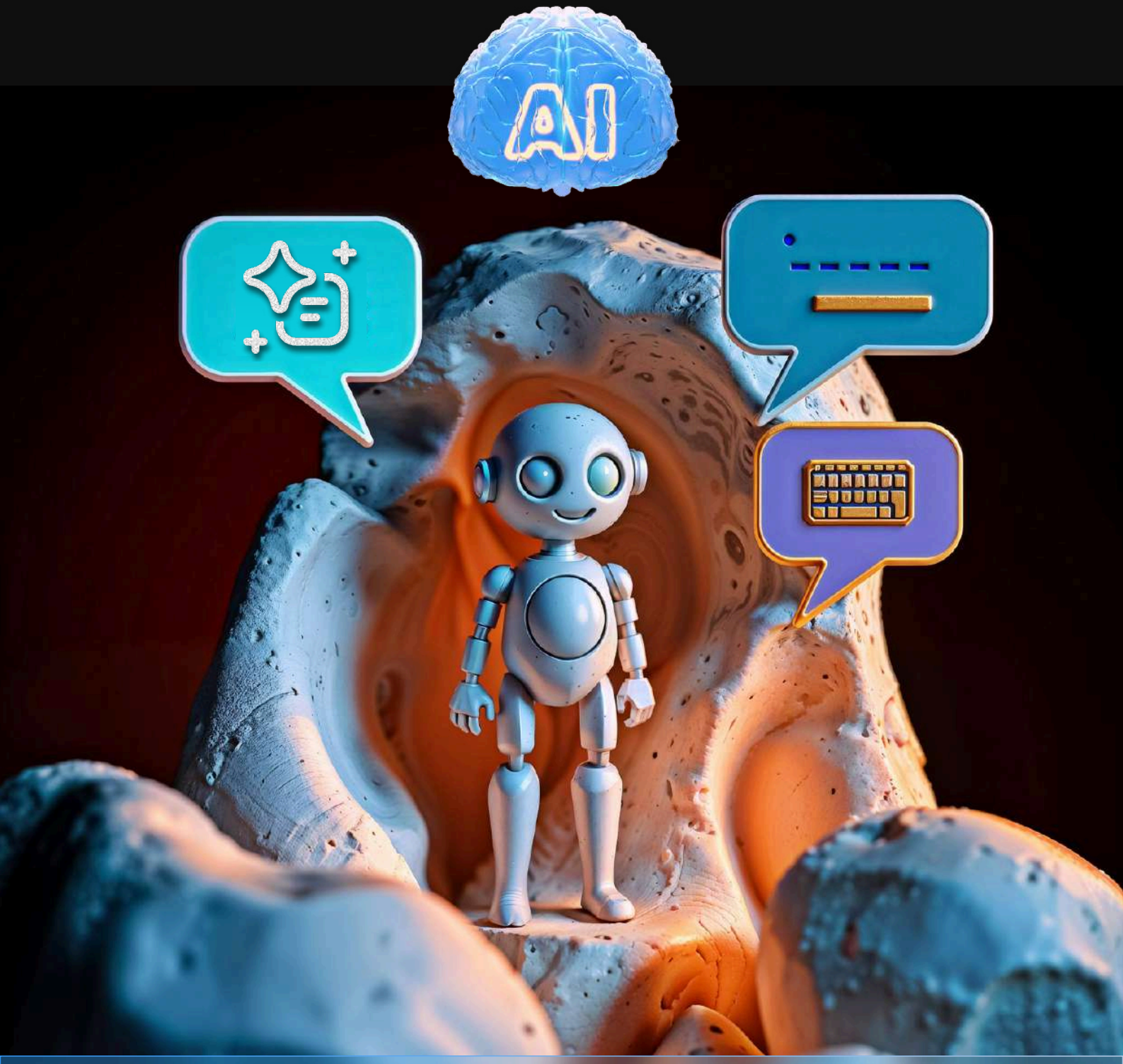
Blockchain Council™

The Beginner's Expedition into Generative and Agentic AI



A Practical Journey from Prompts to Agents

About
Blockchain Council

Embrace Web3, Metaverse & Blockchain with a Range of Certifications Offered by Blockchain Council, Designed to suit Professionals from all Backgrounds. Join our Community & Enjoy Networking Benefits to kickstart your Web3 Journey.

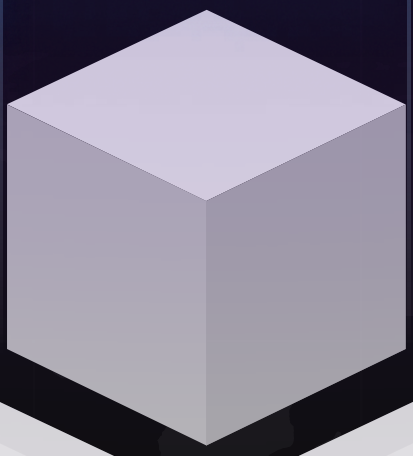
Blockchain Technology is more than just a tech. It is emerging rapidly and has a vast scope in the near future. The Blockchain has multiple use cases ranging from a financial network to a software or as a distributed ledger. Owing to this multitude of benefits and features, a plethora of companies are now shifting from a centralized and traditional working system to this trending and futuristic technology, “Blockchain”. Blockchain Council creates an environment and raises awareness among businesses, enterprises, developers, and society by educating them in the Blockchain space. We are a private de-facto organization working individually and proliferating Blockchain Technology globally.

With our Range of Coding & Non-Coding based Courses, you are sure to have an edge in this competitive Jobs market. Get a Functional understanding of Blockchain, Web3 and The Metaverse Through our Self-paced, Instructor - led or on site (Custom) Training Programs.

50000+
Cetified Professionals

From
127+
Countries

60+
Certifications



Copyright © Blockchain Council | All rights reserved

INDEX

Content	Page No.
Introduction	1
Part A: Foundations (Base Camp)	2-5
Chapter 1: What AI Is (and Isn't)	5-6
Chapter 2: The Map Reader and The Backpack	5-6
Part B: Generative AI (Learning to Navigate)	10-13
Chapter 3: Prompting Basics	5-6
Chapter 4: Few-Shot Prompting and Decomposing Tasks	5-6
Chapter 5: Retrieval Basics	5-6
Chapter 6: Tool Use	5-6
Chapter 7: Evaluation for Beginners (Did We Actually Arrive, or Just Feel Like It?)	5-6
Chapter 2: The Map Reader and The Backpack	
Part C: The Bridge (From Guide to Crew)	2-5
Chapter 8: From Prompts to Workflows (Repeatable Routes)	5-6
Chapter 9: Memory and State: What the Expedition Remembers Between Days	5-6

Copyright © Blockchain Council | All rights reserved

INDEX

Content	Page No.

Part D: Agentic AI (Running the Expedition) 10-13

Chapter 10: What “Agentic” Actually Means: Goals, Tools, Loops 5-6

Chapter 11: Planning and Task Decomposition: Route Planning That Survives Reality 5-6

