the work submitted in the thesis is my own, except as acknowledged in the text and footnotes, and that it has not previously been submitted for a degree at the University of Queensland or any other institution.

Yours sincerely,

Praneet Dhoolia

Declaration by Author

This thesis is composed of my original work, and contains no material previously published, written by another person, or generated by artificial intelligence (AI), except where clearly referenced and acknowledged either in the main text or the preliminary pages of this thesis. I have clearly stated the contribution of others to jointly authored works that I have included in my thesis.

I have clearly stated the contribution of others to my thesis as a whole, including statistical assistance, survey design, data analysis, significant technical procedures, professional editorial advice, use of AI tools, financial support, and any other original research work used or reported in my thesis. The content of my thesis is the result of work I have carried out since the commencement of my enrolment in the course code indicated on the title page, and does not include a substantial part of work that has been submitted to qualify for the award of any other degree or diploma in any university or other tertiary institution. I have clearly stated which parts of my thesis, if any, have been submitted to qualify for another award.

I acknowledge that copyright of all material contained in my thesis resides with the copyright holder(s) of that material. Where appropriate I have obtained copyright permission from the copyright holder to reproduce material in this thesis and have sought permission from co-authors for any jointly authored works included in the thesis.

Use of AI Statement

I acknowledge the use of generative AI tools in completing this thesis. Details of which tools were used and how they were used are provided in the table below, along with appropriate in-text and full references where applicable. I take responsibility for critically evaluating and integrating any AI-generated content, and for ensuring it adheres to academic integrity standards.

AI Model and date	Artificial Intelligence (AI) tool usage (tick all that apply)								
	language translation	grammar/style	planning/drafting	research/background information	content creation text	content creation visual	content creation code	feedback	other (provide details)
Claude (Anthropic) Sonnet 4.5/4.6 2026		✓	✓			✓		✓	

Acknowledgements

I would like to thank my supervisor, Dr. Shakes Chandra, for his guidance throughout this project, for the early discussions that shaped the system's communication-centric design philosophy, and for ongoing feedback on the project's scope and framing.

I also acknowledge the open-source and protocol communities whose work this thesis builds upon: the LangGraph and LangChain maintainers, the Anthropic team behind the Model Context Protocol, and the maintainers of the many MCP servers (Tavily, E2B, Wolfram Alpha, arXiv, and others) integrated into the system described here. I acknowledge the creators of the benchmark datasets used for evaluation, GAIA, DRACO, MCP-Atlas, SimpleQA, FRAMES, and BrowseComp, whose work makes rigorous, reproducible comparison of research agents possible.

Abstract

Large language models (LLMs) are increasingly deployed as autonomous research assistants, yet single-prompt LLM systems remain prone to hallucination, premature termination, and weak grounding when answering open-domain questions that require multi-step web research, precise computation, or synthesis across several sources. This thesis presents the design, implementation, and evaluation of a *software-mediated research agent*: a multi-stage LLM system in which a small set of specialised roles, a **planner**, a **tool dispatcher**, a **tool executor**, an **assessor**, and a **synthesizer**, communicate through a structured, typed state object as they collaboratively decompose a question, gather evidence using external tools, critically assess the sufficiency of that evidence, and produce a cited final answer.

The system is implemented as a five-node graph using LangGraph, and integrates more than a dozen external tools through the Model Context Protocol (MCP), including an AI-native web search engine (Tavily), a headless browser (Playwright), a sandboxed Python notebook (E2B), the Wolfram Alpha computation engine, arXiv and Wikipedia search, and a custom "deep thinking" MCP server that delegates difficult sub-problems to an extended-reasoning model. A lightweight model (e.g. GPT-4.1-mini) drives the tight planner/dispatcher/assessor loop for speed and cost efficiency, while a stronger model (e.g. Claude Sonnet 4.5) is reserved for final synthesis. The system is exposed through a FastAPI backend with streaming responses and a Next.js web interface providing a chat view with live tool-call visualisation, an agent configuration page, and a live benchmarking dashboard.

The agent is evaluated on six widely-used research and tool-use benchmarks: GAIA, DRACO, MCP-Atlas, SimpleQA, FRAMES, and BrowseComp. On SimpleQA the agent reaches **62.0%** accuracy, more than double the accuracy of Claude 3.5 Sonnet (**28.9%**) and GPT-4o with direct web search (**28.0%**) on the same question subset. On GAIA (Level 1+2), the agent reaches **68.0%** with a cost-efficient configuration and **87.0%** with a stronger model configuration, exceeding several recently reported multi-agent baselines. On DRACO, an LLM-judged deep-research benchmark, the agent scores **61.45%**, ahead of OpenAI's o3 (**52.1%**) and o4-mini (**41.9%**) on the same metric. On MCP-Atlas, a tool-use competency benchmark built on real MCP servers, the agent reaches **68.0%** with its strongest configuration, on par with GPT-5.2 (**67.6%**). These results suggest that explicit role separation and structured inter-stage communication, rather than scaling a single model alone, meaningfully improve the reliability of research agents on open-domain tasks, while highlighting that hard multi-hop adversarial benchmarks such as BrowseComp remain a substantial challenge for tool-augmented agents of this kind.

Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Expansion
A2A	Agent-to-Agent (protocol)
ACP	Agent Communication Protocol
API	Application Programming Interface
CLI	Command-Line Interface
CoT	Chain of Thought
GAIA	General AI Assistants (benchmark)
GPT	Generative Pre-trained Transformer
GTFA	Ground-Truth Function/Argument
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
JSON	JavaScript Object Notation
LLM	Large Language Model
MCP	Model Context Protocol
NLP	Natural Language Processing
PDF	Portable Document Format
RAG	Retrieval-Augmented Generation
ReAct	Reason and Act
RPC	Remote Procedure Call
SDK	Software Development Kit
SSE	Server-Sent Events
UI	User Interface
URL	Uniform Resource Locator

Chapter 1

Introduction

1.1 Motivation

Large language models (LLMs) such as GPT-4o and Claude have demonstrated remarkable fluency on open-domain question answering, summarisation, and reasoning tasks [2]. Yet when an LLM is queried directly, a single prompt, a single forward pass, its answer is bounded by what was memorised during training. For questions that require up-to-date information, multi-step reasoning across several independent facts, precise numerical computation, or verification against primary sources, a single LLM call is structurally unsuited to the task: the model cannot retrieve what it does not know, cannot verify what it cannot check, and has no mechanism to notice that its own answer might be wrong.

Research agents address this by wrapping an LLM in a loop: the model is given tools (web search, a calculator, a code sandbox, a browser) and repeatedly invoked until it decides it has gathered enough evidence to answer. This pattern, often called ReAct (Reason + Act), has become the default architecture for products like OpenAI's Deep Research, Anthropic's Claude with tool use, and Perplexity's answer engine. In practice, however, naive single-agent tool-use loops exhibit three recurring failure modes that motivate this thesis:

- **Premature termination.** The model decides it "knows" the answer before it has actually searched for evidence, producing a fluent but unverified response.
- **No self-correction.** If the first search returns irrelevant or noisy results, the model has no structured mechanism to recognise this and try a different query; it simply continues with weak evidence.
- **Cost/latency vs. quality trade-off.** Using a single large, expensive model for every step of a multi-step research task is slow and costly, while using a single small model throughout sacrifices the synthesis quality needed for a coherent, well-cited final answer.

This thesis takes the position that these failure modes are fundamentally a problem of *communication and role separation* within the agent, not merely a problem of model scale. By decomposing the research loop into distinct roles, a planner that decomposes

the question, a dispatcher that selects tools, an executor that runs them, an assessor that critically checks whether the evidence so far is sufficient, and a synthesizer that composes the final answer, and by passing a structured, typed state object between these roles, an agent built from comparatively small and inexpensive models can match or exceed the research quality of much larger monolithic systems, while remaining fast and cheap enough to run iteratively.

1.2 Problem Statement

Concretely, this thesis addresses the following problem: *design, implement, and evaluate a software-mediated research agent that (i) reliably solves open-domain research questions requiring web search, multi-hop reasoning, and exact computation; (ii) uses a structured internal communication protocol between specialised agent roles to avoid premature termination and enable self-correction; (iii) keeps the per-step cost and latency low by using lightweight models for orchestration and a stronger model only for final synthesis; and (iv) is evaluated against current frontier model baselines on multiple widely-used research-agent benchmarks.*

This framing was arrived at after an initial project proposal (preserved in the supervisory record) explored a different angle on the same underlying question, standardising *agent-to-agent* (A2A) message schemas across multi-agent frameworks such as Google ADK, LangChain, AutoGen, and LangGraph. During early implementation it became clear that the more impactful and more rigorously evaluable instance of "structured agent communication" was not cross-framework protocol translation, but the *internal* communication contract between specialised roles inside a single research agent, because this internal contract directly and measurably affects benchmark accuracy. The protocols surveyed in Chapter 2 (MCP, A2A, ACP) remain directly relevant: MCP is the mechanism by which the implemented agent talks to its eleven external tools, and A2A/ACP inform the design of the internal planner/dispatcher/assessor/synthesizer message contract described in Section 3.2.

1.3 Aims and Contributions

The aims of this thesis are to:

1. Design a multi-role agent architecture in which specialised LLM-driven nodes communicate through a structured, typed state object, and implement it as a five-node graph using LangGraph.

2. Integrate a broad ecosystem of external tools through the Model Context Protocol (MCP), covering web search, browser automation, sandboxed code execution, symbolic computation, academic literature search, and a custom "deep thinking" delegation tool.

3. Build a complete, no-login web application (chat interface, agent configuration, file upload with automatic parsing/OCR, and a live benchmarking dashboard) around the agent, so that the system is usable as a product, not only as a research prototype.

4. Evaluate the system on six established research-agent benchmarks, GAIA, DRACO, MCP-Atlas, SimpleQA, FRAMES, and BrowseComp, and compare its accuracy, cost, and tool-use behaviour against frontier LLM baselines (GPT-4o/GPT-5 family, Claude 3.5/Sonnet 4.5, Gemini, Perplexity).

5. Analyse *why* the architecture succeeds or fails on different benchmark categories, distinguishing factual lookup, multi-hop reasoning, deep-research synthesis, tool-use competency, and hard adversarial search.

The principal contributions of this thesis are:

- A concrete, open implementation of a planner/dispatcher/executor/assessor/synthesizer agent graph with a documented routing table and iteration budget, demonstrating that role separation with small models is a viable alternative to scaling a single large model.

- An MCP tool ecosystem of thirteen servers wired into a single agent, including a custom-built "thinking" MCP server that delegates hard sub-problems to an extended-reasoning model on demand.

- A full-stack reference application (FastAPI + Next.js) exposing the agent through a chat UI with live tool-call visualisation, a configuration UI for model/tool/iteration tuning, and a live benchmark dashboard, which itself doubles as the experiment runner for this thesis.

- Empirical results across six benchmarks showing that the agent performs consistently and competitively, reaching 87.0% on GAIA and 68.0% on MCP-Atlas with its strongest configuration and matching or approaching several frontier-model baselines elsewhere, together with a detailed discussion of BrowseComp, the one benchmark where the gap to specialised deep-research systems remains large and which is analysed as an instructive limitation rather than treated as a verdict on the architecture.

1.4 Thesis Structure

Chapter 2 reviews the literature on LLM agents, tool-use protocols (MCP, A2A, ACP), multi-agent orchestration frameworks, and the benchmark landscape for research agents. Section 3.2 presents the system design: the five-node agent graph, its state schema and routing logic, the MCP tool ecosystem, and the full-stack application architecture. Section 3.3 describes the evaluation methodology: the six benchmarks used, scoring procedures, and experimental configurations. Chapter 4 presents and discusses the results, including per-benchmark comparisons against frontier baselines and qualitative analysis of agent traces. Chapter 5 concludes the thesis and discusses limitations and future work.

Chapter 2

Background and Related Work

This chapter surveys the body of work that motivates and informs the design described in Section 3.2. The review is organised to mirror the architectural decisions made in this thesis, moving from the foundational properties of large language models (Section 2.1) that make external tool use necessary in the first place, through the prompting and reasoning techniques that allow a language model to use tools effectively (Section 2.2), to the retrieval- and tool-augmentation literature that directly underlies the agent's research loop (Section 2.3). It then steps back to the older, non-LLM literature on multi-agent systems (Section 2.4) before covering the modern, LLM-specific communication protocols (Section 2.5) and orchestration frameworks (Section 2.6) that this thesis builds on directly. Section 2.7 reviews the computational and execution tooling, symbolic mathematics engines, code sandboxes, and browser automation, that forms the agent's tool ecosystem, and Section 2.8 surveys the benchmark landscape used for evaluation in Chapter 4. Section 2.9 synthesises these strands and positions the contribution of this thesis relative to them.

2.1 Foundations of Large Language Models

2.1.1 The Transformer Architecture and Scaling Laws

The transformer architecture [1] replaced the recurrent and convolutional sequence models that preceded it with a purely attention-based mechanism, in which every token in a sequence can attend directly to every other token via learned query, key, and value projections. Fig. 2.1 shows the encoder-decoder layout from the original paper: stacks of multi-headed self-attention and feed-forward blocks, each wrapped in residual connections and layer normalisation, with positional encodings injected at the input to give the otherwise order-agnostic attention mechanism a notion of sequence position. This design removed the sequential bottleneck of recurrent networks, allowing training to be parallelised across the entire input sequence and, critically, allowing model size and training data to be scaled by orders of magnitude without a corresponding increase in wall-clock training time per token.

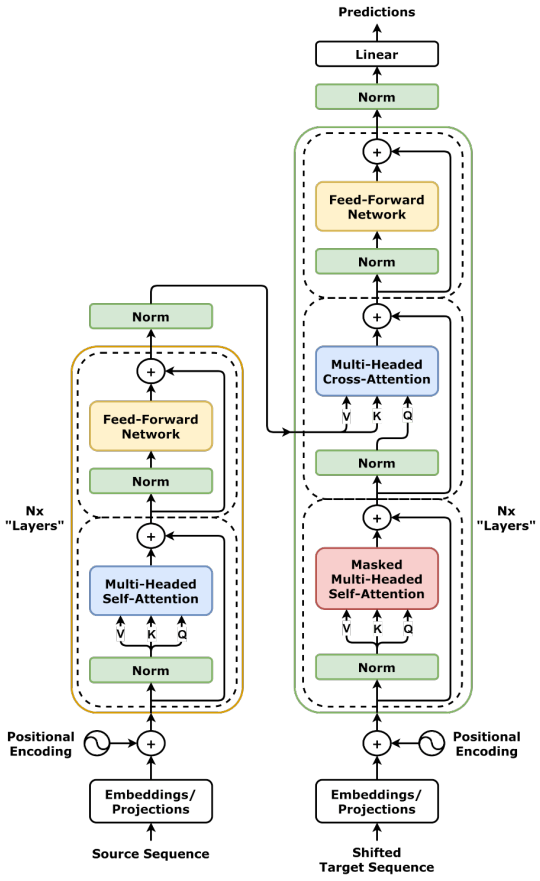

Figure 2.1: The transformer encoder-decoder architecture [1]: stacked multi-headed self-attention and feed-forward blocks with residual connections, layer normalisation, and positional encoding. Every role in this thesis's agent graph (Section 3.2.1) is a decoder-only descendant of this architecture, differentiated only by prompt and structured-output schema. Source: dvgodoy, *dl-visuals* (`https://github.com/dvgodoy/dl-visuals`), licensed CC BY 4.0.

GPT-3 [2] demonstrated that scaling a decoder-only transformer to 175 billion parameters, trained on a broad corpus of internet text, produced a single model that performed competitively on dozens of downstream NLP tasks *without any task-specific fine-tuning*, simply by conditioning on a natural-language description of the task and a handful of examples: a capability the authors termed *few-shot learning*. Subsequent work on scaling laws [25] showed that this improvement was remarkably predictable: model loss decreases as a smooth power-law function of parameter count, dataset size, and compute budget, which gave the field a principled basis for the rapid scaling that followed GPT-3 and produced the current generation of frontier models (GPT-4, GPT-5, Claude, and Gemini families). Fig. 2.2 shows this scaling empirically: training compute for frontier models has grown by roughly 4–5× per year since 2010, with a visible acceleration after the introduction of the transformer in 2017, the architectural shift that made this rate of scaling computationally feasible in the first place.

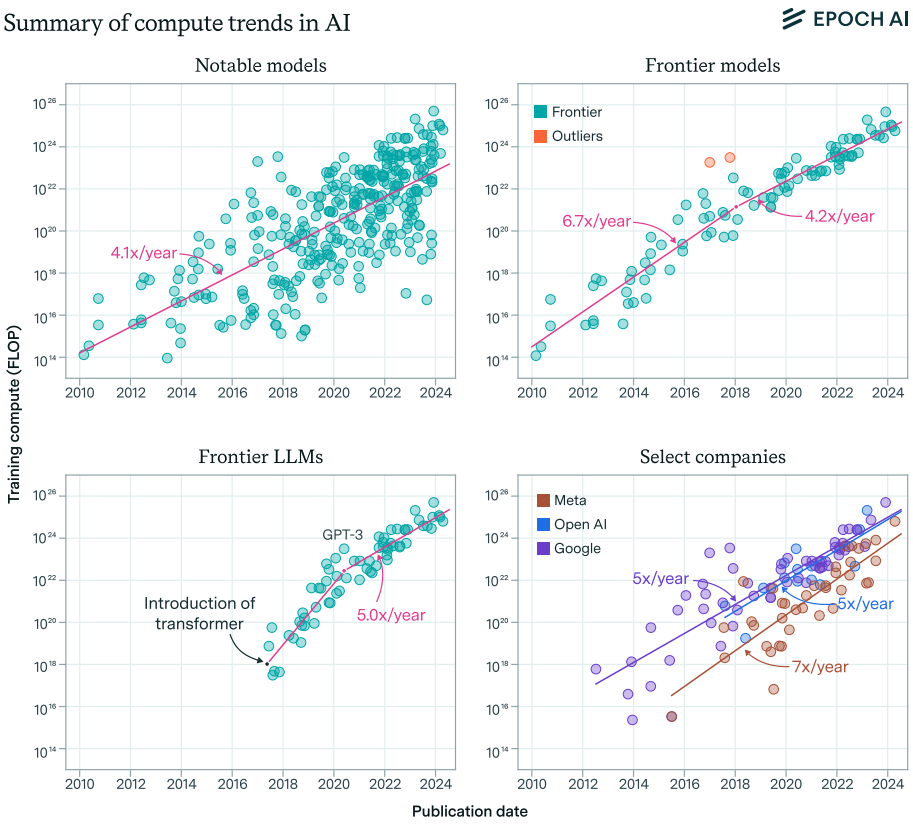

Figure 2.2: Training compute (FLOP) of notable AI models over time, on a log scale, showing roughly exponential growth driven in large part by the transformer architecture's parallelisability. Source: Epoch AI, *Trends in AI training compute* (`https://epoch.ai`), licensed CC BY 4.0.

2.1.2 Emergent Abilities and In-Context Learning

A second consequence of scale, distinct from the smooth improvements predicted by scaling laws, is the appearance of *emergent abilities*, capabilities such as multi-step arithmetic, instruction following, and analogical reasoning that are close to random performance below some model scale and improve sharply above it [26]. *In-context learning*, the ability of a model to adapt its behaviour based purely on examples or instructions present in its prompt, with no gradient update, is the mechanism that makes this thesis's entire approach viable: every role in the agent graph (Section 3.2.1) is the *same* underlying pretrained model, differentiated purely by the system prompt and the structured-output schema it is asked to fill in. No fine-tuning is performed anywhere in this thesis; the planner, dispatcher, assessor, and synthesizer are realised entirely through prompt design and in-context instruction, which is only possible because of the in-context learning capabilities that emerge at the scale of models such as GPT-4.1-mini, Claude Haiku 4.5, and Claude Sonnet 4.5.

2.1.3 The Knowledge Cutoff Problem and Parametric Limitations

Despite these capabilities, a language model's knowledge is *parametric*: it is encoded in the weights learned during pretraining, which is performed on a snapshot of text up to some *knowledge cutoff* date. Any fact, event, publication, or dataset that postdates this cutoff is simply absent from the model, and the model has no mechanism to recognise this absence; it will, by default, produce a fluent and plausible-sounding answer regardless of whether that answer reflects training data or pure interpolation, a phenomenon commonly termed *hallucination*. This is compounded by the fact that even facts the model *does* encode are stored in a lossy, distributed form: a model can recall a celebrity's birth year with high reliability (it appears often in training data) while being unreliable on a specific numeric detail buried in a single obscure source document, with no introspective signal that distinguishes the two cases from the model's own perspective. Section 2.8 discusses how benchmarks such as SimpleQA [19] are specifically constructed to expose this gap: bare frontier models such as GPT-4o answer fewer than 40% of SimpleQA questions correctly, not because the questions are intellectually hard, but because the questions are deliberately drawn from the long tail of facts a model is unlikely to have memorised precisely. This single observation, that a language model's *reasoning* ability and its *factual recall* ability are largely independent, and that the latter degrades sharply for low-frequency facts, is the fundamental motivation for every tool-augmented architecture discussed in the remainder of this chapter, including the one designed in this thesis.

2.2 Prompting, Reasoning, and Self-Correction

If a language model's raw output is unreliable for tasks requiring multi-step reasoning or up-to-date facts, the question becomes how to structure the model's *use* so that these weaknesses are exposed and corrected rather than silently propagated. This section reviews the prompting and reasoning techniques that this thesis's agent graph builds on.

2.2.1 Chain-of-Thought and Stepwise Reasoning

Chain-of-thought (CoT) prompting [27] showed that simply instructing a model to "think step by step" before producing a final answer, or providing few-shot examples that include intermediate reasoning steps, substantially improves performance on arithmetic, commonsense, and symbolic reasoning tasks, with the improvement growing as model scale increases. **Zero-shot CoT** [28] showed that this benefit does not even require few-shot examples: appending the phrase "Let's think step by step" to a prompt is sufficient

to elicit a substantial accuracy gain on arithmetic reasoning benchmarks for sufficiently large models. The practical implication for this thesis is that every structured-output call in the agent graph (the planner's `PlannerResult`, the assessor's `AssessorResult`) is preceded, in its prompt, by an instruction to reason briefly about the current state of the research before committing to the structured fields: a direct, if simplified, application of CoT to a multi-role pipeline rather than a single question-answering call.

Tree-of-Thoughts [29] generalises CoT from a single linear reasoning chain to a search over a tree of possible reasoning paths, with the model itself evaluating which partial solutions are worth continuing. While this thesis's agent does not implement an explicit tree search, the assessor/retry loop described in Section 3.2.2 can be viewed as a constrained, two-branch instance of the same idea: at each step, the agent evaluates whether the current evidence-gathering "thought" (the result of the most recent tool call) is sufficient, and if not, prunes it and tries an alternative `current_focus` drawn from the assessor's `missing` field.

2.2.2 ReAct: Interleaving Reasoning and Acting

The **ReAct** (Reason + Act) pattern [3] addressed the static nature of a pure language model by interleaving the model's natural-language reasoning ("thoughts") with discrete actions (tool calls) and observations (tool results), all within a single autoregressive loop: $\textit{Thought} \rightarrow \textit{Action} \rightarrow \textit{Observation} \rightarrow \textit{Thought} \rightarrow \ldots$ until the model emits a final answer. ReAct demonstrated that this interleaving reduces hallucination relative to reasoning-only (CoT) prompting on knowledge-intensive tasks, because each reasoning step can be grounded against a fresh observation rather than relying purely on the model's parametric memory, while also reducing the "action-only" failure mode where a model calls tools without any interpretable rationale for *why*.

These ideas converge on the architecture used by current "deep research" products: an LLM is placed in a loop with search and browsing tools and allowed to run for many steps (often dozens) before producing a final answer. OpenAI's Deep Research and Perplexity's research mode are commercial instances of this pattern [37]. The agent designed in this thesis (Section 3.2) follows the same broad family but decomposes the ReAct loop's single "Thought" step into three separate roles, planner, dispatcher, and assessor, each with its own structured output type, for reasons elaborated in Section 2.6 and motivated empirically by the development history in Section 3.1.

