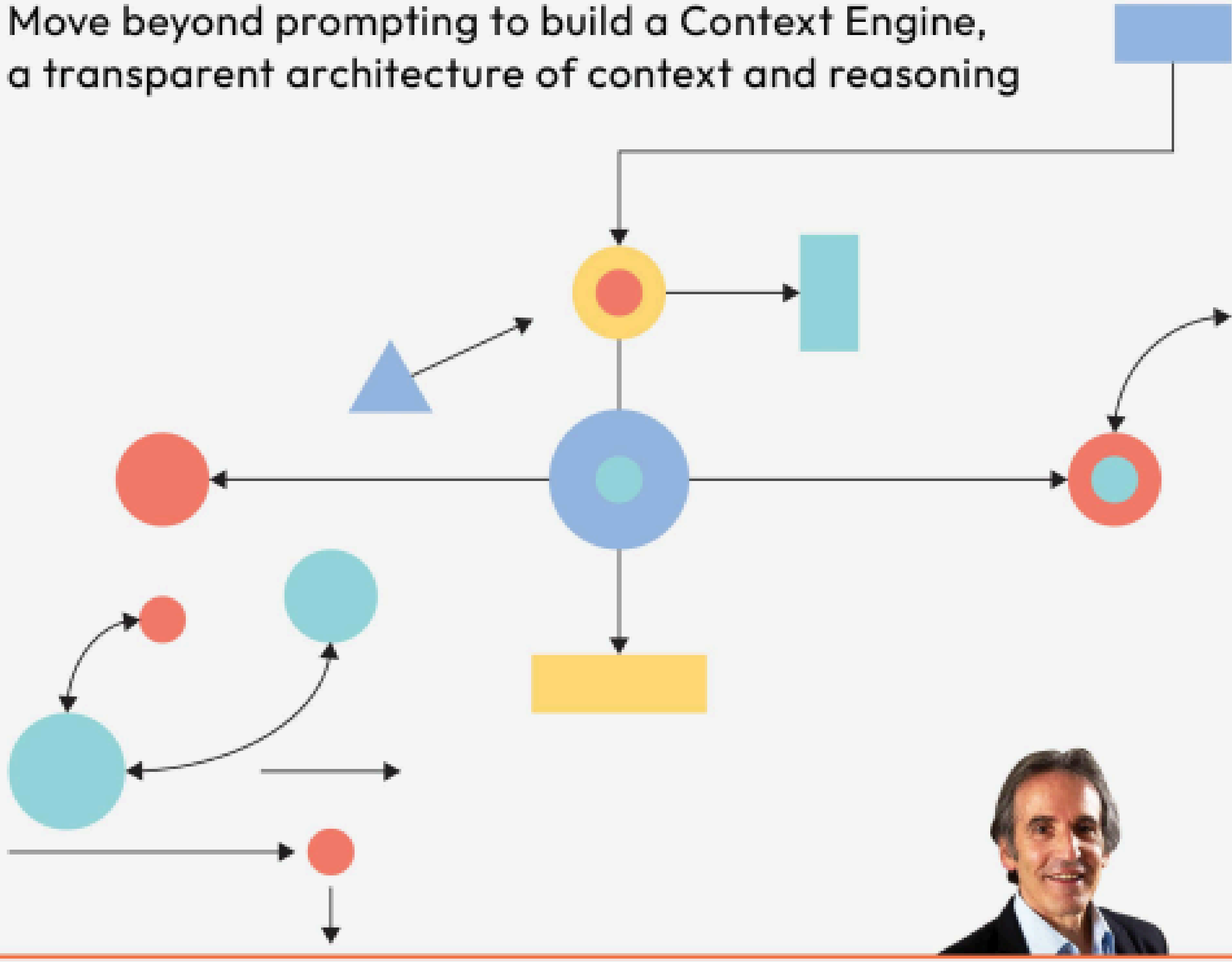
EXPERT INSIGHT

Context Engineering for Multi-Agent Systems

Move beyond prompting to build a Context Engine, a transparent architecture of context and reasoning



Denis Rothman

Context Engineering for Multi Agent Systems

Move beyond prompting to build a Context Engine, a transparent architecture of context and reasoning

Denis Rothman

Context Engineering for Multi-Agent Systems

Portfolio Director: Gebin George
Relationship Lead: Tanya D'cruz
Project Manager: Prajakta Naik
Content Engineer: Tanya D'cruz
Technical Editor: Rahul Limbachiya
Copy Editor: Safis Editing
Indexer: Pratik Shirodkar
Proofreader: Tanya D’cruz
Production Designer: Shantanu Zagade

First Published: November 2025
Production reference: 1141125

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-80669-005-3
`www.packtpub.com`

I would like to dedicate this book to my family and friends who are my source of happiness.

– Denis Rothman

Contributors

About the author

Denis Rothman is a graduate of Sorbonne University and Paris-Diderot University. He pioneered one of the first patented word2matrix embedding algorithms and AI-powered conversational agents. Early in his career, Denis developed a cognitive NLP chatbot adopted by Moët & Chandon and other global brands as an automated language trainer. He later created an AI resource optimizer for Airbus (formerly Aérospatiale), implemented by IBM and apparel industry leaders. His Advanced Planning and Scheduling (APS) solution, used worldwide, shaped supply chain intelligence across sectors.

Through his books, Denis shares his experience of innovation with a global community of thinkers, builders, and learners committed to shaping AI with purpose.

About the reviewers

Paras Patel is a Platform Engineering Leader with over 14 years of Silicon Valley experience spanning cloud computing, DevOps, and SRE. At Rakuten, he leads innovation at scale, building and managing Kubernetes platforms that serve millions of users while driving AI transformation initiatives across the organization. Known for his deep expertise in observability, Kafka, Redis, and Elasticsearch, Paras focuses on designing production-ready AI infrastructure that bridges the gap between cutting-edge research and enterprise reliability. Beyond his engineering leadership, he mentors teams on resilience engineering and speaks on AI-driven DevOps. He holds certifications including CKA, HashiCorp Terraform, and multiple AI credentials.

Heer Samir Patel is currently pursuing a Master of Science in Artificial Intelligence at Boston University, specializing in Generative and Multi-Agentic AI technologies. Driven by a deep curiosity and a passion for continuous learning, she is always eager to invent and simplify. Heer continually pushes herself to look beyond the surface and create solutions that are both elegant and efficient.

Table of Contents

[redacted]

[redacted]

[redacted]

[redacted]

Chapter 9: Architecting for Brand and Agility: The Strategic Marketing Engine 271

[redacted]

A[redacted]

[redacted]

Preface

Generative AI is powerful but unpredictable. This book shows you how to turn that unpredictability into reliability by moving beyond prompt tinkering and thinking like an architect. At its heart lies the emerging discipline of context engineering, the practice of structuring, managing, and governing the information that large language models use to reason, decide, and generate. You'll explore this concept through the Context Engine, a transparent, glass-box system built on multi-agent collaboration and retrieval. You'll learn how to strengthen and deploy this architecture step by step, transforming raw model outputs into verifiable and policy-aligned intelligence.

Across the chapters, you'll build the Context Engine from first principles, starting with context design and semantic blueprints, then orchestrating specialized agents through the Model Context Protocol (MCP). As the engine matures, you'll integrate memory and high-fidelity retrieval with source citations, introduce safeguards against data poisoning and prompt injection, and add moderation layers to ensure every response adheres to defined goals and compliance standards. You'll then harden your architecture for real-world performance, reusing it across domains such as legal compliance and strategic marketing to prove its flexibility and domain independence.

By the end, you'll have a blueprint for production-ready, enterprise-grade AI. The Context Engine becomes your bridge between experimentation and reliability, between black-box prompting and glass-box engineering. It's a guide to designing AI systems that step beyond generating content and start understanding and operating in context.

Who this book is for

This book is for AI engineers, software developers, system architects, and data scientists who want to move beyond ad hoc prompting and learn how to design structured, transparent, and context-aware AI systems. It will also appeal to ML engineers and solutions architects with some familiarity with LLMs who are eager to understand how to orchestrate agents, integrate memory and retrieval, and enforce safeguards. By the end, readers will have the skills to engineer an adaptable, verifiable architecture they can repurpose across domains and deploy with confidence.

What this book covers

Chapter 1, From Prompts to Context: Building the Semantic Blueprint, introduces the principles of context engineering and demonstrates how structured context, semantic blueprints, and agent orchestration transform generative AI from prompt-based unpredictability into reliable, goal driven systems. It establishes the foundation for building a transparent multi-agent architecture, integrating memory, retrieval, and safeguards, that will evolve throughout the book into a production-ready Context Engine.

Chapter 2, Building a Multi-Agent System with MCP, expands context engineering from single agent control to multi-agent collaboration, showing how specialized AI agents can coordinate through the Model Context Protocol (MCP) to complete complex, multi-step workflows. It demonstrates how orchestrators, agents, and validators communicate via structured contexts to ensure reliability, error recovery, and factual accuracy in a robust, production-ready multi agent system.

Chapter 3, Building the Context-Aware Multi-Agent System, extends the architecture into a dual RAG framework that separates factual retrieval from procedural instruction, enabling agents to reason and write using both knowledge and style-based context. It introduces the Context Librarian and Researcher agents, orchestrated through MCP, to dynamically retrieve semantic blueprints and factual data—laying the groundwork for adaptive, context-aware generation within the evolving Context Engine.

Chapter 4, Assembling the Context Engine, consolidates the principles of context engineering into a complete, autonomous architecture that plans, executes, and reflects using specialized agents. It introduces the Planner, Executor, and Tracer modules, integrated through the Model Context Protocol, to create a transparent reasoning system that transforms abstract goals into context-driven outputs.

Chapter 5, *Hardening the Context Engine*, transforms the experimental Context Engine into a production-ready system by applying professional engineering principles such as modularization, dependency injection, and structured logging. It details how to refactor the prototype into independent, testable components—helpers, agents, registry, and engine—creating architecture ready for real-world deployment.

Chapter 6, *Building the Summarizer Agent for Context Reduction*, introduces proactive context management through the Summarizer agent, allowing the Context Engine to dynamically compress and optimize information passed between agents. It focuses on improving efficiency and reasoning stability by reducing token overhead and ensuring context continuity in the multi-agent workflow as a core pillar of scalable context engineering.

Chapter 7, *High-Fidelity RAG and Defense: The NASA-Inspired Research Assistant*, upgrades the Context Engine with verifiability and security, introducing a high-fidelity retrieval pipeline that attaches source metadata to every fact and enables citation-backed reasoning. It also implements a defense layer against data poisoning and prompt injection through input sanitization, establishing enterprise-grade trust, traceability, and backward compatibility within the context-engineered multi-agent architecture.

Chapter 8, *Architecting for Reality: Moderation, Latency, and Policy-Driven AI*, transitions the Context Engine from a controlled prototype into an enterprise-ready system by introducing two-stage moderation, latency budgeting, and policy-based safeguards for real-world deployment. It demonstrates how to integrate moderation gates, policy enforcement, and human-in-the-loop governance through a Legal Compliance Assistant use case, showing that reliable context engineering requires not just code-level safeguards but organizational design and policy alignment.

Chapter 9, *Architecting for Brand and Agility: The Strategic Marketing Engine*, demonstrates the Context Engine's domain independence by re-tasking the same multi-agent architecture from a legal compliance assistant into a strategic marketing engine without changing its core logic. It guides readers through building a marketing knowledge base, enforcing brand consistency, performing competitive analysis, and synthesizing persuasive content, proving that context engineering enables modular reuse and cross-domain adaptability between AI and business objective.

Chapter 10, *The Blueprint for Production-Ready AI*, provides the framework for deploying the glass-box Context Engine as a scalable enterprise service, detailing how to productionize it through containerization, orchestration, environment configuration, asynchronous execution, and observability. It consolidates cost management, verifiable retrieval, data sanitization, and moderation into a cohesive operational blueprint—then demonstrates how these engineering patterns translate into trust, governance, and long-term business value through measurable ROI and compliance assurance.

Appendix, *Context Engine Reference Guide*, serves as the reader's technical companion, consolidating all architectural concepts, agents, and workflows introduced throughout the book into a single, practical implementation guide. It offers a detailed reference for building and maintaining the Context Engine as a enterprise-ready framework.

To get the most out of this book

If you're new to LLMs, *Chapters 1* and *2* will build the necessary conceptual and practical foundation before the book moves into more advanced architecture work.

To make sure everything runs smoothly, set up your development environment before diving into the code. Each hands-on chapter uses reproducible, Python-based environments. The examples were developed primarily in Google Colab and VS Code, ensuring flexibility across platforms.