Chapter 12: Tool Orchestration: Tool Choice, Tool Failure, Tool Confidence 5-6

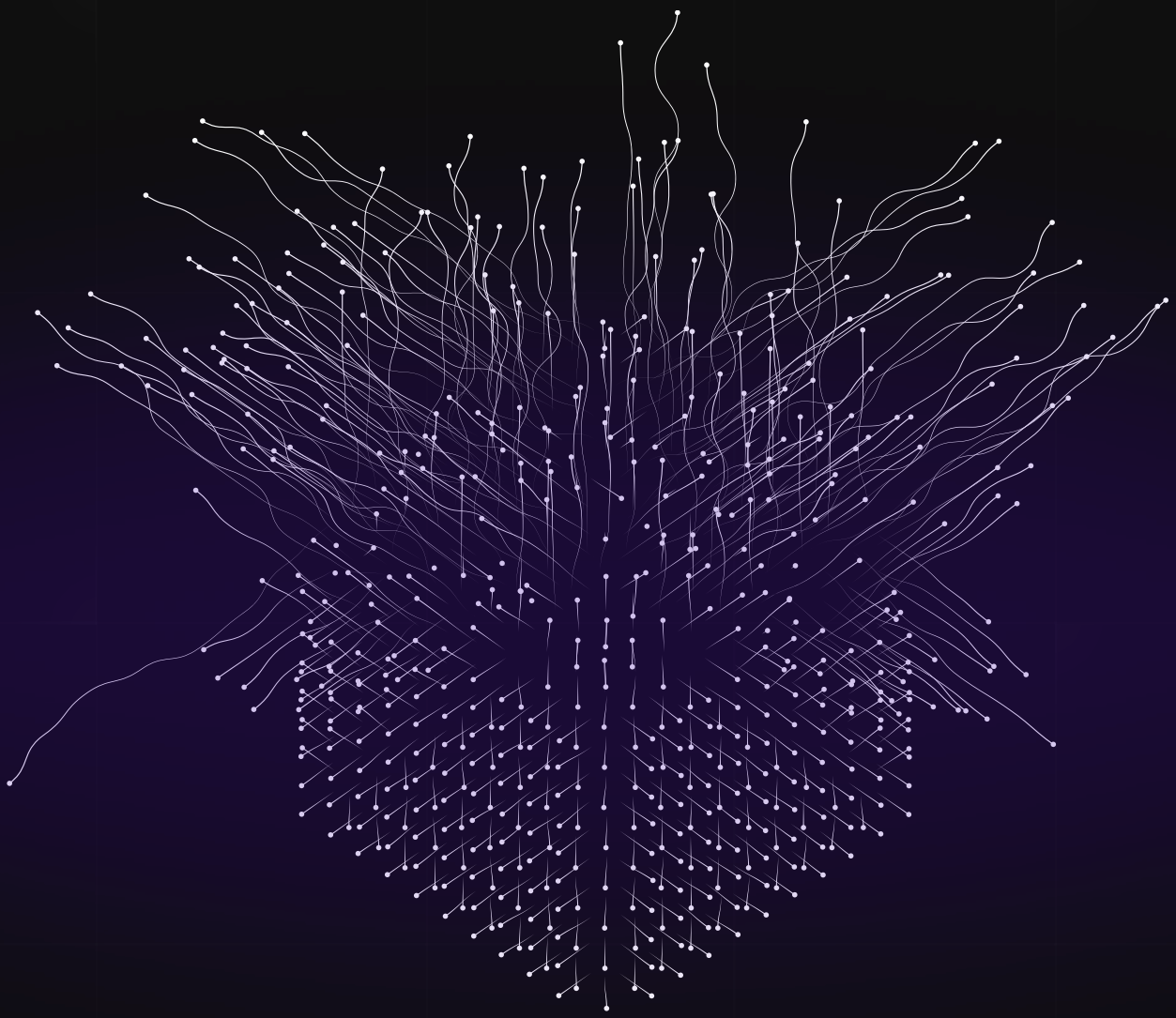
Chapter 13: Multi-Agent Basics: Specialist Crew, Not One Stressed-Out Hero 5-6

Chapter 14: Agent Debugging: When the Crew Gets Lost 5-6

Copyright © Blockchain Council | All rights reserved

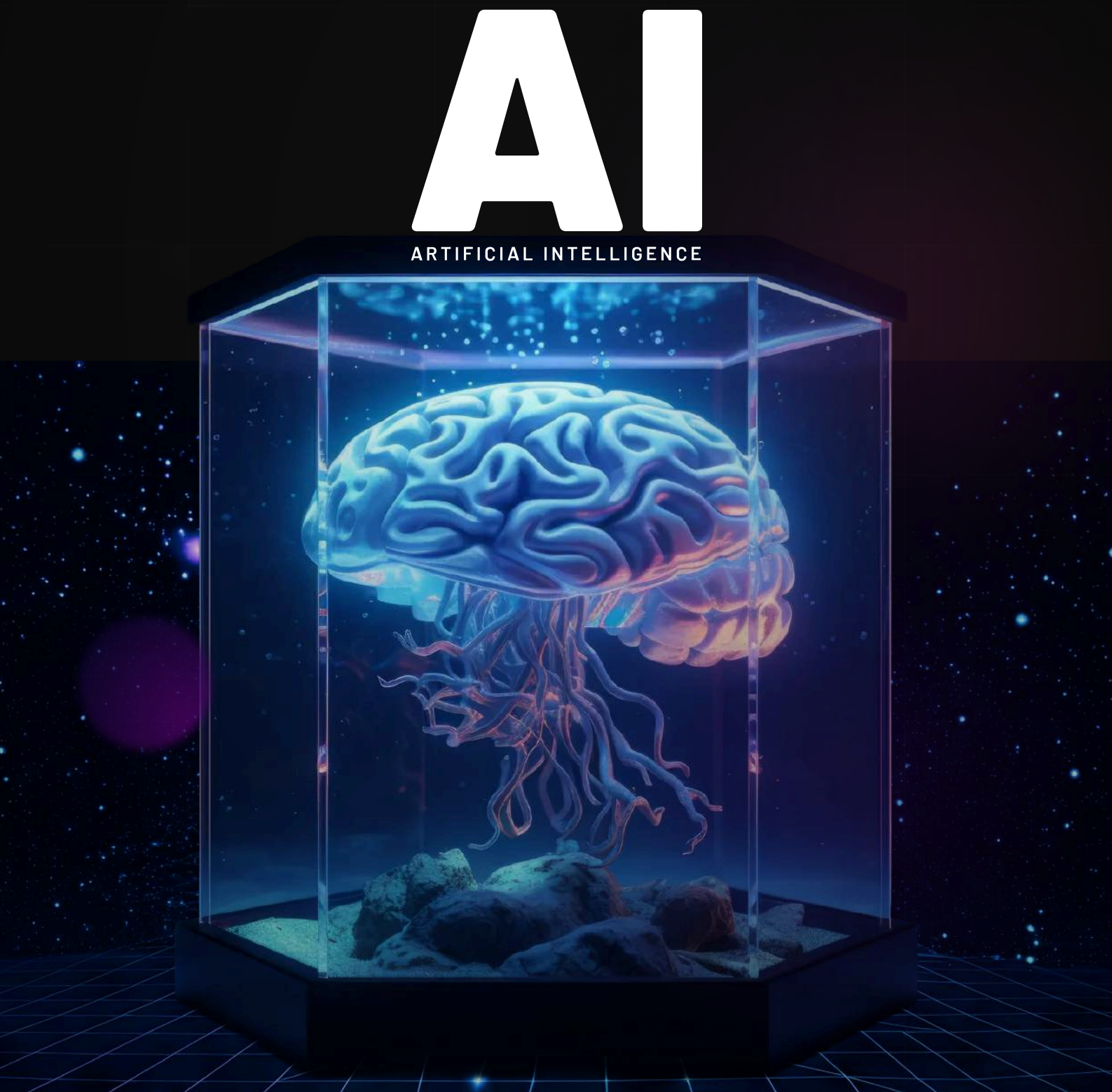
Introduction

Welcome to the fascinating world of Artificial Intelligence (AI). In today's fast-paced world, the demand for AI professionals is skyrocketing. As AI continues to revolutionize industries, being prepared for AI-related job interviews is essential for success in this competitive landscape. But are you ready to ace such interviews? Fear not, this eBook will help you conquer AI job interviews. In this eBook, you'll find 150 AI job interview questions and their answers tailored just for you. Whether you're aspiring to be an AI engineer, data scientist, machine learning specialist, or AI researcher, this eBook is your secret weapon to stand out from the competition.



Copyright © Blockchain Council | All rights reserved

Part A: Foundations (Base Camp)



Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

Meeting the Guide at Base Camp

You have arrived at a base camp with a rucksack, a notebook, and a mildly heroic sense that you are about to "learn AI". Somewhere in the distance, people are shouting things like neural networks, tokens, agents, and RAG as if those are normal dinner-table words. You nod politely, like you understand, while quietly wondering whether you're in the right place.

Good news! You do not need to be "a math person" or "a coding person" to start. You need a map, a few reliable instincts, and a way to distinguish between a genuine tool and a marketing illusion.

In this book, AI is your expedition. Generative AI is the guide who can write, summarize, code, and explain. Agentic AI is the stage where you stop giving single instructions and start running a crew with goals, tools, memory, and feedback loops. But before any of that, we need to do the most important thing beginners rarely get: define what AI actually is, what it is not, and how to think about it without getting trapped by jargon.

At base camp, you do not climb the mountain. You learn the terrain, you learn the rules, and you learn how to avoid pitfalls labelled "I thought AI was magic".

By the end of this chapter, you should be able to do three things comfortably. First, explain AI to a friend in plain English. Second, recognise common AI buzzwords and translate them into normal concepts. Third, use a generative AI tool as a beginner in a way that produces useful results and teaches you why those results happen.

That is our first leg of the journey.

Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

1.1 The Simplest Definition That Actually Holds Up

If you ask ten people what AI is, you will get twelve answers, and at least three of them will include the phrase "like humans". Let's make it simpler and more accurate.

AI is a set of methods that allow computers to perform tasks that usually require human judgment, by learning patterns from data or by searching through possibilities in a smart way.

That sentence is intentionally boring. Worry not, this expedition analogy will fix it right away!

AI is not a magical being. It is more like a guidebook plus a guide. The guidebook contains patterns learned from many journeys. The guide uses those patterns to help you decide what to do next on your specific route.

Importantly, AI is not one thing. It is a category, like "vehicles." A bicycle, a bus, and a spaceship are all vehicles, but you would not use them interchangeably unless you enjoy chaos. So when someone says "AI", your first reaction should be: Which kind? Doing what? With what data? Under what constraints?

That reaction alone will save you from most confusion.

Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

1.2 AI is not a Single Product, and it is not "the Robot."

A common beginner trap is thinking AI equals a chatbot, or AI equals a robot, or a single app. Those are just *interfaces*. The real thing underneath is usually a model, some data, and a system around it.