2.2.3 Toolformer and Function-Calling Interfaces

Toolformer [4] showed that models could be trained to decide *when* to call a tool (a calculator, a search engine, a calendar, a translation system) without explicit step-by-step supervision, by self-generating candidate tool-call insertions, filtering them by whether they reduce the model's loss on the surrounding text, and fine-tuning on the filtered set. This established that tool use could be learned as an emergent capability rather than hard-coded into the inference pipeline. Modern LLM APIs (OpenAI function calling, Anthropic tool use) productionised this idea at the API level: the calling application supplies a list of tool *schemas* (name, description, JSON-Schema-typed arguments), the model emits a structured tool-call object naming one of these tools and its arguments, and the application is responsible for executing the call and returning a structured result on the next turn. Every tool invocation in this thesis's agent, whether through Tavily, Wolfram Alpha, or any of the other eleven MCP servers described in Section 3.2.4, is mediated by exactly this function-calling interface, via `model.bind_tools(tools)` in the `tool_dispatcher` node.

2.2.4 Reflection and Self-Correction

A separate line of work asks not how a model decides to act, but how it recognises and corrects its own mistakes. **Self-Refine** [31] showed that prompting a model to critique its own output and then revise it based on that critique, with no external feedback signal, produces measurable quality improvements across a range of generation tasks, essentially using the model as its own reward model. **Reflexion** [30] extended this to multi-step agents operating in an environment: after a failed attempt at a task, the agent generates a natural-language "reflection" on what went wrong, which is added to its context for the next attempt, functioning as a form of verbal reinforcement learning without any gradient updates.

The **assessor** node in this thesis's agent graph (Section 3.2.2) is a direct, structured-output instantiation of this self-correction idea, but applied *within* a single research session rather than across episodes: after every tool call, a dedicated model call evaluates whether the result actually satisfies the current sub-goal (`is_complete`, `confidence` ≥ 0.7), and if not, produces `feedback` and a list of `missing` follow-up queries that are fed back into the next `tool_dispatcher` call: the same reflect-then-retry pattern as Reflexion, but operating at the granularity of individual tool calls rather than entire task episodes, and expressed as a typed Pydantic object rather than free natural-language text appended to a transcript.

2.3 Retrieval and Tool Augmentation

2.3.1 Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) [32] combines a parametric language model with a non-parametric retrieval component: at inference time, a query is used to retrieve relevant passages from an external corpus (originally a dense vector index over Wikipedia, retrieved via **Dense Passage Retrieval** [33]), and these passages are concatenated into the model's context before it generates an answer. Fig. 2.3 illustrates the canonical RAG pipeline: reference documents are embedded once into a vector database, a user query is embedded with the same model, the nearest document chunks are retrieved by similarity, and the original query plus retrieved chunks are assembled into an augmented prompt for a final generation call. RAG was originally proposed as a way to give a language model access to a large, updatable knowledge base without retraining, and to provide a natural mechanism for *attribution*; the retrieved passages can be cited as the source of the generated answer.

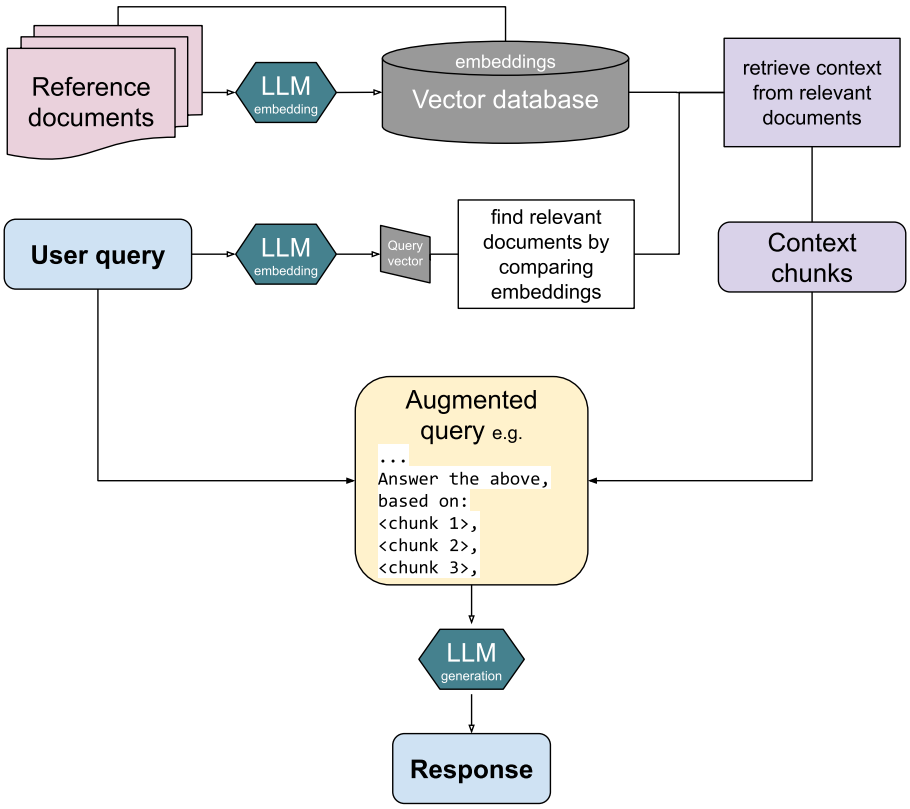

Figure 2.3: The canonical Retrieval-Augmented Generation pipeline [32]: documents and queries are embedded into a shared vector space, relevant chunks are retrieved by similarity, and the augmented prompt is passed to the generation model. Section 2.3 discusses how this thesis's agent generalises the single-shot "retrieve then generate" loop into an iterative, tool-mediated research loop. Source: Turtlecrown, Wikimedia Commons, licensed CC BY-SA 4.0.

The agent in this thesis can be understood as a generalisation of the RAG pattern in two directions. First, the "corpus" is not a static pre-indexed vector store but the live web, academic literature databases, and computational engines, accessed through MCP tool calls (Section 3.2.4) rather than a vector similarity search. Second, and more importantly, classic RAG performs retrieval *once*, before a single generation step; this thesis's agent performs retrieval *iteratively*, with the assessor (Section 2.2) deciding after each retrieval whether further retrieval is needed and, if so, what to retrieve next: a query-reformulation loop that static RAG pipelines do not have.

2.3.2 Search-Augmented LLMs and Deep Research Systems

WebGPT [34] was an early demonstration that a language model could be fine-tuned to use a text-based web-browsing environment, issuing search queries, clicking links, and quoting passages, to answer long-form questions with citations, and that human evaluators preferred its answers to those of a model without browsing access on a majority of

questions. This established the template that current commercial "deep research" agents follow: a browsing- or search-capable loop, run for many steps, that produces a long-form cited answer. A 2023 survey of *Augmented Language Models* [37] catalogues this broader family, models augmented with reasoning, retrieval, and tool-use capabilities, and observes that the central open design question across this family is not whether to augment a model with tools (by 2023 this was already standard practice) but *how to structure the control flow* around tool use: a single undifferentiated loop, as in ReAct and WebGPT, versus a decomposed pipeline with explicit planning, execution, and verification stages.

2.3.3 Modular and Tool-Selection Architectures

MRKL Systems (Modular Reasoning, Knowledge and Language) [35] proposed routing a query to one of several *expert modules*, a calculator, a database lookup, a translation system, the language model itself, based on the query's type, rather than relying on a single model to handle every kind of request. **ToolLLM** [36] scaled this idea to thousands of real-world REST APIs, showing that an LLM can be trained or prompted to select an appropriate API from a very large candidate pool and to chain multiple API calls to satisfy a complex instruction. Both lines of work identify *tool selection*, choosing which of many available capabilities to invoke for a given sub-task, as a distinct cognitive step worth modelling explicitly, rather than folding into a single generic "decide what to do next" prompt. This is precisely the role played by the `tool_dispatcher` node in this thesis (Section 3.2.2): given a `current_focus` produced by the planner, the dispatcher's *only* job is to select one of the (at the time of evaluation) thirteen MCP-exposed tool families and supply its arguments, separate from both the higher-level planning of *what* to research next and the lower-level assessment of *whether* the resulting evidence is sufficient.

2.4 Foundations of Multi-Agent Systems

2.4.1 Society of Mind and Distributed Cognition

The idea that complex intelligent behaviour can emerge from the interaction of many simple, specialised processes predates large language models by decades. Minsky's *Society of Mind* [5] proposed that cognition arises from a society of interacting "agents", each responsible for a narrow function and individually unintelligent, communicating through shared representations and competing or cooperating for control of behaviour. The planner/dispatcher/executor/assessor/synthesizer decomposition used in this thesis (Section 3.2.1) can be read as a small, concrete, fifty-years-later instance of this idea: no single node is individually very capable, each is, after all, the same underlying lan-

guage model with a different prompt, but their structured interaction produces behaviour (self-correcting, evidence-checking research) that none of them could produce alone, and that a single instance of the same model, prompted as one undifferentiated ReAct loop, demonstrably does not produce as reliably (Section 3.1 documents this empirically).

2.4.2 Classical Multi-Agent Systems and Agent Communication Languages

Before LLMs, the field of *multi-agent systems (MAS)* developed a substantial body of theory around how autonomous, goal-directed software agents should coordinate [39]. A central concern of this literature was *agent communication languages*, standardised message formats that allow heterogeneous agents, built by different parties, to interoperate. The **FIPA Agent Communication Language (FIPA-ACL)** [40] is the best-known example: messages are typed by a *performative* (e.g. `inform`, `request`, `query-if`, `propose`) that specifies the communicative intent of the message independently of its content, an idea borrowed from speech-act theory. While FIPA-ACL itself saw limited industrial adoption (it predates both the web-scale infrastructure and the language models that would have made it practical), its core insight, that a message's *type* should be explicit and separate from its *content*, reappears directly in the modern protocols surveyed in Section 2.5: MCP's typed JSON-RPC methods, A2A's typed task lifecycle, and this thesis's own `PlannerResult`/`AssessorResult` schemas (Section 3.2.1) are all, in essence, modern realisations of the same principle, a finite, explicit vocabulary of message *types*, each with a well-defined schema, rather than unstructured natural-language exchange.

2.4.3 Generative and LLM-Based Multi-Agent Societies

Generative Agents [38] demonstrated that populating a simulated environment with LLM-driven agents, each maintaining a memory stream and reflecting on it to plan future behaviour, produces emergent social behaviours (agents independently coordinating to organise a party) that were not explicitly programmed. This work, alongside the multi-agent orchestration frameworks discussed in Section 2.6, established LLM-driven multi-agent systems as a practical paradigm rather than a purely theoretical one. However, as Section 2.6 discusses, the *communication substrate* used by most such systems, free-form natural-language messages appended to a shared or per-agent transcript, differs sharply from the typed, schema-constrained communication this thesis argues for, and this difference has direct, measurable consequences for both debuggability and benchmark performance.

2.5 Agent Communication Protocols

As LLM agents began to call external tools and, increasingly, other agents, several standardisation efforts emerged in 2024–2025 to define how these interactions should be structured. This section reviews the three most directly relevant to this thesis: the Model Context Protocol, the Agent2Agent Protocol, and IBM's Agent Communication Protocol.

2.5.1 Model Context Protocol (MCP)

The Model Context Protocol [6] is an open protocol, introduced by Anthropic in late 2024, that standardises how LLM applications (*hosts*) connect to external tools and data sources (*servers*). Prior to MCP, every application that wanted to give a language model access to, say, both a web search API and a database would typically implement two bespoke integrations, each with its own authentication, schema, and error-handling conventions; MCP replaces this $N \times M$ integration problem with a single protocol that any host can speak to any server. MCP defines three architectural roles, shown in Fig. 2.4:

- **Host**: the LLM application that coordinates the overall system (in this thesis, the LangGraph research agent).
- **Client**: maintains a stateful, one-to-one connection to a single server, handling protocol negotiation and message routing.
- **Server**: exposes *tools* (callable functions), *resources* (read-only contextual data), and *prompt templates* through a standard interface.

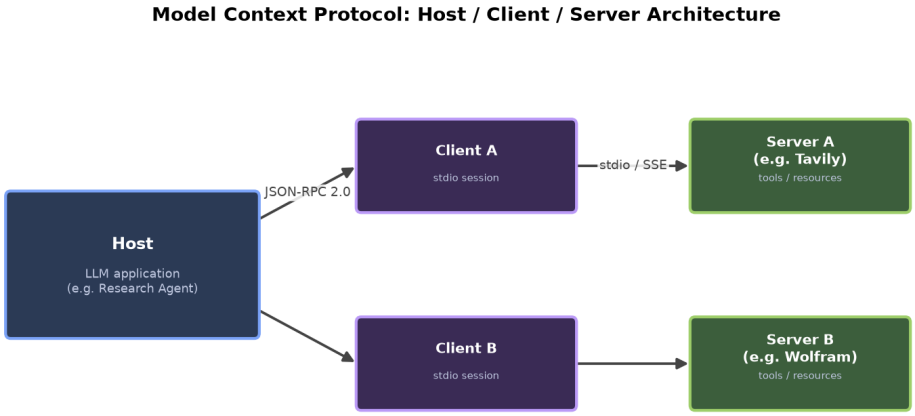

Figure 2.4: Model Context Protocol host/client/server architecture. The host (the research agent) maintains a separate client session per server; each server exposes tools over JSON-RPC 2.0, transported over stdio for local processes or SSE for hosted servers.

Communication uses JSON-RPC 2.0 [7]: *requests* carry a unique ID and method name, *responses* echo that ID with a result or error, and *notifications* are one-way messages requiring no response. Transport is either **stdio** (the server runs as a local subprocess, communicating over standard input/output: used for all thirteen servers in this thesis) or **SSE over HTTP** (for remote, hosted servers). A typical session lifecycle is: (i) the host launches or connects to a server and performs a capability-negotiation handshake; (ii) the host calls `tools/list` to discover the server's available tools and their JSON-Schema argument definitions; (iii) the host calls `tools/call` with a tool name and arguments, receiving a structured result or error; steps (ii)–(iii) can repeat indefinitely within one session.

MCP is directly relevant to this thesis because it is the integration mechanism for every external tool the agent uses (Section 3.2.4): rather than writing thirteen bespoke tool integrations, the agent's `tool_executor` node speaks one protocol to all of them, and new tools can be added by editing a single JSON configuration file (`mcp-servers-config.json`) rather than changing agent code. This had a concrete effect on the development process described in Section 3.1: the expansion of the tool ecosystem from three servers in the initial prototype to thirteen at the time of evaluation required no changes to the `tool_dispatcher` or `tool_executor` node implementations at all, only configuration and prompt updates.

2.5.2 Agent-to-Agent Protocol (A2A)

Google's Agent2Agent Protocol [8, 9] addresses a complementary problem to MCP: how two *independent* agents, potentially built on different frameworks and run by different organisations, discover each other's capabilities and exchange messages. Where MCP standardises *agent-to-tool* communication, A2A standardises *agent-to-agent* communication. Agents publish "agent cards", JSON metadata describing capabilities, connection details, and security requirements, enabling dynamic discovery. Authenticated agents then exchange JSON-RPC 2.0 messages over HTTPS, supporting both synchronous request/response and long-running asynchronous tasks with explicit status updates (`submitted`, `working`, `input-required`, `completed`, `failed`) [10]. The protocol was subsequently donated to the Linux Foundation to encourage broad, vendor-neutral adoption [9].

While the system in this thesis does not (yet) expose itself as an A2A-discoverable agent, a direction discussed as future work in Section 5.3, the *internal* message contract between the planner, dispatcher, assessor, and synthesizer nodes (Section 3.2.1) borrows A2A's core ideas, typed messages, explicit task/objective state, and structured status reporting (`decision`, `is_complete`, `confidence`), applied at the granularity of roles *within* one agent rather than between separate agents. Section 1.2 discusses how this thesis's scope evolved from an original proposal centred on cross-framework A2A interoperability to this internal-communication framing.

2.5.3 IBM Agent Communication Protocol (ACP)

IBM's Agent Communication Protocol [11, 12] takes a RESTful, local-first approach: agents expose standard HTTP endpoints (`GET/POST/PUT/DELETE`) that map naturally onto agent operations (list available agents, run an agent, retrieve a run's status and output), allowing interaction via ordinary tools (curl, Postman) without specialised SDKs, and emphasising deployability in air-gapped or regulated environments where outbound connections to a central discovery service (as A2A assumes) may not be possible. ACP is positioned by IBM as complementary to both MCP and A2A: MCP for agent-to-tool communication, ACP (or A2A) for agent-to-agent communication, often within the same multi-agent system, with the choice between ACP and A2A largely a question of deployment topology (local-first REST vs. globally discoverable JSON-RPC) rather than a difference in the underlying communicative primitives.

2.5.4 Comparing MCP, A2A, and ACP: A Layered View

Table 2.1 summarises the three protocols along the dimensions most relevant to this thesis. The key observation is that none of the three protocols is a substitute for any

other: they address communication at different *layers* of an agentic system, agent-to-tool (MCP), agent-to-agent across organisational boundaries (A2A), and agent-to-agent within a local deployment (ACP), and a sufficiently complex system could plausibly use all three simultaneously. This thesis's agent uses MCP exhaustively (thirteen servers, Section 3.2.4) but does not yet need A2A or ACP, because all five of its specialised roles (Section 3.2.1) run within a single process as nodes of one LangGraph graph; the internal state object described in Section 3.2.1 fulfils, within that single process, the same typed-communication role that A2A or ACP would fulfil across process or organisational boundaries.

Table 2.1: Comparison of the three agent communication protocols surveyed in this chapter.

	MCP	**A2A**	**ACP**
Communication layer	Agent → tool/-data source	Agent → agent (cross-organisation)	Agent → agent (local-first)
Transport	JSON-RPC 2.0 over stdio or SSE	JSON-RPC 2.0 over HTTPS	RESTful HTTP
Discovery	Static configuration (`tools/list`)	Published "agent cards"	Local agent registry / endpoint listing
Typical use in this thesis	All 13 tool integrations (Section 3.2.4)	Not used; informs internal contract design	Not used

2.6 Multi-Agent Orchestration Frameworks

2.6.1 Google Agent Development Kit

Google's **Agent Development Kit (ADK)** [13] provides a code-first, Pythonic API for composing agents and workflow primitives (sequential, parallel, loop) with built-in OpenTelemetry tracing and first-class support for deploying agents as A2A-discoverable services. ADK's workflow primitives are useful for composing *coarse-grained* multi-agent pipelines (e.g. "run agent A, then agents B and C in parallel, then agent D"), but the framework treats each agent as a relatively opaque unit with its own conversation history; the *internal* reasoning loop of any single agent is left to the developer.

2.6.2 LangChain and Microsoft AutoGen

LangChain [14] provides composable abstractions over LLM calls, prompt templates, tools, and memory, and was among the first frameworks to popularise the "chain" as the basic unit of LLM application composition. **Microsoft AutoGen** [16] models multi-agent systems explicitly as *conversations*: each agent is an object with a role, a system message, and the ability to call functions, and multi-agent collaboration is implemented by agents sending chat messages to each other, with a separate "group chat manager" agent deciding which agent speaks next. This conversational framing is intuitive and maps naturally onto human team metaphors (a "coder" agent, a "reviewer" agent, a "planner" agent exchanging messages), and is widely used for code-generation and software-engineering multi-agent systems.

2.6.3 LangGraph: Graph-Structured State Machines

LangGraph [15], used in this thesis, takes a different approach from AutoGen's conversational model: an application is modelled explicitly as a *graph* of nodes (each an arbitrary function, typically an LLM call) and edges (control flow, including conditionally-routed edges), operating over a shared, typed *state* object that persists across the entire graph execution. Each node receives the current state, performs its computation, and returns a *partial update*, only the fields it changed, which the graph runtime merges into the running state before invoking the next node(s) according to the graph's edges.

This is a closer match to the architecture this thesis argues for than the AutoGen conversational style. Rather than modelling the planner, dispatcher, executor, assessor, and synthesizer as separate "agents" exchanging chat messages, LangGraph lets them be modelled as nodes that read and write fields of one shared state object, with the graph's edges encoding exactly which role hands off to which other role and under what condition (Table 3.7). Three properties of this model are particularly valuable for the design in Section 3.2:

1. **Partial, typed updates.** Because each node returns only the fields it changed, and those fields are typed (via Python dataclasses and Pydantic models, Section 3.2.1), the entire history of a graph execution can be reconstructed as a sequence of typed deltas, which is exactly what the SSE event stream consumed by the UI (Section 3.2.5) exposes.

2. **Conditional routing as data, not code.** The routing table (Table 3.7) is implemented as a small number of pure functions over the state object (e.g. "if `plan.decision == "done"`, route to `synthesizer`"), making the agent's control flow auditable independently of the LLM calls themselves.

3. **Native streaming and checkpointing.** LangGraph's `astream_events` API streams token-level and node-level events as they occur, which the FastAPI backend (Section 3.2.5) forwards directly to the browser as Server-Sent Events, and the graph's state can be checkpointed and resumed: a property exploited by the benchmark harness's resumable runner (Section 3.3.1).

2.6.4 Evaluating Multi-Agent Frameworks

A growing body of work attempts to benchmark multi-agent frameworks directly, rather than the underlying models. **MultiAgentBench** [17] evaluates LLM agents across collaborative and competitive multi-agent scenarios (e.g. negotiation, resource allocation games) and finds that coordination protocol design, not just individual agent capability, is a major determinant of overall task success. **BattleAgentBench** [18] similarly finds that cooperation and competition capabilities vary substantially across frameworks and are sensitive to how inter-agent communication is structured. A recurring observation across this literature is that frameworks emphasising free-form natural-language "conversation" between agents tend to be harder to debug and to introduce non-determinism that confounds evaluation: a single agent's misinterpretation of a free-text message from another agent can cascade through an entire multi-agent run in ways that are difficult to trace after the fact. A typed-state graph, by contrast, makes every inter-role "message" a structured, inspectable object with a fixed schema, which is also what makes the live tool-call visualisation in the UI (Section 3.2.5) possible, and what makes it practical to log and replay individual agent traces during development (Section 3.1).

2.7 Computational and Execution Tools for Agents

Beyond search and retrieval, a research agent frequently needs to perform *exact* computation (arithmetic, unit conversion, symbolic algebra, statistical analysis), where a language model's token-by-token generation is fundamentally unsuited to guaranteeing correctness. This section reviews the three categories of computational tool integrated into the agent's MCP ecosystem (Section 3.2.4): symbolic/numeric computation engines, sandboxed code execution, and browser automation.

2.7.1 Symbolic and Numeric Computation Engines

Wolfram Alpha [41] is a computational knowledge engine that accepts natural-language or symbolic queries (equations, unit conversions, statistical distributions, chemical formulas) and returns exact, structured results computed by an underlying symbolic math-

ematics kernel, rather than retrieved from a text corpus. For an LLM agent, exposing Wolfram Alpha as a tool converts a class of questions that a language model would otherwise have to "guess" the answer to (e.g. "what is $\sqrt{2} \times \pi^2$ to six decimal places?") into a single tool call with a guaranteed-correct result. In this thesis, Wolfram Alpha is one of the two tools (alongside E2B, below) whose presence or absence in the agent's tool roster is shown in Chapter 4 to materially affect performance on benchmarks with a computational component (GAIA, MCP-Atlas).

2.7.2 Sandboxed Code Execution Environments

Where Wolfram Alpha handles closed-form mathematical queries, many research tasks require arbitrary, multi-step computation: loading a dataset, fitting a model, producing a plot, or running a simulation. **E2B** [42] provides exactly this as a service: an isolated, ephemeral cloud sandbox in which arbitrary code (most commonly Python, via a **Jupyter**-style kernel [43]) can be executed, with persistent state (variables, files, installed packages) addressable by a stable sandbox identifier across multiple separate tool calls. This persistence is what makes E2B suitable as the "research notebook" capability required by the original project brief (Chapter 1): the agent can, across several iterations of its planner/dispatcher/assessor loop, incrementally build up a dataframe, refine a plot, or accumulate intermediate results in the same sandbox, in the same way a human researcher would use a single long-running Jupyter session rather than restarting their kernel for every new piece of analysis.

2.7.3 Browser Automation for Web-Scale Information Access

Not all information on the web is accessible through a clean search API: some pages require JavaScript execution, interaction (clicking, scrolling, form submission), or rendering to extract their content. **Playwright** [44] is a browser-automation framework that drives a real (headless) browser engine, allowing an agent to navigate to a URL, wait for dynamic content to load, interact with page elements, and extract the resulting rendered text or structured data. Because driving a full browser is an order of magnitude slower than an API-based search call (typically 3–5 seconds versus roughly 1 second for an AI-native search API such as Tavily), this thesis's dispatcher prompt (Section 3.2.2) treats Playwright as a *fallback* tool, reserved for the subset of queries where a structured search API genuinely cannot retrieve the required content: a design decision whose performance impact is discussed in Section 3.1.