Before you begin, ensure you have:

- Python 3.10 or later
- Google Colab or a local environment configured with openai, pinecone-client, tiktoken, tenacity, and fastapi
- A GitHub or local directory structure to store project files (helpers.py, agents.py, registry.py, engine.py, and notebook files for each chapter).
- API keys for:
    - OpenAI (for model access and moderation)
    - Pinecone (for vector database storage and retrieval)
    - *Optional:* Google Cloud or AWS (for deployment-related sections in *Chapter 10*).

Chapters 5 onward, we begin using modular components that depend on earlier notebooks. Make sure your environment is correctly configured before proceeding, as setup steps might not be repeated in detail in every chapter.

Your system doesn't need to be high-end, but meeting these hardware baselines will help you avoid performance issues:

- **Minimum:** Dual-core CPU, 8 GB RAM (for local runs).
- **Recommended:** A system with at least 16 GB RAM or a cloud runtime (Google Colab Pro or equivalent).
- **GPU acceleration** is optional but useful for embedding generation and token-intensive tasks.

If you're running locally, be mindful of token and API costs when experimenting with large contexts. *Chapter 6* introduces the Summarizer agent specifically to help manage these costs to an extent.

Before you start building, create a dedicated workspace to keep your helper scripts and notebooks organized. Familiarize yourself with retrieval workflows (RAG) and agent orchestration (MCP), which will be the foundation for almost all chapters. Review *Appendix:*

Context Engine Reference Guide when needed; it consolidates code structures and component explanations from every chapter for quick lookup.

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/Denis2054/Context-Engineering-for-Multi-Agent-Systems`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here:`https://packt.link/gbp/9781806690053`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "The `count_tokens` utility provides the measurement, and the Summarizer agent provides the action. "

A block of code is set as follows:

```
class AgentRegistry:
    def __init__(self):
        self.registry = {
            "Librarian": agents.agent_context_librarian,
            "Researcher": agents.agent_researcher,
            "Writer": agents.agent_writer,
            # --- NEW: Add the Summarizer Agent ---
            "Summarizer": agents.agent_summarizer,
 }
```

Any command-line input or output is written as follows:

```
Prepared 3 context blueprints.
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: " Both data types are processed by the **embedding model**."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book or have any general feedback, please email us at `customercare@packt.com` and mention the book's title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packt.com/submit-errata`, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packt.com/`.

Join our Discord and Reddit Space

You're not the only one navigating fragmented tools, constant updates, and unclear best practices. Join a growing community of professionals exchanging insights that don't make it into documentation.

Join our Discord at `https://packt.link/ z8ivB` or scan the QR code below:	Follow us on Reddit at `https://packt.link/0rExL` or scan the QR code below:

Share your thoughts

Once you've read *Context Engineering for Multi-Agent Systems*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.

https://packt.link/r/1806690047

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Free Benefits with Your Book

This book comes with free benefits to support your learning. Activate them now for instant access (see the "*How to Unlock*" section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:

PDF and ePub Copies Next-Gen Web-Based Reader

Access a DRM-free PDF copy of this book to read anywhere, on any device.

Use a DRM-free ePub version with your favorite e-reader.

Multi-device progress sync: Pick up where you left off, on any device.

Highlighting and notetaking: Capture ideas and turn reading into lasting knowledge.

Bookmarking: Save and revisit key sections whenever you need them.

- Dark mode: Reduce eye strain by switching to dark or sepia themes.

How to Unlock

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

UNLOCK NOW

Note: Keep your invoice handy. Purchases made directly from Packt don't require one

1

From Prompts to Context: Building the Semantic Blueprint

Context engineering is the discipline of transforming generative AI from an unpredictable collaborator into a fully controlled creative partner. Where a prompt often opens a door to random chance, a context provides a structured blueprint for a predictable outcome. It is a fundamental shift from asking an LLM to continue a sequence to engineering a closed environment where it executes a precise plan. This evolution takes the interaction beyond simple requests into the realm of directed creation, telling the model not just what to do, but how to think within the boundaries you define.

For too long, we've treated generative AI like an oracle by sending prompts into the void and hoping for a coherent reply. We've praised its moments of brilliance and overlooked its inconsistencies, accepting unpredictability as part of the experience. But this is the art of *asking*, not the art of creating. This chapter is not about asking better questions; it is about providing better plans and *telling* the LLM what to do.

Our journey begins with a hands-on demonstration that progresses through five levels of contextual complexity, showing how each additional layer transforms output from random guesses into structured, goal-driven responses. We then move from linear sequences of words to multidimensional structures of meaning by introducing **Semantic Role Labeling** (**SRL**), a linguistic technique that reveals who did what to whom, when, and why. With SRL as our foundation, we build a Python program that visualizes these structures as semantic blueprints. Finally, we synthesize these skills in a complete meeting analysis use case, where we will introduce **context chaining** and demonstrate how multi-step workflows can turn a raw transcript into insights, decisions, and professional actions.

By the end of this chapter, you will no longer be searching for answers in a digital wilderness. You will be the architect of that wilderness, capable of designing the very landscape of the AI model's thought and directing it toward any destination you choose.

This chapter covers the following topics:

- Progressing through five levels of context engineering to build a semantic blueprint
- Transitioning from linear text to multidimensional semantic structures through SRL
- Building a Python program to parse and structure text using SRL
- Applying context chaining as a method for step-by-step, controlled reasoning
- Using a complete meeting analysis use case to turn raw transcripts into a professional email

Understanding context engineering

Context engineering is the art and science of controlling and directing the informational world that **Large Language Model** (**LLM**) has learned. It will transform your role from a questioner into a confident director. It is the difference between handing an actor a single line and giving them a full script, complete with character motivations and stage directions. You are no longer asking for a performance; you are designing it and telling the LLM what to do.

The best way to understand context engineering is to experience it. We will skip the theory and begin with a hands-on demonstration. This journey is designed to let you *feel* the AI's output transform and to witness its raw, statistical guesses evolve into nuanced, reliable, and perfectly aligned responses. Together, we will progress through five levels of complexity, as shown in *Figure 1.1*. We will go from a simple prompt to a fully architected semantic blueprint using three major copilots (Google, Microsoft, and OpenAI):

- **Level 1: The basic prompt (zero context)**. This is a simple instruction with no background. The AI guesses based on training data, producing generic or clichéd outputs.

- **Level 2: The better context (linear context)**. This is a small step forward. Adding a linear thread improves factual accuracy compared to zero context, but the model still lacks style, purpose, or direction.

- **Level 3: The good context (goal-oriented context)**. This would be the first *true* context level. By giving the model a clear goal, its responses become intentional and aligned. This is the first acceptable milestone in context engineering.

- **Level 4: The advanced context (role-based context)**. This is more structured than goal-only prompts. By assigning explicit roles, the model can follow conflict and motivation, producing narratively intelligent responses.

- **Level 5: The semantic blueprint**. This is the ultimate engineered ontext. A precise, unambiguous plan using semantic roles transforms creativity into a reliable, repeatable engineering process.

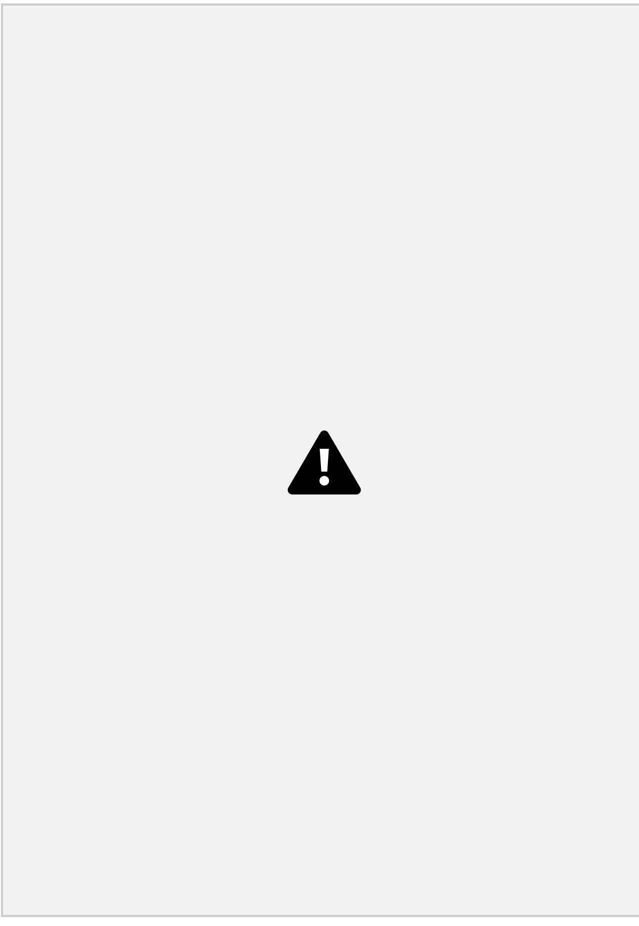


Figure 1.1: Evolving from prompts to contexts

> The responses you obtain may vary as each LLM copilot is updated, or between two requests, due to the stochastic nature of generative AI. However, directed, well designed contexts will always provide better outputs than improvised, undirected prompts.

Let's begin with a basic prompt.

Level 1: The basic prompt (zero context)

A basic prompt is a simple, direct instruction with no background information. The AI relies entirely on the most common statistical patterns from its training data, as we will see in this example. Our input is as follows:

```
Complete this sentence: "The cat played with the...
```

Gemini 2.5's response is this:

```
 ... ball of yarn.
Here are a few other ways to complete the sentence:
 ... toy mouse.
 ... string.
 ... cardboard box.
 ... laser pointer dot.
 ... other kitten.
```

As you can see, the output is a cliché. Without context, the AI defaults to the most predictable and uninspired completion.

Microsoft Copilot's response is this:

```
The cat played with the ball of yarn until it unraveled across the floor."
...crumpled receipt it found under the couch, batting it around like it was the
most precious treasure in the kingdom of dust bunnies.
```

The output, in this case, is creative, but if we wanted a precise response, we would consider this response a hallucination. Without context, the AI defaults to the most predictable completion it has learned statistically.

OpenAI GPT-5's response is this:

```
The cat played with the ball of yarn until it unraveled across the floor.
```

The output is a cliché again. Without context, the AI defaults to the most predictable and uninspired completion.

All three models could have stated:

```
I'm sorry, but without more context, I can't complete this sentence correctly.
```

We can see that basic prompts will lead to basic, imprecise responses or even hallucinations. The goal here is not to try to find workarounds with model hyperparameters, such as temperature, but to engineer better contexts. Let's move on to the next level.

Level 2: The better context (linear context)

Here, we add a simple preceding sentence as added context, which will provide a linear thread of information, which improves factual accuracy but doesn't guide the AI's style or purpose. Our input will be as follows:

```
Complete this sentence: The ball fell on the floor and it rolled for a few feet.
The cat then played with the...
```

Gemini 2.5's response is this:

```
The ball fell on the floor and it rolled for a few feet. The cat then played with
the ball.
```

Notice that the response is factual and correct, but lacks creativity, which is normal without more context.

Microsoft Copilot's response is this:

```
...ball, pouncing on it with swift little swats and chasing it across the room
like it was the most thrilling prey in the world.
```

The response, in this case, is factually correct within our micro-story. It's also creative, but it didn't really know what we wanted it to generate.

OpenAI GPT-5's response is this:

```
The ball fell on the floor and it rolled for a few feet. The cat then played with
the ball, batting it back and forth with its paws.
```

Here, too, the response is factually correct and also creative, but again, it didn't know what to generate and continued randomly.

The responses are not particularly convincing beyond producing basic probabilities, but they are relatively accurate. This might work for simple tasks, but not when we need the generative AI model to do specific tasks. Let's move on, then, to engineering a context and setting a goal.

Level 3: The good context (goal-oriented context)

This level is the first true step in context engineering. We will provide a clear goal and richer descriptive details, which enable the LLM to co-create a scene that aligns with our intended purpose. Our input is as follows:

```
Goal: Write a short, suspenseful scene for a children's story.
---
Complete this sentence:
"The ancient grandfather clock chimed midnight, its shadow stretching like a claw.
A red ball, seemingly moving on its own, rolled out from under the dusty armchair.
The black cat, Onyx, crouched low, his eyes gleaming. He then played with the..."
```

Here’s Gemini 2.5's response:

```
The ancient grandfather clock chimed midnight, its shadow stretching like a claw.
A red ball, seemingly moving on its own, rolled out from under the dusty armchair.
The black cat, Onyx, crouched low, his eyes gleaming. He then played with the...
tiny, shimmering thread tied to the ball.
He pounced, but the thread danced away, pulling the red ball with it. The ball
didn't just roll; it skittered across the floorboards as if it were alive, a
little red mouse being tugged towards the heavy velvet curtains by the window.
Onyx let out a low growl, a sound he never made when playing. The ball slipped
silently behind the thick, dark fabric. The thread hung in the air for a second,
taut and humming, before it too was yanked into the darkness. A tiny whisper, no
louder than the rustle of a leaf, echoed from behind the curtain. "Mine."
```

Observe that the response is more directed. We're beginning to get what we asked for. Remember, LLMs, like humans, need context to provide more effective responses. Humans and LLMs that have well-designed longer context windows provide better results.

Microsoft Copilot's response is this:

```
...ball for only a moment before freezing. The ball had stopped—dead still—but
Onyx's fur bristled. From behind the armchair came a soft creak, like a whispering
floorboard. Onyx backed away, tail puffed, eyes locked on the darkness. Then, a
tiny giggle echoed through the room.
Not a human giggle.
The curtains fluttered though the windows were shut. Onyx hissed, and the ball
rolled again—this time toward the hallway, leaving a faint trail of glowing red
dust. The clock ticked louder now, each second like a drumbeat.
And just before the hallway swallowed the ball, a pale hand reached out from the
shadows and snatched it.
Onyx bolted.
But the giggle remained.
```

Notice that in this case, Microsoft Copilot starts exactly at the end of our input. Additionally, we can observe that we are transitioning from a prompt (unstructured context) to a directed request. We are the directors.

This is OpenAI GPT-5's response:

```
...ball, batting it across the wooden floor. But instead of stopping, the ball
bounced back toward him—as if pushed by an invisible hand. Onyx froze, his tail
puffing up, while the clock ticked louder than ever, each tick echoing through the
empty room.
```

In this case, too, the response is quite nice and starts exactly where we asked it to. Overall, these responses are more creative and intentional than those in *Level 2*. Yet, they remain loosely guided. To achieve even greater precision, we need to introduce structure into the context itself.

Level 4: The advanced context (role-based context)

Rather than leaving the AI to infer the dramatic structure, we now define the roles of each participant. By providing the LLM with characters, objects, and relationships, we give it the scaffolding of conflict and motivation. This makes the response narratively intelligent. Our input will be as follows:

```
Goal: Write one suspenseful sentence to continue a children's story.
---
```

```
BACKGROUND:
- Protagonist: Onyx, a brave black cat.
```



```
- Antagonist: An ancient, haunted grandfather clock.
- Catalyst Object: A mysterious red ball.
- Setting: A dark, dusty room at midnight.
- Preceding Action: The clock chimed, and the ball rolled out.
---
Continue the story from this point: "The black cat, Onyx, crouched low... he then
played with the...
```

Here is Gemini 2.5's response:

```
...mysterious red ball, batting it gently until a low groan echoed from the clock
and its tall, glass door swung open into the darkness within.
```

This time, we didn't simply *ask*. We *told* the LLM exactly what we wanted, and the result is acceptable and more directed.

Microsoft Copilot's response is this:

```
...ball until it pulsed with a strange glow, and the clock's hands spun wildly
backward, unleashing a chilling whisper that slithered across the floor toward
him.
```

The LLM, in this case, is attentive to the director we have become. The story follows our structured instructions.

Here is OpenAI GPT-5's response:

```
The black cat, Onyx, crouched low… he then played with the red ball—only to freeze
as the grandfather clock's pendulum creaked on its own, swinging faster and faster
in the midnight silence.
```

The response, in this case, follows the structure of our context. With that, we have gone from undirected linear prompts to structured contexts. Instead of *asking*, we are *telling*—and the model responds with narrative discipline. The next step is to formalize this process into a **semantic blueprint**.

Level 5: The semantic blueprint

This level represents the full realization of context architecture. Here, we provide the model with a precise and unambiguous plan using a structured format. The creative act becomes a reliable engineering process, guided by semantic roles: the scene goal, the participants, their descriptions, the specific action to complete, the agent (who performs the action), and the patient (who is most affected by the action).

Our input will be as follows:

```
TASK: Generate a single, suspenseful sentence.
---
SEMANTIC BLUEPRINT:
{
  "scene_goal": "Increase tension by showing defiance",
```

```
  "participants": [
 { "name": "Onyx", "role": "Agent", "description": "black cat" },  { "name": "Red Ball", "role": "Patient", "description": "mysterious" },  { "name": "Grandfather Clock", "role": "Source_of_Threat", "description": "ancient, looming" }
 ],
"action_to_complete": {
            "predicate": "play with",
 "agent": "Onyx",
 "patient": "Red Ball"
 }
}
---
SENTENCE TO COMPLETE: "He then played with the..."
```

Gemini 2.5's response is this:

```
He then played with the red ball, batting it with deliberate slowness directly
under the shadow of the looming grandfather clock as its heavy pendulum began the
first inexorable swing toward midnight.
```

The output follows the blueprint exactly. We are no longer improvising; the model is executing a defined plan.

Here is Microsoft Copilot's response:

```
He then played with the red ball, his shadow flickering defiantly beneath the
looming tick of the grandfather clock, as if daring time itself to strike.
```

The story now has a suspenseful tone that effectively captures our instructions.

OpenAI GPT-5's response is this:

```
He then played with the Red Ball, its echoing bounce defying the relentless tick
of the Grandfather Clock that seemed to grow louder with every daring swipe of
Onyx's paw.
```

The suspenseful tone mirrors the goal. The structure we provided carries through into the narrative. We're obtaining what we told the LLM to do.

At this stage, we are no longer spectators of the model's improvisation. We are directors, and the LLM is the actor working from our script. But how does a semantic blueprint such as the one in *Level 5* work? To answer that, we turn to **Semantic Role Labelling** (**SRL**), a method that will take us on our first journey from linear sequences of language to multidimensional semantic structures.

SRL: from linear sequences to semantic structures

Our journey through the five levels of context engineering culminated in the semantic blueprint, a method that gives an LLM a structured plan instead of a loose request. But how do we construct such a blueprint from the linear, often ambiguous flow of human language? To do so, we must stop seeing sentences as strings of words and start viewing them as *structures of meaning*.

The key to upskilling our perception of linear sequences into *structures* is SRL, a powerful linguistic technique initially formalized by Lucien Tesnières and later by Charles J. Fillmore that takes language representations from linear sequences to semantic structures. You can find links to their work in the *References* section. SRL emerged over the years following the work of these great pioneers, with the goal of deconstructing a linear sentence to answer the most fundamental question: *Who did what to whom, when, where, and why?* It moves beyond simple grammar to identify the functional role each component plays in the overall action. Instead of linear sequences of text, we get a hierarchical map of meaning centered around an action or predicate.

> For readers who want to see how these foundational ideas, such as SRL and semantic blueprints, fit into the larger context engine architecture, the *Appendix* provides a concise overview of how the full system ties these concepts together.

Consider a simple sentence:

```
Sarah pitched the new project to the board in the morning.
```

An LLM interprets a sequence of words as a chain of tokens. A context engineer, using SRL, sees more: a *stemma*, or graph, that maps each word to its semantic role. The central action is "*pitched*", while every other component is assigned a role in relation to that action.

By labeling these roles, we will do the following:

- Reconstruct the multidimensional semantic structure of an otherwise linear string of words
- Define a semantic blueprint that an LLM can follow, as demonstrated in the *Level 5* example earlier

This process is the foundational skill of advanced context engineering. To understand the SRL process, let's build a Python program to put this powerful theory into practice.

Building an SRL notebook in Python

The purpose of this Python script is to take the essential parts of a sentence and turn them into a visual diagram of meaning. Rather than leaving structure hidden in text, the program draws a picture of the relationships between words and roles. *Figure 1.2* illustrates the overall process:

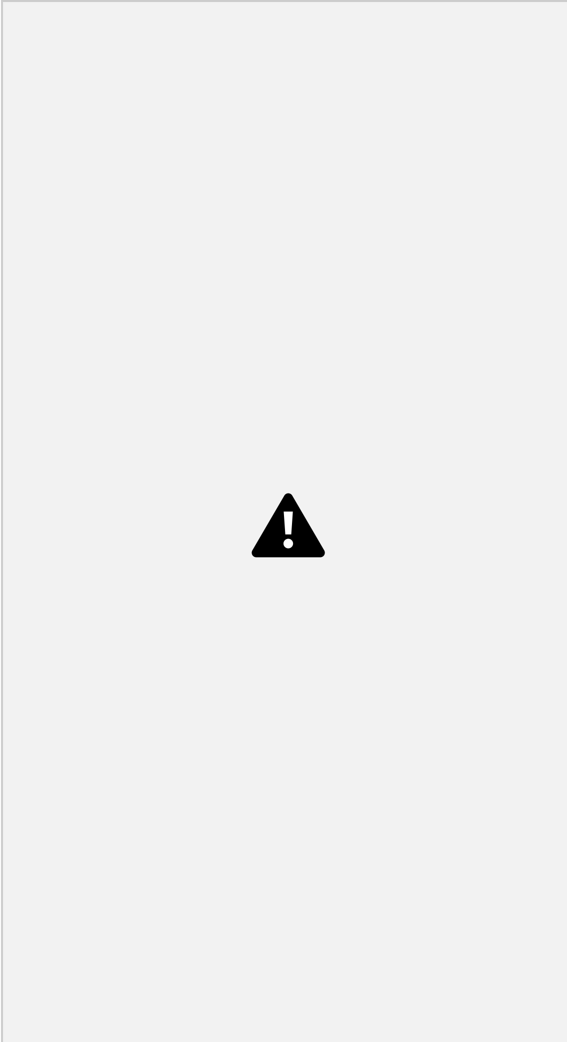


Figure 1.2: Flowchart of the SRL program

Let's walk through each stage of the process:

1. **User Input**: The journey begins when you call the main function, `visualize_srl()`. Here, you provide the building blocks of the sentence, such as its verb (predicate), the agent (the entity performing the action), the patient (the entity receiving or affected by the action), and other semantic roles represented as textual arguments.

2. **Data Structuring:** The main function organizes these components into a Python dictionary. Each entry is assigned the proper SRL label, so what began as a loose list of words becomes a structured map of roles.

3. **The Plotting Engine:** Once the dictionary is ready, it is passed to an internal helper function, `_plot_stemma`. This function does one thing only: draw.

4. **Canvas Setup:** The plotting engine creates a blank canvas using Matplotlib,

preparing the stage for the diagram.

5. **Dynamic Positioning:** The function calculates where to place each role node so the layout remains clear and balanced, regardless of how many components are included.
6. **Drawing the Stemma (Graph)**: The engine then draws the core verb as the root node, adds each role s a child node, and connects them with labeled arrows. What was once a linear sentence is now a visual map of meaning.
7. **Final Display**: Finally, the function adds a title and displays the completed semantic blueprint.

At this point, we are ready to implement SRL in practice. Open `SRL.ipynb` in `Chapter01` of the GitHub repository.

> A word about the code itself: it has been written for clarity and teaching rather than for production. The focus is on showing the flow of information as directly as possible. For that reason, the notebook avoids heavy control structures or error handling that would distract from the experience. Those refinements can always be added later when building production systems. Also, to run the notebook locally, you will need to install spaCy, Matplotlib, and Graphviz, and then download the English model for spaCy using the `python -m spacy download en_core_web_sm` command.

Let's get building a semantic blueprint visualizer in Python that will generate our stemma, which is basically a graph with semantic nodes and edge visualizations. We will break down the script block by block, explaining the purpose of each section.

First, import the necessary tools. Our visualizer relies on `matplotlib` for plotting:

- `matplotlib.pyplot as plt`: The main plotting interface, imported with the conventional alias `plt` for ease of use
- `matplotlib.patches import FancyArrowPatch`: A utility for drawing clean, directed arrows that connect the verb to its roles

With these tools in place, we can now define the SRL visualizing function.

The main function: visualize_srl

The heart of our program is the main function, `visualize_srl()`. This is the primary point of interaction, represented by step **1** in *Figure 1.2*. As the user, you provide the core components of a sentence as arguments, and the function takes care of the rest, organizing the data and sending it to the plotting helper function.

The arguments correspond directly to the semantic roles we defined earlier. To keep the interface flexible, the function also accepts `**kwargs`, which allows you to pass in any number of optional modifiers (for example, temporal or location details) without complicating the function signature.

The sole purpose of `visualize_srl()` is to assemble these roles into a dictionary (`srl_roles`) and then hand them off, along with the predicate (verb), to the internal `_plot_stemma()` function for visualization:

```
def visualize_srl(verb, agent, patient, recipient=None, **kwargs):
    """
    Creates a semantic blueprint and visualizes it as a stemma.
    This is the main, user-facing function.
    """
    srl_roles = {
        "Agent (ARG0)": agent,
        "Patient (ARG1)": patient,
    }
    if recipient:
        srl_roles["Recipient (ARG2)"] = recipient
    # Add any extra modifier roles passed in kwargs
    for key, value in kwargs.items():
        # Format the key for display, e.g., "temporal" -> "Temporal (ARGM-TMP)"
        role_name = f"{key.capitalize()} (ARGM-{key[:3].upper()})"
        srl_roles[role_name] = value
    _plot_stemma(verb, srl_roles)
```

This function is deliberately simple: it shows the *flow of meaning* rather than the full complexity of a production-ready parser. By doing so, it keeps the focus on how semantic roles are organized into a blueprint that an LLM can use.

Before we start drawing the stemma itself, let's take a closer look at the semantic roles that drive this structure.

Defining the semantic roles

Let's review the roles defined in `visualize_srl`, which is one of the methods for performing context engineering. Let's return to our running example:

```
Sarah pitched the new project to the board in the morning.
```

At first glance, this looks like a simple sentence. To an LLM, it is nothing more than a sequence of tokens, one word after another. But for a context engineer, every part of the sentence plays a role in the unfolding action. To capture those roles, we use a set of labels, the primary tools for defining meaning and adding multidimensional structure to an otherwise linear sequence of words:

1. **Predicate (the verb)**: The predicate is the heart of the sentence, the central action or state of being. Every other role is defined in relation to it. In our example, `pitched` is the predicate.

2.

Agent (ARG0): The agent is the entity that performs the action, the "doer." In our example, `Sarah` is the agent.

3. **Patient (ARG1)**: The patient is the entity directly affected or acted upon by the verb. In our example, `the new project` is the patient.

4. **Recipient (ARG2)**: The recipient is the entity that receives the patient or the result of the action. In our example, `the board` is the recipient.

5. **Argument Modifiers (ARGM-)**: These are additional roles that provide context but are not central to the core action. They answer questions such as *when, where, why,* or *how*:

    - **Temporal (ARGM-TMP)**: Specifies when the action occurred. In our example, … `in the morning` is the Temporal modifier.
    - **Location (ARGM-LOC)**: Specifies where the action occurred. For example, …`in the conference room` is the location modifier.
    - **Manner (ARGM-MNR)**: Specifies how the action was performed. For example, …`with great confidence` is the manner modifier.

With these roles in place, a sentence is no longer just a linear/flat sequence of words. It becomes a structured map of meaning. And now, we are ready to plot the stemma.

The plotting engine: _plot_stemma and canvas setup

Our internal helper function is `_plot_stemma` (*step 3* in *Figure 1.2*). It does the heavy lifting of drawing the visualization. We keep it focused on one responsibility: **draw**.

First, we create a blank canvas (`fig`, `ax`) and set its dimensions. We turn off the axes because we are creating a diagram, not a traditional chart.

```
def _plot_stemma(verb, srl_roles):
    """Internal helper function to generate the stemma visualization."""
fig, ax = plt.subplots(figsize=(10, 6))
 ax.set_xlim(0, 10)
 ax.set_ylim(0, 10)
 ax.axis('off')
```

We can customize the styles as referenced in *Figure 1.2* as step 4. We can define the visual appearance of our nodes. We use Python dictionaries to store the styling rules (box style, colors) for the verb and the roles, making our code clean and easy to modify if we want to change the look later:

```
 verb_style = dict(boxstyle="round,pad=0.5", fc="lightblue", ec="b")
role_style = dict(boxstyle="round,pad=0.5", fc="lightgreen", ec="g")
```

With the canvas prepared and styles in place, we're ready for dynamic positioning, placing each role so the diagram remains clear and balanced regardless of how many components we include.

Dynamic positioning and drawing the stemma (graph)

We begin with the root node (the verb). Since the verb is the anchor of our structure, we give it a fixed position near the top center of the canvas. Using `ax.text()`, we draw it on the diagram and apply `verb_style` we defined earlier.

```
verb_pos = (5, 8.5)
 ax.text(verb_pos[0], verb_pos[1], verb, ha="center", va="center",
bbox=verb_style, fontsize=12)
```

We will now position the child nodes (the roles) of the root node. To make our diagram look clean no matter how many roles we have, we need to calculate their positions dynamically. We

count the number of roles and then create a list of evenly spaced `x_positions` along a horizontal line:

```
srl_items = list(srl_roles.items())
 num_roles = len(srl_items)
 x_positions = [10 * (i + 1) / (num_roles + 1)
        for i in range(num_roles)]
 y_position = 4.5
```

Let's add connections to our stemma graph by drawing the roles, arrows, and labels.

We will loop through each role in our `srl_roles` dictionary. In each iteration of the loop, we perform three actions:

1. **Draw the role node**: We use `ax.text()` again to draw the box for the current role at its calculated position
2. **Draw the connecting arrow**: We create `FancyArrowPatch` that connects the verb's position to the role's position and add it to our plot
3. **Draw the arrow label**: We calculate the midpoint of the arrow and place the role's name (e.g., `Agent (ARG0)`) there, so it's clear what the relationship is

The code will then manage the positioning:

```
   for i, (role, text) in enumerate(srl_items):
 child_pos = (x_positions[i], y_position)
 ax.text(child_pos[0], child_pos[1], text,
 ha="center", va="center",
 bbox=role_style, fontsize=10, wrap=True)

 arrow = FancyArrowPatch(
```

```
    verb_pos,
    child_pos,
    arrowstyle='->',
    mutation_scale=20,
    shrinkA=15,
    shrinkB=15,
    color='gray'
    )
    ax.add_patch(arrow)

    label_pos = (
```



```
    (verb_pos[0] + child_pos[0]) / 2,
    (verb_pos[1] + child_pos[1]) / 2 + 0.5
            )
    ax.text(label_pos[0], label_pos[1],
    role, ha="center", va="center",
    fontsize=9, color='black', bbox=dict(boxstyle="square,pad=0.1",
   fc="white", ec="none"))
```

Finally, we give our visualization a title and use `plt.show()` to display the finished stemma:

```
fig.suptitle("The Semantic Blueprint (Stemma Visualization)",
 fontsize=16)
 plt.show()
```

At this point, our main function is fully defined. We can now run examples to see our stemma visualizer in action.

Running SRL examples

By calling our `visualize_srl` function with different arguments, we can instantly transform linear sentences into structured semantic blueprints. The following examples will reinforce your understanding of the core semantic roles and demonstrate the flexibility of the tool we've built.

Example 1: Business pitch

Let's begin with the sentence we've been deconstructing throughout this

section: `Sarah pitched the new project to the board in the morning.`

The corresponding SRL definition is:

```
print("Example 1: A complete action with multiple roles.")
visualize_srl(
 verb="pitch",
 agent="Sarah",
 patient="the new project",
 recipient="to the board",
 temporal="in the morning"
)
```

In this example, we provide values for all the core roles and one modifier: •

Predicate: The central action is `pitch`.

- **Agent (ARG0):** The one doing the pitching is `Sarah`.
- **Patient (ARG1):** The thing being pitched is `the new project`.
- **Recipient (ARG2):** The entity receiving the pitch is `to the board`.
- **Temporal (ARGM-TMP):** The action took place `in the morning`.

Running this code produces the stemma shown in *Figure 1.3*, with **pitch** as the root node and four child nodes representing the roles:

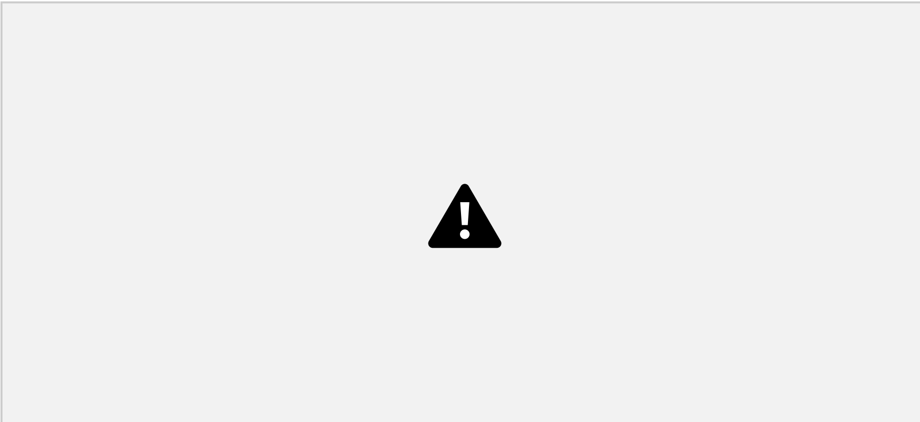


Figure 1.3: A root node and four child nodes

Let’s move on and explore another example.

Example 2: Technical update

Now, let's model a different type of sentence, this time including a location. Consider this sentence:

```
The backend team resolved the critical bug in the payment gateway.
```

The SRL definition will be as follows:

```
print("\nExample 2: An action with a location")
visualize_srl(
 verb="resolved",
 agent="The backend team",
 patient="the critical bug",
```



```
 location="in the payment gateway"
)
```

Here’s how the components map to the semantic roles:

- **Predicate**: The action is resolved
-

- **Agent (ARG0)**: The entity that performed the resolution is the backend team
- **Patient (ARG1)**: The thing that was resolved is the critical bug
- **Location (ARGM-LOC)**: The context for *where* the bug was resolved is in the payment gateway

The visualizer generates the stemma shown in *Figure 1.4*, with **resolved** at the top, connected to its three key participants. This clearly structures the technical update.

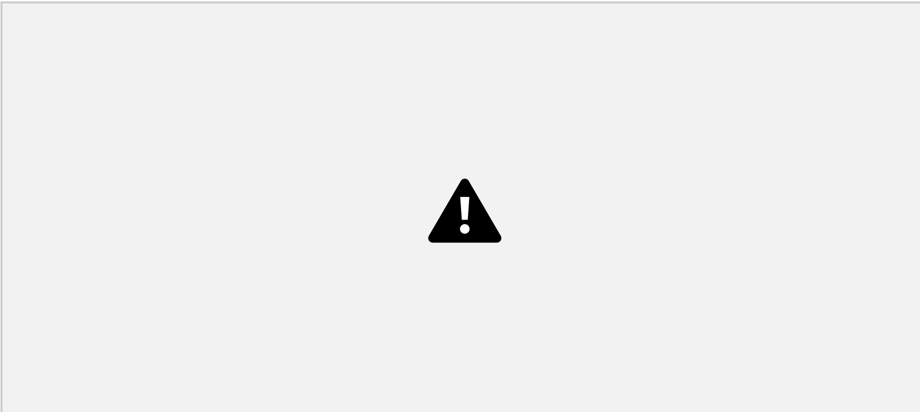


Figure 1.4: A stemma with an action and a location

Let's see how to visualize *how* something is done.

Example 3: Project milestone

Finally, let's visualize a sentence that describes *how* something was done, using a manner modifier. The sentence is this:

```
Maria's team deployed the new dashboard ahead of schedule.
```

The SRL definition is this:

```
print("\nExample 3: Describing how an action was performed")
visualize_srl(
 verb="deployed",
 agent="Maria's team",
```