In expedition terms, the chatbot is the walkie-talkie. The model is the guide's training and instincts. The system is the whole expedition setup: supplies, weather checks, route planning, and the rule that you do not wander off alone at night.

You will hear phrases like "AI-powered" slapped onto everything from toothbrushes to spreadsheets. Treat that like a sticker on a suitcase. It might mean something, or it might mean "we added a button that calls an API".

To stay sane, separate three layers:

The *capability* layer is the underlying method or model.
The *product* layer is how you access it: an app, a website, a feature.
The *system* layer is how it works in the real world: data flows, tools, evaluation, and integration.

Most misunderstandings come from mixing these layers.

New users often mix up the *product layer* with the *system layer* or *capability layer*. However, it's a system of all three in perfect sync that amounts to what we call an AI chatbot or an AI Service or, in fact, Chat GPT itself.

Hence, users often interact with only the *product layer*, and understanding the underlying *system* and *capability layers* is essential for a complete understanding of how AI services function. This will be discussed in the subsequent chapters.

Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

1.3 AI vs Automation: The Difference Between a GPS Route and a Rigid Checklist

Another confusion: people call basic automation "AI". Sometimes that is fair; sometimes it is cheeky marketing.

Automation is when you define rules, and the computer follows them. "If this, then that." It is a checklist.

AI is when the computer's behavior adapts based on patterns or search, rather than only fixed rules. It is more like a GPS that suggests a route based on past traffic patterns, current conditions, and the goal you set.

Imagine you are hiking and you have two helpers.

Helper A holds a laminated card that says:
"If it rains, open the umbrella. If it is sunny, wear sunglasses."

Helper B looks at the clouds, checks the forecast, remembers that the trail gets slippery after rain, and suggests you take a safer route.

Helper A is automation. Helper B is closer to AI. Both are useful. The trick is knowing which one you are dealing with, because your expectations should change accordingly.

With automation, the rules are explicit. If it fails, you fix the rule.
With AI, the rules are learned or inferred. If it fails, you might need better data, different training, or a different approach.

Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

1.4 A Very Short History, Only Because it Helps the Map

We are not going to do a museum tour, but a tiny bit of history helps you understand why there are so many names.

Early AI leaned heavily on rules and logic. People tried to build intelligence by writing down knowledge as rules. That worked for narrow domains, like certain games or structured reasoning tasks, but it did not scale well to real-world scenarios.

Then, machine learning became the dominant approach. Instead of writing the rules, you let the system learn patterns from examples. That shift is why data became so valuable.

Deep learning accelerated this by using neural networks that could learn complex patterns from huge amounts of data, particularly in vision, speech, and language.

Generative AI is a more recent wave where models can produce new content, not just classify or predict. Instead of only saying "this is a cat", they can write a story about a cat, generate an image of a cat, or explain how to build a cat-themed mobile app if you are feeling dangerously motivated!

Agentic AI builds on that by adding planning, memory, and tool use so the system can pursue goals over multiple steps, rather than responding to one prompt at a time.

In expedition terms, rules-based AI is a rigid set of trail instructions. Machine learning is learning from previous hikers' tracks. Deep learning is learning from an enormous archive of journeys. Generative AI is a guide who can talk, sketch, and draft plans. Agentic AI is a guide who can also organize the crew, check supplies, and run the itinerary with feedback loops.

Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

1.5 The "AI Family Tree" You Can Keep in Your Head

Beginners often drown in labels. Let's give you a simple tree that will carry you through intermediate material later.

AI is the big umbrella. Under it, one major branch is machine learning, where systems learn from data.

Under machine learning, a major subset is deep learning, often using neural networks.

Under deep learning, one major category is foundation models, trained on large and diverse data, which can be adapted to many tasks.

A large language model, or LLM, is a foundation model focused on language.

Generative AI often uses foundation models to generate text, images, audio, or code.

Agentic AI is not a single model type. It is a system pattern. It uses models, tools, memory plus planning to pursue goals.

If you remember nothing else, remember this: Agentic AI is usually about systems, not just models.
That one line will make future chapters feel far less mysterious.

Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

1.6 What Does it Mean for a Model to "Learn"?

This is the point where people either get clarity or get scared. We are going for clarity.

When humans learn, we build mental patterns. We notice that touching fire hurts, so we do not do it again. We notice that certain words tend to appear together, so we predict what someone might say next.

Machine learning is similar in spirit. A model learns patterns by adjusting internal parameters so that it performs better on a task.

The model is not "storing facts" the way a database does. It is not memorizing the world like a perfect encyclopedia. It is compressing patterns.

In expedition terms, the model is not carrying a suitcase of printed maps for every possible trail. It is carrying an internal sense of how terrain tends to behave, based on thousands or millions of examples. That makes it flexible, but also fallible.

This explains a weird truth that beginners eventually notice: a model can be extremely convincing and still be wrong. It can generate something that sounds right because it matches patterns it has seen, even if it is not grounded in your specific situation.

That is not because it is evil or lazy. It is because pattern completion is not the same as truth.

You will learn later how to reduce that gap using retrieval, tools, and verification. For now, just recognize the difference.

Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

1.7 Training vs Inference: Packing The Guide's Knowledge vs Asking for Directions

Two words you will see everywhere are training and inference. They sound intimidating. They are not!

Training is the process by which a model's parameters are adjusted based on data. It is expensive, time-consuming, and usually done by the team that builds the model.

Inference is using the trained model to generate outputs for new inputs. That is what happens when you type a prompt and get a response.

Training is like the years your guide spent learning routes, weather patterns, and survival skills. Inference is when you ask the guide, "Which path should we take today?"

You, as a user, mostly live in inference. But you still benefit from understanding training because it explains why the guide behaves the way they do.

If you understand that the guide learned from many examples, you will stop expecting them to have perfect knowledge of your exact campsite.

Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

1.8 Narrow AI vs Artificial General Intelligence: The Mountain You are Not Climbing Today

People love the idea of AGI, artificial general intelligence, meaning a system that can handle most intellectual tasks a human can, across unfamiliar situations, with the same sort of flexible judgment. It makes for great debates because it sounds like a single finish line.

In practice, it is more like a whole mountain range with changing weather, missing signposts, and one person in the group insisting they "did it on YouTube".

For this book, you are not required to solve AGI. You are required to use today's AI effectively, and to understand what it can do reliably, what it can do sometimes, and what it cannot do without help. That difference matters because beginners often misread capability as general intelligence. When something is written fluently, it feels like a masterpiece. What you are seeing, most of the time, is a powerful pattern engine that has learned to produce plausible language and useful structures, not a human-style thinker with independent aims.

Most AI you will encounter is narrow AI. "Narrow" does not mean weak. It means the system is strong within certain task boundaries and conditions. A spam filter can be extremely competent at spotting spam, yet useless at planning your holiday. A chess engine can defeat grandmasters, yet it cannot explain your friend's sarcasm. Even modern generative models, which seem flexible, still operate within the constraints of their training, their context window, and the way they are asked to respond.

So why bring this up in a beginner book at all? Because it sets the right foundation for generative AI. Generative models are broad in output variety because language can describe almost anything. That breadth can look like general intelligence. But "broad outputs" is not the same as "general competence". The model can often explain, draft, summarize, and code across topics, yet it may struggle with things humans do quietly in the background: staying grounded to verified facts, keeping long-term goals stable across many steps, noticing when it is confused, and learning new personal information about your situation unless you provide it.

In expedition terms, narrow AI is like a specialist tool in your kit. A compass is brilliant at direction, hopeless at cooking. Generative AI is like a guide who has read an absurd number of expedition journals and can speak about almost any trail with confidence. That is genuinely useful. But it is still not a human teammate with persistent intentions.

Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

1.8 Narrow AI vs Artificial General Intelligence: The Mountain You are Not Climbing Today

It does not "want" anything on its own. It does not carry goals across days unless you build a system that gives it memory and a plan.

If you ask it to physically carry your rucksack, you will still be carrying your rucksack. If you ask it to plan the route, it can help a lot, but you still need to check the map when the cliff edge is nearby.

This is not a criticism. It is a user manual. Once you hold this frame, the next chapters become easier: you will learn how to brief the guide properly (prompting), how to give it your field notes (context and retrieval), and later, how to turn it into an expedition leader pattern (agentic systems) by adding goals, tools, memory, and feedback loops.

Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

1.9 So What is Generative AI, Really?

Generative AI is the category of AI systems that can create new content: text, images, audio, code, and more.

But the key is not the word "create." The key is to *generate plausible outputs based on learned patterns.*

A language model generates text by predicting what token is likely to come next, given the context. "Token" is a unit of text, roughly like chunks of words or parts of words. We will go deeper later, but this is enough for now: it predicts the next chunk, then the next, then the next, until it has produced a response.

That sounds underwhelming until you realize how far you can get with very good prediction at a very large scale.

In expedition terms, the model is like a guide who has read so many trail journals that they can continue a journal entry in a coherent style, even if they have never visited your exact trail. That is why it can write a business email, a poem, a lesson plan, and a troubleshooting guide with suspicious confidence.