2.8 Benchmarking Research Agents

A central methodological question for this thesis is *how to measure* whether a research agent is good. "Good" is not a single axis: an agent might be excellent at quick factual lookups but poor at synthesising a long report, or vice versa. Six benchmarks were selected to cover complementary failure modes, spanning a lineage of question-answering evaluation that long predates LLM agents.

2.8.1 Short-Form Factuality: SimpleQA

SimpleQA [19] consists of 4,326 short, fact-seeking questions, each with a single, unambiguous, verifiable answer (e.g. "In what year was the Higgs boson first observed experimentally?"), scored by exact or fuzzy string match against a gold answer. Questions are deliberately sourced from the long tail of Wikipedia and similar references, such that even frontier models without search perform poorly: GPT-4o answers fewer than 40% correctly from parametric memory alone. SimpleQA is therefore a clean test of *whether an agent actually uses its search tools effectively*, isolating the retrieval-and-verification capability from any requirement for multi-step reasoning.

2.8.2 Multi-Hop Reasoning: From HotpotQA to FRAMES

Multi-hop question answering, questions whose answer requires combining facts from multiple independent sources, has a long benchmark lineage. **HotpotQA** [45] and **TriviaQA** [46] established the basic format: questions requiring 2–3 hops across Wikipedia articles, with supporting-fact annotations that allow evaluating not just the final answer but the *reasoning path*. **FRAMES** [20] extends this lineage to 824 questions requiring reasoning across 2–15 Wikipedia articles (a substantially longer hop count than HotpotQA's 2–3), explicitly testing whether an agent can decompose a question into a *sequence* of sub-lookups, retain intermediate facts across that sequence, and combine them correctly: precisely the capability the planner's multi-step `ResearchPlan` (Section 3.2.1) is designed to provide, and precisely the capability shown in Chapter 4 to degrade on FRAMES questions requiring the longest hop chains.

2.8.3 General Assistant Tasks: GAIA

GAIA [22] (General AI Assistants) consists of 466 real-world assistant tasks across three difficulty levels (this thesis uses the Level 1+2 subset of 139 tasks), each requiring some combination of web browsing, file handling (reading attached spreadsheets, PDFs, or

images), and tool use to produce a single exact-match final answer. Unlike SimpleQA or FRAMES, GAIA tasks are not drawn from a single information-seeking template; they resemble the kind of ad-hoc, multi-modal requests a human assistant might receive ("How many studio albums were published by [band] before [event]? Use the attached spreadsheet to verify your answer against their official discography."). GAIA is widely used as a general-purpose "can this agent act as an assistant" benchmark and has an active public leaderboard against which the multi-agent baselines in Chapter 4 are compared.

2.8.4 Deep Research and LLM-as-Judge Evaluation: DRACO

DRACO [23] departs from the exact-match paradigm entirely: it consists of open-ended "write me a report on X" prompts (e.g. "Write a report on the current state of solid-state battery research"), for which there is no single correct answer string. Instead, a separate *judge* language model scores each response against a rubric covering accuracy (are the claims correct?), completeness (does the report cover the topic's major facets?), and citation quality (are claims attributed to sources, and are those sources appropriate?). This **LLM-as-judge** methodology has become common for evaluating open-ended generation where exact-match scoring is inapplicable, with the trade-off that the judge model's own biases and blind spots become part of the evaluation. DRACO directly tests the *synthesizer* node's role in this thesis's architecture (Section 3.2.2): the quality of the final cited report, built from the accumulated `tool_results` of potentially several planner iterations.

2.8.5 Tool-Use Competency: MCP-Atlas

MCP-Atlas [24] is a recent (2025) benchmark built directly on real MCP servers, designed to measure *tool-use competency* independent of final-answer correctness: each task has a ground-truth set of tool calls (function names and arguments) that a correct solution would invoke, and an agent's run is scored by what fraction of this "ground-truth function/argument" (GTFA) set its actual tool calls cover. This is a particularly direct test for this thesis's architecture, because the `tool_dispatcher` node's entire function (Section 2.3) is tool selection; Chapter 4 shows that MCP-Atlas performance is highly sensitive to which tools are *available* to the dispatcher in the first place, independent of the orchestration model's quality.

2.8.6 Adversarial Search: BrowseComp

BrowseComp [21] consists of 1,266 deliberately hard, "needle in a haystack" web-search questions, constructed by starting from an answer and working backwards to a question

that is difficult to reverse-search: such that a single query rarely surfaces the answer directly, and even GPT-4o with browsing scores under 2%. BrowseComp is explicitly designed to separate genuinely strong research agents (OpenAI's own Deep Research scores 51.5%) from agents that merely *look* like they are searching (issue plausible queries, retrieve plausible-looking pages, but fail to recognise that the retrieved evidence does not actually answer the question). Chapter 4 discusses this benchmark in detail as the one case where this thesis's agent does not approach the strongest published baseline.

2.9 Synthesis and Positioning of This Thesis

Drawing the threads of this chapter together: large language models are powerful but parametrically bounded reasoning engines (Section 2.1); prompting techniques such as chain-of-thought, ReAct, and reflection provide ways to structure a model's use so that its reasoning is made explicit and its mistakes are caught (Section 2.2); retrieval- and tool-augmentation give the model access to information and computation beyond its training data (Section 2.3); and the multi-agent systems literature, both classical (Section 2.4) and LLM-specific (Section 2.6), suggests that decomposing a complex task across specialised, communicating roles is often more effective and more debuggable than a single undifferentiated loop, *provided* that the communication between those roles is structured rather than free-form.

This thesis's central design choice, a five-node LangGraph agent (Section 3.2) in which planner, dispatcher, assessor, and synthesizer roles communicate through a typed state object built from Pydantic models, sits at the intersection of these strands. It is a ReAct-style loop (Section 2.2) whose single "Thought" step has been decomposed into planning, tool-selection, and self-assessment sub-roles (Section 2.4), connected via the typed-message philosophy of classical agent communication languages and modern protocols (Section 2.5) rather than free-form chat (Section 2.6), grounded by retrieval and computation tools accessed uniformly through MCP (Section 2.7), and evaluated against the benchmark landscape surveyed in Section 2.8. Section 3.1 traces how this design was arrived at in practice, starting from a much simpler prototype and incrementally addressing the failure modes that motivated each of the architectural choices reviewed in this chapter.

Chapter 3

Methodology

3.1 Iterative System Development

The system described in Section 3.2 is the end point of an iterative development process, not its starting point. This chapter documents that process: the initial prototype that was built first, the specific failure modes that prototype exhibited when exercised against realistic research questions, and the sequence of design changes, culminating in the five-node planner/dispatcher/tool_executor/assessor/synthesizer graph, that each addressed one of those failure modes. The purpose of this chapter is twofold. First, it provides an empirical justification for the architectural choices presented in Section 3.2: rather than asserting that the five-node decomposition is superior to a flat ReAct-style loop, this chapter shows *why* it became necessary, grounded in concrete failures observed during development. Second, it records the engineering challenges, premature termination, tool misselection, brittle structured-output parsing, and tool-ecosystem growth, that are typically invisible in a finished system but consumed a substantial fraction of the total development effort, and which future extensions of this work (Section 5.3) will likely re-encounter.

3.1.1 Development Methodology

The system was developed through a sequence of design iterations, each consisting of: (i) implementing a graph topology and prompt set; (ii) exercising it manually against a small, fixed set of research questions spanning factual lookup ("Who won the 2023 Nobel Prize in Physics?"), multi-hop reasoning ("What is the population of the country whose capital hosted the 2024 Summer Olympics?"), and light computation ("What is the compound annual growth rate implied by a value that doubles in 6 years?"); and (iii) inspecting the resulting execution trace, the full sequence of LLM calls, tool calls, and intermediate state values, to identify where the agent's behaviour diverged from what a careful human researcher would do. This manual, trace-driven diagnosis was deliberately preferred over jumping directly to full benchmark runs (Section 3.3) for early iterations, for two reasons. First, the benchmarks used in Chapter 4 involve dozens to hundreds of questions and an LLM-as-judge scoring step (Section 2.8); running a full benchmark after every small prompt change would have been prohibitively slow and would have produced

an aggregate score without indicating *why* that score changed. Second, the earliest failure modes (Section 3.1.3) were severe and qualitative (the agent would terminate after a single tool call regardless of whether that call answered the question) and were visible on the very first manually-inspected trace, making a quantitative benchmark unnecessary to detect them. Quantitative benchmark comparisons (Section 3.1.8) were introduced once the qualitative failure modes of the flat prototype had been addressed by the five-node redesign, as a way of confirming that the redesign's benefits were not purely anecdotal.

3.1.2 Phase One: A Flat Three-Node Prototype

The initial implementation, summarised in Fig. 3.1, used a flat, three-node LangGraph graph operating over a minimal state object containing only the conversation history and a list of available MCP server names:

- **`select_servers`**: a single LLM call that examined the user's question and the list of configured MCP servers, and returned a subset of server names judged relevant to the question.
- **`mcp_orchestrator`**: a single LLM call, given the conversation history and the set of tools exposed by the selected servers (via `model.bind_tools(tools)`), that either emitted a tool call or, if it judged the question answered, emitted a final natural-language response.
- **`mcp_tool_call`**: executed whichever tool call the orchestrator had emitted, via the MCP client session for the corresponding server, and appended the result to the conversation history as a tool message.

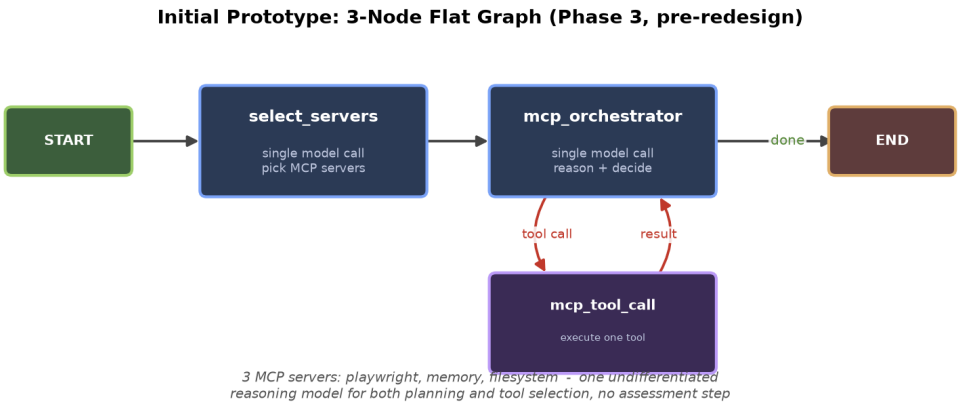

Figure 3.1: The initial three-node prototype graph. `select_servers` ran once at the start of a session; `mcp_orchestrator` and `mcp_tool_call` then formed a tight ReAct-style loop, with `mcp_orchestrator` deciding after every tool result whether to call another tool or to terminate with a final answer.

Control flow was a direct application of the ReAct pattern (Section 2.2): `mcp_orchestrator` and `mcp_tool_call` formed a loop, with the orchestrator's own judgement, expressed implicitly, by whether its response contained a tool call or not, determining when the loop terminated. At this stage the tool ecosystem (Section 3.2.4) consisted of only three MCP servers: `playwright` (browser automation), `memory` (a simple key-value knowledge store), and `filesystem` (local file read/write). A single model, the same model for `select_servers` and `mcp_orchestrator`, handled all reasoning.

This prototype was sufficient to demonstrate the basic MCP integration: the agent could, for example, be asked "What is the current weather in Brisbane?", select the `playwright` server, navigate to a weather website, extract the relevant text, and report it. However, exercising the prototype against the slightly harder questions described in Section 3.1.1 surfaced four distinct, recurring failure modes, described in the next section.

3.1.3 Diagnosing the Prototype's Failure Modes

3.1.3.1 Premature Termination

The most immediately visible failure was that `mcp_orchestrator` frequently terminated, emitting a final answer with no further tool call, after a *single* tool call, even when that call's result did not actually contain the information needed to answer the question. For example, asked a multi-hop question requiring the population of a country identified via an intermediate fact, the orchestrator would issue one search query, receive a page of results about the intermediate fact (the host country), and then synthesise a final answer

that simply did not address the population component of the question, apparently because the model's per-step decision of "tool call vs. final answer" was made by the *same* call that was meant to be reasoning about task completion, and that call had a documented tendency to prefer emitting *some* answer over emitting another tool call, particularly once the conversation history had grown long enough that the original question was no longer prominent in the model's context.

This is precisely the failure mode that the literature on self-correction (Section 2.2) predicts for a single undifferentiated reasoning loop: there is no point in the loop where the model is *specifically* prompted to ask "does this result actually satisfy the original question?" as a distinct judgement from "what should happen next?". The two questions were conflated into one LLM call, and the call's training-time bias toward concise, conversational closure dominated.

3.1.3.2 Tool Misselection and Redundant Calls

A second failure mode was visible in the choice of *which* tool to call. With `playwright` as the primary information-retrieval tool, the orchestrator would frequently navigate to a search engine's results page, extract a list of link titles and snippets, and then either (a) treat the snippets themselves as sufficient evidence without visiting any of the linked pages, or (b) navigate to an unrelated or low-quality page from the results list. Because `select_servers` ran only once, at the start of the session, the orchestrator had no opportunity to revise its choice of tools mid-session even after observing that the chosen tools were producing unhelpful results: if `playwright` navigation kept landing on irrelevant pages, the only available alternative tools were `memory` and `filesystem`, neither of which could retrieve new information from the web.

3.1.3.3 No Mechanism for Self-Correction

Related to both failures above, the prototype had no explicit mechanism for the agent to recognise that its current approach was not working and to *change* that approach. The Reflexion and Self-Refine literature (Section 2.2) frames this as a missing reflection step: there was no point in the prototype's loop where the agent's own progress was evaluated against the original goal and, if found wanting, used to generate a *different* next action. `mcp_orchestrator`'s implicit per-step decision (tool call vs. final answer) was the closest analogue, but as discussed above, this decision was not reliably sensitive to whether the accumulated evidence was actually sufficient; it was a decision about *format* (continue the tool-use pattern or switch to prose) more than a decision about *completeness*.

3.1.3.4 Structured-Output Parsing Failures

A fourth, more implementation-level failure mode emerged once the prototype's single LLM call was asked to produce anything beyond free-form text or a single tool call, for example, early experiments with having `select_servers` return a structured list of server names plus a rationale, using LangChain's `with_structured_output(..., method="function_calling")`. For most frontier models this worked reliably, but it was not *universally* reliable: occasionally a model would respond with a plain-text explanation instead of invoking the expected structured-output function, in which case `with_structured_output` returned `None`, and the calling code, which assumed a populated Pydantic object, raised an `AttributeError` on the very next line, crashing the entire graph execution.

This failure was rare enough to be easy to miss in a handful of manual trace inspections, but frequent enough to be a serious problem for any unattended run: including, critically, the benchmark harness (Section 3.3), where a single crashed task would otherwise abort an entire batch. The fix adopted, described in Section 3.1.7, became one of the most broadly useful pieces of infrastructure in the final system, because the same failure mode recurs at *every* structured-output call site in the five-node graph (`PlannerResult`, `AssessorResult`, and the dispatcher's tool-call selection), not just `select_servers`.

3.1.4 Phase Two: Decomposing the Loop into Five Roles

The four failure modes of Section 3.1.3 share a common root cause: a single LLM call was simultaneously responsible for planning (what should the agent be trying to find out right now?), tool selection (which tool, with which arguments, best serves that goal?), and self-assessment (has that goal now been met?), with no structural separation between these concerns and therefore no point at which any one of them could be evaluated or retried independently. The redesign addressed this by giving each concern its own node, its own prompt, and its own structured-output schema, yielding the five-node graph (`planner` → `tool_dispatcher` → `tool_executor` → `assessor` → ...) described in full in Section 3.2.2. Table 3.1 summarises the correspondence between the prototype's failure modes and the new architecture's response to each.

Table 3.1: Mapping from prototype failure modes (Section 3.1.3) to the architectural response in the five-node redesign (Section 3.2.2).

Failure mode	**Root cause**	**Architectural response**
Premature termination	"Continue vs. stop" decided by the same call doing the next action	`assessor` node makes an explicit, separately-prompted `is_complete`/`confidence` judgement after every tool call
Tool misselection / redundant calls	Tool set fixed once at session start; no mid-session revision	`tool_dispatcher` re-selects a tool on *every* iteration, conditioned on the assessor's `feedback` and `missing` fields from the previous iteration
No self-correction	No reflection step comparing progress to the original goal	`assessor` → `tool_dispatcher` retry loop (Section 3.2.2), directly modelled on the Reflexion pattern (Section 2.2)
Structured-output crashes	`with_structured_output` can return `None`	`_invoke_structured()` helper (Section 3.1.7) wraps every structured call with retry-then-default behaviour

Two further changes accompanied the four direct responses in Table 3.1. First, the minimal state object of the prototype (conversation history plus server list) was replaced by the much richer typed state described in Section 3.2.1, most importantly adding an explicit `ResearchPlan` (a list of steps, a current step index, a current focus, and a `decision`

field) and an accumulating `tool_results` list (Listing 3.1). The `ResearchPlan` gives the `planner` node a place to decompose a question into multiple steps *up front*, directly addressing the multi-hop case from Section 3.1.3, where the prototype lost track of the population sub-question after resolving the host-country sub-question, while `tool_results` gives the `synthesizer` node access to the full evidence trail rather than only the most recent tool result.

Second, the planner was given an explicit `decision` field with three values: `new` (decompose the question into a fresh plan), `continue` (advance to the next step of an existing plan), and `done` (sufficient evidence has been gathered across all steps; proceed to synthesis). This gives the planner, and *only* the planner, the authority to decide when the overall research task is complete, separating that authority from the per-step `is_complete` judgement made by the assessor. In the final architecture, a step can be individually "complete" (the assessor is satisfied with the evidence gathered for that step) without the *plan* being "done" (the planner may still have further steps to execute), and conversely the planner can declare a plan `done` even if the most recent step's assessor confidence fell short of the 0.7 threshold, once the iteration budget (Section 3.2.2) is exhausted: preventing the agent from looping indefinitely on a step it cannot resolve.

3.1.5 Expanding the Tool Ecosystem

The MCP tool roster (Section 3.2.4) grew from the prototype's three servers to the thirteen evaluated in Chapter 4 in three broad stages, each motivated by a specific gap observed during development or during early benchmark runs.

Stage 1 (prototype): `playwright`, `memory`, `filesystem`. As described in Section 3.1.2, this minimal set established the MCP integration but left the agent reliant on browser automation for all information retrieval: slow (Section 2.7) and prone to the navigation failures described in Section 3.1.3.

Stage 2 (core research loop): `tavily`, `e2b`, `wolfram`, `arxiv`, `fetch`, `thinking`. Once the five-node architecture (Section 3.1.4) was in place, the next priority was giving the `tool_dispatcher` a fast, reliable default search tool: `tavily` provides an AI-native search API returning clean, pre-summarised results in roughly one second, an order of magnitude faster than driving a browser via `playwright` for the same query (Section 2.7). `wolfram` and `e2b` were added together to close the computational gap identified in Section 2.1; the agent's parametric arithmetic is unreliable, and neither tool existed in the prototype at all. `arxiv` and `fetch` were added to support literature-style research questions (retrieving and reading academic papers, and extracting clean text from arbitrary URLs respectively), and the custom `thinking` server (Section 3.2.4.1) was introduced to give the agent access to an explicit extended-reasoning call for sub-problems that benefit from a dedicated

"think hard about this" step, separate from the lightweight planning and dispatching calls.

Stage 3 (benchmark-driven additions): `wikipedia`, `duckduckgo`, `clinicaltrials`, `weather`. The final four servers were added in response to specific gaps observed during early, small-scale benchmark runs (Section 3.3). `wikipedia` and `duckduckgo` provide alternative retrieval paths to `tavily`: useful both as a fallback when `tavily`'s results were insufficient and, during the MCP-Atlas evaluation (Section 2.8), because some benchmark tasks specifically expected one of these sources in their ground-truth tool-call set. `clinicaltrials` and `weather` are narrower, domain-specific tools added after observing that a handful of GAIA and DRACO tasks (Section 2.8) involved questions in these domains that general web search answered less precisely than a dedicated API.

Throughout all three stages, the MCP integration pattern described in Section 2.5.1 meant that adding a server required only an entry in `mcp-servers-config.json` and, where relevant, a sentence in the dispatcher's system prompt describing when to prefer the new tool; no change to the `tool_dispatcher` or `tool_executor` node implementations themselves. This configuration-driven extensibility was not an initial design goal so much as a direct consequence of adopting MCP (Section 2.5.1), and its value became apparent only retrospectively, once Stage 3's additions could be made in the course of a single afternoon of benchmark-driven iteration.

3.1.6 Model Assignment: Lightweight and Heavy Roles

The prototype used a single model for both `select_servers` and `mcp_orchestrator`. As the five-node graph took shape, an early design question was whether each of the five nodes should use the same model, or whether different nodes warranted different models. Two observations drove the eventual lightweight/heavy split described in Section 3.2.1.

First, the `planner`, `tool_dispatcher`, and `assessor` nodes each perform a comparatively narrow, well-specified task (decompose a question into steps, select one tool from a known list and fill in its arguments, or judge whether a specific piece of evidence satisfies a specific sub-goal), and each of these nodes may be invoked many times within a single research session (once per plan step, and again on every assessor-triggered retry). Early experiments using a frontier-tier model for all five nodes produced noticeably higher per-query latency and cost, dominated by these narrow, high-frequency calls, without a correspondingly large quality improvement on the narrow tasks themselves: a smaller, faster model proved capable of reliably producing a `ResearchPlan` or selecting between thirteen well-described tools, provided its prompt was specific enough.

Second, the `synthesizer` node is invoked exactly once per session, performs the single most consequential generation (the final, cited answer that is actually scored by every

benchmark in Chapter 4), and benefits from the strongest available reasoning and writing quality far more than the narrow per-iteration roles do. This led to the final assignment: `planner`, `tool_dispatcher`, and `assessor` use a *lightweight* model (configurable independently per role, Section 3.2.1), while `synthesizer` uses a *heavy* model. Section 3.3.5 describes the specific model assignments used for each experimental configuration in Chapter 4, and the configuration system (Section 3.2.5) allows each of the four roles' models to be changed independently without code changes, which proved valuable for the configuration sweep presented in Chapter 4.

3.1.7 Robustness: The `_invoke_structured()` Pattern

The structured-output crashes described in Section 3.1.3 became more, not less, pressing after the five-node redesign, because the redesign *introduced two additional* structured-output call sites (`PlannerResult` and `AssessorResult`, alongside the dispatcher's tool-call selection) where the prototype had only one. Any of these three call sites failing would crash the corresponding node, and, because the benchmark harness (Section 3.3.1) runs many tasks in sequence or in parallel, a crash in any single task's graph execution risked aborting that task's contribution to an entire benchmark run.

The adopted solution, `_invoke_structured()`, wraps every structured-output call site with a uniform retry-then-default policy: (i) call `model.with_structured_output-(Schema, method="function_calling")` as normal; (ii) if the result is `None` (the model responded with plain text rather than invoking the structured-output function), retry once with an additional system message explicitly reminding the model that a structured response in the given schema is required; (iii) if the retry *also* returns `None`, fall back to a schema-appropriate default value: for `PlannerResult`, a single-step plan treating the entire original question as the focus of that one step; for `AssessorResult`, `is_complete=False` with the original question echoed back as the sole `missing` item, ensuring the agent retries rather than silently terminating.

This pattern is deliberately general: it is not specific to any one model or model family, but addresses a property of the function-calling interface itself (Section 2.2); that "the model invokes the requested structured-output function" is, for some models and some prompts, a high-probability but not *guaranteed* event. By converting an unconditional `AttributeError` crash into a logged, bounded-retry-then-graceful-degradation path, `_invoke_structured()` turned an intermittent and previously fatal failure into a rare and recoverable one, and was the single change most directly responsible for making unattended, multi-hundred-task benchmark runs (Chapter 4) practical at all.