```
 patient="the new dashboard",
 manner="ahead of schedule"
)
```

The roles map as follows:

- **Predicate**: The action is `deployed`
- **Agent (ARG0)**: The doer of the action is `Maria's team`
- **Patient (ARG1)**: The entity acted upon is `the new dashboard`

- **Manner (ARGM-MNR)**: This modifier describes *how* the deployment was done: `ahead of schedule`.

Running this code generates the stemma shown in *Figure 1.5*, which illustrates how a Manner modifier adds a contextual layer to the core agent-patient relationship.

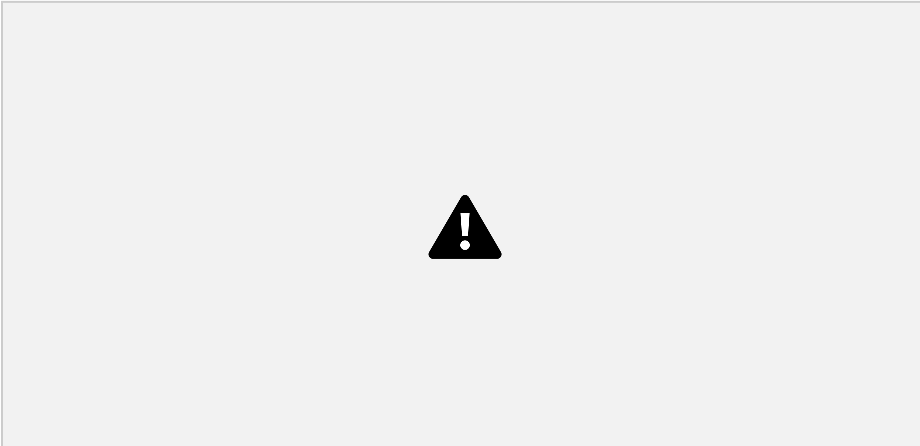


Figure 1.5: A stemma explaining the "how" of an action

These three examples illustrate how a sentence, a simple sequence of words, can be broken down into its semantic components using SRL. What was once linear text becomes a structured blueprint. With this, you gain control over the information you provide to an LLM. With the SRL tool in hand, we are ready to apply this framework to a practical use case.

Engineering a meeting analysis use case

This meeting analysis demonstrates a powerful technique called **context chaining**, where we guide an LLM through a multi-step analysis. Instead of one large, complex prompt, we use a series of simpler, focused prompts, where the output of one step becomes the input for the next. This creates a highly controlled and logical workflow.

We use a context chaining process because an LLM, despite its power, has no true memory or long-term focus. Giving an LLM a single, massive prompt with a complex, multi-step task is akin to giving a brilliant but forgetful assistant a lengthy list of verbal instructions and hoping for the best. The model will often lose track of the primary goal, get bogged down in irrelevant details, and produce a muddled, unfocused result.

Context chaining solves this problem by transforming a complex task into a controlled, step by-step dialogue. Each step has a single, clear purpose, and its output becomes the clean, focused input for the next step. This method gives you, the context engineer, three critical advantages:

- **Precision and control**: You can guide the AI's *thought process* at each stage, ensuring the analysis stays on track
- **Clarity and debugging**: If one step produces a poor result, you know exactly which prompt to fix, rather than trying to debug a single, monolithic instruction
-

Building on insight: It creates a narrative flow, allowing the AI to build upon the refined insights from the previous step, leading to a far more sophisticated and coherent final outcome

Let's review the program flowchart, as shown in *Figure 1.6*:

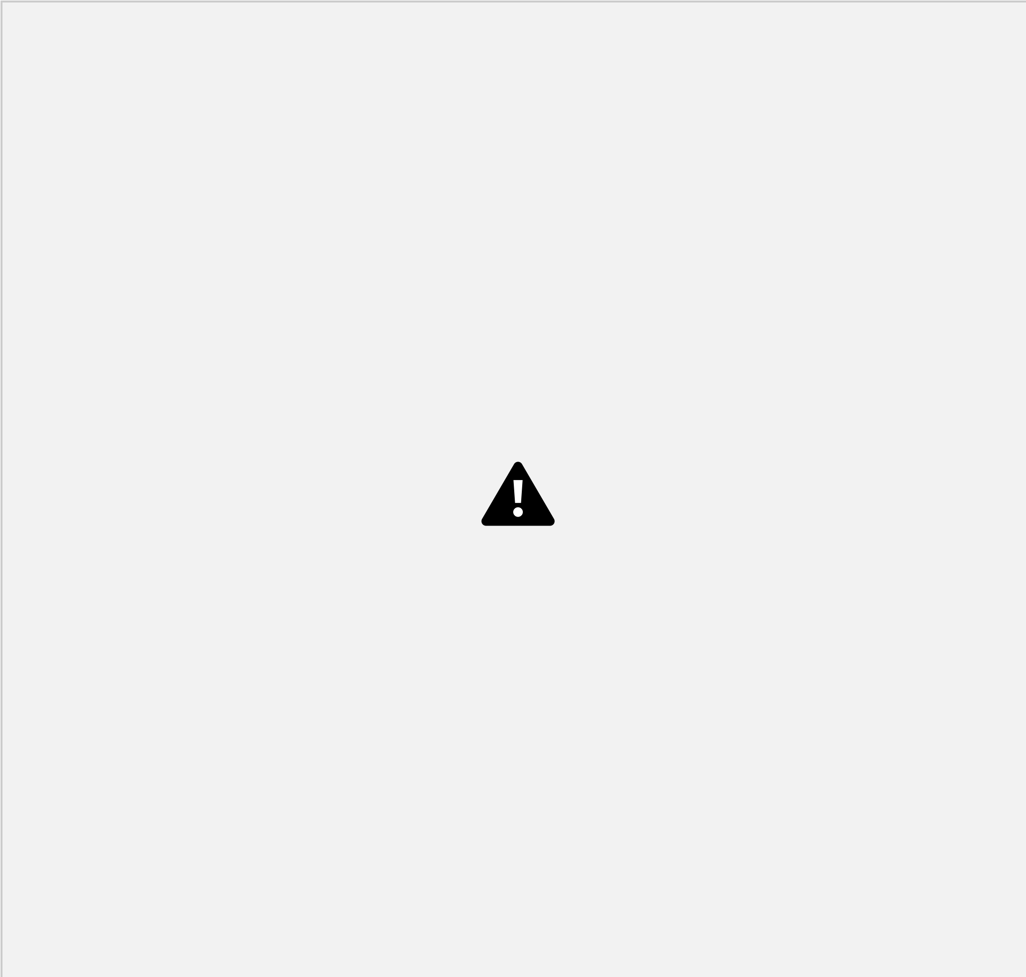


Figure 1.6: Context chaining flowchart

This branching structure is the key to the technique's power. It transforms the workflow from a single path into a parallel, multi-task analysis, starting with identifying new developments, analyzing implicit dynamics, and generating a novel solution. The complexity of this flow explains why we will be building a Context Engine in later chapters to manage these relationships automatically:

1. **Input raw transcript** (`meeting_transcript`)
    - **Purpose**: Provide the single source of raw data

   - **Predecessor**: None (This is the starting data)
   - **Successor**: g2 (Isolate Key Content)

2. **Isolate key content** (g2)
    - **Purpose**: Clean the data; separate signal from noise.
    - **Predecessor**: transcript (Input Raw Transcript).
    - **Successors**: This is the main branching point. Its output
    - (`substantive_content`) is the direct predecessor for three different, parallel steps:
        - g3 (Identify New Developments)
        - g4 (Analyze Implicit Dynamics)
        - g5 (Generate Novel Solution)
3. **Identify new developments** (g3)
    - **Purpose**: Find new information vs. an old summary
    - **Predecessors**: `substantive_content` (from g2) and `prev_summary`
    - **Successor**: g6 (Create Structured Summary)
4. **Analyze implicit dynamics** (g4)
    - **Purpose**: Analyze feelings and social subtext
    - **Predecessor**: `substantive_content` (from *Step 2*)
    - **Successor**: None (leads to a final output, `implicit_threads`)
5. **Generate novel solution** (g5)
    - **Purpose**: Synthesize facts into a new, creative idea
    - **Predecessor**: `substantive_content` (from *Step 2*)
    - **Successor**: None (leads to a final output, `novel_solution`).
6. **Create structured summary** (g6)
    - **Purpose**: Format the new developments into a table.

- **Predecessor**: g3 (Identify New Developments).
- Successor: g7 (Draft Follow-up Action).

7. **Draft follow-up action** (g7)
    - **Purpose**: Convert the structured summary into an action item.
    - **Predecessor**: g6 (Create Structured Summary).
    - **Successor**: None (Leads to the final output, `follow_up_email`).

Let's now translate this flowchart idx_89d5f1af plan into actual code and run it step by step: 1.

Open `Use_Case.ipynb` in the chapter directory. We will first install the necessary OpenAI library.

```
# Cell 1: Installation
!pip install openai
```

2. This workflow uses Google Colab Secrets to store the OpenAI API key. Load the key, set the environment variable, and initialize the client:

```
# Cell 2: Imports and API Key Setup
# We will use the OpenAI library to interact with the LLM and Google
Colab's
# secret manager to securely access your API key.

import os
from openai import OpenAI
from google.colab import userdata

# Load the API key from Colab secrets, set the env var, then init the
client
try:
 api_key = userdata.get("API_KEY")
    if not api_key:
        raise userdata.SecretNotFoundError("API_KEY not found.")

    # Set environment variable for downstream tools/libraries
 os.environ["OPENAI_API_KEY"] = api_key

    # Create client (will read from OPENAI_API_KEY)
 client = OpenAI()
    print("OpenAI API key loaded and environment variable set
successfully.")

except userdata.SecretNotFoundError:
    print('Secret "API_KEY" not found.')
```

```
        print('Please add your OpenAI API key to the Colab Secrets Manager.')
```



```
    except Exception as e:
        print(f"An error occurred while loading the API key: {e}")
```

3. Add the meeting transcript as a multi-line string. This will be the input for the chained steps that follow:

```
# Cell 3: The Full Meeting Transcript
meeting_transcript = """
 Tom: Morning all. Coffee is still kicking in.
 Sarah: Morning, Tom. Right, let's jump in. Project Phoenix
timeline. Tom, you said the backend components are on track?
 Tom: Mostly. We hit a small snag with the payment gateway
integration. It's... more complex than the docs suggested. We might need
another three days.
 Maria: Three days? Tom, that's going to push the final testing  phase
right up against the launch deadline. We don't have that buffer.
Sarah: I agree with Maria. What's the alternative, Tom?
 Tom: I suppose I could work over the weekend to catch up. I'd  rather
not, but I can see the bind we're in.
 Sarah: Appreciate that, Tom. Let's tentatively agree on that. Maria,
what about the front-end?
 Maria: We're good. In fact, we're a bit ahead. We have some extra
bandwidth.
 Sarah: Excellent. Okay, one last thing. The marketing team wants to  do a
big social media push on launch day. Thoughts?
 Tom: Seems standard.
 Maria: I think that's a mistake. A big push on day one will swamp  our
servers if there are any initial bugs. We should do a soft launch,
invite-only for the first week, and then do the big push. More controlled.
Sarah: That's a very good point, Maria. A much safer strategy. Let's go
with that. Okay, great meeting. I'll send out a summary.
 Tom: Sounds good. Now, more coffee.
 """
```

You're ready to start context chaining. The first action will separate signal from noise in `meeting_transcript`: extract decisions, updates, and issues; set aside greetings and small talk. Let's get into it!

Layer 1: Establishing the scope (the "what")

We'll define the scope of analysis first. As established, each step's output will feed the next, creating a chain of context.

1. Tell the model exactly what to extract and what to ignore. The goal is to separate substantive content (decisions, updates, problems, proposals) from conversational noise (greetings, small talk).

```
# Cell 4: g2 - Isolating Content from Noise
prompt_g2 = f"""
```

```
 Analyze the following meeting transcript. Your task is to isolate  the
substantive content from the conversational noise.
 - Substantive content includes: decisions made, project updates,
problems raised, and strategic suggestions.
 - Noise includes: greetings, pleasantries, and off-topic remarks  (like
coffee).
 Return ONLY the substantive content.

 Transcript:
 ---
        {meeting_transcript}
 ---
 """
```

2.

Now we activate OpenAI to isolate the substantive content:

```
from openai import OpenAI
try:
 client = OpenAI()

 response_g2 = client.chat.completions.create(
 model="gpt-5",
 messages=[
 {"role": "user", "content": prompt_g2}
 ]
 )

 substantive_content = response_g2.choices[0].message.content
    print("--- SUBSTANTIVE CONTENT ---")
    print(substantive_content)
```



```
except Exception as e:
    print(f"An error occurred: {e}")
```

The output displays the substantive content:

```
--- SUBSTANTIVE CONTENT ---
- Project Phoenix timeline: Backend mostly on track, but payment gateway
integration is more complex than expected; needs an additional three days.
- Impact: Extra three days would push final testing up against the launch
deadline, reducing buffer.
- Mitigation decision: Tom will work over the weekend to catch up
(tentatively agreed).
- Front-end status: Ahead of schedule with extra bandwidth.
- Marketing/launch strategy: Initial plan for a big social media push on
launch day flagged as risky (potential server load with early bugs).
Decision: Use a soft launch (invite-only) for the first week, then execute
the big push.
```

3.

Next, we simulate a **retrieval-augmented generation** (**RAG**) context by comparing the new meeting with a summary of the previous one. This narrows the focus to *what's new*, showing the importance of historical context:

```
# Cell 5: g3 - Identifying NEW Information (Simulated RAG)
previous_summary =
"In our last meeting, we finalized the goals for Project Phoenix and
assigned backend work to Tom and front-end to Maria."

prompt_g3 = f"""
Context: The summary of our last meeting was: "{previous_summary}"

Task: Analyze the following substantive content from our new meeting.
Identify and summarize ONLY the new developments, problems, or decisions
that have occurred since the last meeting.