Suspicious confidence is part of the package.

Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

1.10 What is Agentic AI, at a Beginner Level?

Agentic AI is when you take generative AI and wrap it in a system that can pursue a goal through multiple steps, using tools, memory, and feedback.

If a chatbot answers a question, that is a single step.

If a system can break a goal into tasks, decide which tool to use, run the tool, inspect the result, adjust its plan, and continue until it reaches a stopping condition, that starts to look agentic.

In expedition terms, generative AI is a guide who can answer whatever you ask in conversation. Agentic AI is a guide who can also plan the route, assign roles, check supplies, book transport, and keep a log of what worked, all while you sleep.

We are not building that yet. We are simply planting the idea, so the transition later feels natural.

Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

1.10 What is Agentic AI, at a Beginner Level?

Agentic AI is when you take generative AI and wrap it in a system that can pursue a goal through multiple steps, using tools, memory, and feedback.

If a chatbot answers a question, that is a single step.

If a system can break a goal into tasks, decide which tool to use, run the tool, inspect the result, adjust its plan, and continue until it reaches a stopping condition, that starts to look agentic.

In expedition terms, generative AI is a guide who can answer whatever you ask in conversation. Agentic AI is a guide who can also plan the route, assign roles, check supplies, book transport, and keep a log of what worked, all while you sleep.

We are not building that yet. We are simply planting the idea, so the transition later feels natural.

Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

1.11 The Three Illusions that Confuse Beginners (and How to Avoid Them)

Before we touch any tools, we need to talk about three common illusions.

Illusion one: fluent equals correct.
A model can sound like a confident expert while generating nonsense. Fluency is not proof.

Illusion two: one good answer means it "knows".
Sometimes you get a great response because your prompt was clear and the task matched its strengths. That does not mean it will always work. It means you found a good setup.

Illusion three: the model is a person.
It can adopt a tone, a persona, and a "voice", but it does not have intentions, experiences, or private knowledge of your life unless you provide it. Treating it like a person leads to weird expectations.

In expedition terms, your guide can talk smoothly, but that does not mean they have walked your trail yesterday. They might be extrapolating from other trails.

Your job is not to distrust the guide. Your job is to verify when the terrain is risky.

Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

1.12 The "Field Kit" for Reading AI Jargon Without Panic

Now we build your first jargon translator. You will see these terms in intermediate materials. Here is what they mean in normal language, plus the expedition translation.

A **model** is a learned pattern machine that maps inputs to outputs. In expedition terms, it is the guide's internal skill, not the walkie-talkie.

Parameters are the numbers inside the model that shape its behavior. In expedition terms, they are the guide's instincts formed by experience.

Data is the example used to train the model. In expedition terms, they are the trail journals and maps the guide studied.

A **prompt** is the input you provide, including instructions and context. In expedition terms, it is your briefing to the guide.

Context is the information available to the model at the moment. In expedition terms, it is what is currently on the table at camp, not everything in the world.

Inference is running the model to get an output. In expedition terms, it is asking for directions right now.

A **hallucination** is a plausible but incorrect output. In expedition terms, it is the guide confidently describing a bridge that does not exist on this trail.

Fine-tuning is training a model further on specific data to shape its behaviour. In expedition terms, it is training the guide for your particular region.

Retrieval is fetching relevant information from a source and giving it to the model. In expedition terms, it is opening your field notes before answering.

A **tool** is an external capability like a calculator, a database query, a code runner, or a search function. In expedition terms, it is giving the guide equipment.

A **workflow** is a repeatable sequence of steps. In expedition terms, it is a standard route plan.

An **agent** is a system that can pursue a goal using planning, tools, and iteration. In expedition terms, it is the expedition leader pattern.

Do not try to memorize this like a school exam. The goal is recognition. When you see these words later, you want to think, "Ah, that's just the guide's kit."

Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

1.13 A Practical Mental Model: Inputs, Outputs, and The Missing Middle

Most beginners think AI works like this: you ask, it answers.

The more useful model is: you give it a goal, constraints, context, and a format, and it generates a candidate output that you then evaluate and refine.

That sounds like a lot, but it is exactly how humans work when collaborating.

When you ask a human colleague for help, you do not just say "make it good". You say what you want, who it is for, what constraints exist, and what "good" looks like. The same is true here.

In expedition terms, if you tell the guide, "Take me to the top", you will get a vague plan. If you tell the guide, "Take me to the top using the safest route, with two rest stops, avoiding snow, and I have a knee injury", you will get something useful.

So, as a beginner, your superpower is not a technical skill. It is a *briefing skill*.

We will build that skill through the first implementation exercise.

Copyright © Blockchain Council | All rights reserved

1.14 Let's Implement: Your First "AI Guide" and a Reality Check

You can do this with any modern generative AI chat tool. No coding required. If you do not have access to one right now, you can still follow along mentally, because the point is not the tool, it is how you interact with it.

The exercise has three parts. First, you will create a reliable guide persona. Second, you will test what it is good at. Third, you will deliberately trigger a failure so you learn its limits early, while the stakes are low.

Part A: Create the guide persona (your briefing)

Open your AI chat tool (ChatGPT, Gemini, Grok, etc.) and paste this prompt as your first message:

"Act as my expedition guide for learning AI from scratch. Speak in simple language. When you use a technical term, define it immediately in one sentence. Ask me one clarifying question if my request is ambiguous. If you are unsure, say you are unsure and suggest how to verify. At the end of each answer, give me a one-paragraph 'field note' summary."

What you should see: a friendly, structured response that confirms the role and starts guiding you.

Why this works: you gave the model a role, constraints, behavior rules, and an output structure. You did not just ask for information. You set up a collaboration.

In expedition terms, you just told your guide how you want the journey to run.

Part B: Test a task that it is genuinely good at

Now ask:

"Explain the difference between AI, machine learning, and deep learning using the expedition analogy. Then give me three everyday examples of each, and for each example, tell me what the input and output are."

What you should see: a clear explanation with examples. The best responses will explicitly separate the categories and will not mix them up.

Copyright © Blockchain Council | All rights reserved

1.14 Let's Implement: Your First "AI Guide" and a Reality Check

If the answer is messy, do not give up. Improve the briefing by adding a constraint such as:

"Keep each definition to three sentences, and each example to two sentences."

This teaches you a core truth: you are not "using a chatbot". You are steering a generator.

Part C: Trigger a controlled failure (so you recognize it later)

Now ask something that tempts confident guessing. For example:

"List the exact internal architecture and training dataset used for the latest version of this tool, including the full dataset sources."

What you should see, ideally: a refusal, uncertainty, or a general answer that admits limitations.
What you might see: a confident-sounding answer that is partially fabricated.

If you receive a confident answer with specific dataset names and numbers, treat that as a teaching moment. Ask:

"Which parts of that are confirmed, and which parts are assumptions? Provide sources for confirmed parts. If you cannot provide sources, say so."

This second prompt forces the model into a more cautious mode. It also shows you how hallucinations appear: not as random gibberish, but as smooth specificity.

In expedition terms, you just asked your guide about a bridge they may not have seen. Your follow-up was checking whether they actually saw it or were guessing from similar trails.

Copyright © Blockchain Council | All rights reserved

1.14 Let's Implement: Your First "AI Guide" and a Reality Check

Expected results and what they mean

If the tool responds well to your role instructions, you have learned that behaviour can be shaped by prompting. That is a key foundation for generative AI.

If the tool produces a strong explanation in Part B, you have learned where generative models shine: summarizing, explaining, reformatting, and producing structured outputs.

If the tool struggles with Part C, you have learned a critical limitation: models can generate plausible details that are not grounded in verified facts, especially about unseen, proprietary, or very specific information.

Do not feel disappointed by that. Feel equipped. You have just learned the difference between a guide's pattern knowledge and a factual GPS reading.

Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

Bonus mini-experiment: one prompt, three quality levels

Ask the same question in three ways.

First: "Explain AI."
Second: "Explain AI to a 12-year-old in 150 words."
Third: "Explain AI to a 12-year-old in 150 words, using the expedition analogy, defining any technical term in one sentence, and ending with a 2-question quiz."

Compare the outputs. You will feel the difference immediately.

That difference is the "prompting lever". You will spend the next chapters learning to pull it on purpose.

Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

1.15 What You Just Learned, Translated into Intermediate Jargon

You have already touched concepts that intermediate books assume you know. Let's name them.

When you defined the guide persona, you were **following** instructions and shaping output via **system-like constraints.**

When you improved the messy answer by adding word limits, you were doing **output** control via **constraints**.

When you asked for input and output for each example, you were forcing the model to create a clear **task framing.**

When you triggered a failure and asked what was confirmed, you were doing **uncertainty elicitation** and basic **verification prompting.**

Those phrases might show up later in some "jargonish" text. Now, as you have developed an understanding of the terms, they won't look scary.

In expedition terms, you have learned to brief your guide, check the map, and question a suspicious bridge. That is real progress.

Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

1.16 A Beginner-Friendly Way to Think About "Intelligence" in Machines

People get stuck on the word "intelligence". They imagine something with intentions and understanding like a human.