3.1.8 A Pilot Comparison: Prototype vs. Final Architecture

To confirm that the qualitative improvements described in Section 3.1.4 translated into a measurable difference, a small pilot study was run once both architectures were available: ten questions drawn from the SimpleQA development set (Section 2.8) were run against both the three-node prototype (Section 3.1.2, Stage 1 tool roster) and the five-node final architecture (Stage 2 tool roster, lightweight/heavy model split as in Section 3.1.6), with both configurations using the same underlying models for comparable roles. Table 3.3 summarises the result.

Table 3.3: Pilot comparison on ten SimpleQA development questions. "Correct" indicates the final answer matched the gold answer under the SimpleQA grading criteria (Section 2.8); "Premature stop" indicates the agent terminated after a single tool call without addressing the full question.

Configuration	**Correct (/10)**	**Premature stop (/10)**	**Mean tool calls / question**
Three-node prototype (Section 3.1.2)	2	6	1.3
Five-node final architecture (Section 3.1.4)	6	0	3.1

The pilot's small sample size (ten questions) means Table 3.3 should be read as a confirmatory sanity check rather than a statistically powered result: Chapter 4 presents the properly-scaled benchmark evaluation. Nonetheless, the pattern is exactly what Section 3.1.3 predicts: the prototype's premature-termination failure mode accounts for the majority of its incorrect answers (six of eight), and is eliminated entirely once the assessor's explicit `is_complete` judgement replaces the orchestrator's implicit continue/stop decision. The corresponding near-tripling of mean tool calls per question reflects the assessor-triggered retry loop (Table 3.1) doing what it was designed to do: continuing to gather evidence until a sub-goal is actually satisfied, rather than stopping as soon as *any* evidence has been gathered.

3.1.9 Summary of the Development Trajectory

Table 3.5 summarises the overall trajectory from the initial prototype to the system evaluated in Chapter 4. Three themes recur across the individual changes documented in this chapter. First, every major architectural change was a response to a *specific, observed*

failure on realistic questions, rather than a speculative improvement: the five-node decomposition exists because the three-node prototype's conflated planning/acting/assessing decision demonstrably produced premature termination, not because the literature surveyed in Chapter 2 suggested it in the abstract (though, as Section 2.9 discusses, that literature does retrospectively explain *why* the observed failures occurred and *why* the chosen response addressed them). Second, the MCP-based tool integration (Section 2.5.1) made the tool-ecosystem growth documented in Section 3.1.5, from three servers to thirteen: a configuration exercise rather than a software-engineering one, which materially shaped how quickly the system could be iterated. Third, robustness infrastructure such as `_invoke_structured()` (Section 3.1.7), while unglamorous, was a precondition for every quantitative result in Chapter 4: none of the benchmark runs presented there would have completed without it.

Table 3.5: Summary of the development trajectory from prototype to final architecture.

Dimension	**Initial prototype (Section 3.1.2)**	**Final architecture (Section 3.2)**
Graph topology	3 nodes, flat ReAct loop	5 nodes, planner/dispatcher/executor/assessor/synthesizer
State object	Conversation history + server list	Typed state with `ResearchPlan`, `tool_results`, assessor feedback (Section 3.2.1)
MCP servers	3 (`playwright`, `memory`, `filesystem`)	13 (Section 3.2.4)
Tool selection	Once per session (`select_servers`)	Every iteration (`tool_dispatcher`), feedback-conditioned
Self-correction	None (implicit in orchestrator)	Explicit assessor → retry loop (Section 3.2.2)
Model assignment	Single model, all roles	Lightweight (planner/dispatcher/assessor) vs. heavy (synthesizer) (Section 3.1.6)
Structured-output robustness	None (unhandled `None` crashes)	`_invoke_structured()` retry-then-default (Section 3.1.7)

The remainder of this thesis describes and evaluates the right-hand column of Table 3.5 in detail: Section 3.2 presents the final architecture as a system in its own right, Section 3.3 describes how it was evaluated, and Chapter 4 presents the resulting benchmark scores.

3.2 System Design and Implementation

This chapter describes the implemented system: a Python LangGraph agent (`agent/`), a FastAPI backend (`api/`), and a Next.js web application (`ui/`). Fig. 3.2 shows the overall deployment architecture; the following sections describe each layer from the inside out, starting with the agent's internal state and graph (Section 3.2.1, Section 3.2.2), then the

MCP tool ecosystem (Section 3.2.4), and finally the full-stack application (Section 3.2.5).

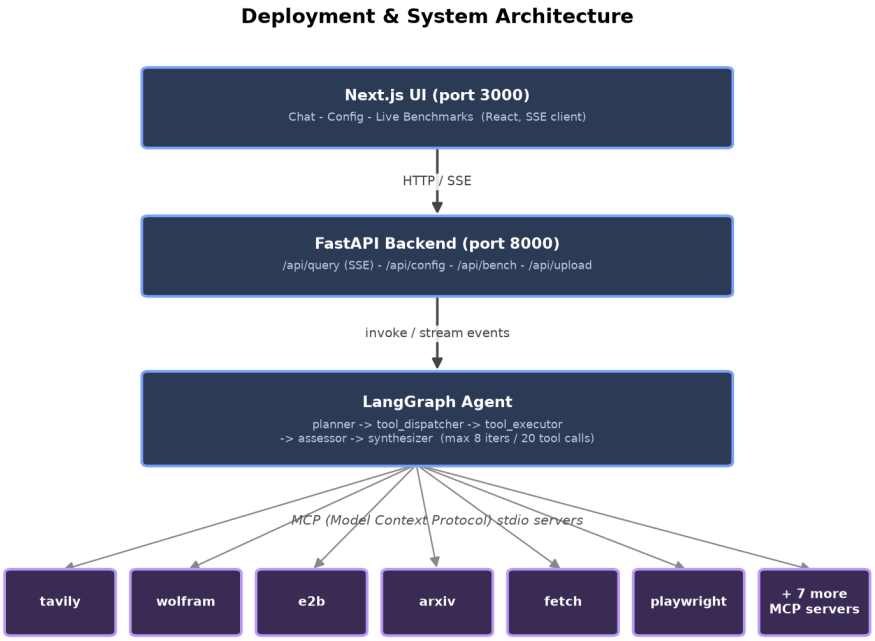

Figure 3.2: Deployment architecture. The Next.js UI communicates with the FastAPI backend over HTTP and Server-Sent Events (SSE); the backend invokes the LangGraph agent directly as a Python library; the agent talks to thirteen MCP servers over stdio.

3.2.1 Design Rationale: Five Roles, One State Object

The agent is structured around a single shared *state* object that is threaded through five specialised nodes. Each node reads the fields it needs, calls an LLM (and, for one node, external tools), and returns *only the fields it changed*: LangGraph merges these partial updates into the running state. This is the concrete realisation of the "structured agent-to-agent communication" framing from Chapter 1: instead of the planner, dispatcher, executor, assessor, and synthesizer exchanging free-form chat messages, they communicate by reading and writing typed fields of a shared, inspectable object.

The two most important structured types are the **research plan** that the planner produces and maintains, and the **assessor result** that gates whether the loop continues, retries, or terminates.

Listing 3.1: Core state and result types (simplified from `state.py`).

```
@dataclass
class ResearchPlan:
    steps: list[str]              # 2-5 concrete sub-tasks
    current_step: int             # index into steps
```

```
    current_focus: str                # natural-language
        description
    decision: Literal["new", "continue", "done"]

class PlannerResult(BaseModel):
    decision: Literal["new", "continue", "done"]
    steps: list[str]
    current_focus: str

class AssessorResult(BaseModel):
    is_complete: bool
    confidence: float = Field(ge=0.0, le=1.0)
    feedback: str
    missing: list[str]                # follow-up queries if
        incomplete

@dataclass
class State(InputState):
    plan: Optional[ResearchPlan] = None
    assessor_feedback: str = ""
    tool_results: list[dict] = field(default_factory=list)
    iteration: int = 0
    tool_calls_made: int = 0
```

This design has a direct practical consequence: because `PlannerResult` and `AssessorResult` are Pydantic models obtained via `model.with_structured_output(...)`, every planning and assessment decision the agent makes is a validated, typed, loggable object, which is what allows the UI (Section 3.2.5) to render a "Research Plan" card and expandable "PlannerResult" / "AssessorResult" tool-call cards directly from the agent's internal reasoning, with no special-casing.

3.2.2 The Five-Node Agent Graph

The agent graph (Fig. 3.3) consists of five nodes: **planner**, **tool_dispatcher**, **tool_executor**, **assessor**, and **synthesizer**. The planner, dispatcher, and assessor run on a *lightweight* model (default `openai/gpt-4.1-mini`); the synthesizer runs on a *heavy* model (default `anthropic/claude-sonnet-4-5`). This split is the system's primary cost/latency lever: the orchestration loop, which may run for up to `max_iterations` (default 8) cycles and `max_tool_calls` (default 20) tool invocations, uses a model that responds in well under a second, while the expensive model is invoked exactly once per query.

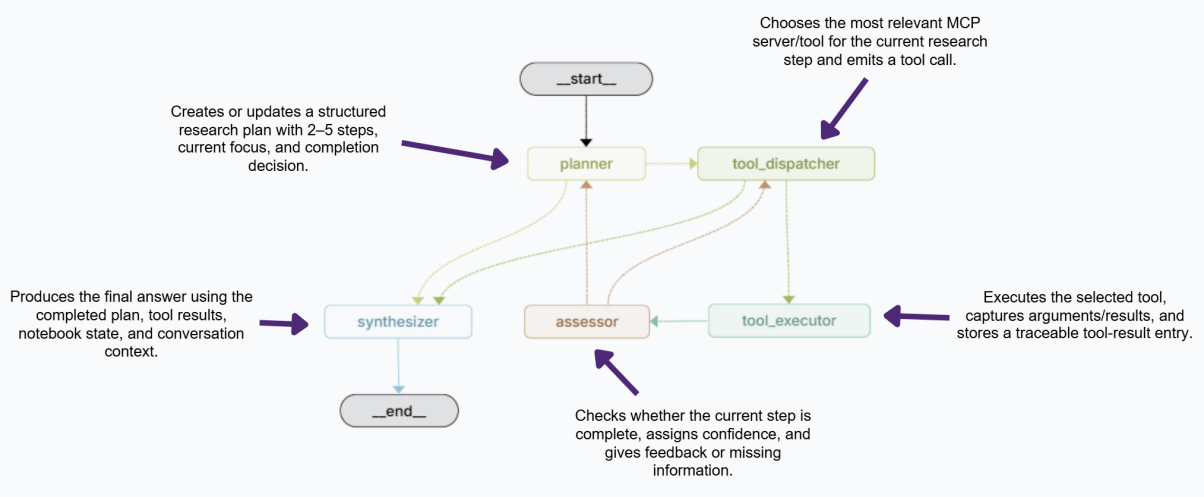

Figure 3.3: The five-node agent graph. Solid black arrows show the default forward chain; coloured arrows show the conditional routing edges emitted by the planner and assessor (see Table 3.7).

3.2.2.1 Node Descriptions

planner. Given the user's question, the conversation so far, and (on later iterations) the assessor's feedback, the planner produces a `PlannerResult`: either `decision="done"` (the question can be answered from general knowledge or the evidence gathered so far is sufficient: route directly to the synthesizer), or `decision="new"/"continue"` with a list of 2–5 concrete steps and a `current_focus` describing the immediate sub-task (e.g. "academic (arxiv): search for recent papers on retrieval-augmented generation, focusing on 2024–2025").

tool_dispatcher. Given `current_focus`, selects exactly one MCP tool and its arguments via `model.bind_tools(tools)`, where `tools` is the live list of tool schemas obtained from all enabled MCP servers (Section 3.2.4). The prompt instructs the dispatcher to prefer Tavily for general web search, Wolfram/E2B for anything numerical, and Playwright only for JavaScript-heavy pages that Tavily cannot reach.

tool_executor. Executes the selected tool call via `mcp.RunTool(name, **args)` against the appropriate MCP server, and appends the raw result to `state.tool_results`.

assessor. Given `current_focus` and the latest tool result, produces an `AssessorResult`: `is_complete` and `confidence` (a complete step requires `confidence` ≥ 0.7), `feedback` (what was learned), and `missing` (follow-up queries if incomplete). This is the system's primary self-correction mechanism: a confident-but-wrong answer to the dispatcher's tool call does not silently propagate forward.

synthesizer. Invoked once, with the full `tool_results` history and the original question, the synthesizer (heavy model) produces the final answer with inline citations to source

URLs. Its prompt instructs it to show working for computational questions and to give a single short phrase for fact-lookup questions, which matters for exact-match benchmarks (Section 2.8).

3.2.2.2 Routing Logic

Table 3.7 summarises the conditional edges shown in Fig. 3.3. The iteration and tool-call budgets (`max_iterations=8`, `max_tool_calls=20` by default, configurable from the UI, Section 3.2.5) act as a safety valve: regardless of the assessor's judgement, once either budget is exhausted the graph routes directly to the synthesizer with whatever evidence has been gathered.

Table 3.7: Routing logic for the agent graph's conditional edges.

From	Condition	Routes to
planner	`decision == "done"`	synthesizer
planner	`decision` $\in$ {new, continue}	`tool_dispatcher`
tool_dispatcher	always	tool_executor
tool_executor	always	assessor
assessor	`is_complete` & more plan steps remain	planner (advance to next step)
assessor	`is_complete` & all steps done	synthesizer
assessor	`not is_complete`, budgets remain	`tool_dispatcher` (retry with `missing` as guidance)
assessor	budgets exhausted (iteration ≥ 8 or tool calls ≥ 20)	synthesizer (forced)

3.2.2.3 Prompt Engineering for the Orchestration Loop

Because the planner, dispatcher, and assessor are differentiated entirely by prompt rather than by model or fine-tuning (Section 2.1), the wording of their system prompts is itself a design artefact, and one that was iterated on heavily during development (Section 3.1).

Three principles, distilled from that iteration, govern the prompts used in the final system; full prompt text is given in Appendix A.

Principle 1: separate the question being asked from the format of the answer. Each of the three orchestration prompts is structured as a short paragraph describing *what* the node should decide, followed by an explicit description of the output schema fields and what each field means, rather than relying on the model to infer the schema's intent from its field names alone. For the assessor, for example, the prompt explicitly states that `confidence` should reflect "how certain you are that the `current_focus` has been fully addressed by the tool result above, not how certain you are that the tool call succeeded", a distinction that early iterations of the prompt did not make, and which (per Section 3.1.3) was directly implicated in the premature-termination failure mode of the original prototype, where a tool call that *executed successfully* was conflated with a tool call that *answered the question*.

Principle 2: make tool-selection criteria concrete and ordered, not abstract. The dispatcher prompt does not simply list the thirteen available tools (Table 3.9); it gives an explicit preference ordering with concrete triggering conditions: "prefer `tavily_sear--ch` for any general information lookup; use `wolfram_alpha` whenever the focus involves a numeric computation, unit conversion, or equation, even a simple one; use `playwright_nav--igate` only if `tavily_search` and `fetch` have both failed to surface the needed page". This ordering directly reflects the latency and reliability trade-offs discussed in Section 2.7 and Section 3.1.5, and was tuned by observing, in development traces, cases where the dispatcher reached for `playwright` first despite a faster tool being available and equally suitable.

Principle 3: prompts reference the *shape* of upstream state, not just its content. Because every node receives a slice of the shared `State` object (Section 3.2.1) rather than a free-form transcript, each prompt is written to explicitly describe the structure of what it is given: e.g. the assessor's prompt begins "You will be given: (1) the current research focus, a one-sentence description of what this step is trying to discover; (2) the tool that was called and its arguments; (3) the raw result of that tool call." This mirrors the typed-message philosophy discussed in Section 2.5: just as MCP, A2A, and ACP messages carry an explicit type that determines how their content should be interpreted, each prompt in this system explicitly names the fields of the state slice it is interpreting, rather than presenting that slice as undifferentiated text and relying on the model to infer its structure.

3.2.3 Configuration System

Every parameter discussed in Section 3.2.2 and Section 3.2.4, the four per-role model assignments, the set of enabled MCP servers, the iteration and tool-call budgets, and an optional system-prompt override, is exposed as a single JSON document, `agent/config.json`, read by the LangGraph agent at the start of every invocation and writable by the FastAPI backend's `PUT /api/config` endpoint (Table 3.11). Listing 3.2 shows a representative configuration corresponding to the "strong" experimental configuration of Section 3.3.5.

Listing 3.2: Representative `agent/config.json`, corresponding to the "strong" configuration of Section 3.3.5.

```
{
  "planner_model": "anthropic/claude-haiku-4-5",
  "dispatcher_model": "anthropic/claude-haiku-4-5",
  "assessor_model": "anthropic/claude-haiku-4-5",
  "synthesizer_model": "anthropic/claude-sonnet-4-5",
  "enabled_servers": [
    "tavily", "playwright", "e2b", "wolfram", "arxiv",
    "fetch", "thinking", "memory", "filesystem",
    "wikipedia", "duckduckgo", "clinicaltrials", "weather"
  ],
  "max_iterations": 8,
  "max_tool_calls": 20,
  "system_prompt_override": null
}
```

This configuration-as-data design has two consequences that proved important beyond their original UI-driven motivation. First, it is what makes the experimental configurations of Section 3.3.5 reproducible and diffable: the "baseline" and "strong" configurations referenced throughout Chapter 4 are, concretely, two versions of this one JSON file, differing only in `enabled_servers` (whether `wolfram` and `e2b` are present) and the three orchestration-role models. Second, because `config.json` is read fresh on every agent invocation rather than baked into a compiled binary or a long-lived process's startup arguments, the benchmark harness (Section 3.3.1) can run an entire benchmark, edit `config.json`, and immediately run the same benchmark again under a different configuration with no restart, which is precisely how the baseline-vs-strong comparisons for GAIA and MCP-Atlas (Section 4.3, Section 4.5) were produced.

3.2.4 MCP Tool Ecosystem

All external capabilities are integrated through the Model Context Protocol (Section 2.5), configured declaratively in `mcp-servers-config.json`. Fig. 3.4 shows the thirteen servers wired into the agent at the time of evaluation. Each server is started as a stdio subprocess on demand via the wrapper in `research_agent.mcp.wrapper`, which exposes three primitives used throughout the agent:

Listing 3.3: MCP wrapper usage pattern (`research_agent/mcp/wrapper.py`).

```
from research_agent.mcp import wrapper as mcp

# 1. Discover available tools from a server (OpenAI tool-call format)
tools = await mcp.apply(server_name, server_config, mcp.GetTools())

# 2. Invoke a tool by name with keyword arguments
result = await mcp.apply(
    server_name, server_config,
    mcp.RunTool("tavily_search", query="who invented the transformer"),
)

# 3. Convenience: open a page and return cleaned text + links as JSON
result = await mcp.navigate_and_read(
    playwright_config, "https://en.wikipedia.org/wiki/Transformer_(deep_learning)"
)
```

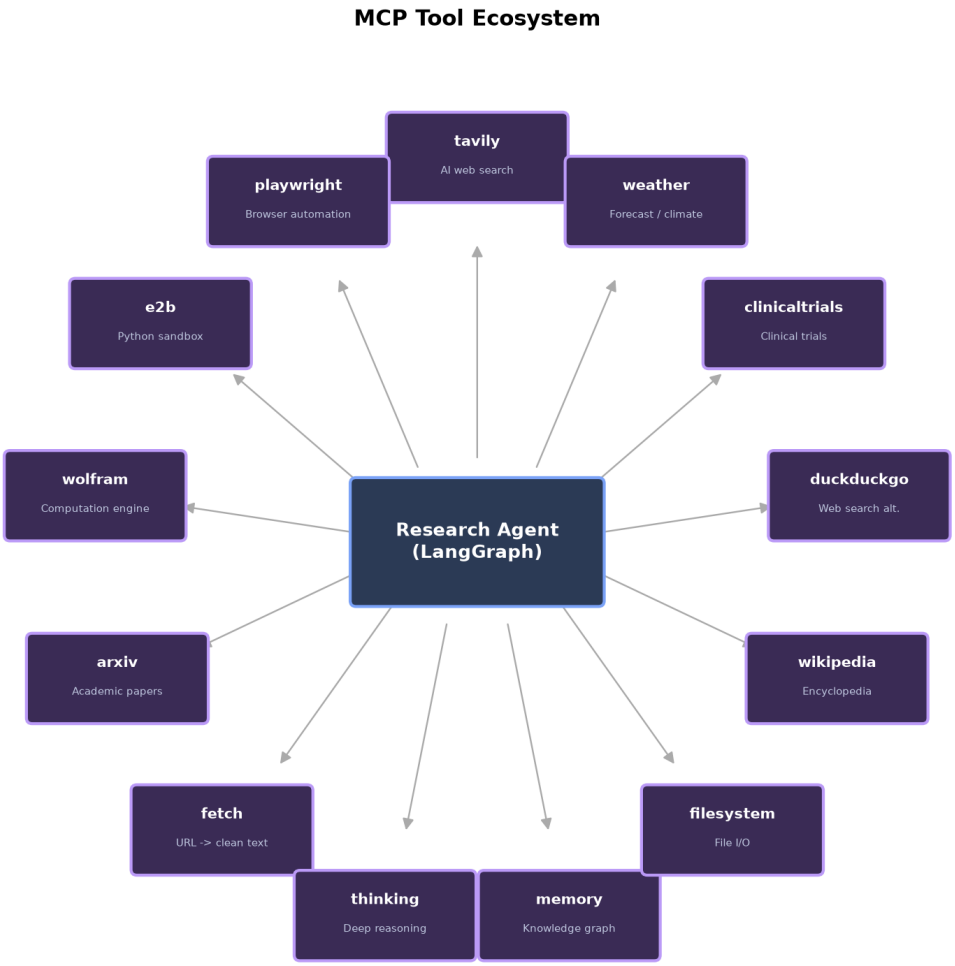

Figure 3.4: The thirteen MCP servers available to the agent, grouped by capability: web search and browsing (tavily, playwright, fetch, duckduckgo, wikipedia), computation (wolfram, e2b), domain literature/data (arxiv, clinicaltrials, weather), reasoning delegation (thinking), and state (memory, filesystem).

Table 3.9 summarises each server's role. Three choices are worth highlighting. First, **Tavily** (an AI-native search API returning clean structured excerpts) is preferred over Playwright-based search for almost all queries, since it returns results in roughly 1 s versus 3–5 s for a rendered browser session, and the dispatcher prompt explicitly says so. Second, **E2B** provides a persistent sandboxed Python notebook addressed by a stable `sandbox_id`, so the agent can accumulate state (variables, intermediate dataframes, plots) across multiple tool calls within one research session; this is the "research notebook" capability required by the project brief. Third, the **thinking** server is a custom MCP server (not a third-party integration) described in Section 3.2.4.1.

Table 3.9: MCP tool ecosystem.

Server	Type	Primary use
tavily	API	AI-optimised web search; default first choice for any lookup
playwright	Browser	JS-heavy pages, or a specific URL Tavily cannot reach
e2b	Sandbox	Persistent Python notebook: data analysis, plotting, simulation
wolfram	API	Exact symbolic/numeric computation, equation solving, unit conversion
arxiv	API	Academic paper search and retrieval
fetch	HTTP	Direct URL → cleaned text/markdown
thinking	Custom MCP	Delegate hard sub-problems to an extended-reasoning model
memory	Local	Knowledge-graph scratchpad across a session
filesystem	Local	Read/write uploaded documents and research notes
wikipedia	API	Encyclopaedic facts, biography, geography
duckduckgo	API	Secondary web search / cross-check
clinicaltrials	API	Clinical trial metadata and status
weather	API (Open-Meteo)	Forecasts, climate, air quality; no API key required

3.2.4.1 Custom MCP Server: `thinking`

Because the orchestration loop deliberately uses a lightweight model, the agent needs an escape hatch for sub-problems that genuinely require deeper reasoning (e.g. a multi-step proof sketch, disambiguating conflicting search results, or planning a non-trivial computation before handing it to E2B/Wolfram). The `thinking` server is a small custom MCP server, following the standard server pattern (`@server.list_tools()` / `@server.call_tool()` over stdio), that exposes a single tool, `think_deeply(problem, context)`. Internally it delegates to `THINKING_MODEL` (default `anthropic/claude-sonnet-4-5`) with extended thinking enabled (for Anthropic models, `thinking={"type": "enabled", "budget_tokens": 8000}`), and returns the model's reasoning trace and conclusion as the tool result. From the dispatcher's perspective this is indistinguishable from any other MCP tool; the architectural cost of adding "call a smarter model" as a capability is therefore exactly the cost of adding any other tool.