New Meeting Content:
---
{substantive_content}
---
"""
```

We now put OpenAI to work again for the RAG-like task:

```
from openai import OpenAI
from google.colab import userdata

# Your chat completion request
try:
 response_g3 = client.chat.completions.create(
 model="gpt-5",
 messages=[{"role": "user", "content": prompt_g3}]
 )
 new_developments = response_g3.choices[0].message.content
    print("--- NEW DEVELOPMENTS SINCE LAST MEETING ---")
    print(new_developments)
except Exception as e:
    print(f"An error occurred: {e}")
```

The output narrows down the expectations and the core information of the meeting:

```
--- NEW DEVELOPMENTS SINCE LAST MEETING ---
- Backend issue: Payment gateway integration is more complex than expected;
needs an additional three days.
- Schedule impact: The extra three days compress final testing, pushing it
up against the launch deadline and reducing buffer.
- Mitigation decision: Tentative agreement that Tom will work over the
weekend to catch up.
- Front-end status: Ahead of schedule with extra bandwidth.
- Launch/marketing decision: Shift from a big day-one social push to a one
week invite-only soft launch, followed by the major push.
```

We have uncovered the scope; in other words, the *what* of the transcript. Let's dig into the *how*.

Layer 2: Conducting the investigation (the "how")

Now we move

from identifying facts to generating insights, the core of the semantic context interpretation journey. This is where the prompt asks the AI to read between the lines.

1. Beyond explicit facts, every meeting carries **subtext**: hesitations, tensions, and moods. This step asks the AI to analyze the underlying dynamics:

```
# Cell 6: g4 - Uncovering Implicit Threads
prompt_g4 = f"""
```



```
Task: Analyze the following meeting content for implicit social dynamics
and unstated feelings. Go beyond the literal words.
- Did anyone seem hesitant or reluctant despite agreeing to something?
- Were there any underlying disagreements or tensions?
- What was the overall mood?

Meeting Content:
---
{substantive_content}
---
"""
```

2. We now run the prompt to explore the implicit dynamics of the meeting beyond the literal words:

```
try:
 response_g4 = client.chat.completions.create(
 model="gpt-5",
 messages=[{"role": "user", "content": prompt_g4}]
 )
 implicit_threads = response_g4.choices[0].message.content
    print("--- IMPLICIT THREADS AND DYNAMICS ---")
    print(implicit_threads)
except Exception as e:
    print(f"An error occurred: {e}")
```

The output at this stage highlights something new: LLMs aren't just parroting facts or summarizing events. They are also surfacing the *subtext*:

```
 --- IMPLICIT THREADS AND DYNAMICS ---
Here's what seems to be happening beneath the surface:

Hesitation/reluctance despite agreement
 - Tom's "tentative" agreement to work over the weekend reads as reluctant.
It suggests he felt pressure to volunteer rather than genuine willingness. -
Marketing likely agreed to the soft launch with some reluctance; shifting
from a big day-one push to invite-only implies a concession to engineering
risk.

Underlying disagreements or tensions
  - Pace vs quality: Engineering wants stability and buffer; marketing
```

```
originally aimed for impact. The soft launch is a compromise, but the
differing risk appetites remain.
- Workload equity: Backend is behind while frontend has "extra bandwidth."
The decision to have Tom work the weekend (vs redistributing tasks) hints
at siloing or a norm of individual heroics, which can breed quiet
resentment.
- Testing squeeze: Pushing testing against the deadline implies QA will be
under pressure, potentially creating friction if bugs slip through or late
changes occur.
- Estimation confidence: The payment gateway being "more complex than
expected" may subtly challenge earlier estimates, inviting unspoken doubt
about planning or vendor integration assumptions.

Overall mood
- Sober, pragmatic, and slightly tense. The group is solution-oriented and
collaborative, but there's a sense of urgency and strain, with relief at
having a plan tempered by concerns about workload, risk, and reduced
buffer.
```

This is where context chaining shifts from recording what happened to interpreting why it matters. The result feels less like a raw transcript and more like an analyst's commentary, giving us insights into team dynamics that would otherwise remain unstated.

3. Next, we prompt the AI to be creative and solve a problem by synthesizing different ideas from the meeting, demonstrating its thinking power:

```python
# Cell 7: g5 - Generating a Novel Solution
prompt_g5 = f"""
Context: In the meeting, Maria suggested a 'soft launch' to avoid server
strain, and also mentioned her team has 'extra bandwidth'.
Tom is facing a 3-day delay on the backend.

Task: Propose a novel, actionable idea that uses Maria's team's extra
bandwidth to help mitigate Tom's 3-day delay. Combine these two separate
pieces of information into a single solution.
"""
try:
 response_g5 = client.chat.completions.create(
 model="gpt-5",
 messages=[{"role": "user", "content": prompt_g5}]
```



```python
 )
 novel_solution = response_g5.choices[0].message.content
    print("--- NOVEL SOLUTION PROPOSED BY AI ---")
    print(novel_solution)
except Exception as e:
    print(f"An error occurred: {e}")
```

The prompt challenges the AI to take two separate threads from the meeting, Maria's extra frontend bandwidth and Tom's three-day backend delay, and weave them into a novel solution. This is where context chaining demonstrates its real strength: instead of asking for raw recall or passive interpretation, we push the model to generate something new, something that didn't exist in the transcript:

```
--- NOVEL SOLUTION PROPOSED BY AI ---
Idea: Soft launch behind a temporary "Edge Bridge" that Maria's team builds
to buffer reads/writes until Tom's backend is ready.

What Maria's team does (uses their extra bandwidth)
- Stand up a thin serverless/API facade that matches the real API contracts
(e.g., API Gateway/Lambda + SQS/DynamoDB or Cloudflare Workers + Durable
Objects).
- Reads: Serve from a prewarmed cache or static snapshots (stale-while
revalidate). Update snapshots hourly via a lightweight data export from
staging or existing endpoints.
- Writes: Capture requests into a durable queue with idempotency keys;
return immediate "queued" success to the client and show "syncing" UI. When
Tom's backend is live, a replay worker drains the queue and applies
changes.
- Add feature flags/traffic gating (e.g., LaunchDarkly) to limit the soft
launch cohort and throttle requests to avoid server strain.

How this mitigates the 3-day delay
- The product can soft-launch to a small cohort without waiting for the
backend; users get read access and buffered writes.
- When Tom's backend is ready, flip routing to the real backend and drain
the queue to reconcile data.

Action plan and timeline
- Day 0 (today): Identify minimal critical endpoints for the soft launch.
Classify by read vs write. Define API contracts and idempotency rules. Set
success/error thresholds and a kill switch.
```



```
- Day 1: Maria's team builds the Edge Bridge, cache, and write queue;
implement basic observability and encryption-at-rest for any PII in the
queue. Front-end adds "syncing" UI states and feature flags.
- Day 2: QA with mocked data, then with a tiny internal cohort. Prewarm
caches. Set traffic cap (e.g., 5-10% of target users).
- Day 3: Soft launch goes live on the Edge Bridge. When Tom's backend
unlocks, switch routing gradually, start replay worker, monitor for
conflicts, then retire the bridge.

Risk controls
- Data consistency: Use idempotency keys and a simple conflict policy
(latest-write-wins or version checks).
- Rollback: Feature flag to disable writes or pause replay if error rate
exceeds threshold.
- Privacy: Encrypt queued payloads; limit PII scope.

Owners
- Maria's team: Edge Bridge, caching, queue/replay, monitoring.
- Tom's team: Final backend endpoints, schema, and replay acceptance hooks.
- Front-end: Feature-flag routing and "queued/syncing" UX.

This combines Maria's extra bandwidth with a controlled soft launch to keep
```

```
momentum while absorbing Tom's 3-day backend delay.
```

The output shows how an LLM can function as a creative collaborator, proposing a technically feasible workaround that blends system design, risk controls, and role assignments. The result feels less like a "guess" and more like a carefully reasoned plan a senior engineer might sketch out in a design session.

Layer 3: Determining the action (the "what next")

Finally, we turn the analysis into concrete, forward-looking artifacts. Up to this point, we've been generating raw insights: new facts, implicit dynamics, and creative solutions. But insights are only valuable if they can be communicated clearly. The next step is to compile everything into a structured, final summary.

This serves two purposes:

1. It **forces clarity** since every item is reduced to topic, outcome, and owner.
2. It **makes information reusable**. Whether dropped into an email, report, or dashboard, the summary is clean and immediately actionable.

```
# Cell 8: g6 - Creating the Final, Structured Summary
prompt_g6 = f"""
Task: Create a final, concise summary of the meeting in a markdown table.
Use the following information to construct the table.

- New Developments: {new_developments}

The table should have three columns: "Topic", "Decision/Outcome", and "Owner".
"""
try:
 response_g6 = client.chat.completions.create(
 model="gpt-5",
 messages=[{"role": "user", "content": prompt_g6}]
 )
 final_summary_table = response_g6.choices[0].message.content
    print("--- FINAL MEETING SUMMARY TABLE ---")
    print(final_summary_table)
except Exception as e:
    print(f"An error occurred: {e}")
```

The output we get represents the essence of the meeting: all the noise, subtext, and negotiation distilled into a crisp reference point:

```
--- FINAL MEETING SUMMARY TABLE ---
| Topic | Decision/Outcome | Owner |
|---|---|---|
| Backend payment gateway integration | More complex than expected; requires an
additional three days | Backend Team |
| Schedule impact | Extra three days compress final testing, reducing buffer
before launch | Project Manager |
| Mitigation | Tentative plan: Tom will work over the weekend to catch up | Tom |
| Front-end status | Ahead of schedule with extra bandwidth available | Front-end
```

```
Team |
| Launch/marketing plan | Shift to a one-week invite-only soft launch, then major
day-one push | Marketing + Product |
```


The final step in context chaining is to close the loop from insight to action: turning the structured analysis into a professional follow-up email:

```
# Cell 9: g7 - Drafting the Follow-Up Action
prompt_g7 = f"""
Task: Based on the following summary table, draft a polite and professional
follow-up email to the team (Sarah, Tom, Maria).
The email should clearly state the decisions made and the action items for each
person.

Summary Table:
---
{final_summary_table}
---
"""
```

The LLM will do the heavy-lifting again and get the job done:

```
try:
 response_g7 = client.chat.completions.create(
 model="gpt-5",
 messages=[{"role": "user", "content": prompt_g7}]
 )
 follow_up_email = response_g7.choices[0].message.content
    print("--- DRAFT FOLLOW-UP EMAIL ---")
    print(follow_up_email)
except Exception as e:
    print(f"An error occurred: {e}")
```

The final action is a follow-up email based on a deep analysis of the meeting:

```
--- DRAFT FOLLOW-UP EMAIL ---
Subject: Follow-up: Decisions and next steps from today's sync

Hi Sarah, Tom, and Maria,

Thanks for the productive discussion earlier. Here's a quick recap of decisions
and the action items for each of us.

Decisions
- Backend payment gateway integration is more complex than expected and will
```



```
require an additional three days.
- This pushes the schedule by three days and compresses the final testing window,
reducing our pre-launch buffer.
- Mitigation: Tom will work over the weekend to help us catch up.
- Front-end is ahead of schedule and has extra bandwidth to support. -
Launch/marketing plan will shift to a one-week invite-only soft launch, followed
by the larger day-one push.
```

```
Action items
- Tom:
 - Confirm weekend availability and share a brief plan (key milestones,
dependencies, and any risks).
 - Proceed with the gateway integration and coordinate early integration testing
with Front-end and QA.
 - Provide short daily progress updates and flag blockers immediately.

- Sarah:
 - Update the project timeline to reflect the three-day shift and the compressed
QA window.
 - Coordinate with QA on a risk-based test plan that fits the shortened testing
period.
 - Align with Marketing/Product on the invite-only soft launch scope, success
metrics, and comms; circulate the plan to the team.

- Maria:
 - Reallocate Front-end bandwidth to support the backend integration (payment UI
hooks, error handling, instrumentation).
 - Partner with Tom on mocks/stubs as needed to unblock early integration and QA.
- Ensure front-end readiness for the soft launch (feature flags/toggles,
tracking) and share any gaps.

Please reply to confirm your action items and note any constraints or support you
need. I'm happy to set up a brief daily check-in while we work through this;
propose a time if you have a preference.

Thanks all, and appreciate the quick coordination.