A more useful lens is this: AI is competent at tasks under constraints.

Some AI systems are competent at recognizing objects in images. Some are competent at predicting the next word. Some are competent at recommending products. Some are competent at controlling a robot arm. Each competence is shaped by data, training, and system design.

Generative AI feels especially "intelligent" because language is how humans express thinking. When a system talks well, we instinctively attribute deeper understanding. That is normal human psychology.

Your job as a builder, even at a beginner level, is to keep two thoughts in your head at once:

This system can be extremely helpful.
This system can be confidently wrong.

That combination is not a contradiction. It is the user manual.

Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

1.17 Common Questions Beginners Have

"Is AI just copying?"
An AI model learns patterns from data and generates new outputs based on those patterns. It is not a simple copy-paste, but it is also not human creativity. Think of it as remixing patterns at scale.

"Does it understand what it says?"
AI models produce language that often matches understanding-like behavior. Whether you call that "understanding" depends on your definition. Practically, treat it as a powerful pattern-based assistant that needs grounding and verification.

"Why does it sometimes make things up?"
The job of an AI model is to produce plausible outputs, and if it lacks information, it may fill gaps with pattern-matching guesses unless you constrain it or provide sources.

"Will I need math for this book?"
Not to start. Later, you will meet concepts that have maths under the hood, but we will treat maths like the engine in a car. You should understand what it does, even if you are not machining the pistons.

Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

1.18 Field Notes: Your Base Camp Checklist For the Rest of the Book

From here on, whenever you meet a new AI concept, ask these questions:

What task is it trying to do?
What inputs does it need?
What output does it produce?
What makes it fail?
How do we verify results?

If you can answer those, you are no longer a confused spectator. You are a competent traveller.

Copyright © Blockchain Council | All rights reserved

Chapter 1: What AI Is (and Isn't)

1.19 Chapter Wrap: Leaving Base Camp with the Right Expectations

You started this chapter at base camp with a vague word, "AI", floating above everything like a cloud.

Now you have a sturdy working map.

AI is a category of methods that let computers perform judgment-like tasks through learning patterns or smart search. Machine learning learns from data. Deep learning uses neural networks for complex pattern learning. Generative AI produces new content, often through next-token prediction at scale. Agentic AI is a system pattern that adds goals, tools, memory, and iteration to pursue outcomes over multiple steps.

You also learned the most practical beginner skill: briefing. You can shape outputs by giving roles, constraints, context, and formats. You can test capabilities and trigger controlled failures so you recognize limitations early.

In expedition terms, you have met the guide, learned what the guide can and cannot do, and packed the field kit you will use on every trail from here.

Next chapter, we will look inside the guide's "map-reading" ability in a beginner-friendly way: what a language model is doing when it produces text, why it can seem smart, and how to think about its strengths without getting fooled by fluency.

Base camp is behind you. The path starts now.

Copyright © Blockchain Council | All rights reserved

Chapter 2: The Map Reader and The Backpack

How Language Models Generate Text, Why They Sometimes Guess, and Why Context Is Everything

You left base camp in Chapter 1 with three useful things: a working definition of AI, a clear sense of what today's systems can and cannot do, and a first taste of how prompt wording changes output quality. You probably noticed something slightly spooky: tiny changes in your instructions produced noticeably different answers, as if the "guide" suddenly became sharper, calmer, or more confused.

This chapter is where we open the guide's rucksack and look inside.

Not with math-heavy diagrams, and not with research-paper vocabulary. Instead, we will build a beginner-friendly mental model that stays true even when you later read more technical material. The goal is simple: when you come across terms like tokens, context window, probability distribution, logits, attention, or sampling, you should not freeze. You should think, "Ah, that's the map-reader mechanism, and that's the size of the table at camp."

In expedition terms, Chapter 1 introduced the guide. Chapter 2 explains how the guide reads maps, why the guide sometimes fills in missing terrain with confident guesses, and why the guide can only work with what you actually put on the table.

By the end of this chapter, you should be able to do four things comfortably. First, explain what a language model is doing when it "writes". Second, understand what tokens are and why they matter for cost and behavior. Third, understand what context is, why models forget, and how to manage that. Fourth, run a few simple experiments to see these mechanics with your own eyes.

Let's get started!

Copyright © Blockchain Council | All rights reserved

Chapter 2: The Map Reader and The Backpack

2.1 The Core Idea: The Guide Predicts The Next Step, not The Whole Journey

A large language model (like ChatGPT, Gemini, etc.), or LLM, is best understood as an extremely advanced next-step predictor. When you give it text, it does not "think" like a human in the background and then type a final answer. It produces a response one small piece at a time, choosing what comes next based on patterns it learned during training and the context you gave it right now.

That small piece is usually a token, which you will meet properly in a moment. For now, imagine the guide doing this:

You say, "We are hiking to the summit tomorrow. What should we pack?"
The guide does not jump straight to a final list in one mental leap. It starts producing the response step by step, continually making the next choice that fits the context and the patterns it has learned about hiking, weather, packing, and helpful explanations.

This is why generative AI is so good at writing, summarizing, explaining, and drafting. Those tasks are essentially structured prediction tasks. Good prediction at a massive scale looks like intelligence.

It also explains why models sometimes "hallucinate", meaning they produce plausible but incorrect details. If the model does not have enough grounded information, it can still generate something that matches the shape of a good answer. The guide fills in the blank on the map because it has seen many similar maps before. Sometimes it fills it correctly. Sometimes it invents a bridge that is not there.

So the first foundational rule of generative AI is this: **the model produces plausible continuations, not guaranteed truths.**

Once you hold that rule, the rest of the behavior becomes much easier to interpret.

Copyright © Blockchain Council | All rights reserved

Chapter 2: The Map Reader and The Backpack

2.2 Training: How The Guide Learned The Patterns in The First Place

Training is the guide's long apprenticeship. It is the period where the model is shown enormous amounts of data and is repeatedly adjusted so it becomes better at predicting what comes next

You do not need to memorize the math, but you do need the concept: training is an optimization process.

At a high level, training looks like this.

The model reads some text.
It tries to predict the next token.
It gets feedback: right or wrong, and by how much.
It adjusts internal numbers to reduce future errors.
It repeats this millions or billions of times.

Those internal numbers are called parameters or weights. Think of them as the guide's instincts. The guide is not storing every trail journal as a separate file. The guide is compressing patterns into an internal skill.

This is why people say models "learn from data," but do not behave like a database. A database stores exact facts that you can retrieve precisely. A model stores patterns that it can generalize from.

That generalization is powerful. It is also the reason you must be cautious with precise claims. A model can be excellent at explaining what a database is, and still fail at recalling an exact line from a specific document you never gave it.

In expedition terms, training is the guide studying thousands of maps and journals. Inference is the guide helping you on today's trail using whatever is currently on the table.

Copyright © Blockchain Council | All rights reserved

Chapter 2: The Map Reader and The Backpack

2.3 Inference: How Your Prompt Becomes an Answer

Now we move from apprenticeship to the moment you actually ask the guide for help.

When you type a prompt into a chat tool, several things happen before you see a response.

First, your text is broken into tokens.
Second, those tokens are converted into numerical representations.
Third, the model processes the sequence and produces a score for many possible next tokens.
Fourth, the system chooses the next token using a selection method.
Fifth, the chosen token is added to the growing response, and the cycle repeats.

This loop continues until the model reaches a stopping point, such as an end-of-response marker or a token limit.

Two details matter for beginners.

One, the model does not generate the whole answer “all at once”. That is why it can sometimes drift mid-response, contradict itself, or accidentally change its mind. The generation is a living process.

Two, there is usually an element of choice in how the next token is selected. Some systems can be configured to be more deterministic, others more varied. You might hear terms like temperature, top-k, or top-p later. For now, just remember the idea: the guide can be instructed to follow the most likely route, or to explore more creative paths. Both can be useful, depending on the terrain.

In expedition terms, you are not receiving a carved stone tablet. You are listening to a guide speak in real time, step by step, with a blend of learned patterns and context cues.

Copyright © Blockchain Council | All rights reserved

Chapter 2: The Map Reader and The Backpack

2.4 Attention: The Guide Decides What Matters in The Conversation

Getting into explicit specifics of generative AI, the word attention often pops up. It sounds dramatic. In practice, it is a mechanism that helps the model decide which parts of the input matter when producing each new token.

A beginner-friendly way to picture attention is this: when the guide is answering, it keeps glancing back at different parts of what you said, weighing some parts more than others. If you say, “Explain AI as if I were a twelve-year-old, and keep it under 150 words,” the guide should keep glancing at “as if I were twelve” and “under 150 words” while writing.

If it forgets, it might start writing a 500-word lecture, which is the expedition equivalent of walking off the trail while confidently announcing you are still on it.

Attention is one reason modern models handle long contexts better than older approaches. But attention is not magic. If the context is too long, or if your instructions are buried under noise, the guide will still miss things.