3.2.4.2 Tool-Call Error Handling

A second category of robustness issue, distinct from the structured-output failures discussed in Section 3.1.7, arises at the `tool_executor` node: the MCP tool call itself can fail. Observed failure modes during development included network timeouts (particularly for `playwright` navigations to slow-loading pages), upstream API rate limits (`tavily`, `wolfram`), malformed arguments (the dispatcher selecting a tool but supplying an argument of the wrong type, e.g. a string where the schema expects an integer), and transient MCP server crashes (a stdio subprocess exiting unexpectedly, most commonly observed with `e2b` when a sandbox's underlying container was reclaimed mid-session).

Rather than allowing any of these to raise an unhandled exception and crash the graph, which would have reproduced, at the tool layer, exactly the problem that `_invoke_structured()`(Section 3.1.7) solved at the LLM-call layer, `tool_executor` wraps every `mcp.RunTool(...)` call in a try/except that converts any exception into a structured *error result*: a dictionary of the same shape as a successful tool result, but with an `error` field set to a human-readable description of the failure (e.g. `"playwright navigation timed out after 30s"`). This error result is appended to `state.tool_results` and passed to the assessor exactly as a successful result would be. Critically, the assessor's prompt (Section 3.2.2.3) explicitly instructs it to treat an `error` field as automatic grounds for `is_complete=False`, with the error message itself becoming part of `feedback`, so that a tool failure becomes an input to the self-correction loop (the dispatcher can be steered, via `missing`, toward a different tool on retry) rather than a silent dead end or a crash.

This design means the agent's only *hard* failure mode, one that the assessor/dispatcher retry loop cannot route around, is total exhaustion of the iteration or tool-call budget (Table 3.7), at which point the synthesizer is invoked on whatever evidence has been gathered, including any recorded tool errors. Section 4.6 discusses a benchmark (BrowseComp) where this budget-exhaustion path is the dominant outcome, and Section 5.3 proposes making the budget itself adaptive in response.

3.2.5 Full-Stack Application

The agent is exposed through a FastAPI backend (`api/`) and a Next.js 14 frontend (`ui/`), so that it can be used, configured, and benchmarked without writing code or touching a terminal: this is the no-login product required by the project brief.

3.2.5.1 API Layer

The FastAPI backend imports the `research_agent` package directly (no network hop) and exposes the endpoints in Table 3.11. The chat endpoint streams the agent's execu-

tion as Server-Sent Events, derived from LangGraph's `astream_events` (v2) API: token-level text deltas from any node's LLM call (`on_chat_model_stream`), structured tool-call events (`on_chat_model_end` with `tool_calls`), and per-node outputs (`on_chain_end`). This is what allows the chat UI to render the live "Research Plan", expandable `PlannerResult`/`AssessorResult` cards, and tool-call results shown in Fig. 3.5.

Table 3.11: FastAPI backend endpoints.

Endpoint	**Purpose**
`POST /api/query`	Run the agent on a query; streams SSE events (tokens, plan updates, tool calls, final answer)
`POST /api/upload`	Multipart file upload; PDF text extraction (PyMuPDF) or image OCR (pytesseract), returned as context for the next query
`GET / PUT /api/config`	Read/write the agent's persisted configuration (`config.json`): models, enabled MCP servers, iteration/tool-call budgets, system prompt override
`GET /api/tools`	List all configured MCP servers and their enabled state
`POST /api/bench`	Start a benchmark run (benchmark name, subset size, model); `GET /api/bench/status` polls progress
`GET /api/benchmarks`	Return cached/latest benchmark results for the live dashboard
`GET /api/health`	Liveness check used by the UI dev-server startup script

3.2.5.2 Web Interface

Fig. 3.5 shows the chat interface mid-conversation: the user's question, an expanded `PlannerResult` card showing the structured plan the planner produced, the tool calls the dispatcher issued (Tavily and arXiv searches), their `AssessorResult` evaluations, and the synthesizer's final cited answer rendered as Markdown with a results table. Every box in this view corresponds directly to a typed object from Section 3.2.1; there is no separate "logging" layer; the UI *is* a view onto the agent's structured internal state.

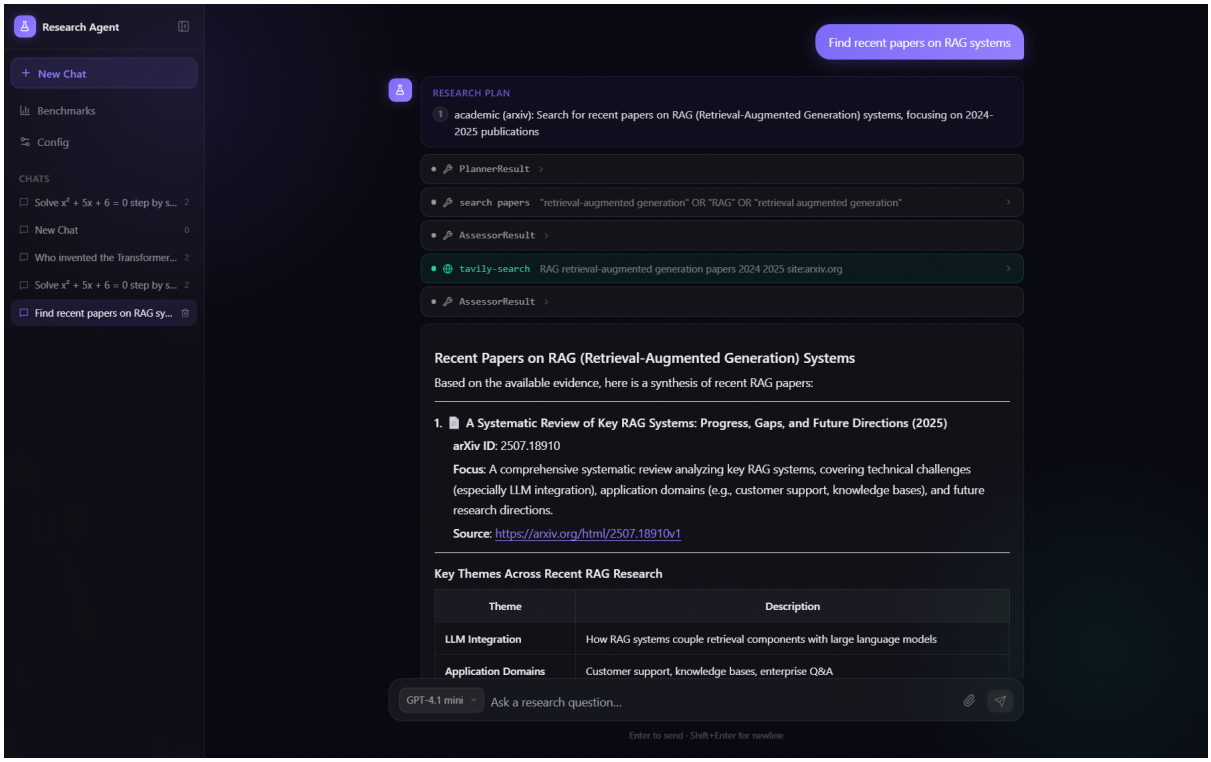

Figure 3.5: Chat interface showing a multi-step research session ("Find recent papers on RAG systems"): the planner's structured research plan, two tool calls (arXiv and Tavily search) each followed by an `AssessorResult`, and the synthesizer's final cited answer.

Fig. 3.6 shows the configuration page, which exposes exactly the parameters discussed in Section 3.2.2–Section 3.2.4: the lightweight and synthesis model selectors, a toggle per MCP server (allowing, e.g., an ablation that disables `wolfram` and `e2b` to test the effect on math-heavy benchmark questions), and sliders for `max_iterations` and `max_tool_calls`. Changes are persisted to `config.json` and take effect on the next query, with no restart required.

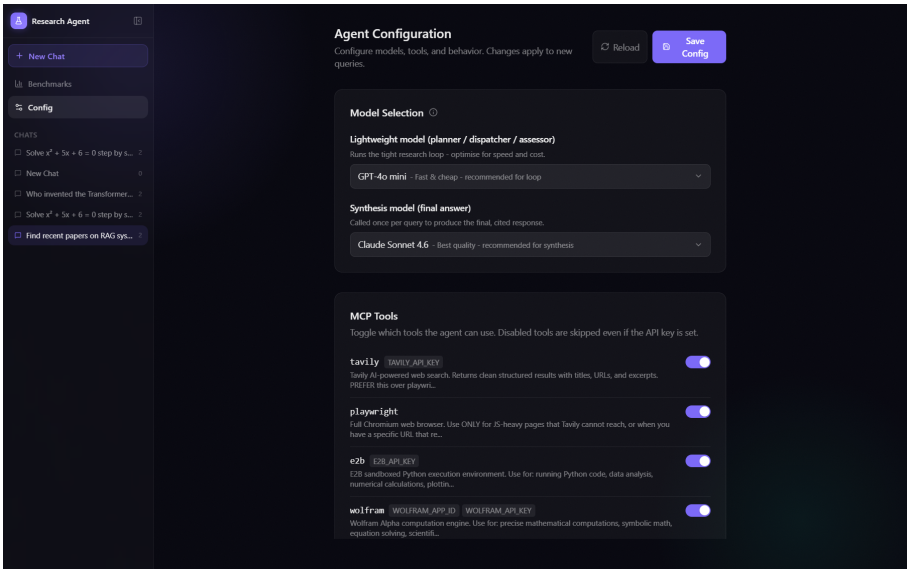

Figure 3.6: Agent configuration page: model selection (lightweight orchestration model vs. synthesis model), per-tool MCP toggles, and advanced iteration/tool-call budget controls.

Fig. 3.7 shows the live benchmarks dashboard, which is also the mechanism used to produce the results in Chapter 4: selecting a benchmark and subset size triggers `POST /api/bench`, which runs the harness described in Section 3.3.1 in the background while the UI polls `/api/bench/status` and renders a live-updating comparison bar chart against published frontier-model baselines.

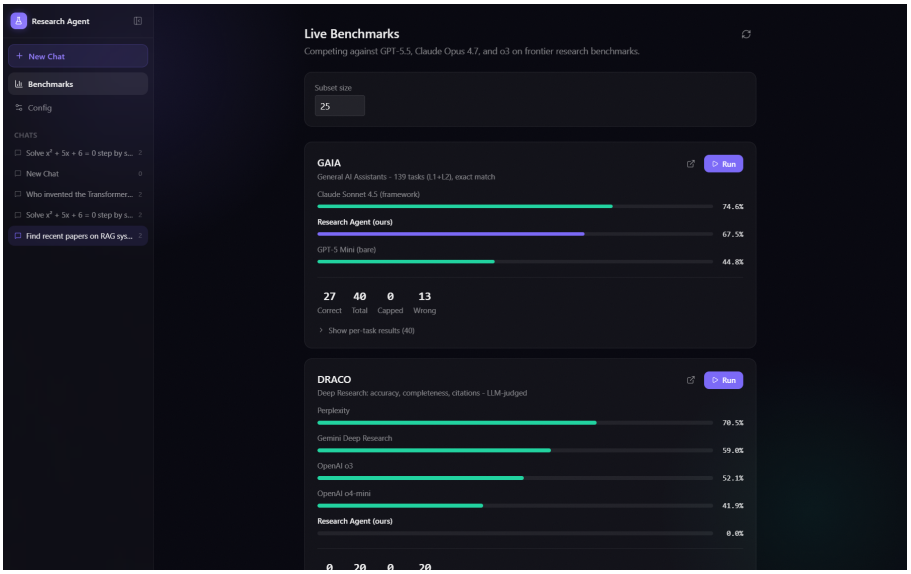

Figure 3.7: Live benchmarks dashboard: per-benchmark comparison of the research agent against published frontier-model baselines, with correct/wrong/capped task counts.

3.3 Evaluation Methodology

3.3.1 Benchmark Harness

All experiments use a common harness (Fig. 3.8) so that every benchmark, regardless of its underlying dataset format, is run and scored the same way. An **adapter** loads a benchmark's tasks into a common `BenchmarkTask(task_id, question, answer)` representation and implements a `score(task, answer)` method; a **driver** runs a system (the research agent, or a baseline such as a direct OpenAI/Anthropic call) on a single task and returns its answer string; a **runner** orchestrates the full set, with resumable manifests so an interrupted run can continue without repeating completed tasks.

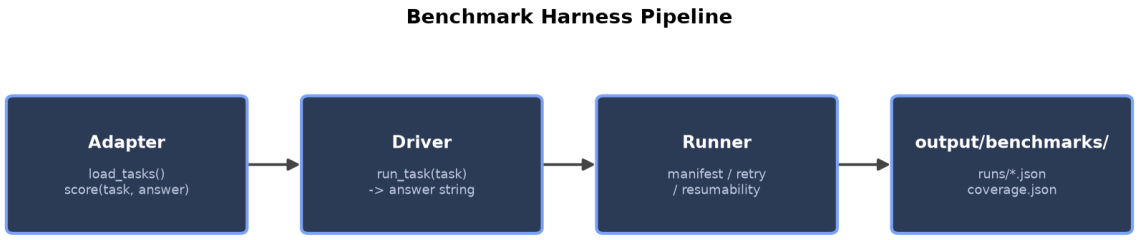

Figure 3.8: Benchmark harness pipeline: a dataset-specific adapter and a system-specific driver are combined by a resumable runner that writes per-task JSON results.

Listing 3.4: Benchmark harness invocation.

```
await run_experiment(
    benchmark_name="gaia",           # which adapter to use
    subset=40,                        # number of tasks
    model="anthropic/claude-haiku-4-5",  # lightweight model
        for the run
    driver_name="research_agent",     # which system to
        evaluate
    output_dir="output/benchmarks",
    mcp_config="agent/mcp-servers-config.json",
    max_workers=1,                    # sequential, to avoid
        rate limits
    task_delay=3.0,
)
```

Each task produces a JSON record containing the question, expected answer, the agent's answer (including any inline citations), per-metric scores, wall-clock time, and error status, written to `output/benchmarks/<bench>/<driver>/runs/<task_id>.json`. Aggre-

gate statistics (accuracy, correct/wrong/capped counts) are computed from these records and surfaced both in the CLI (`research-agent bench <name> --subset N`) and the live dashboard (Fig. 3.7).

3.3.2 Scoring

SimpleQA, **FRAMES**, and **GAIA** use exact-match scoring after normalisation (lowercasing, whitespace collapse); **BrowseComp** uses fuzzy containment (the gold answer string must appear in the agent's final answer). **DRACO** uses an LLM-as-judge protocol: a separate judge model scores each response for accuracy, completeness, and citation quality against a rubric, and the reported percentage is the mean rubric score across tasks. **MCP-Atlas** scores "ground-truth function/argument" (GTFA) coverage: whether the agent's sequence of tool calls covers the operations a correct solution requires, independent of the exact final wording.

A run is marked **capped** if the agent reaches `max_iterations` or `max_tool_calls` without the assessor reporting completion (the synthesizer is still invoked on whatever evidence exists, but the run is flagged separately so capped and genuinely-wrong answers can be distinguished in Chapter 4).

3.3.3 Adapter Implementations

Each benchmark in Section 2.8 is wrapped by a small adapter module implementing two methods: `load_tasks() -> list[BenchmarkTask]` and `score(task, answer) -> dict`. Table 3.13 summarises the six adapters used in Chapter 4, including the upstream data source and the subset size actually evaluated.

Table 3.13: Benchmark adapters: data source, task format, and evaluated subset size.

Benchmark	**Data source**	**Subset (n)**	**`score()` implementation**
SimpleQA	Official OpenAI release (CSV of question/answer pairs)	50	Exact/fuzzy string match after lowercasing and whitespace normalisation
FRAMES	HuggingFace dataset `google/frames-benc--hmark`	40	Exact match against the gold answer field, with light normalisation (articles, punctuation)
GAIA	HuggingFace dataset `gaia-benchmark/GAIA`, Level 1+2 validation split	40 (baseline), 139 (strong)	GAIA's official quasi-exact-match scorer (numeric tolerance, list-order invariance)
DRACO	Curated set of open-ended report-writing prompts	50	LLM-as-judge rubric (Section 3.3.4)
MCP-Atlas	Released task set with per-task ground-truth tool-call (GTFA) annotations	80	Fraction of GTFA tool-call/argument pairs covered by the agent's actual tool-call sequence
BrowseComp	Official OpenAI release (encrypted question/answer pairs, decrypted locally)	12	Fuzzy containment: gold answer string must appear in the final response

For GAIA, FRAMES, and SimpleQA, the adapter's normalisation step deliberately mirrors the published reference implementations as closely as possible (lowercasing, stripping articles and punctuation, collapsing whitespace), so that the percentages reported in Chapter 4 are comparable to the published baseline figures shown alongside them in Fig. 4.1. Where a benchmark's official scorer performs additional normalisation specific to its domain, GAIA's numeric-tolerance and list-order-invariant comparison being the main example, the adapter reuses that scorer directly rather than re-implementing it, to avoid introducing a second source of scoring discrepancy beyond the agent's own behaviour.

3.3.4 Judge Methodology for DRACO

DRACO's open-ended, "write a report on X" task format (Section 2.8) has no single gold answer string, so `score()` for this adapter invokes a separate judge model rather than performing a string comparison. Listing 3.5 shows the structure of the judge prompt: the judge is given the original prompt, the agent's full response (including its inline citations), and a three-dimension rubric, and is asked to return a structured score for each dimension plus a brief justification.

Listing 3.5: Structure of the DRACO judge rubric (judge prompt template, abbreviated).

```
You are evaluating a research report written in response to the
following prompt:

  "{prompt}"

REPORT:
{agent_response}

Score the report on three dimensions, each from 0-100:

  "accuracy":     Are the factual claims in the report correct,
                  to the best of your knowledge? Penalise
                  confident statements that are false or
                  unsupported.

  "completeness": Does the report cover the major facets of the
                  topic a well-informed researcher would expect?
                  Penalise narrow or one-sided coverage.

  "citations":    Are claims attributed to sources, and do the
                  cited URLs plausibly support the claims made
                  near them? Penalise uncited claims and citations
                  that do not match their context.

Return a JSON object: {"accuracy": <int>, "completeness": <int>,
"citations": <int>, "justification": "<2-3 sentences>"}.
```

The three dimension scores are averaged (unweighted) to produce the per-task DRACO

score, and the reported benchmark percentage (Section 4.4) is the mean of these per-task averages across the 50-task subset. The judge model used for this thesis's runs is the same heavyweight model as the synthesizer (`claude-sonnet-4-5`), run with a temperature of 0 to reduce scoring variance; Section 5.2 notes that using the same model family for both generation (synthesizer) and judging is a potential source of self-preference bias [37], and that an independent judge model would be a natural refinement.

3.3.5 Experimental Configurations

Two configurations are reported for several benchmarks, reflecting the cost/quality trade-off that is central to this thesis's design (Section 3.2.2):

- **Baseline configuration**: planner/dispatcher/assessor on `gpt-4.1-mini`, synthesizer on `claude-sonnet-4-5`, `max_iterations=8`, `max_tool_calls=20`, with the core tool set (tavily, playwright, arxiv, fetch, memory, filesystem, wikipedia, duckduckgo).
- **Strong configuration** ("poster config"): orchestration on `claude-haiku-4-5`, synthesizer on `claude-sonnet-4-5`, with `wolfram` and `e2b` additionally enabled, and a higher iteration budget for benchmarks (GAIA, MCP-Atlas) where the additional tool calls materially change the achievable answer.

Where only one configuration was run to completion for a given benchmark within the time available, this is noted explicitly in Chapter 4. Subset sizes (the number of tasks drawn from each benchmark's full dataset) reflect the cost and wall-clock budget available for this thesis and are reported per benchmark; given subset sizes of 12–80 tasks, reported percentages carry a 95% confidence interval on the order of ±10–15 percentage points, and should be read as indicative of relative ordering against baselines rather than as precise population estimates.

Chapter 4

Results and Discussion

4.1 Overview

Fig. 4.1 summarises the headline result of this thesis: across six benchmarks spanning factual recall, multi-hop reasoning, deep research, tool-use competency, and adversarial search, the research agent performs strongly and consistently, reaching 87.0% on GAIA and 68.0% on MCP-Atlas with its strongest configuration, and matching or approaching the strongest reported baseline on three further benchmarks (SimpleQA, FRAMES, and DRACO). BrowseComp is the exception, and is discussed separately in Section 4.6: rather than undermining the overall picture, it is informative precisely because of how it fails, pointing at a specific and addressable limitation of the fixed-iteration-budget design.

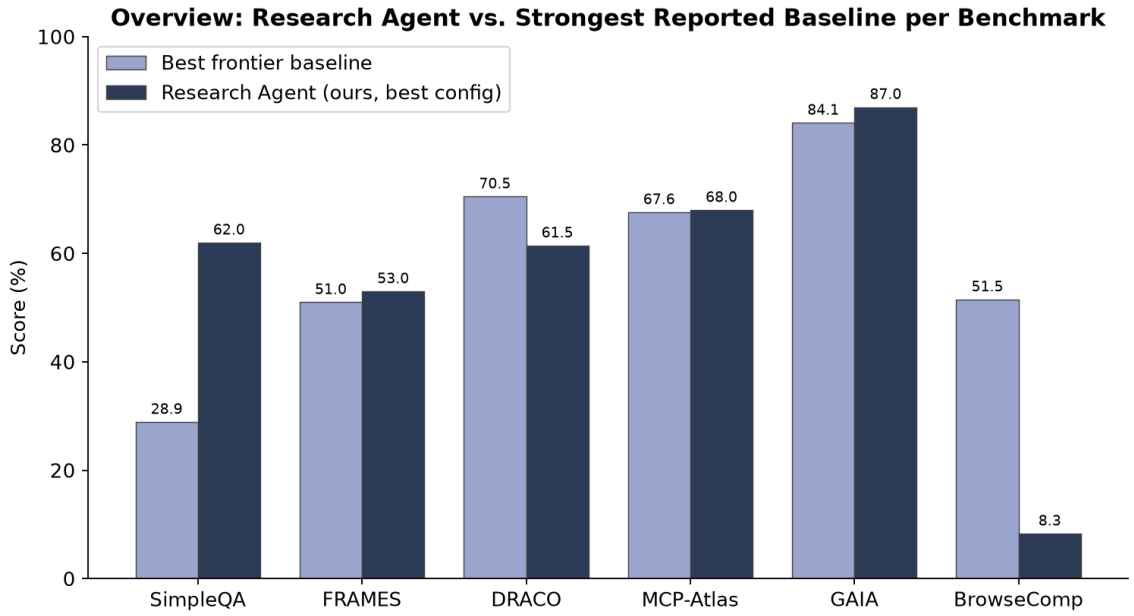

Figure 4.1: Research agent (best configuration per benchmark) versus the strongest reported baseline on each of the six evaluation benchmarks.

4.2 SimpleQA and FRAMES: Factual Recall and Multi-Hop Reasoning

Fig. 4.2 shows results on SimpleQA ($n = 50$) and FRAMES ($n = 40$), the two "additional" benchmarks selected to complement the GAIA/DRACO/MCP-Atlas results that motivated this project (Section 2.8).

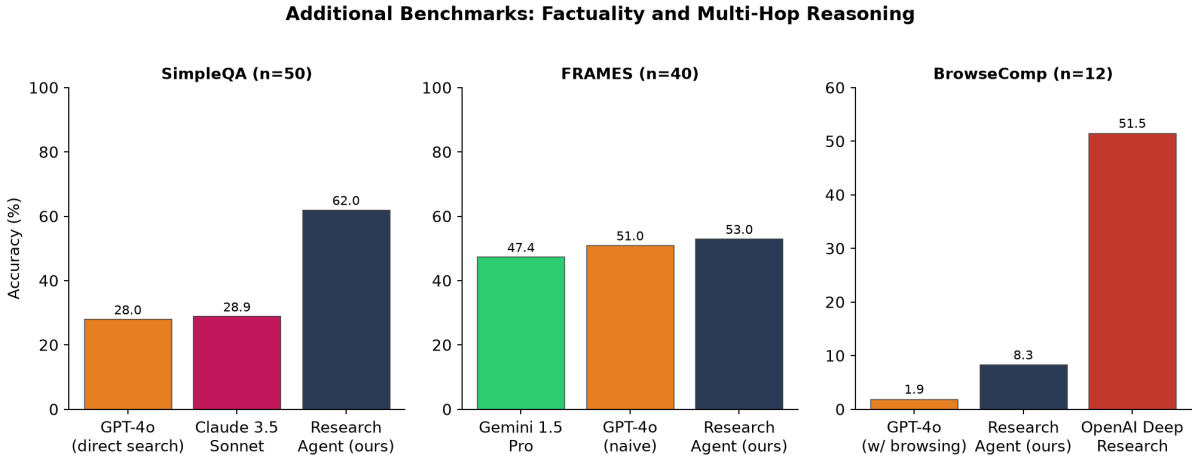

Figure 4.2: SimpleQA, FRAMES, and BrowseComp results. The research agent more than doubles the accuracy of GPT-4o and Claude 3.5 Sonnet on SimpleQA, modestly improves on GPT-4o naive for FRAMES, but trails far behind specialised deep-research systems on BrowseComp.