Best,
[Your Name]
```


This final artifact demonstrates the full promise of context chaining. The email reads like it was crafted by a diligent project manager. Instead of ending with abstract insights or a loose list of points, the workflow produces professional communication. The AI has not only summarized the meeting but has also translated it into a format that does the following:

- **Captures decisions clearly** so there's no ambiguity about what was agreed upon
- **Assigns ownership**, ensuring every task is tied to a responsible person
- **Sets expectations** such as timelines, next steps, and accountability
- **Reduces follow-up friction** as the draft is already polished enough to send, saving the human time and energy

This is the moment where the LLM stops being a "note-taker" and becomes a *creative partner*, as we mentioned at the start of this chapter. In this use case, we didn't just *get a summary*. We witnessed how to think with the AI as a partner. The human remains at the center of the process and can create templates of context chaining for meetings, email processing, reporting, and anything you can imagine. Used well, context chaining can elevate the way teams, companies, and clients operate.

Let's now step back, summarize what we've accomplished, and move on to the next exploration in context engineering.

Summary

This chapter introduced context engineering as an emerging skill for turning LLMs into reliable, goal-oriented systems. Instead of relying on unstructured prompts, we showed how control comes from engineering the informational environment, culminating in the semantic blueprint as the most precise form of direction.

We traced this shift through a five-level progression: from zero-context prompts that yielded generic outputs to linear, goal-oriented, and role-based contexts, each proving that structured input drives better results. To formalize this approach, we introduced SRL, a method that breaks sentences into predicate, agent, patient, and modifiers, supported by a Python visualizer that renders these roles as a stemma diagram.

Finally, we applied these skills in a meeting analysis use case, where context chaining turned a raw transcript into actionable outcomes. Step by step, the process reduced noise, highlighted new developments, surfaced implicit dynamics, and produced structured summaries and follow-up actions.

Together, SRL and context chaining provide both the theoretical framework and the practical workflow, respectively, to move beyond prompting. We are now ready to engineer agentic contexts in the next chapter.

Questions

1. Is the primary goal of context engineering, as defined in this chapter, simply to ask an LLM more creative questions? (Yes or no)
2. Does a "Level 2: Linear Context" provide enough information to control an LLM's narrative style and purpose reliably? (Yes or no)
3. Is the "Semantic Blueprint" at Level 5 presented as the most effective method for architecting a precise and reliable AI response? (Yes or no)
4. Is the main function of **Semantic Role Labeling** (**SRL**) to check the grammatical correctness of a sentence? (Yes or no)
5. In the sentence "Sarah pitched the new project," is "Sarah" identified as the patient (ARG1)? (Yes or no)
6. Do **Argument Modifiers** (**ARGM-**), such as temporal or location, represent the central and essential components of an action in SRL? (Yes or no)
7.

Does the chapter's final use case rely on a single, large, and complex prompt to analyze the meeting transcript? (Yes or no)

8. Is the technique of "context chaining" defined as using the output from one LLM call as the input for the next? (Yes or no)

9. In the use case workflow, is the step *Analyze Implicit Dynamics* designed to extract only the explicit facts and decisions from the text? (Yes or no)

10. Does the chapter's meeting analysis workflow end with the creation of an *actionable artifact*, such as a draft email? (Yes or no)

References

- Tesnière, L. (1959). *Éléments de syntaxe structurale*. Klincksieck.
- Fillmore, Charles J. 1968. "The Case for Case." In *Universals in Linguistic Theory*, edited by Emmon Bach and Robert T. Harms, 1–88. New York: Holt, Rinehart and Winston. `https://linguistics.berkeley.edu/~syntax-circle/syntax-group/spr08/fillmore.pdf`
- Palmer, Martha, Daniel Gildea, and Paul Kingsbury. 2005. "The Proposition Bank: An Annotated Corpus of Semantic Roles." *Computational Linguistics* 31 (1): 71–106. `https://aclanthology.org/J05-1004.pdf`

Further reading

- Brown, Tom, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, et al. 2020. "Language Models Are Few-Shot Learners." *Advances in Neural Information Processing Systems* 33: 1877–1901. `https://arxiv.org/abs/2005.14165`
- Wei, Jason, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models." *Advances in Neural Information Processing Systems* 35: 24824–24837. `https://arxiv.org/abs/2201.11903`

Note: Keep your invoice handy. Purchases made directly from Packt don't require one.

2

Building a Multi-Agent System with MCP

In the previous chapter, we established the foundational skill of context engineering: transforming a vague prompt into a structured semantic blueprint. Now, we will apply that skill at a much larger scale to build a system of specialized AI agents that collaborate to solve complex problems using contexts to communicate. A single LLM is a brilliant generalist, but it's not an expert. For any complex, multi-step task, using a single LLM is inefficient and often fails. Therefore, we will architect a **Multi-Agent System** (**MAS**), where the *context* we engineer expands beyond the content of a single message to define the very design of the system. This includes defining which agents are needed, their specific roles and capabilities, and the structured methods they use to communicate without losing information. This is the true scope of context engineering.

Further, to make our system of agents reliable, we will implement the **Model Context Protocol** (**MCP**), a shared language that ensures our*engineered contexts* are passed between agents with perfect fidelity. In this hands-on section, we will build a complete MAS-MCP system from scratch. You will learn how to design the **Orchestrator** that manages the workflow and the **specialist agents** (Researcher and Writer) that xecute the tasks. You will also see how error handling, message validation, and automated quality control are built directly into the workflow, ensuring the system can recover from failures and maintain factual accuracy. By the end, you will have a fully functional system that can take a high-level goal and autonomously manage a complex, multi-step process. In a nutshell, we will cover the following topics:

- Architecting a context-driven MAS workflow
- Building an MAS with MCP to manage contexts

- Error handling and validation
- The evolution of AI architecture
- Tools for building agent systems

Let's begin by architecting our system.

Architecting the MAS workflow with MCP

We need a clear architectural plan before writing even a single line of code. This section lays out that blueprint. We will deconstruct the overall goal into core components, define the role of each part of the system, and map the flow of communication that will bring our agent team to life.

First, let's define the two core concepts at the heart of this design, the **MAS** and **MCP**:

- **MAS**: We will design a system that can run multiple independent agents, each one specialized in a distinct task such as research, writing, or data analysis. By giving each agent a clear context, we ensure it can excel at its specific responsibility.
- **MCP**: For our agents to collaborate, they need a shared language. MCP gives us the rules for how our agents pass tasks and information to one another. It provides a framework that ensures every message is structured, reliable, and perfectly understood.

Our MAS will consist of three distinct components. Each plays a specific role, and together they form a workflow capable of transforming a high-level user goal into a finished product:

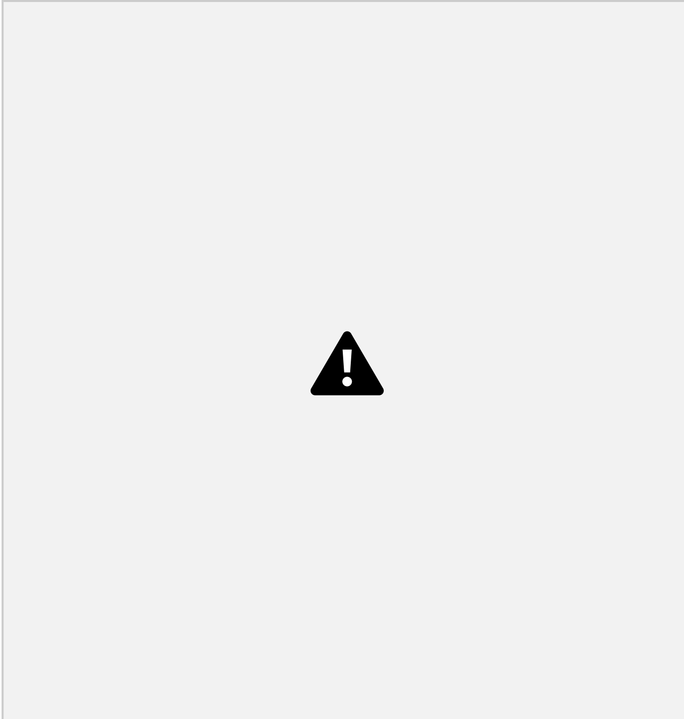


Figure 2.1: The architectural blueprint of our MAS workflow

The flowchart illustrates the system's complete workflow. Let's break down the role of each component in this cognitive pipeline:

- **Orchestrator (the project manager)**: The Orchestrator is the brain of the operation. It doesn't perform specialized tasks itself but manages the entire workflow. It receives the user's high-level goal, breaks it down into logical steps, and delegates each step to the right agent. It is also responsible for receiving the results from one agent and passing them as *context* to the next. In other words, it applies *context chaining* at the system level to our MAS.
- **Researcher agent (the information specialist)**: This is our first specialized agent. Its purpose is to take a specific topic, find relevant information, and synthesize that information into a structured summary. In our project, it will receive a research task from the Orchestrator and return the results as a clear, bullet-pointed list.
- **Writer agent (the content creator)**: This is our second specialized agent. Its strength lies in communication and creative expression. It takes the structured summary from

the Researcher and transforms it into a polished, human-readable piece of content, with careful attention to tone, style, and narrative.

Information doesn't just flow as raw text. Every interaction between agents is packaged as a structured **MCP message**. This ensures that tasks and results are always passed with *full context*, in a consistent, predictable, and reliable format. MCP is the connective tissue that turns a collection of individual agents into an integrated system.

With this blueprint in place, we're ready to begin building our MAS with MCP.

In this book, we explore the cutting-edge direction of applying MCP principles to agent-to-agent communication. While MCP was originally designed for agent-tool interaction, recent explorations (such as Microsoft's *Can You Build Agent2Agent Communication on MCP? Yes!*) demonstrate how MCP's evolving capabilities can support emerging inter-agent coordination patterns.

Link to the article: `https://developer.microsoft.com/blog/can-you-build agent2agent-communication-on-mcp-yes`

Building an MAS with MCP

Now that we have a blueprint, it's time to start coding. In this section, we will implement the MAS and MCP step by step. We will begin with the system's core functionality, then return later to add error handling and validation in the *Error handling and validation* section.

Open `MAS_MCP.ipynb` from this chapter's repository to follow along. The initial OpenAI installation is the same as in `SRL.ipynb` from *Chapter 1*.

We will work through the Colab notebook block by block: first defining our communication protocol, then building each specialist agent, constructing the Orchestrator to manage them, and finally, running the entire system to see our agent team in action. Let's first initialize the OpenAI client.

Initializing the client

We first initialize the OpenAI client, which will serve as our gateway to the LLM. We also import the `json` library to display our structured messages in a clean, readable format:

```
#@title 1. Initializing the Client
# -----------------------------------------------------------------------------
# We'll need the `openai` library to communicate with the LLM.
```



```
# Note: This notebook assumes you have already run a setup cell in your Colab
# environment to load your API key from Colab Secrets into an environment #
variable, as you specified.
# -----------------------------------------------------------------------------
import json

# --- Initialize the OpenAI Client ---
# The client will automatically read the OPENAI_API_KEY from your environment.
client = OpenAI()
print("OpenAI client initialized.")
```

The output displays a confirmation message:

```
OpenAI client initialized.
```

Defining the protocol

As we established earlier, our agents need a shared language to collaborate. That language is defined by MCP. In this section, we'll implement a simplified version to illustrate the process. While our Python dictionary approach works perfectly for learning, it's worth understanding the official MCP rules that govern how two systems communicate.

Message format

The structure of every MCP message is strictly defined to ensure consistency:

- All messages follow the **JSON-RPC 2.0** format as clean JSON objects
- Messages must be **UTF-8 encoded** for universal compatibility
- Each message must appear on a single line with no embedded newlines, making parsing fast and reliable

Once the message is defined, MCP defines the transport layers.

Transport layers

The transport layer defines how messages are transmitted between agents. The two primary methods are as follows:

- **STDIO (standard input/output)**: For agents running on the same machine (such as in our Colab notebook), they can communicate directly through standard input/output. This is the simplest and most direct method.
- **HTTP**: For agents running on different servers, messages are sent over the internet using standard HTTP requests.

Finally, we need a protocol management framework.

Protocol management

MCP also includes rules for compatibility and safety:

- **Versioning**: When using HTTP, a version header is required to ensure the client and server are using the same set of rules
- **Security**: There are rules for validating connections to prevent common cyberattacks and ensure you are communicating with the intended server

For the hands-on projects in this book, we will focus on the spirit of MCP by practicing structured communication. A simple Python dictionary is more than enough to illustrate this idea. It serves as our stand-in for the formal JSON-RPC object, letting us see clearly how messages are built and passed without getting lost in protocol overhead.

With that in place, we can now create a helper function called `create_mcp_message`. This function is our template for messages that travel through the system. By using the same structure each time, we make sure information never gets lost or misunderstood. Here's the code:

```
#@title 2. Defining the Protocol: The MCP Standard
# ------------------------------------------------------------------------
# Before we build our agents, we must define the language they will speak.
# MCP provides a simple, structured way to pass context. For this example,
# our MCP message will be a Python dictionary with key fields.
# ------------------------------------------------------------------------
def create_mcp_message(sender, content, metadata=None):
    """Creates a standardized MCP message."""
    return {
        "protocol_version": "1.0",
        "sender": sender,
        "content": content,
        "metadata": metadata or {}
 }
print("--- Example MCP Message (Our Simplified Version) ---")
example_mcp = create_mcp_message(
 sender="Orchestrator",
 content="Research the benefits of the Mediterranean diet.",
 metadata={"task_id": "T-123", "priority": "high"}
)
print(json.dumps(example_mcp, indent=2))
```

The output displays an example of our structured MCP message in JSON format:

```
--- Example MCP Message (Our Simplified Version) ---
{
 "protocol_version": "1.0",
 "sender": "Orchestrator",
 "content": "Research the benefits of the Mediterranean diet.",
 "metadata": {
 "task_id": "T-123",
 "priority": "high"
 }
}
```

The `create_mcp_message` function takes three inputs: the sender, the content (the task or information), and any optional metadata. It always returns a standard Python dictionary in the same structure. This consistency is the foundation of our system's reliability. Messages can flow between agents without risk of being lost, misread, or misunderstood.

With our protocol defined, we are ready to build the specialist agents.

Building the agents

This section contains the code that brings our agents from the workflow diagram in *Figure 2.1* to life. We will define each agent as a Python function. Every agent function will accept a structured MCP message as input and return another MCP message as output. An agent's specific role is shaped by its **system prompt**, which tells the LLM how to behave. To keep ommunication consistent, we will also create a single helper function called `call_llm` that manages all interactions with the OpenAI API.

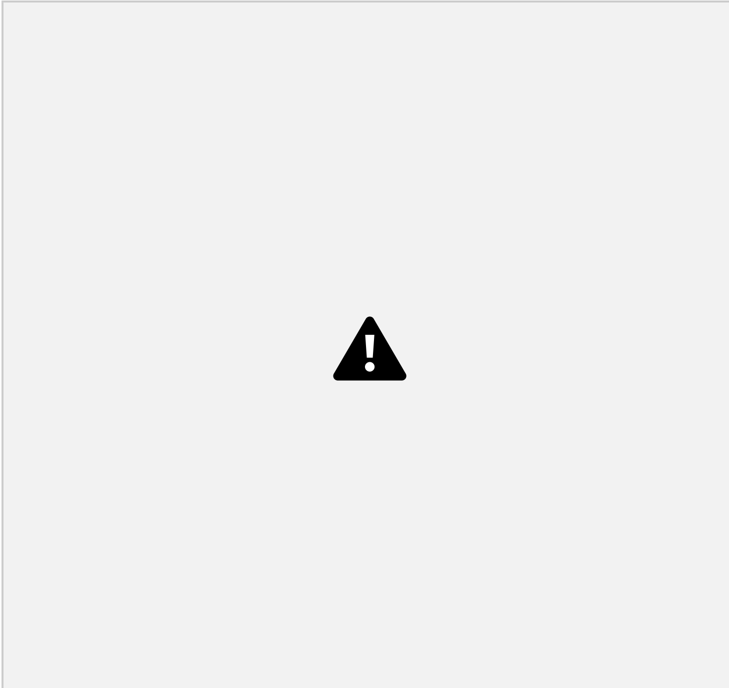


Figure 2.2: The architecture of the multi-agent functions

The figure illustrates a workflow that uses two AI agents to create a blog post. Information flows from an initial idea to a finished piece of content in a series of steps:

1. The process begins with a research topic, the prompt that starts the workflow. A message containing the topic is sent to our first agent, the **Researcher agent**, which gathers relevant facts and information from a simulated internal database.
2. Once the Researcher has the raw data, it uses the LLM API, an external language model service, to process the information and create a *summary* of the key findings. This summary is the bridge between the previous and next stages of the process.
3. The summary is then passed to the second agent, the **Writer agent**, whose role is to transform the research points into a finished article. The Writer also communicates with the LLM API, but or the complex and creative task of writing.
4. The final result of the system is a blog post, a clear demonstration of how specialized agents can work together to turn a high-level goal into a complete artifact. We are now ready to implement the agents that communicate with one another.

Creating the helper function

Before we build the agents themselves, we need a reliable and consistent way for them to communicate with the LLM. Our helper function, `call_llm`, handles all API calls to OpenAI. It takes a **system prompt** and **user content** as input and returns the model's text response:

```
def call_llm(system_prompt, user_content):
    """A helper function to call the OpenAI API using the new client syntax."""
    try:
    # Using the updated client.chat.completions.create method
  response = client.chat.completions.create(
  model="gpt-5",
  messages=[
  {"role": "system", "content": system_prompt},
  {"role": "user", "content": user_content}
  ]
  )
        return response.choices[0].message.content…
```

This function has two inputs:

- `system_prompt`, which is a string that tells the model how to behave
- `user_content`, which is the string containing the specific information or question we are sending

Inside the function, we call the OpenAI client to create a chat completion. The messages are passed as a list of dictionaries, where we explicitly set the **system role** (to frame the agent's behavior) and the **user role** (to deliver the actual input). The function then returns the text from the model's response. The `try` and `except` block, meanwhile, ensures basic error
handling, so the function doesn't crash if the API call fails.

With this helper function in place, we can now build our first specialized agent: the Researcher agent.

Defining the Researcher agent

Our first agent is the **Researcher**. Its job is to take a topic, look up relevant information, and summarize it for the next stage in the workflow.

We start by defining the function. The `researcher_agent` function takes in `mcp_input`, which follows the standard MCP message format we defined earlier. When activated, it prints a log message so we can trace when the agent starts working:

```
def researcher_agent(mcp_input):
    """
    This agent takes a research topic, finds information, and returns a summary.
    """
    print("\n[Researcher Agent Activated]")
```

Next, we create `simulated_database`, a simple dictionary that mimics a real data source. This is enough for our example at this point, though in later chapters, we will replace it with a vector database to support RAG:

```
simulated_database = {
```

```
        "mediterranean diet": "The Mediterranean diet is rich in fruits,
vegetables, whole grains, olive oil, and fish. Studies show it is associated with
a lower risk of heart disease, improved brain health, and a longer lifespan. Key
components include monounsaturated fats and antioxidants."
 }
```

From the input message, we extract the research topic. The agent reads the content field of the MCP message and looks up the corresponding entry in the database. If the topic isn't found, it returns a default message:

```
research_topic = mcp_input['content']
  research_result = simulated_database.get(research_topic.lower(),
              "No information found on this topic.")
```

Now we define `system_prompt`. A prompt may seem simple, but within an MAS, it becomes a carefully engineered instruction. Here, the prompt tells the LLM to behave like a research analyst and condense the retrieved information into 3–4 concise bullet points:

```
system_prompt = "You are a research analyst. Your task is to synthesize the
provided information into 3-4 concise bullet points. Focus on the key findings."
```


The agent then calls our helper function, `call_llm`, passing in the system prompt and the research result. The summary generated by the LLM is stored in `summary`, and we print a log confirming that the work is complete:

```
summary = call_llm(system_prompt, research_result)
   print(f"Research summary created for: '{research_topic}'")
```

Finally, the agent packages the summary into a new MCP message using `create_mcp_message`. This ensures the output is consistent with the protocol and includes both the content and relevant metadata:

```
   return create_mcp_message(
sender="ResearcherAgent",
content=summary,
metadata={"source": "Simulated Internal DB"}
)
```

The message output, complete with content and metadata, is the Researcher agent's final output. We can now build our Writer agent, which acts as the system's content creator and will transform the summary into a blog post.

Defining the Writer agent

The second agent in our system is the Writer. It receives the summarized research from the Researcher agent and transforms those summary points into a short blog post.

We begin by defining the function. The `writer_agent` function also takes an MCP message as input. The activation log helps us track when the Writer starts its work:

```
def writer_agent(mcp_input):
```

```
    """
   This agent takes research findings and writes a short blog post.
  """
    print("\n[Writer Agent Activated]")
```

Next, we extract the research summary from the message. This is the content produced earlier by the Researcher agent:

```
research_summary = mcp_input['content']
```

We then create `system_prompt` that sets the Writer's role. Unlike the Researcher, which focused on factual synthesis, the Writer is instructed to adopt an engaging, informative, and

encouraging tone suitable for a health and wellness blog. The prompt also specifies the length (around 150 words) and asks for a catchy title:

```
system_prompt = "You are a skilled content writer for a health and wellness
blog. Your tone is engaging, informative, and encouraging. Your task is to take
the following research points and write a short, appealing blog post (approx. 150
words) with a catchy title."
```

The agent calls `call_llm`, passing in the system prompt and the research summary. The returned text is our draft blog post. We also print a message to confirm that the draft was created:

```
blog_post = call_llm(system_prompt, research_summary)
    print("Blog post drafted.")
```

Finally, the blog post is wrapped into a new MCP message using `create_mcp_message`. This ensures the output follows the same structured format as all other messages, with the content plus metadata; in this case, the word count of the draft:

```
    return create_mcp_message(
 sender="WriterAgent",
 content=blog_post,
 metadata={"word_count": len(blog_post.split())}
 )
```

This completes the Writer agent. It takes structured input from the Researcher, applies a new system prompt, and outputs a fully written blog post in a consistent MCP message format.

With both the Researcher and the Writer in place, we now have the core of our MAS defined. Each agent has a distinct role, and together they can carry out a task from raw information to finished content.

Building the Orchestrator

We now have our specialized agents, but we need a way to manage them. That role belongs to the **Orchestrator**. Think of it as the project manager of our AI team. Its job is to take a high level goal, break it into a sequence of tasks, and delegate those tasks to the right agent. It also manages the flow of information, taking the output from one agent and passing it as input to the next. This creates a seamless process from start to finish:

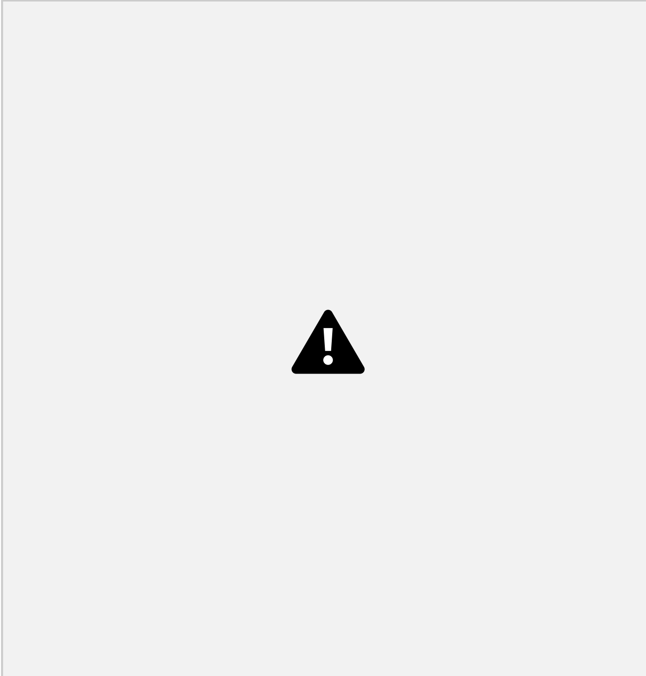


Figure 2.3: The Orchestrator of the agents

The workflow in the preceding figure begins when an **initial goal** is sent to the **Orchestrator**. The Orchestrator acts as a central hub, first sending a task to the **Researcher agent**. Once the research is complete, the agent sends its findings back to the Orchestrator. The Orchestrator then processes this information and sends a new task to the **Writer agent**. After the Writer finishes, it sends the completed content back to the Orchestrator. Finally, the Orchestrator assembles everything to produce the **final output**, completing the entire process.

The following code defines our Orchestrator. This single function manages the entire multi agent workflow from start to finish. Its role is to call the Researcher, then the Writer, and finally, assemble the output into a completed artifact. Let's walk through it step by step:

```
def orchestrator(initial_goal):
    """
 Manages the multi-agent workflow to achieve a high-level goal.
 """
    print("=" * 50)
    print(f"[Orchestrator] Goal Received: '{initial_goal}'")
    print("=" * 50)
```