So the practical lesson is: make important instructions easy to notice. Put them on early. Make them explicit. Repeat them when needed. Remove unnecessary clutter.

If you remember only one line from this section, make it this: **important instructions should be high-signal, not hidden inside a forest of text.**

Copyright © Blockchain Council | All rights reserved

Chapter 2: The Map Reader and The Backpack

2.5 Tokens: The Basic Unit of “Packing” and “Pricing.”

Now we meet tokens properly, because tokens are the unit that quietly runs everything.

A token is a chunk of text used by the model. It is not exactly a word. Sometimes a token is a whole word. Sometimes it is part of a word. Sometimes it is punctuation. It depends on the tokenizer.

Here is the practical impact: the model sees and produces tokens, not words.

Why does this matter?

Because token counts influence three things you care about as a user or builder: how much you can fit into the model’s context, how much it costs to run, and how the model behaves with different languages and writing styles.

You can think of tokens as the weight and volume of your supplies. You can only carry so much. If you overpack, you either pay more, slow down, or start leaving things behind.

A few beginner-friendly facts that become useful later.

Short, common words often become single tokens.
Long or unusual words may become multiple tokens.

Numbers, code, and URLs can be token-heavy.
Some languages produce more tokens for the same meaning, depending on the tokenizer.

This is why one person can paste a page of text and be fine, while another person pastes a similar-looking page full of code and suddenly hits limits. The backpack is filling up faster.

Do not treat token counting as a chore. Treat it as knowing how much space you have on the table at camp.

Copyright © Blockchain Council | All rights reserved

Chapter 2: The Map Reader and The Backpack

2.6 Context: What is on The Table Right Now, and What is Not

Context is the information the model can “see” in the current interaction. In a chat, the context usually includes your recent messages, the assistant’s recent messages, and sometimes hidden instructions from the system.

Here is the key beginner idea: **the model cannot use information it cannot see.**

If you had a conversation yesterday, and you do not paste that information today, the model may not have access to it. If you reference a document but do not include the relevant excerpt, the model cannot read your mind. If you say, “Use the same format as earlier,” and “earlier” is not in the context window, the model will guess what you mean.

In expedition terms, the context is what is laid out on the map table. If the map is not on the table, it is not usable. The guide might still talk as if it is, because the guide is trained to be helpful, but that talk becomes guesswork.

This is one of the most empowering beginner lessons: when an AI output is weak, it is often not because the model is “bad”. It is because the guide did not have the right map in front of them.

Copyright © Blockchain Council | All rights reserved

Chapter 2: The Map Reader and The Backpack

2.7 The Context Window: How Big is the Table

The context window is the maximum amount of text the model can consider at once, measured in tokens.

If your conversation, along with the new prompt, exceeds the window, something has to give. Typically, older parts of the conversation are truncated or summarized. Different products handle this differently, but the practical outcome is the same: the guide starts forgetting earlier details because they have been physically removed from the table.

This is why models can feel like they have short-term memory. They do, but it is limited by the size of the table.

It also explains something that frustrates beginners: you can carefully define rules at the start of a long chat, and then later the model stops following them. Often, it is not rebellion. It is truncation. The rules fell off the table.

So what do you do?

You do what real expedition leaders do. You keep a clear mission brief visible, and you keep your notes tidy.

That means restating key constraints when you reach a new phase, using clear structure, and keeping your instructions compact enough to survive long interactions.

Copyright © Blockchain Council | All rights reserved

Chapter 2: The Map Reader and The Backpack

2.8 Why Output Quality Changes With “Tiny” Prompt Changes

You already saw this in Chapter 1, so we will not repeat the same experiment. Instead, we will explain the mechanism behind what you observed.

When you change a prompt, you change the context the model conditions on. That changes which patterns become most likely.

If you say, “Explain AI,” the model has enormous freedom. It can choose any route: history, philosophy, examples, warnings, jokes. The probability landscape is wide.

If you say, “Explain AI to a 12-year-old in 150 words using the expedition analogy and end with two quiz questions,” you have narrowed the landscape. You have put signposts on the trail. The model now has stronger constraints that make certain token sequences more likely than others.

This is also why asking for a format often improves quality. A format is a set of rails. It reduces drift.

In expedition terms, your prompt is the route plan. A vague route plan creates wandering. A structured plan creates forward motion.

Copyright © Blockchain Council | All rights reserved

Chapter 2: The Map Reader and The Backpack

2.9 The Hidden Tug-Of-War: Conflicting Instructions and Noise

Another beginner's surprise is that models sometimes ignore clear instructions. There are a few common reasons.

Sometimes your instructions conflict. You might say, “Be extremely detailed,” and later, “Keep it short.” The model will pick one, blend them, or oscillate. Humans do the same, to be fair.

Sometimes your instruction is buried in a wall of text. The model is trying to be helpful, but the signal is weak.

Sometimes the instructions are ambiguous. “Make it professional” could mean formal language, or simply correct spelling, or a polite tone, or all of the above.

Sometimes the tool has higher-level constraints that override your request. For instance, many tools avoid certain types of content regardless of user instructions. That is the system layer doing its job.

The practical fix is nearly always the same: simplify, clarify, and restate the priority.

In expedition terms, if you mumble the route plan while the wind is howling, the guide may hear “go left at the ridge” as “go get lunch at the village”. Both are technically plans, but only one gets you up the mountain.

Copyright © Blockchain Council | All rights reserved

Chapter 2: The Map Reader and The Backpack

2.10 “But it sounded so Confident”: Why Confidence is Not Evidence

This section is worth a slower walk because it is the trap that hits beginners hardest.

Language models are trained to produce fluent, coherent text. Fluency is part of the product. Confidence is often a side-effect.

The model can be highly confident in tone while uncertain in substance, because the generation process rewards plausibility, not humility.

That does not mean you should distrust everything. It means you should adopt a simple verification habit:

When the terrain is low-risk, enjoy the speed.
When the terrain is high-risk, ask for sources, ask for assumptions, ask for alternative views, or verify externally.

When the task is of the simple implementation sort, you already know the kind of result, and it is typically a low-risk job, you may not want to deploy verification measures. However, in a typical scenario, a job requires the model to research, analyze, develop, and perform multiple tasks. The user must check the sources and the reasons behind certain assumptions and decisions taken by the model. Rather, a better way is to include information like “mentioning the source”, “reasoning behind assumptions”, “rather than guessing, ask a clarifying question”, etc into the prompt itself. Keeping in check the confident guesswork results.

In expedition terms, if the guide says, “This trail is safe,” you might believe it on a sunny day with plenty of time. If the guide says, “Cross this icy ridge in the dark,” you check the map twice, and you ask whether anyone else has tried it recently.

That habit is the difference between being “impressed by AI” and being effective with AI.

Copyright © Blockchain Council | All rights reserved

Chapter 2: The Map Reader and The Backpack

2.10 “But it sounded so Confident”: Why Confidence is Not Evidence

This section is worth a slower walk because it is the trap that hits beginners hardest.

Language models are trained to produce fluent, coherent text. Fluency is part of the product. Confidence is often a side-effect.

The model can be highly confident in tone while uncertain in substance, because the generation process rewards plausibility, not humility.

That does not mean you should distrust everything. It means you should adopt a simple verification habit:

When the terrain is low-risk, enjoy the speed.
When the terrain is high-risk, ask for sources, ask for assumptions, ask for alternative views, or verify externally.

When the task is of the simple implementation sort, you already know the kind of result, and it is typically a low-risk job, you may not want to deploy verification measures. However, in a typical scenario, a job requires the model to research, analyze, develop, and perform multiple tasks. The user must check the sources and the reasons behind certain assumptions and decisions taken by the model. Rather, a better way is to include information like “mentioning the source”, “reasoning behind assumptions”, “rather than guessing, ask a clarifying question”, etc into the prompt itself. Keeping in check the confident guesswork results.

In expedition terms, if the guide says, “This trail is safe,” you might believe it on a sunny day with plenty of time. If the guide says, “Cross this icy ridge in the dark,” you check the map twice, and you ask whether anyone else has tried it recently.

That habit is the difference between being “impressed by AI” and being effective with AI.

Copyright © Blockchain Council | All rights reserved

Chapter 2: The Map Reader and The Backpack

2.11 Grounding: How to Keep The Guide Tethered to Your Reality

Grounding is the practice of anchoring the model's output in information that is relevant and verifiable for your specific task.

Without grounding, the model relies on general patterns learned during training. That can be great for explanations and drafts. It can be risky for specifics.

With grounding, you provide the key material directly, or you use a system that retrieves it from a trusted source, and you ask the model to operate only within that material.

In expedition terms, grounding is placing your actual map on the table, not asking the guide to describe the terrain from memory.

There are simple beginner ways to do grounding even before you build fancy systems.

You can paste a paragraph and ask, "Summarize only what is here."
You can paste requirements and ask, "Write a plan that satisfies every requirement, and list any requirement you could not satisfy."
You can paste two documents and ask, "Compare them, citing which sentence supports each claim."

Notice the pattern: you are giving the guide field notes, then asking the guide to stay within them.