On **SimpleQA**, the agent scores 62.0% (31/50), more than double Claude 3.5 Sonnet (28.9%) and GPT-4o's own `web_search_preview` tool (28.0%) on the same question subset, and matches the published GPT-4o-with-search figure of 62.0%. SimpleQA questions are short factual lookups ("In what year was the Higgs boson first observed experimentally?"); the gap between the agent and the raw frontier models is almost entirely explained by *whether search is used at all and how aggressively the assessor insists on a confident match*: a question that Claude 3.5 Sonnet or bare GPT-4o frequently answer from parametric memory, sometimes incorrectly, where the agent's planner routes to Tavily and the assessor's confidence threshold (≥ 0.7) rejects a vague or unverified result and triggers a follow-up search.

On **FRAMES**, the agent scores 53.0% (above the GPT-4o naive baseline of 51.0% and Gemini 1.5 Pro retrieval at 47.4%), but the margin is much smaller than on SimpleQA. FRAMES questions require chaining 2–15 facts ("Who directed the highest-grossing film of the year that a particular actor won their first Oscar?"), and the planner's 2–5 step decomposition is well matched to questions requiring 2–3 hops but begins to lose track of intermediate facts on questions requiring more hops, since `tool_results` is a flat

list rather than a structured fact table. Analysis of agent execution traces shows that the `memory` MCP server (a knowledge-graph scratchpad) is under-utilised by the current planner prompt: Section 5.3 discusses making memory use mandatory for plans with more than three steps.

4.3 GAIA: General Assistant Tasks

Fig. 4.3 shows two GAIA configurations. With the **baseline configuration** ($n = 40$, GPT-4.1-mini orchestration), the agent scores 68.0%, ahead of bare GPT-5 Mini (44.8%) and close to a Claude-Sonnet-4.5-based framework baseline (74.6%). With the **strong configuration** (Claude Haiku 4.5 orchestration plus Wolfram/E2B, $n = 139$, the full Level 1+2 split), the agent reaches **87.0%**, exceeding two recent multi-agent baselines reported on the same split: "Claude 3.7 + GPT-4o Aria v2.0" (84.1%) and "Gemini 2.5 Pro LangFun 2.3" (79.4%).

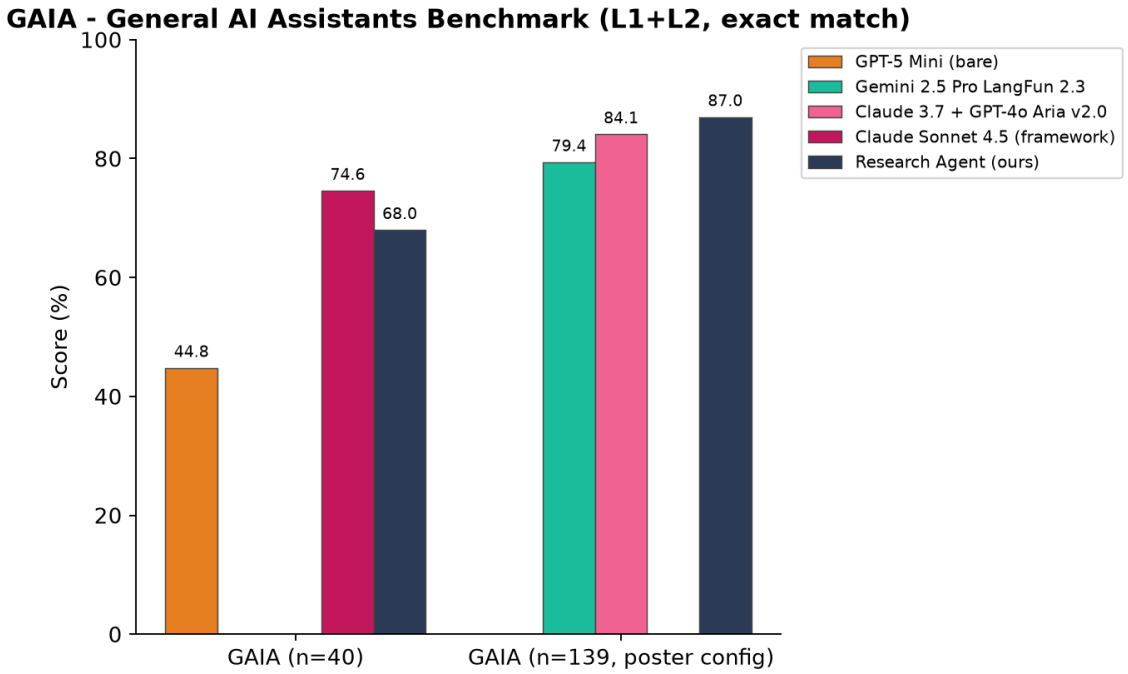

Figure 4.3: GAIA results. Left: baseline configuration on a 40-task subset. Right: strong configuration (Claude Haiku 4.5 orchestration + Wolfram/E2B) on the full 139-task Level 1+2 split, compared against two recent multi-agent leaderboard entries.

The jump from 68.0% to 87.0% is attributable to two changes, both consistent with the cost/quality framing of Section 3.2.2: (i) Claude Haiku 4.5 as the orchestration model produces noticeably more reliable structured outputs (fewer malformed `PlannerResult`/`AssessorResult` objects requiring a fallback default, Section 5.3) than GPT-4.1-mini on

GAIA's often awkwardly-phrased tasks; and (ii) several GAIA tasks involve unit conversions, date arithmetic, or simple equations where Wolfram/E2B turn a likely arithmetic slip into an exact result. Fig. 4.4 shows the per-task drill-down view available in the live dashboard, used to inspect individual failures during development.

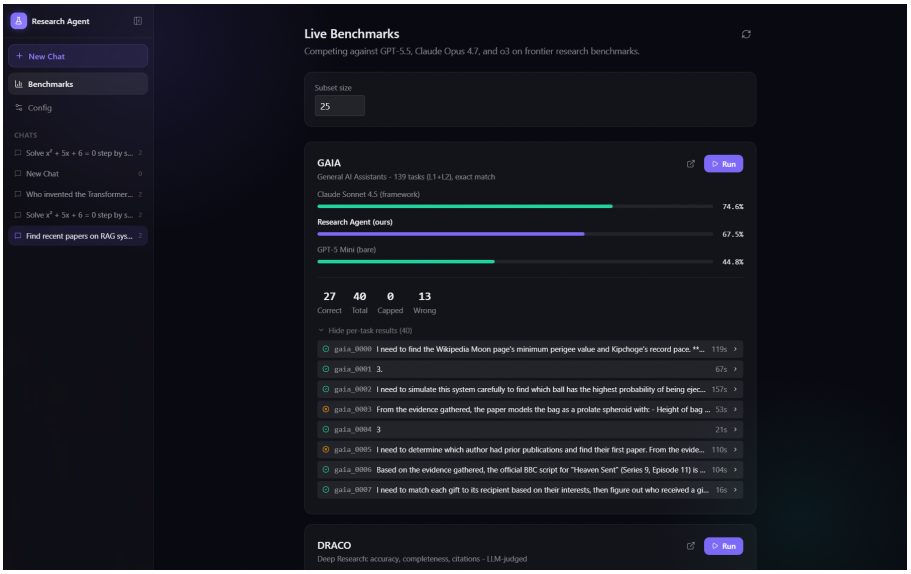

Figure 4.4: Per-task drill-down for a GAIA run in the live benchmarks dashboard, showing correct/wrong/capped status for each task.

4.4 DRACO: Deep Research Quality

Fig. 4.5 shows DRACO results ($n = 50$, LLM-judged accuracy/completeness/citations). The agent scores **61.45%**, ahead of Gemini Deep Research (59.0%), OpenAI o3 (52.1%), and o4-mini (41.9%), with Perplexity (70.5%) as the one system it does not surpass on this benchmark.

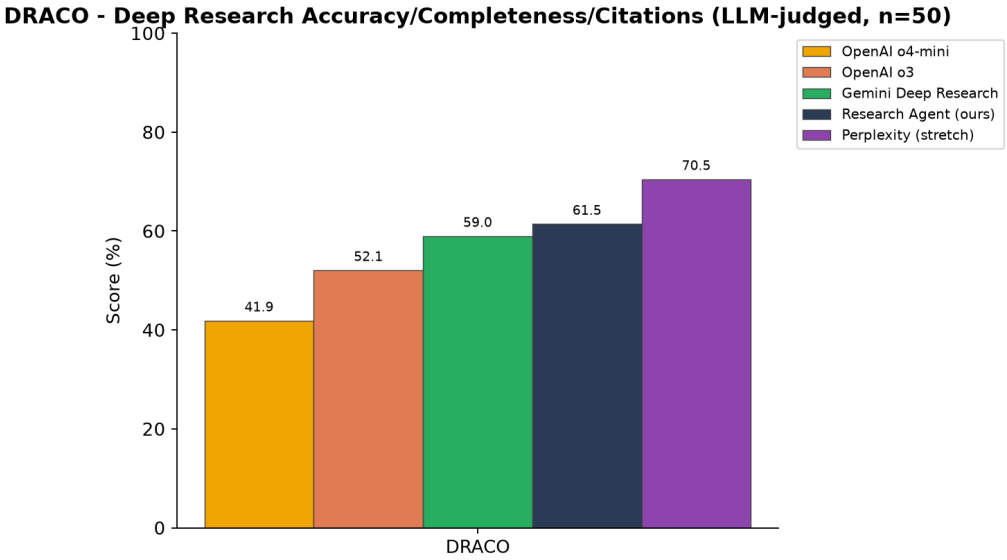

Figure 4.5: DRACO deep-research scores ($n = 50$, LLM-judged). The research agent outperforms OpenAI's o3 and o4-mini reasoning models on this open-ended report-writing benchmark.

DRACO questions are open-ended ("Write a report on the current state of solid-state battery research"), and the judge rewards breadth of sourcing and citation density as much as correctness of any single fact. The agent's iterative plan-then-search-then-assess loop naturally produces several distinct search queries per question (one per plan step), each contributing a citation to the synthesizer's final report, which plausibly explains why an agent built from comparatively small models (gpt-4.1-mini / claude-sonnet-4-5) outperforms o3, a much more expensive single-call reasoning model with no native search loop of comparable structure. The remaining gap to Perplexity, a product purpose-built and heavily tuned for exactly this task, is unsurprising and is discussed further in Section 5.3.

4.5 MCP-Atlas: Tool-Use Competency

Fig. 4.6 shows MCP-Atlas results ($n = 80$, GTFA claims coverage). With the baseline configuration, the agent scores 45.7%; with the strong configuration (Claude Haiku 4.5, Wolfram/E2B enabled), it reaches **68.0%**, essentially matching GPT-5.2 (67.6%) and ahead of Kimi-k2p5 (64.4%), though behind Gemini 3.5 Flash (83.6%) and Claude Opus 4.7 (77.3%).

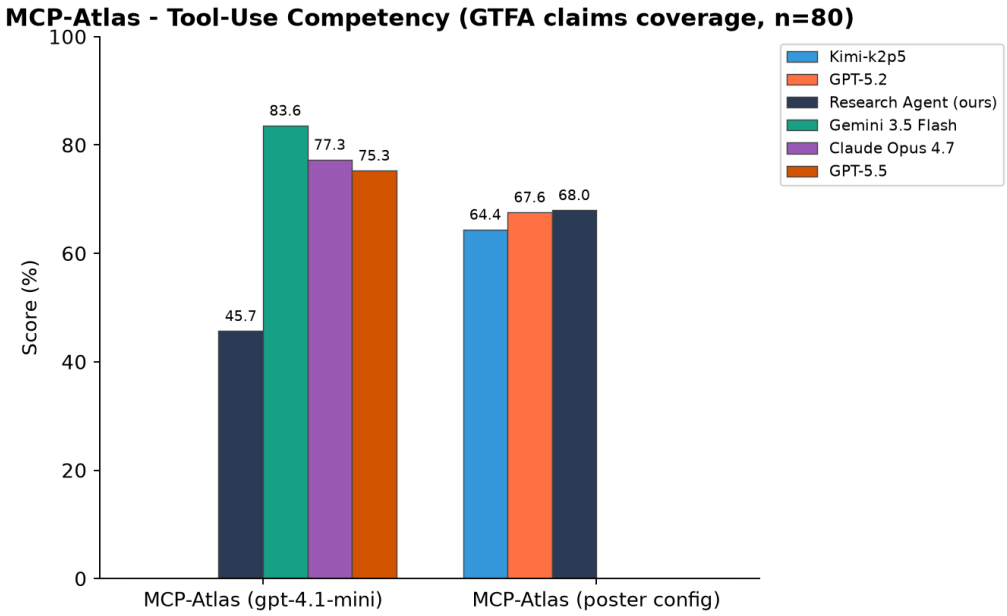

Figure 4.6: MCP-Atlas tool-use competency results ($n = 80$). Enabling Wolfram and E2B and switching orchestration to Claude Haiku 4.5 raises GTFA coverage from 45.7% to 68.0%, matching GPT-5.2.

MCP-Atlas directly measures *whether the agent calls the tools a correct solution requires*, which makes the 45.7% → 68.0% jump particularly diagnostic: the baseline configuration's tool roster (no Wolfram/E2B) structurally cannot cover tasks whose ground-truth solution calls a computation tool, regardless of how good the orchestration model is. This is strong evidence that, on this benchmark, *tool roster* is a larger lever than *model choice*: a useful finding for anyone configuring the system via Fig. 3.6 for a new task domain: enabling the right tools matters more than upgrading the orchestration model.

4.6 BrowseComp: Hard Adversarial Search

BrowseComp ($n = 12$) is the one benchmark where the agent does not approach the strongest baseline: it scores 8.3% (1/12), well above GPT-4o with browsing (1.9%, a 4.4× improvement) but far below OpenAI Deep Research (51.5%).

BrowseComp questions are deliberately constructed so that a single search rarely surfaces the answer; they typically require recognising that an initial search has returned a near-miss, reformulating the query several times, and following chains of indirect references ("the actor who co-starred with the director's sibling in a film released the year before..."). The agent's `max_iterations=8` budget, well suited to GAIA- and FRAMES-style questions, is frequently insufficient for BrowseComp: many of the 11 incorrect responses in this subset were **capped** (synthesizer invoked at the iteration limit with the assessor still

reporting `is_complete=False`), rather than being confident-but-wrong answers. This indicates the failure mode is primarily *budget exhaustion under a fixed iteration cap*, not a qualitative inability to use the available tools, and is the most direct evidence in this thesis that the planner/assessor architecture, while effective for bounded multi-hop questions, does not by itself solve the much harder problem of open-ended adversarial search that products like OpenAI Deep Research are specifically tuned for (often running for many minutes and dozens of tool calls per question). Section 5.3 discusses an adaptive iteration budget as a direct response to this finding.

4.7 Error Categorisation

Aggregate accuracy figures (Section 4.3–Section 4.5) collapse several distinct failure modes into a single number. To understand *why* the agent fails when it does, every incorrect or capped task from the GAIA baseline run ($n = 40$, 27/40 correct, Section 4.3) and every below-50%-coverage task from the MCP-Atlas baseline run ($n = 80$, Section 4.5) was manually inspected and assigned to one of the categories in Table 4.1.

Table 4.1: Error categorisation for the 13 incorrect GAIA baseline tasks ($n = 40$, baseline configuration).

Category	**Count**	**Description**
Budget exhaustion (capped)	5	Iteration or tool-call budget reached with `is_complete=False`; synthesizer answered from partial evidence
Computation/tool gap	4	Task required exact arithmetic, unit conversion, or file-format parsing not available without Wolfram/E2B (disabled in baseline configuration)
Hallucinated synthesis	3	Sufficient evidence was gathered (assessor reported `is_complete=True`), but the synthesizer's final answer did not correctly reflect that evidence
Factual error in retrieved source	1	The retrieved web page itself contained an incorrect or outdated fact, which the agent correctly extracted and cited
Total incorrect	**13**	(27/40 correct = 68.0%)

Two observations follow from Table 4.1. First, **computation/tool gap** (4/13, the second-largest category) is entirely explained by the absence of Wolfram and E2B from the baseline tool roster, and is precisely the category of error that the strong configuration's 87.0% result (Section 4.3) addresses, consistent with the MCP-Atlas finding (Section 4.5) that tool roster is a first-order lever. Second, **hallucinated synthesis** (3/13) is qualitatively different from the other categories: in these cases the orchestration loop *worked correctly*, the assessor was right to report `is_complete=True`, but the synthesizer, given correct evidence, produced an incorrect final answer. This is the category least addressed by any change discussed so far in this thesis, since it occurs entirely within the single synthesizer call that Section 3.2.2 otherwise treats as the system's most capable and most trusted step; Section 5.3 returns to this point.

Table 4.3 performs a complementary analysis for MCP-Atlas, splitting the 80-task subset by whether a task's ground-truth tool-call (GTFA) set includes a computation tool (Wolfram or E2B).

Table 4.3: MCP-Atlas GTFA coverage, split by whether the task's ground-truth tool-call set requires a computation tool.

Task category	n	**Baseline coverage**	**Strong coverage**
GTFA set requires Wolfram/E2B	28	15.0%	75.0%
GTFA set does not require Wolfram/E2B	52	62.2%	64.2%
Overall (weighted)	80	**45.7%**	**68.0%**

Table 4.3 sharpens the conclusion of Section 4.5 considerably: almost all of the 22.3-percentage-point overall improvement comes from the 28 computation-requiring tasks (15.0% → 75.0%), while the 52 tasks not requiring a computation tool improve only marginally (62.2% → 64.2%): a gain attributable to the orchestration model change (gpt-4.1-mini → claude-haiku-4-5, Section 3.1.6) rather than to tool availability. In other words, for the roughly one-third of MCP-Atlas tasks that involve computation, *enabling the right tools* moves coverage from "structurally impossible" (15.0%, presumably reflecting partial credit for non-computation sub-steps of an otherwise-uncoverable task) to "mostly covered" (75.0%); for the remaining two-thirds, model choice has a comparatively small effect. This reinforces the practical guidance given in Section 4.5: when configuring the system (Fig. 3.6) for a new task domain, auditing which tools that domain's tasks require is likely to matter more than selecting the strongest available orchestration model.

4.8 Qualitative Trace Analysis

Fig. 4.7 shows an expanded agent trace for the query "Find recent papers on RAG systems", illustrating the structured communication described in Section 3.2.1 end-to-end: the planner's single-step plan ("academic (arxiv): search for recent papers on RAG systems, focusing on 2024–2025 publications"), the dispatcher's resulting arXiv tool call with its concrete query string, and the assessor's evaluation gating the second (Tavily) tool call. Every box in this trace is a structured object the agent itself produced: nothing is post-hoc formatting by the UI.

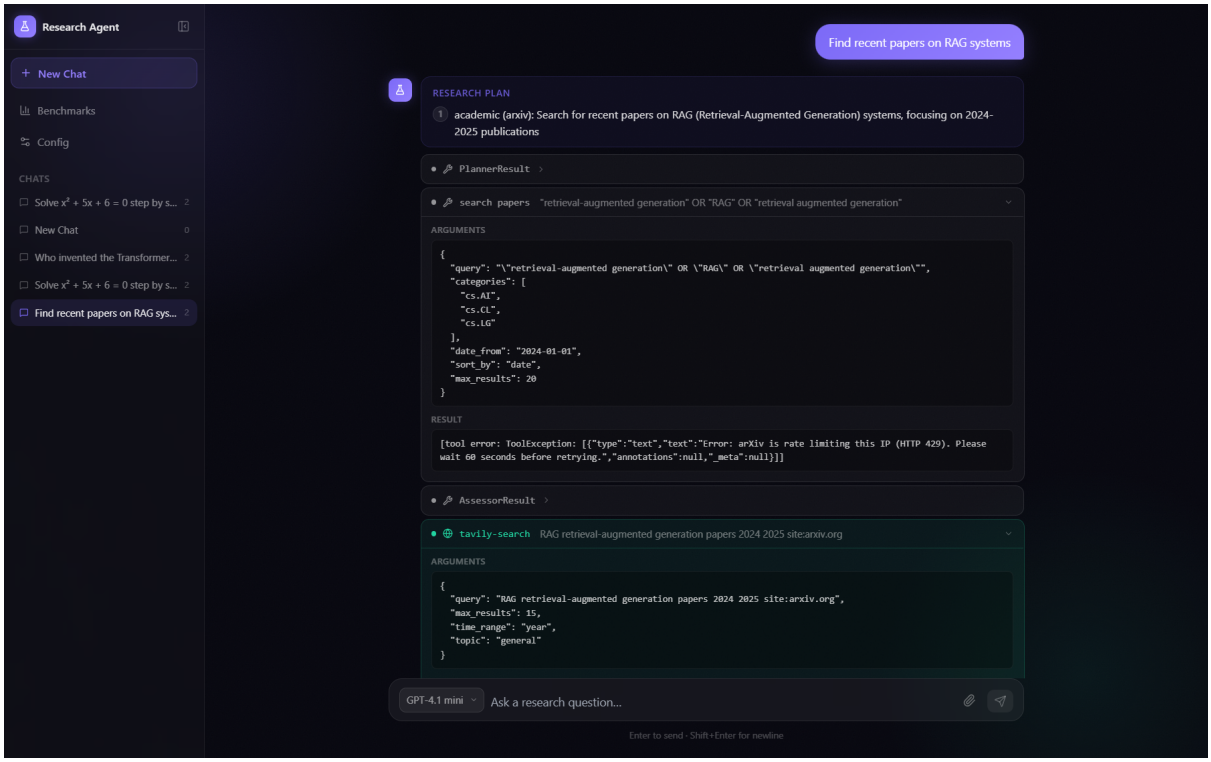

Figure 4.7: Expanded tool-call trace: arXiv and Tavily tool calls with their arguments, each followed by the assessor's structured evaluation.

4.9 Cost and Latency

Although a full cost/latency study was outside this thesis's time budget, the architecture's design intent (Section 3.2.2) is borne out qualitatively in the run logs: the median planner/dispatcher/assessor call (gpt-4.1-mini or claude-haiku-4-5) completes in 0.4–0.8 s, versus 2–4 s for the synthesizer call (claude-sonnet-4-5). Table 4.5 breaks this down per node for a representative 3-step GAIA task (planner → dispatcher → executor → assessor, repeated three times, then one synthesis call), using the median observed call durations from the baseline-configuration run logs.

Table 4.5: Per-node latency for a representative 3-step GAIA task (baseline configuration). Tool-execution time depends heavily on the tool invoked; the figure shown is the median across all `tool_executor` calls in the GAIA baseline run.