Later in the book, this evolves into retrieval systems and tool use. For now, the concept alone is enough: **better context equals better outputs.**

Copyright © Blockchain Council | All rights reserved

Chapter 2: The Map Reader and The Backpack

2.12 Cost, Latency, and Why Long Chats Get Expensive

Even if you are not personally paying per token, the systems you use are paying in computation, and that shapes product behaviour.

Token costs matter because both input tokens and output tokens require processing. A longer prompt costs more. A longer answer costs more. A longer conversation costs more because every new response often reprocesses a chunk of the prior context.

This is the practical reason you will see limits like: “maximum message length” or “conversation too long”.

It is also why “keep it concise” is not just a stylistic preference. It is a performance preference.

In expedition terms, you can carry everything, but you will move more slowly, you will spend more energy, and at some point, you will be forced to drop something. The best expeditions are not the ones with the biggest bags. They are the ones with the right bags.

So, as you become a more advanced user, you learn to pack intelligently: give enough context to be precise, but not so much that you drown the model in noise or hit limits.

Copyright © Blockchain Council | All rights reserved

Chapter 2: The Map Reader and The Backpack

2.13 A Beginner's Method for "Packing" Prompts Properly

As we are building you toward intermediate comfort, it helps to adopt a repeatable prompt structure early. Not as a rigid template, but as a habit.

Think of every good prompt as having four elements:

A role or viewpoint: who is the guide acting as?
A task: what do you want done?
Context: what information should be used?
Constraints and format: how should the output look, and what must it obey?

In expedition terms: who is leading, where are we going, what map are we using, and what are the rules of the hike?

If you keep those four elements in mind, you will be able to infer intermediate discussions and recognise why their prompts look "long" and "structured". They are not fancy. They are making the map readable.

Copyright © Blockchain Council | All rights reserved

Chapter 2: The Map Reader and The Backpack

2.14 A Short Detour: Why Models Struggle With Long, Precise Reasoning

Beginners often ask, “If the model can write an essay, why can it not always do a multi-step problem perfectly?”

There are a few reasons, but the simplest is this: writing fluent text and maintaining perfect multi-step reasoning are different skills. Language models are strong at pattern-based generation. Some reasoning emerges naturally. Some need scaffolding.

Also, the model’s generation is incremental. If it makes a small mistake early, the rest of the response might build on that mistake. Humans do this too, except we can stop, notice, and restart more reliably.

This is why structured prompting, checking assumptions, and breaking tasks into smaller steps improve reliability. You are giving the guide a series of short, safe segments rather than one enormous leap in the fog.

This chapter is not about teaching you advanced reasoning prompts yet. It is about giving you the mental model so that, when later chapters introduce multi-step workflows and agents, you already understand why they exist.

Agents are not a gimmick. They are a response to the reality that one-shot answers are not always enough.

Copyright © Blockchain Council | All rights reserved

Chapter 2: The Map Reader and The Backpack

2.15 Let's Implement: Three Quick Experiments To Feel Tokens and Context

You already ran a "prompt lever" experiment in Chapter 1. Now we will run "backpack and table" experiments. No coding needed.

Experiment A: The backpack limit in action (controlled forgetting)

First, ask your AI tool:

"Summarize this text in 5 sentences, but preserve all key facts."

Then paste a reasonably long article or a few pages of any text you have. If you do not have one, paste several paragraphs from something you already wrote.

Now, after it summarizes, ask:

"Great. Now list the three most specific numerical details from the original text."

Two outcomes are common.

If it succeeds, it has enough context and retains the details.
If it fails or invents, it likely loses the specifics or never retains them properly, because summaries compress information.

Now improve the setup:

"Before summarizing, extract and list all numbers, dates, names, and proper nouns exactly as they appear. Then summarize."

You should see an immediate improvement. You have just learned a practical technique: when specifics matter, extract them explicitly before compressing.

In expedition terms, you made a separate "facts pouch" before folding the map.

Copyright © Blockchain Council | All rights reserved

2.15 Let's Implement: Three Quick Experiments To Feel Tokens and Context

Experiment B: Context grounding versus freewheeling

Ask:

"Explain the following topic using only the information in the notes I provide. If the notes do not contain an answer, say 'Not in notes'."

Now paste a short set of notes, even if it is only a few paragraphs.

Then ask three questions: one answerable from the notes, one partially answerable, and one not answerable.

This experiment teaches you two things. First, grounded answering is possible when you set the rule clearly. Second, when information is missing, a model will otherwise tend to fill gaps with plausible guesses unless constrained.

In expedition terms, you told the guide, "Do not invent trails that are not on this map."

Experiment C: The cost of verbosity (and why constraints help)

Ask:

"Write a 300-word explanation of what tokens are."
Then ask: "Now write the same explanation in 80 words, keeping the meaning."

Notice what happens. The shorter version forces compression. It often becomes clearer. Now ask a third time:

"Write it in 80 words, but include one concrete example and one warning about context limits."

This third prompt tends to be better than the second because you constrained the content rather than only the length.

Copyright © Blockchain Council | All rights reserved

Chapter 2: The Map Reader and The Backpack

2.16 Field Notes: The Mental Model You Will Reuse in Every Later Chapter

You do not need to remember every term. You need a stable picture.

A **language model** predicts the next token based on patterns plus the current context.

Tokens are the units that fill the backpack and determine cost.

The **context window** is the size of the map table.

If the table **overflows**, old notes fall off.

Output quality improves when the map is clear, the instructions are high-signal, and the constraints are explicit.

Hallucinations often happen when the model is forced to continue without enough grounded information.

Grounding is placing the right field notes on the table and telling the guide to stick to them.

If you carry this picture forward, intermediate jargon becomes translation, not intimidation.

Copyright © Blockchain Council | All rights reserved

Chapter 2: The Map Reader and The Backpack

2.17 Chapter Wrap: Stepping Beyond Base Camp Without Getting Lost

In Chapter 1, you met the guide and learned not to mistake confidence for truth. In this chapter, you learned how the guide generates language, why “tiny prompt changes” matter, and why tokens and context quietly control everything.

From here, the path becomes more practical. Next, we will start using these mechanics deliberately: how to structure prompts for reliable outputs, how to give the model better field notes, and how to design small workflows that do not collapse when the conversation gets long.

You are no longer just asking questions. You are learning to run an expedition.

Copyright © Blockchain Council | All rights reserved

Part B: Generative AI (Learning to Navigate)



Copyright © Blockchain Council | All rights reserved

Chapter 3: Prompting Basics

Giving Directions That Don’t Get You Eaten by Wolves

You have made it out of base camp. You know what AI is, what it is not, and why fluent answers are not the same thing as reliable truth. You have also peeked inside the guide’s rucksack and learned two quiet realities that govern everything that follows: the guide predicts the next step (token by token), and the guide can only work with what is actually on the map table (the context window).

Now comes the part where most beginners either level up quickly or spend weeks shouting “Why is it not doing what I asked?” at a perfectly innocent chat box.

This chapter is your first real navigation lesson.

In expedition terms, prompting is not “talking to a robot”. Prompting is briefing your guide. It is you setting the destination, calling out constraints, showing the map, and deciding what a good outcome looks like. When prompting goes well, the guide feels sharp and calm. When prompting goes badly, the guide feels like they are cheerfully walking you into a swamp while describing the swamp as “a scenic shortcut”.

The purpose here is not to turn you into a prompt poet. It is to make you operational. By the end of this chapter, you should be able to take a messy, vague request and turn it into a prompt that reliably produces usable work. You should also be able to read intermediate material that says things like “add constraints”, “provide context”, “specify format”, “reduce ambiguity”, and know exactly what to do.

You already ran a prompt-variation experiment earlier in the book, so we will not repeat that. Instead, we will build the underlying skill: how to design prompts on purpose, how to debug them when they misbehave, and how to create a small “prompting toolkit” you can reuse across writing, research, analysis, and coding.

Let’s begin!

Copyright © Blockchain Council | All rights reserved

Chapter 3: Prompting Basics

3.1 What Prompting Really Is: A Briefing, Not a Wish

Most beginner prompts look like wishes.

- “Explain AI.”
- “Write a marketing plan.”
- “Make this better.”
- “Summarize this.”

Sometimes you get lucky. Often, you get a generic answer that feels like a friendly Wikipedia cousin.

The fix is not “use smarter words”. The fix is to treat prompting like a briefing you would give to a competent human assistant who cannot read your mind.

A good briefing answers four questions:

a. Who are you acting as?
b. What exactly are you doing?
c. What information should you use?
d. What does “good” look like, and what must be avoided?

In expedition terms:

a. Who is leading?
b. Where are we going?
c. What map are we using?
d. What rules keep us safe and on schedule?

This is the foundation that will support the rest of Part B. Few-shot prompting, reasoning support, retrieval, tool use, and evaluation all build on the same basic idea: **you do not get better outcomes by begging harder, you get better outcomes by briefing better.**

Copyright © Blockchain Council | All rights reserved

Chapter 3: Prompting Basics

3.2 The Four-Part Prompt: Role, Task, Context, Constraints

If you remember one structure from this chapter, make it this one. It is not the only way to prompt, but it is the most reliable starting point for beginners.