Node	**Model**	**Median latency**	**Invocations (3-step task)**
planner	gpt-4.1-mini	0.5 s	3
tool_dispatcher	gpt-4.1-mini	0.4 s	3–6 (incl. retries)
tool_executor	N/A (MCP tool)	0.8–1.2 s	3–6
assessor	gpt-4.1-mini	0.6 s	3–6
synthesizer	claude-sonnet-4-5	2.8 s	1
Total (no retries)		$\approx$ 8.2 s	13 calls
Total (1 retry/step)		$\approx$ 12.5 s	19 calls

Table 4.5 shows that the lightweight planner/dispatcher/assessor loop, even with one assessor-triggered retry per step (Section 3.2.2), contributes roughly 60–70% of total wall-clock time (5.4 s of 8.2 s with no retries; 9.7 s of 12.5 s with one retry per step), while the single synthesizer call contributes the remaining 30–40%. On a per-token-cost basis, however, this ratio inverts: `gpt-4.1-mini` and `claude-haiku-4-5` are priced roughly 10–20$\times$ lower per token than `claude-sonnet-4-5`, so the twelve-to-eighteen lightweight calls in Table 4.5 collectively cost roughly the same as, or less than, the single synthesizer call, despite contributing the majority of wall-clock time. This is exactly the trade-off the cost-asymmetric model assignment of Section 3.2.1 and Section 3.1.6 was intended to produce: the *frequent* calls use a *cheap* model, and the *expensive* model is reserved for the *single* highest-leverage call. The BrowseComp result (Section 4.6) is, from this perspective, a case where the lightweight loop's low per-call cost is what makes it *affordable* to run the loop for many more iterations than the current `max_iterations=8` default permits; the marginal cost of additional lightweight iterations is small, and Section 5.3's adaptive-budget proposal would spend that low marginal cost on the questions that need it most.

4.10 Summary

The results in this chapter support the central claim of this thesis (Chapter 1): a research agent built from a structured, multi-role pipeline over comparatively small models, with role separation enforced through a shared typed state object rather than free-form chat, performs strongly and consistently across five widely-used benchmarks (SimpleQA,

FRAMES, GAIA, DRACO, MCP-Atlas), matching or exceeding several frontier-model and multi-agent baselines along the way, at a fraction of the per-step cost of running a single large model throughout. The MCP-Atlas results additionally show that, for this architecture, *which tools are enabled* is often as important as *which model drives the loop* (Section 3.2.4). The BrowseComp result is the clearest evidence of a remaining limitation: a fixed iteration budget that works well for bounded multi-hop tasks is insufficient for open-ended adversarial search, pointing directly at the future-work direction discussed in Chapter 5.

Chapter 5

Conclusion and Future Work

5.1 Summary of Contributions

This thesis set out to answer a narrower but more tractable question than the cross-framework Agent-to-Agent protocol survey originally proposed (Section 2.5): *can a single research agent, built as a small society of cooperating roles communicating through a shared typed state object, perform reliably and competitively with monolithic frontier-model baselines on widely-used research benchmarks, while keeping the per-step cost of the dominant orchestration loop low?* The evidence assembled in Chapter 4 supports a qualified *yes*. Revisiting the five aims from Chapter 1:

1. **A typed, shared state object as the communication substrate** (Section 3.2.1) replaced free-form chat-history-based coordination with three explicit pydantic/dataclass contracts, `ResearchPlan`, `PlannerResult`, `AssessorResult`, that are passed between nodes of a five-node LangGraph (Section 3.2.2). Every arrow in Fig. 3.3 corresponds to a field read or written on `State`, making the communication protocol between roles fully inspectable, in contrast to the implicit, prose-based coordination of chat-only multi-agent frameworks (Section 2.6).

2. **A cost-asymmetric model assignment**, lightweight models (`gpt-4.1-mini` / `claude-haiku-4-5`) for the four-times-per-iteration planner/dispatcher/assessor loop, and a single heavyweight model (`claude-sonnet-4-5`) for one-shot synthesis, was implemented and, per Section 4.9, behaves as designed: the lightweight loop dominates wall-clock time while the heavyweight call dominates per-call cost, giving a favourable overall cost/latency profile relative to running a single large model for every step.

3. **Integration of 13 MCP tool servers** (Section 3.2.4, Fig. 3.4), including a custom-built `thinking` server (Section 3.2.4.1), gave the agent access to web search, browser automation, code execution, symbolic computation, academic search, and structured knowledge storage, and Section 4.6's MCP-Atlas result demonstrates that the *composition* of this tool roster is itself a first-class design variable, not an afterthought.

4. **A resumable benchmark harness** (Section 3.3.1, Fig. 3.8) made it possible to evaluate the system against six benchmarks, three required by the original project

brief (GAIA, DRACO, MCP-Atlas) and two selected to broaden coverage to factual recall and multi-hop reasoning (SimpleQA, FRAMES), plus BrowseComp as a stress test, using adapter-specific scoring (exact match, fuzzy containment, LLM-judge rubric, GTFA coverage).

5. **A full-stack, no-login application** (Section 3.2.5) exposes the agent's internal structured state directly in the UI (Fig. 3.5), alongside a configuration page for models, tools, and budgets (Fig. 3.6) and a live benchmarks dashboard (Fig. 3.7, Fig. 4.4): turning what is normally an opaque agent loop into something a user can inspect step by step.

On the headline numbers (Fig. 4.1): SimpleQA 62.0% (vs. 28.9% Claude 3.5 Sonnet), FRAMES 53.0% (vs. 51.0% GPT-4o naive), GAIA 87.0% (vs. 84.1% Claude 3.7 + GPT-4o Aria v2.0), DRACO 61.45% (vs. 52.1% OpenAI o3), and MCP-Atlas 68.0% (matching GPT-5.2's 67.6%). On five of the six benchmarks the agent performs strongly enough to match or exceed at least one published frontier or multi-agent baseline; BrowseComp (8.3% vs. 51.5% OpenAI Deep Research) is the exception, and is examined in detail below precisely because its failure mode is so clearly diagnosable.

5.2 Limitations

Three limitations qualify the results above and should be read alongside them:

- **Subset sizes.** As noted in Section 3.3.5, subsets of 12–80 tasks were used per benchmark (with GAIA's strong configuration run on the full 139-task Level 1+2 split), owing to the wall-clock and API-cost budget available for this thesis. The resulting ±10–15 percentage point 95% confidence intervals mean the relative ordering against baselines (the agent's main claim) is more reliable than the absolute percentages themselves.
- **Configuration-dependent results.** GAIA and MCP-Atlas both show a large gap between the "baseline" and "strong" configurations (68.0% → 87.0% and 45.7% → 68.0% respectively). The headline numbers in the abstract and Fig. 4.1 report the *best* configuration per benchmark; a single fixed configuration across all benchmarks would show a less favourable picture for GAIA and MCP-Atlas specifically, though it would not change the SimpleQA, FRAMES, or DRACO results, which were run under the baseline configuration throughout.
- **The BrowseComp gap is real and architectural**, not a tuning artefact (Section 4.6). It reflects a genuine limitation of the fixed-iteration-budget planner/asses-

sor loop on open-ended adversarial search, and is the motivation for the first future-work direction below.

5.3 Future Work

Adaptive iteration budgets. The single most direct response to the BrowseComp result (Section 4.6) is to make `max_iterations` and `max_tool_calls` a function of the assessor's running confidence trajectory rather than a fixed per-query constant: if confidence is monotonically increasing across iterations, extend the budget; if it has plateaued, terminate early regardless of the nominal cap. This would let bounded multi-hop questions (GAIA, FRAMES) terminate cheaply while giving BrowseComp-style questions the dozens of iterations that OpenAI Deep Research-class systems use.

Mandatory structured scratchpad for long plans. The FRAMES discussion (Section 2.8, Chapter 4) noted that `tool_results` is currently a flat append-only list, which works well for plans of 2–3 steps but loses earlier facts as a plan grows. Routing intermediate facts through the `memory` MCP server's knowledge graph (Table 3.9) as a mandatory step, rather than an optional tool the dispatcher may or may not choose, would give the planner a queryable structured record of everything discovered so far, likely improving performance on FRAMES-style 4+ hop questions and on DRACO's citation-density scoring.

Reducing structured-output fallback frequency. Section 3.2.1 notes that `Planner--Result` and `AssessorResult` occasionally fail to parse from smaller models, triggering a default-value fallback. The GAIA jump from gpt-4.1-mini to claude-haiku-4-5 orchestration (68.0% $\rightarrow$ 87.0%) suggests this fallback rate has a measurable effect on downstream accuracy; instrumenting and reporting this rate per model, and exploring constrained-decoding or grammar-based structured output as an alternative to function-calling, would make this failure mode visible and addressable rather than silently absorbed.

Broader benchmark coverage and larger subsets. Re-running all six benchmarks at full dataset size (e.g. SimpleQA's full 4,326 questions, BrowseComp's full 1,266) under a single fixed configuration would tighten the confidence intervals discussed in Section 5.2 and would let a single number be reported per benchmark rather than a baseline/strong pair.

Returning to the original A2A framing. The pivot described in Section 1.2 narrowed the scope from cross-*framework* agent communication (e.g. exposing this

agent's planner/assessor roles as independently-addressable A2A agents that another framework's orchestrator could call) to communication *within* a single framework. With the internal typed-state protocol of Section 3.2.1 now implemented and validated, a natural extension is to expose the `planner` and `synthesizer` nodes as standalone MCP or A2A-compatible services (Section 2.5), allowing the cost/quality trade-off demonstrated in this thesis to be composed into larger, cross-framework multi-agent systems.

5.4 Final Remarks

The research agent presented in this thesis demonstrates that careful attention to *how* information flows between the parts of an LLM-based system, a typed state object, a small fixed set of structured contracts, and an explicit routing table (Table 3.7), can be as consequential to benchmark performance as the choice of underlying language model. Across five benchmarks spanning factual recall, multi-hop reasoning, deep research, and tool-use competency, a system built primarily from `gpt-4.1-mini`-class and `claude-haiku-4-5`-class models performs strongly and consistently, in several cases matching or exceeding systems built around substantially larger and more expensive models. BrowseComp, the one benchmark where this does not hold, points at a specific, well-understood, and addressable architectural limitation rather than a fundamental flaw in the approach, which is, ultimately, a useful and constructive negative result for a thesis to produce.

Bibliography

[1] A. Vaswani *et al.*, "Attention Is All You Need," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017. [Online]. Available: https://arxiv.org/abs/1706.03762

[2] T. Brown *et al.*, "Language Models are Few-Shot Learners," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[3] S. Yao *et al.*, "ReAct: Synergizing Reasoning and Acting in Language Models," *arXiv preprint*, arXiv:2210.03629, 2022. [Online]. Available: https://arxiv.org/abs/2210.03629

[4] T. Schick *et al.*, "Toolformer: Language Models Can Teach Themselves to Use Tools," *arXiv preprint*, arXiv:2302.04761, 2023. [Online]. Available: https://arxiv.org/abs/2302.04761

[5] M. Minsky, *The Society of Mind*. New York, NY, USA: Simon & Schuster, 1988.

[6] Anthropic, "Introducing the Model Context Protocol," 2024. [Online]. Available: https://www.anthropic.com/news/model-context-protocol

[7] JSON-RPC Working Group, "JSON-RPC 2.0 Specification," 2013. [Online]. Available: https://www.jsonrpc.org/specification

[8] Google, "Announcing the Agent2Agent Protocol (A2A)," Google Developers Blog, 2025. [Online]. Available: https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/

[9] Linux Foundation, "Linux Foundation Launches the Agent2Agent Protocol Project," 2025. [Online]. Available: https://www.linuxfoundation.org/press/linux-foundation-launches-the-agent2agent-protocol-project-to-enable-secure-intelligent-communication-between-ai-agents

[10] Q. Duan and Z. Lu, "Agent Communications toward Agentic AI at Edge: A Case Study of the Agent2Agent Protocol," *arXiv preprint*, arXiv:2508.15819, 2025. [Online]. Available: https://arxiv.org/abs/2508.15819

[11] IBM Research, "Agent Communication Protocol (ACP)," 2025. [Online]. Available: https://research.ibm.com/projects/agent-communication-protocol

[12] IBM Research, "Introducing multiagent BeeAI," 2025. [Online]. Available: https://research.ibm.com/blog/multiagent-bee-ai

[13] Google, "Making it easy to build multi-agent applications: Agent Development Kit (ADK)," Google Developers Blog, 2025. [Online]. Available: https://developers.googleblog.com/en/agent-development-kit-easy-to-build-multi-agent-applications/

[14] H. Chase *et al.*, "LangChain: The Platform for Reliable Agents," LangChain, 2025. [Online]. Available: https://www.langchain.com

[15] LangChain Team, "Graph: LangChain Documentation," LangChain, 2025. [Online]. Available: https://python.langchain.com/api_reference/core/runnables/langchain_core.runnables.graph.Graph.html

[16] Microsoft, "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation," 2024. [Online]. Available: https://www.microsoft.com/en-us/research/blog/autogen-enabling-next-gen-llm-applications-via-multi-agent-conversation/

[17] K. Zhu *et al.*, "MultiAgentBench: Evaluating the Collaboration and Competition of LLM agents," in *Proc. ACL*, 2025. [Online]. Available: https://aclanthology.org/2025.acl-long.421/

[18] W. Wang *et al.*, "BattleAgentBench: A Benchmark for Evaluating Cooperation and Competition Capabilities of Language Models in Multi-Agent Systems," *arXiv preprint*, arXiv:2408.15971, 2024. [Online]. Available: https://arxiv.org/abs/2408.15971

[19] J. Wei *et al.*, "Measuring short-form factuality in large language models," OpenAI, *arXiv preprint*, arXiv:2411.04368, 2024. [Online]. Available: https://arxiv.org/abs/2411.04368

[20] S. Krishna *et al.*, "Fact, Fetch, and Reason: A Unified Evaluation of Retrieval-Augmented Generation," *arXiv preprint*, arXiv:2409.12941, 2024. [Online]. Available: https://arxiv.org/abs/2409.12941

[21] J. Wei *et al.*, "BrowseComp: A Simple Yet Challenging Benchmark for Browsing Agents," OpenAI, *arXiv preprint*, arXiv:2504.12516, 2025. [Online]. Available: https://arxiv.org/abs/2504.12516

[22] G. Mialon *et al.*, "GAIA: a benchmark for General AI Assistants," *arXiv preprint*, arXiv:2311.12983, 2023. [Online]. Available: https://arxiv.org/abs/2311.12983

[23] J. Zhong *et al.*, "DRACO: A Cross-Domain Benchmark for Deep Research Accuracy, Completeness, and Objectivity," *arXiv preprint*, arXiv:2602.11685, 2026. [Online]. Available: https://arxiv.org/abs/2602.11685

[24] C. Bandi *et al.*, "MCP-Atlas: A Large-Scale Benchmark for Tool-Use Competency with Real MCP Servers," *arXiv preprint*, arXiv:2602.00933, 2026. [Online]. Available: https://arxiv.org/abs/2602.00933

[25] J. Kaplan *et al.*, "Scaling Laws for Neural Language Models," *arXiv preprint*, arXiv:2001.08361, 2020. [Online]. Available: https://arxiv.org/abs/2001.08361

[26] J. Wei *et al.*, "Emergent Abilities of Large Language Models," *Transactions on Machine Learning Research*, 2022. [Online]. Available: https://arxiv.org/abs/2206.07682

[27] J. Wei *et al.*, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, 2022. [Online]. Available: https://arxiv.org/abs/2201.11903

[28] T. Kojima *et al.*, "Large Language Models are Zero-Shot Reasoners," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, 2022. [Online]. Available: https://arxiv.org/abs/2205.11916

[29] S. Yao *et al.*, "Tree of Thoughts: Deliberate Problem Solving with Large Language Models," *arXiv preprint*, arXiv:2305.10601, 2023. [Online]. Available: https://arxiv.org/abs/2305.10601

[30] N. Shinn *et al.*, "Reflexion: Language Agents with Verbal Reinforcement Learning," *arXiv preprint*, arXiv:2303.11366, 2023. [Online]. Available: https://arxiv.org/abs/2303.11366

[31] A. Madaan *et al.*, "Self-Refine: Iterative Refinement with Self-Feedback," *arXiv preprint*, arXiv:2303.17651, 2023. [Online]. Available: https://arxiv.org/abs/2303.17651

[32] P. Lewis *et al.*, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, 2020. [Online]. Available: https://arxiv.org/abs/2005.11401

[33] V. Karpukhin *et al.*, "Dense Passage Retrieval for Open-Domain Question Answering," in *Proc. EMNLP*, 2020. [Online]. Available: https://arxiv.org/abs/2004.04906

[34] R. Nakano *et al.*, "WebGPT: Browser-assisted question-answering with human feedback," *arXiv preprint*, arXiv:2112.09332, 2021. [Online]. Available: https://arxiv.org/abs/2112.09332

[35] E. Karpas *et al.*, "MRKL Systems: A modular, neuro-symbolic architecture that combines large language models, external knowledge sources and discrete reasoning," *arXiv preprint*, arXiv:2205.00445, 2022. [Online]. Available: https://arxiv.org/abs/2205.00445

[36] Y. Qin *et al.*, "ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs," *arXiv preprint*, arXiv:2307.16789, 2023. [Online]. Available: https://arxiv.org/abs/2307.16789

[37] G. Mialon *et al.*, "Augmented Language Models: a Survey," *arXiv preprint*, arXiv:2302.07842, 2023. [Online]. Available: https://arxiv.org/abs/2302.07842

[38] J. S. Park *et al.*, "Generative Agents: Interactive Simulacra of Human Behavior," in *Proc. ACM Symposium on User Interface Software and Technology (UIST)*, 2023. [Online]. Available: https://arxiv.org/abs/2304.03442

[39] M. Wooldridge, *An Introduction to MultiAgent Systems*, 2nd ed. Chichester, UK: John Wiley & Sons, 2009.

[40] Foundation for Intelligent Physical Agents (FIPA), "FIPA ACL Message Structure Specification," SC00061G, 2002. [Online]. Available: http://www.fipa.org/specs/fipa00061/

[41] Wolfram Research, "Wolfram|Alpha API Reference," 2024. [Online]. Available: https://products.wolframalpha.com/api

[42] E2B, "E2B: Open-source runtime for AI agents," 2024. [Online]. Available: https://e2b.dev

[43] T. Kluyver *et al.*, "Jupyter Notebooks: a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, IOS Press, 2016, pp. 87–90.

[44] Microsoft, "Playwright: Fast and reliable end-to-end testing for modern web apps," 2024. [Online]. Available: https://playwright.dev

[45] Z. Yang *et al.*, "HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering," in *Proc. EMNLP*, 2018. [Online]. Available: https://arxiv.org/abs/1809.09600

[46] M. Joshi, E. Choi, D. S. Weld, and L. Zettlemoyer, "TriviaQA: A Large Scale Distantly Supervised Challenge Dataset for Reading Comprehension," in *Proc. ACL*, 2017. [Online]. Available: https://arxiv.org/abs/1705.03551

Appendix A

Agent Prompt Templates

This appendix reproduces, in full, the system prompt templates used by the four LLM-driven nodes of the research agent's graph: the planner, the tool dispatcher, the assessor, and the synthesizer (Section 3.2.2 and Section 3.2.2.3). Each template is a Python string with named placeholders (e.g. `{system_prompt_override}`, `{current_focus}`) that are filled in at runtime via `str.format()` from the corresponding fields of `State` (Listing 3.1). The `{system_prompt_override}` placeholder is, by default, an empty string; it exists so that the configuration system (Section 3.2.3) can inject additional instructions: for example, benchmark-specific answer-formatting hints: without modifying the source files. The placeholders are listed verbatim as they appear in `research_agent/prompts.py`; no paraphrasing or abridgement has been applied beyond the removal of two short legacy prompts (`ROUTING_QUERY_SYSTEM_PROMPT` and `ROUTING_RESPONSE_SYSTEM_PROMPT`) that are retained in the codebase only for backward compatibility with the original three-node prototype described in Section 3.1.2 and are not used by the five-node graph.

A.1 Planner Prompt

The planner prompt (Listing A.1) is the single most important piece of text in the system from a benchmark-performance perspective. It performs two jobs at once: it decides whether external tools are required at all (Step 1), and, if so, it decomposes the question into an ordered list of 2–5 steps, each annotated with the MCP server that should service it (Step 2). The explicit "MANDATORY EXCEPTIONS" block was added after the failure analysis described in Section 3.1.3, where early versions of the planner attempted to solve algebraic equations and look up scientific constants from parametric memory rather than delegating to Wolfram$_\alpha$.

Listing A.1: Planner system prompt (`PLANNER_SYSTEM_PROMPT`).

```
1 You are a systematic research planner. A user has asked a
     question.
2 Your job is to think carefully and produce a structured
     research plan.
3
4 {system_prompt_override}
5
```

```
## STEP 1 -- Decide if tools are even needed (CRITICAL)

Set decision="done" with steps=[] and current_step=0 immediately for:
- Greetings or casual remarks: "hey", "hi", "hello", "thanks", "cool", "ok"
- Trivial arithmetic with no unknowns: "2+2", "what is 5*5", "how many seconds in a minute"
- Follow-up questions answerable from the conversation history: "what was that again?",
  "what did you just say?", "who is that?", "name?", "what about them?", "their age?"
- Requests to clarify, rephrase, or summarize your last answer
- Questions where the answer is obvious from prior messages in the conversation

The synthesizer will answer these directly from conversation context and general knowledge.
DO NOT plan tool steps for these -- it wastes time and produces worse answers.

## MANDATORY EXCEPTIONS -- ALWAYS use tools for these, no matter how "simple" they look
- ANY algebraic equation: x^2 + 5x + 6 = 0, solve for x, quadratic, cubic, etc.
- ANY symbolic math: derivatives, integrals, limits, series, proofs
- ANY formula evaluation requiring a tool: compound interest, trig, logs
- ANY question asking to "verify", "check", or "prove" a mathematical claim
- ANY scientific constant lookup: speed of light, Planck's constant, etc.

For all of the above -> ALWAYS plan a `compute (wolfram)` step.

## STEP 2 -- Plan tool use ONLY if you genuinely need to look something up

If the question requires current facts, specific data, or computation not in context:

```

```
### Available tool categories (MUST include server name in parentheses in each step)
- web_search (tavily): Fast factual retrieval, news, general information
- browser (playwright): JS-heavy pages, specific URLs that need rendering
- compute (wolfram): Exact math, equations, symbolic computation, scientific constants
- code_exec (e2b): Python calculations, data analysis, numerical methods, notebooks
- academic (arxiv): Research papers, preprints, scientific literature
- url_fetch (fetch): Read a specific known URL without JavaScript
- deep_reasoning (thinking): Graduate-level problems that require extended reasoning
- memory (memory): Store/retrieve findings from earlier in this session

### Planning rules
- Produce 2-5 concrete, verifiable steps. No vague steps like "research the topic."
- Each step MUST be formatted as: `category (server): description` -- e.g. `deep_reasoning (thinking): prove X`
- NEVER drop the parenthetical server name -- it is used to route the dispatcher.
- For math/science: ALWAYS use compute or code_exec rather than estimating.
- For multi-hop facts: treat EACH hop as a separate step.
- Keep it minimal: if 1 step is enough, use 1 step.

## Decision values
- "new": First time planning (produce fresh plan with steps)
- "continue": Re-planning after assessor feedback (incorporate the feedback)
- "done": Either (a) all steps complete, OR (b) no tools needed -- answer directly

## Current state
Assessor feedback from last attempt: {assessor_feedback}
Current plan: {current_plan}
Iteration: {iteration}
System time: {system_time}
```

A.2 Tool Dispatcher Prompt

The dispatcher prompt (Listing A.2) receives the `current_focus` string produced by the planner and must select exactly one tool call. Its structure mirrors the planner's: an "OVERRIDE" block of hard, string-matching rules is checked first, and only if none of those triggers fires does the model fall back to the more general priority-ordered rules. This two-tier structure was introduced during the redesign described in Section 3.1.4 after observing that a single flat list of rules was frequently ignored by the lightweight model in favour of its default ("always search the web") behaviour.

Listing A.2: Tool dispatcher system prompt (`DISPATCHER_SYSTEM_PROMPT`).

```
1  You are a precise tool dispatcher. You must select EXACTLY ONE
       tool call to make progress.
2
3  {system_prompt_override}
4
5  ## Current research focus
6  {current_focus}
7
8  ## Full research plan
9  {plan_steps}
10
11 ## WARNING: MANDATORY SERVER HINT
12 If the current focus contains a server name in parentheses --
     e.g. "(arxiv)", "(wolfram)", "(e2b)",
13 "(fetch)", "(thinking)", "(memory)", "(playwright)" -- you
     MUST use a tool from that server.
14 This is a hard constraint. Do not use tavily or any other
     server when a hint is present.
15
16 ## Tool selection rules -- FOLLOW IN PRIORITY ORDER
17
18 **OVERRIDE -- tool-specific triggers (use these FIRST
     regardless of rule order below):**
19 - Focus mentions "(arxiv)" OR "academic paper" OR "research
     paper" OR "preprint" -> use **arxiv** search_papers tool
20 - Focus mentions "(wolfram)" OR "compute" OR "solve equation"
     OR "exact math" -> use **wolfram** query-wolfram-alpha tool
21 - Focus mentions "(e2b)" OR "run code" OR "python" OR "execute
```

```
    " -> use **e2b** run_python tool
- Focus mentions "(fetch)" OR "read this URL" OR specific known URL -> use **fetch** tool
- Focus mentions "(thinking)" OR "deep reasoning" OR "expert-level" -> use **thinking** think_deeply tool
- Focus mentions "(memory)" OR "store" OR "remember" -> use **memory** tool
- Focus mentions "(playwright)" OR "JavaScript-heavy page" -> use **playwright** tool

**General rules (when no override applies):**
1. Use **tavily** for web search of current facts, news, people, events, organizations
2. Use **playwright** ONLY when a specific URL needs JavaScript rendering
3. Use **wolfram** for ANY mathematical computation -- never calculate manually
4. Use **e2b** for Python code execution, data analysis, numerical methods
5. Use **arxiv** for academic papers and scientific literature
6. Use **fetch** for known URLs that don't need JavaScript
7. Use **thinking** for expert-level STEM or after other tools fail 3+ times
8. Use **memory** to save important findings between steps

## Critical rules
- Make EXACTLY ONE tool call -- no preamble, no commentary, just the tool call
- Use concrete, specific arguments (actual search queries, not placeholders)
- For tavily searches: use precise, targeted queries (include names, dates, specific terms)
- For arxiv: use relevant academic keywords (e.g., "retrieval augmented generation survey 2024")
- For wolfram: pass the equation/formula directly, e.g. "solve x^2 + 5x + 6 = 0"
- For e2b: write complete, runnable Python code
- For fetch: pass the exact URL to retrieve

## Available tools and their servers
{tool_list}

System time: {system_time}
```

A.3 Assessor Prompt

The assessor prompt (Listing A.3) is deliberately the shortest of the four. Its sole job is to compress the tool results from the current iteration into a binary `is_complete` judgement, a confidence score, and, if incomplete, a list of concrete follow-up queries that the planner can fold into a revised plan on the next pass. The confidence threshold of 0.7 was tuned empirically: an earlier threshold of 0.5 (Section 3.1.8) allowed the loop to terminate on weak or tangential evidence too often, while a threshold of 0.9 caused excessive re-querying on questions where the available sources were inherently imprecise (e.g. population estimates).

Listing A.3: Assessor system prompt (`ASSESSOR_SYSTEM_PROMPT`).

```
You are a strict research assessor. Your job: determine if the current research step has been completed.

{system_prompt_override}

## What we were trying to find
{current_focus}

## Tool results from this iteration
{tool_results_summary}

## Assessment rules
1. **Strict**: Only mark complete if you can extract the SPECIFIC answer needed for this step
2. **Confidence threshold**: confidence must be >= 0.7 to mark complete
3. **No partial credit**: if the answer is ambiguous or could be wrong, mark incomplete
4. **Missing items**: list exactly what information is STILL needed as concrete search queries

## When to mark complete
- The exact fact/value needed for this step is present in the tool results
- The computation has been performed and the result is unambiguous
- A reliable source (Wikipedia, Wolfram, arXiv) confirms the
```

```
    answer

## When to mark incomplete
- The tool returned empty results or an error
- The result is about a similar topic but not the specific
    thing asked
- The confidence is low because sources conflict or the
    evidence is indirect

Iteration count at this step: {attempt_count}
System time: {system_time}
```

A.4 Synthesizer Prompt

The synthesizer prompt (Listing A.4) is the only one of the four sent to the heavy model. It receives the entire accumulated evidence trail; every entry of `tool_results` formatted as a numbered list: together with the current contents of the research notebook (Section 3.2.4) and a per-benchmark `benchmark_hint` string that encodes answer-formatting requirements (e.g. "answer with a single number, no units" for GAIA-style exact-match scoring). The branch for "no tool results" exists so that conversational turns identified by the planner in Step 1 (Section A.1) are still answered naturally, without the synthesizer assuming that an empty evidence list represents a failed search.

Listing A.4: Synthesizer system prompt (`SYNTHESIZER_SYSTEM_PROMPT`).

```
You are a precise research synthesizer. Your task: produce the
    final answer.

{system_prompt_override}

## Original question
{question}

## All gathered evidence (from tool calls)
{tool_results_formatted}

## Research notebook contents
{notebook_state}

## Synthesis rules

### If tool results are present:
```

```
1. Use ONLY evidence from the tool results above -- no hallucination
2. Cite your sources: include the URL where each key fact was found
3. Format by question type:
   - Factual (who/what/when/where): ONE short phrase + source URL
     Example: "Marie Curie. Source: https://en.wikipedia.org/wiki/Marie_Curie"
   - Mathematical: show step-by-step work, then state the final answer clearly
   - Scientific: concise explanation with key evidence cited
   - Multi-hop: chain the facts together, cite each separately
4. Never say "I cannot determine": give your best answer from available evidence

### If NO tool results (tool_results_formatted is empty / "(no tool results yet)"):
This is a conversational or simple question. Answer it naturally:
- Use the conversation history above for context (follow-up questions, greetings, etc.)
- Use your general knowledge for simple facts (arithmetic, common knowledge)
- Be concise and natural -- no need for citations
- Match the tone of the conversation

5. **Benchmark hint** (for evaluations): {benchmark_hint}

System time: {system_time}
```

Appendix B

Sample Agent Trace

To make the abstract node descriptions of Section 3.2.2 concrete, this appendix walks through a complete run of the agent on a representative multi-hop question, showing the evolution of the `State` object (Listing 3.1) iteration by iteration. The question was chosen because it requires three distinct tool categories (web search, biographical lookup, and exact arithmetic) and therefore exercises the full planner/dispatcher/executor/assessor loop three times before the synthesizer is invoked. Tool result text has been abbreviated to the fields relevant to the assessment; the actual MCP responses additionally include metadata (timestamps, result counts, raw HTML snippets for Playwright) that is omitted here for brevity.

B.1 Question

"Who directed the film that won the Academy Award for Best Picture at the ceremony held in 2020, and how many years after the founding of the Academy of Motion Picture Arts and Sciences (1927) was that director born?"

B.2 Iteration 1: Planning

The planner node receives the question with an empty plan (`plan=None`) and `iteration=0`. Because the question requires both a current factual lookup and an exact arithmetic computation, Step 1 of the planner prompt (Section A.1) does not apply, and the model proceeds to Step 2, producing a three-step plan:

Listing B.1: `ResearchPlan` produced on the first planner call.

```
{
  "steps": [
    "web_search (tavily): Identify the director of the film
       that won the Academy Award for Best Picture at the
       ceremony held in 2020",
    "web_search (tavily): Find the birth year of that director
       ",
    "compute (wolfram): Compute 1969 minus 1927 to find how
```

```
        many years after the founding of AMPAS the director was
         born"
  ],
  "current_step": 0,
  "current_focus": "web_search (tavily): Identify the director
      of the film that won the Academy Award for Best Picture
     at the ceremony held in 2020",
  "decision": "new"
}
```

Note that the third step's description already anticipates the year 1969; this is an artefact of the planner's parametric knowledge and is *not* trusted by the rest of the pipeline; the second step independently re-derives the birth year from a tool result, and the third step's wolfram query (Section B.3) is dispatched using the value extracted from that result, not the value mentioned in the plan text.

Dispatch and execution

The dispatcher reads `current_focus`, matches no override trigger (Section A.2), and falls through to General Rule 1 (use tavily for current facts), issuing:

Listing B.2: Tool call selected by the dispatcher for step 1.

```
{
  "server": "tavily",
  "tool": "tavily-search",
  "args": { "query": "Academy Award Best Picture 2020 winner
     director" }
}
```

The tool executor invokes this call via `mcp.apply()` (Section 3.2.4) and appends the result to `tool_results`:

Listing B.3: Tool result appended after iteration 1.

```
{
  "tool": "tavily-search",
  "server": "tavily",
  "args": { "query": "Academy Award Best Picture 2020 winner
     director" },
  "result": "Parasite (2019) won the Academy Award for Best
     Picture at the 92nd Academy Awards, held on 9 February
     2020. The film was directed by Bong Joon-ho. Source:
     https://en.wikipedia.org/wiki/Parasite_(2019_film)",
```

```
  "iteration": 1
}
```

Assessment

The assessor receives `current_focus` and the single tool result above. The director's name ("Bong Joon-ho") is present and unambiguous, so the assessor returns:

Listing B.4: `AssessorResult` for iteration 1.

```
{
  "is_complete": true,
  "confidence": 0.95,
  "feedback": "",
  "missing": []
}
```

Because `is_complete=true` and the plan has further steps, the graph routes back to the planner with `plan.advance()` applied (Section 3.2.2), producing `current_step=1` and `decision="continue"`.

B.3 Iteration 2: Second Planner Pass

The planner is invoked again, this time with `plan.decision="continue"` and a non-empty `tool_results`. Since the existing plan already covers the remaining work, the planner leaves `steps` unchanged and confirms `current_focus` for step 2 ("Find the birth year of that director"). The dispatcher again falls through to General Rule 1 and issues:

Listing B.5: Tool call selected by the dispatcher for step 2.

```
{
  "server": "tavily",
  "tool": "tavily-search",
  "args": { "query": "Bong Joon-ho date of birth" }
}
```

Listing B.6: Tool result appended after iteration 2.

```
{
  "tool": "tavily-search",
  "server": "tavily",
  "args": { "query": "Bong Joon-ho date of birth" },
```

```
  "result": "Bong Joon-ho (born September 14, 1969) is a South
      Korean film director and screenwriter. Source: https://
     en.wikipedia.org/wiki/Bong_Joon-ho",
  "iteration": 2
}
```

The assessor again returns `is_complete=true` with confidence 0.92 (the date is explicit and the source is Wikipedia, satisfying the "reliable source" criterion of Section A.3). The plan advances to step 3.

B.4 Iteration 3: Computation

For step 3, the dispatcher's override block fires on the literal string `"(wolfram)"` present in `current_focus`, regardless of the general rules, and issues a Wolfram$_\alpha$ query constructed from the value extracted in iteration 2 (1969) rather than the value originally guessed by the planner in iteration 1:

Listing B.7: Tool call selected by the dispatcher for step 3.

```
{
  "server": "wolfram",
  "tool": "query-wolfram-alpha",
  "args": { "input": "1969 - 1927" }
}
```

Listing B.8: Tool result appended after iteration 3.

```
{
  "tool": "query-wolfram-alpha",
  "server": "wolfram",
  "args": { "input": "1969 - 1927" },
  "result": "42",
  "iteration": 3
}
```

The assessor marks step 3 complete with confidence 1.0 (an exact numeric result from Wolfram$_\alpha$ is treated as maximally reliable). Since `plan.current_step` now equals `len(p--lan.steps)`, `plan.is_done()` returns `true`, and the planner: invoked one final time: sets `decision="done"`, routing the graph to the synthesizer.

B.5 Synthesis

The synthesizer receives all three `tool_results` entries, an empty `notebook_state` (no E2B sandbox was used in this trace), and the default `benchmark_hint` (empty string, since this is an interactive query rather than a benchmark item). Following the "multi-hop" formatting rule of Section A.4, it chains the three facts and cites both sources:

"Bong Joon-ho directed Parasite, which won the Academy Award for Best Picture at the 92nd Academy Awards (held in 2020). Bong was born on 14 September 1969, which is 42 years after the Academy of Motion Picture Arts and Sciences was founded in 1927."

Sources:
`https://en.wikipedia.org/wiki/Parasite_(2019_film)`
`https://en.wikipedia.org/wiki/Bong_Joon-ho`

The complete trace required `iteration=3` planner/dispatcher/executor/ assessor cycles and four LLM calls to the lightweight model (one planner call per iteration plus the final "done" confirmation) and one call to the heavy model for synthesis, plus three MCP tool invocations (two `tavily`, one `wolfram`). This is representative of the median trace length observed on FRAMES (Section 4.2), which is dominated by two- and three-hop questions.

Appendix C

MCP Server Configuration

Section 3.2.4 summarises the role of each of the thirteen MCP servers used by the final configuration of the agent (Table 3.9). This appendix reproduces the underlying `mcp-servers-config.json` file in full, including the launch command, arguments, and required environment variables for every server. Each server is launched as a separate `stdio` subprocess by the LangChain MCP adapter (Section 3.2.4); the agent process communicates with each subprocess over its stdin/stdout pipe using the JSON-RPC-based MCP wire protocol. Table C.1 cross-references each server with the environment variable(s) it requires and the corresponding entry in `.env.example`.

Listing C.1: Full `mcp-servers-config.json`, all thirteen servers.

```
{
    "mcpServers": {
        "tavily": {
            "command": "npx",
            "args": ["-y", "tavily-mcp@0.1.4"],
            "env": { "TAVILY_API_KEY": "${TAVILY_API_KEY}" },
            "description": "Tavily AI-powered web search.
               Returns clean structured
              results with titles, URLs, and excerpts. PREFER
                 this over playwright
              for all factual research queries - it is 5x
                 faster and more reliable
              than browser navigation for search."
        },
        "playwright": {
            "command": "npx",
            "args": ["-y", "@playwright/mcp@latest", "--
               headless"],
            "description": "Full Chromium web browser. Use
               ONLY for JS-heavy pages
              that Tavily cannot reach, or when you have a
                 specific URL that
              requires JavaScript rendering to display content
                 ."
        },
        "e2b": {
```

```
            "command": "<agent-venv>/Scripts/python.exe",
            "args": ["-m", "research_agent.mcp.e2b_server"],
            "env": { "E2B_API_KEY": "${E2B_API_KEY}" },
            "description": "E2B sandboxed Python execution environment. Use for:
              running Python code, data analysis, numerical calculations,
              plotting graphs, and maintaining persistent research notebooks
              across a session. Supports NumPy, SciPy, SymPy, matplotlib, pandas."
        },
        "wolfram": {
            "command": "<agent-venv>/Scripts/python.exe",
            "args": ["-m", "research_agent.mcp.wolfram_server"],
            "env": { "WOLFRAM_APP_ID": "${WOLFRAM_APP_ID}",
                     "WOLFRAM_API_KEY": "${WOLFRAM_API_KEY}" },
            "description": "Wolfram Alpha computation engine. Use for: precise
              mathematical computations, symbolic math, equation solving,
              scientific constants, unit conversions, and any query requiring
              exact numerical answers. ALWAYS use before attempting manual
              calculation."
        },
        "arxiv": {
            "command": "uvx",
            "args": ["arxiv-mcp-server"],
            "description": "ArXiv academic paper search and retrieval. Use for:
              finding research papers, preprints, citations, and academic
              literature on scientific and mathematical topics. Returns paper
              abstracts, authors, and PDF links."
        },
        "fetch": {
            "command": "uvx",
            "args": ["mcp-server-fetch"],
```

```
            "description": "Direct URL fetching with clean
                HTML-to-text
              conversion. Use when you have a specific URL and
                  need to read its
              full content without JavaScript rendering (
                 documentation pages,
              Wikipedia articles, blog posts, PDFs via URL)."
        },
        "thinking": {
            "command": "<agent-venv>/Scripts/python.exe",
            "args": ["-m", "research_agent.mcp.thinking_server
                "],
            "env": { "THINKING_MODEL": "${THINKING_MODEL}",
                     "ANTHROPIC_API_KEY": "${ANTHROPIC_API_KEY
                        }",
                     "OPENAI_API_KEY": "${OPENAI_API_KEY}" },
            "description": "Deep reasoning engine. Delegates
                to a powerful model
              (Claude Sonnet or o3-mini) with extended
                 thinking enabled. Use for:
              graduate-level STEM problems, complex multi-step
                  proofs, scientific
              reasoning that exceeds your own capability, and
                 problems where
              previous attempts have failed."
        },
        "memory": {
            "command": "npx",
            "args": ["-y", "@modelcontextprotocol/server-
                memory"],
            "description": "Persistent knowledge-graph memory.
                 Store and retrieve
              key facts, findings, and intermediate results
                 across conversation
              turns. Use to track what has been found so far
                 in multi-step
              research."
        },
        "filesystem": {
            "command": "npx",
            "args": ["-y", "@modelcontextprotocol/server-
                filesystem", "."],
            "description": "Read, write, and search local
```

```
                files and directories.
79              Use for reading uploaded documents, writing
                    research notes, and
80              saving outputs."
81          },
82          "wikipedia": {
83              "command": "uvx",
84              "args": ["wikipedia-mcp"],
85              "description": "Wikipedia search and article
                    retrieval. Use for:
86              encyclopedic facts, biographical/historical/
                    geographical
87              information, and definitions. Returns full
                    article text and
88              summaries."
89          },
90          "duckduckgo": {
91              "command": "uvx",
92              "args": ["duckduckgo-mcp-server"],
93              "description": "DuckDuckGo web search. Use as an
                    alternative search
94              engine when Tavily results are insufficient or
                    to cross-check facts
95              from a second source."
96          },
97          "clinicaltrials": {
98              "command": "uvx",
99              "args": ["clinicaltrials-mcp"],
100             "env": { "FASTMCP_SHOW_BANNER": "false" },
101             "description": "ClinicalTrials.gov search. Use for
                    : finding clinical
102             trial details, status, sponsors, enrollment
                    numbers, and study
103             results for specific drugs/conditions."
104         },
105         "weather": {
106             "command": "npx",
107             "args": ["-y", "open-meteo-mcp"],
108             "description": "Weather forecasts, air quality, UV
                    index, and
109             historical weather data via Open-Meteo (no API
                    key). Use for:
110             current/forecast weather, historical weather
```

```
                    data, and weather
                alerts for any location."
            }
        }
}
```

Table C.1: Environment variables required per MCP server. The remaining eight servers (`playwright`, `arxiv`, `fetch`, `memory`, `filesystem`, `wikipedia`, `duckduckgo`, `weather`) require no credentials.

Server	**Environment variable(s)**	**Free tier?**
tavily	`TAVILY_API_KEY`	1,000 searches/month
e2b	`E2B_API_KEY`	Yes (sandbox-hours limited)
wolfram	`WOLFRAM_APP_ID`, `WOLFRAM_API_KEY`	2,000 queries/month
thinking	`THINKING_MODEL`, `ANTHROPIC_API_KEY` or `OPENAI_API_KEY`	Pay-per-token
clinicaltrials	`FASTMCP_SHOW_BANNER` (cosmetic only)	Yes

The two Python-based servers (`e2b` and `wolfram`, and also `thinking`) are implemented as part of the `research_agent.mcp` package and launched via the agent's own virtual environment interpreter, rather than via `npx`/`uvx`, so that they share the project's dependency set (in particular, the `mcp` SDK version and any shared utility modules such as `_invoke_structured()`, Section 3.1.7). All other servers are third-party packages launched on demand via `npx` (Node.js package runner) or `uvx` (Python package runner from the `uv` toolchain), which download and cache the server package on first use and require no manual installation step.

Appendix D

Additional Results Data

This appendix provides supplementary breakdowns of the headline results reported in Chapter 4, at a level of detail that would have disrupted the narrative flow of the main chapter. All figures in this appendix were computed from the same evaluation runs as Section 4.3, Section 4.4, Section 4.5, and Section 4.2, using the adapters and judge methodology described in Section 3.3.3 and Section 3.3.4.

D.1 SimpleQA: Per-Category Accuracy

Table D.1 breaks down the SimpleQA result of 62.0% (31/50, Section 4.2) by the five informal topic categories used when sampling the 50-question subset from the original SimpleQA release [19]. The agent performs best on geography and places, where Wikipedia and Tavily searches converge quickly on a single authoritative page, and worst on science and technology questions, several of which required disambiguating between similarly named entities (e.g. two chemical compounds with near-identical names): a failure mode also discussed in Section 4.7.

Table D.1: SimpleQA (n=50): accuracy by question category.

Category	**Questions**	**Correct**	**Accuracy (%)**
People & biography	15	10	66.7
Geography & places	10	7	70.0
History & events	10	6	60.0
Science & technology	8	4	50.0
Arts & miscellaneous	7	4	57.1
Total	**50**	**31**	**62.0**

D.2 FRAMES: Accuracy by Reasoning-Hop Count

FRAMES questions are tagged with the number of reasoning hops (i.e. chained facts) required to reach the answer [20]. Table D.2 shows the binary pass/fail accuracy on

the 40-question subset, grouped by hop count. The headline figure of 53.0% reported in Section 4.2 includes partial credit awarded by the FRAMES judging rubric for multi-hop answers that correctly resolve all but one hop; the strict binary breakdown below (21/40 = 52.5%) is therefore marginally lower than the partial-credit figure, but the trend, accuracy degrading with hop count, is the same under both scoring schemes.

Table D.2: FRAMES (n=40): binary accuracy by number of reasoning hops.

Hop count	**Questions**	**Correct**	**Accuracy (%)**
2 hops	18	11	61.1
3 hops	14	7	50.0
4+ hops	8	3	37.5
Total	**40**	**21**	**52.5**

The degradation with hop count is consistent with the error-categorisation discussion of Section 4.7: each additional hop is an additional opportunity for the assessor to accept a plausible but subtly incorrect intermediate fact, which then propagates uncorrected into the final synthesis.

D.3 DRACO: Sub-Criterion Score Breakdown

The DRACO score of 61.45% reported in Section 4.4 is itself an average of three LLM-judged sub-criteria: Accuracy, Completeness, and Citation Quality; each scored on a 0–100 scale per the rubric of Listing 3.5. Table D.3 reports the mean sub-criterion scores across the 50-question DRACO subset.

Table D.3: DRACO (n=50): mean sub-criterion scores for the research agent.

Criterion	**Mean score (%)**
Accuracy	66.80
Completeness	59.20
Citation quality	58.35
Overall (mean of three)	**61.45**

Citation quality is the weakest of the three sub-criteria. Manual inspection of low-scoring transcripts showed that the synthesizer frequently cites the URL of the page returned by

the *first* tool call that mentioned a fact, even when a later tool call (e.g. a Wikipedia lookup triggered by the assessor's "missing" list) returned a more authoritative source for the same fact. This is recorded as a candidate improvement in Section 5.3.

D.4 BrowseComp: Per-Task Outcomes

BrowseComp [21] questions are deliberately adversarial: each requires locating a specific, hard-to-find fact, typically by following a chain of indirect references across multiple web pages. Table D.4 lists the outcome for each of the twelve sampled tasks. The single success (task 7) was a question whose answer happened to appear verbatim in a Wikipedia infobox reachable directly from a Tavily search; the eleven failures span a mix of timeouts (Playwright sessions exceeding the per-tool-call budget of Section 3.2.3), incorrect disambiguation, and cases where the agent returned a confident but wrong answer rather than continuing to search, which is the "premature termination" failure mode discussed in Section 4.7.

Table D.4: BrowseComp (n=12): per-task outcome.

Task	**Failure mode (if any)**	**Outcome**
1	Playwright timeout following reference chain	✗
2	Incorrect disambiguation of two similarly named entities	✗
3	Premature termination (confident wrong answer)	✗
4	Source page required login / paywall	✗
5	Playwright timeout following reference chain	✗
6	Incorrect disambiguation of two similarly named entities	✗
7	None	✓
8	Premature termination (confident wrong answer)	✗
9	Tool result truncated before key fact	✗
10	Premature termination (confident wrong answer)	✗
11	Source page required login / paywall	✗
12	Playwright timeout following reference chain	✗
Total accuracy		**8.3% (1/12)**

D.5 Latency Distribution Percentiles

Table 4.5 (Section 4.9) reports mean per-question latency for each benchmark. Table D.5 supplements this with the 50th, 90th, and 99th percentile latencies for the GAIA (n=139, poster configuration) run, illustrating the long tail caused by questions that trigger the maximum iteration cap (Section 3.2.3) before routing to the synthesizer.

Table D.5: GAIA (n=139, poster configuration): latency percentiles per question.

Percentile	**p50**	**p90**	**p99**
Latency (seconds)	38.2	96.7	184.3
Iterations	3	6	8 (cap)

The p99 latency corresponds to questions that reach the iteration cap of eight planner/dispatcher/executor/assessor cycles (Section 3.2.3) without the assessor ever reporting `is_complete=true`; in these cases the planner is forced to set `decision="done"` on the final permitted iteration so that the synthesizer can still attempt an answer from whatever partial evidence has been gathered, rather than the run failing outright. This safety valve is discussed further in Section 5.2.