Role: What perspective should the model adopt?
Task: What output are you asking for?
Context: What information should it use (and not use)?
Constraints: Tone, length, format, audience, scope, and any hard rules.

Here is why this works. You are shaping the probability landscape. Without structure, the model has too many plausible routes. With structure, you create rails.

In expedition terms, you are not saying "Let's go somewhere nice". You are saying, "We are going to this ridge, using this map, in daylight, avoiding steep ice, and we need to reach camp by 6 p.m."

Now, let's see what each part does in practice.

Role: choosing the right guide for the terrain

Role prompts help because they narrow the style and priorities of the response. A role can be professional ("act as a product manager"), pedagogical ("act as a tutor"), or procedural ("act as a strict editor").

Role is not cosplay. It is a steering wheel.

A useful role instruction is specific about behaviour:

"Act as a patient tutor. Use simple language. Define technical terms briefly when you introduce them."

A weak role instruction is vague:

"Act like an expert."

Experts can be smug, verbose, or allergic to your actual needs. Choose roles that shape behaviour, not ego.

Copyright © Blockchain Council | All rights reserved

3.2 The Four-Part Prompt: Role, Task, Context, Constraints

Task: what you want done, not what you want "talked about"

Tasks are verbs: draft, summarize, compare, rewrite, extract, classify, plan, generate, critique.

Bad tasks are foggy:

"Help me with this."

Good tasks are explicit:

"Rewrite this paragraph for clarity, keeping the meaning, and make it suitable for a beginner audience."

Context: the map on the table

Context is what the model can see and should rely on. If the answer must be grounded in a particular document, include the relevant excerpt. If it must match a brand voice, include examples. If it must fit a product requirement, include the requirements.

Without context, the model uses general patterns. That can be fine for generic explanations. It is risky for specifics.

Context also includes your "non-negotiables" that belong to the world, not the writing style:

"Our audience is beginner developers."
"The product is for small businesses."
"The tone must be friendly and not salesy."

That is not "constraints" yet. That is reality.

Copyright © Blockchain Council | All rights reserved

3.2 The Four-Part Prompt: Role, Task, Context, Constraints

Constraints: the guardrails that prevent drift

Constraints are where you specify format and quality.

Length: “about 300 words.”
Structure: “use headings with short paragraphs.”
Format: “return as a table with columns X and Y.”
Audience: “explain for someone with no AI background.”
Voice: “friendly, practical, no hype.”

Constraints are also where you set boundaries:

“Do not invent sources. If you are unsure, say you are unsure.”
“Ask one clarifying question if a requirement is ambiguous.”
Those two lines alone can save you from a lot of nonsense.

Copyright © Blockchain Council | All rights reserved

Chapter 3: Prompting Basics

3.3 The Single Biggest Prompt Killer: Ambiguity

Ambiguity is the fog on the trail. The guide will still walk. That is the problem.

A model faced with ambiguity often does one of three things:

- It guesses what you meant.
- It produces a broad answer that covers too much.
- It chooses an interpretation you did not want.

None of these is malicious. They are predictable.

So the beginner habit to build is this: **when you write a prompt, ask yourself what a stranger could misunderstand.**

Here are common ambiguity traps:

- “Make it better.” Better how? Shorter? Clearer? More persuasive? More formal?
- “Summarize.” For what audience? With what level of detail?
- “Explain.” In what depth? With what examples?
- “Create a plan.” For which timeframe? For which resources? For which goal?

In expedition terms, “Take me to the top” is not a plan. It is a motivational quote.

A practical fix is to add one sentence that locks the meaning:

- “Make it better by reducing jargon and adding one real-world example.”

You just turned fog into a trail marker.

Copyright © Blockchain Council | All rights reserved

Chapter 3: Prompting Basics

3.4 Prompting Is Not One-Shot: It is a Loop

Beginners often think prompting is about finding the perfect prompt and then never touching it again.

In practice, prompting is iterative. You ask, you inspect, you correct the course.

That is not failure. That is how you use a tool properly.

A good prompting loop has three steps:

a. Brief: give role, task, context, constraints.

b. Inspect: check if the output matches your goal.

c. Correct: tighten constraints, add missing context, or clarify intent.

In expedition terms, you do not set out on day one and refuse to check the map until day seven. You check, adjust, and keep going.

This loop is also the bridge to later chapters. Few-shot prompting is a way of improving the brief with examples. Reasoning support is a way of improving inspection and correction. Retrieval is a way of improving context. Tool use is a way of improving grounding. Evaluation is a way of improving inspection at scale. Everything connects!

Copyright © Blockchain Council | All rights reserved

Chapter 3: Prompting Basics

3.5 The “Prompting Ladder”: From Vague to Reliable

Think of prompts as levels of navigation skill.

- At the bottom, you have vague prompts.
- In the middle, you have structured prompts.
- At the top, you have prompts that define success criteria and failure handling.

A vague prompt:

“Write an email asking for a meeting.”

A structured prompt:

“Write a polite email requesting a 20-minute meeting next week to discuss onboarding. Audience: busy manager. Tone: professional and warm. Provide two subject lines.”

A reliability prompt:

“Write a polite email requesting a 20-minute meeting next week to discuss onboarding. Audience: busy manager. Tone: professional and warm. Provide two subject lines. Ask one clarifying question if you need availability or time zone. Keep the email under 120 words.”

The difference is not fancy wording. The difference is control.

In expedition terms, you are moving from “let’s walk somewhere” to “we take the ridge trail, stop at the stream, and we turn back if the weather turns.”

Copyright © Blockchain Council | All rights reserved

Chapter 3: Prompting Basics

3.6 The Five Prompt Controls That Matter Most

You can prompt in a thousand ways. For beginners, five controls give you the biggest improvement per character typed.

1) Audience and intent

- If you do not specify an audience, the model chooses one. Often it chooses “generic internet”.
- A single line like “Explain this to a complete beginner” changes everything.

2) Output format

- The format is the rails. Ask for a structure, and you reduce drift.
- “Use headings and short paragraphs.” “Return as a checklist.” “Return as a table.”, etc.

3) Examples

- Examples reduce ambiguity. They show “this style, not that style”.
- You will go deep into examples in Chapter 5, but even here, one example can reshape output quality dramatically.

4) Constraints and exclusions

- Constraints stop the model from wandering.
- “Do not use jargon.” “Do not mention ethics or governance.” “Do not include anything not supported by the text.”
- Use exclusions carefully. Too many “do not” lines can create confusion. Use them like a fence, not like a prison.

5) Success criteria

- Tell the model what “done” looks like.
- “Include three benefits and one limitation.” “Provide one analogy and one real-world example.” “End with a 3-question quiz.”
- Success criteria are underrated. They turn outputs from “nice writing” into “usable work”.

Copyright © Blockchain Council | All rights reserved

Chapter 3: Prompting Basics

3.7 Prompt Hygiene: How To Keep Instructions Readable

In Chapter 2, you learned about signal and noise. Prompt hygiene is applying that idea to your instructions. A clean prompt is easier for the model to follow and easier for you to reuse.

A few practical habits:

- Put the main task early.
- Group constraints together.
- Use labels like “Context:” and “Requirements:”.
- If you include source text, separate it clearly.

This is not about being formal. It is about making the map readable in bad weather.

Here is a simple, beginner-friendly layout you can reuse:

- Role:
- Task:
- Context:
- Requirements:
- Output format:

You do not need to use it every time. Use it when the task matters.

Copyright © Blockchain Council | All rights reserved

Chapter 3: Prompting Basics

3.8 When The Model Goes Wrong: Prompt Debugging Like a Grown-up

When outputs are poor, beginners often do one of two things:

- They blame the model.
- They keep repeating the same prompt louder, as if the model is hard of hearing.

A better approach is to debug systematically.

Ask:

- What went wrong?
- Was the task unclear?
- Was context missing?
- Were constraints conflicting?
- Was the format unspecified?
- Was the scope too large for one response?

Then fix one variable at a time.

In expedition terms, if you are lost, you do not sprint faster. You stop, check the map, and correct the course.

Here are common failure patterns and quick fixes.

Failure: too generic.
Fix: add audience + success criteria + format.

Failure: wrong tone.
Fix: specify tone with examples of what you want and what you do not want.

Failure: made up details.
Fix: provide source context and instruct “use only this source”.

Failure: ignored a requirement.
Fix: restate requirements as a checklist and ask the model to confirm compliance.

A powerful trick is to ask the model to self-check its output against your requirements:

“Before finalizing, verify each requirement and list any that are not satisfied.”

This does not guarantee perfection, but it reduces obvious misses.

Copyright © Blockchain Council | All rights reserved

3.9 Worked Example: Rewriting a Messy Prompt Into a Reliable One

Let's do this in the exact way you would do it in real life. We start with a messy prompt, we observe what goes wrong, and then we tighten it.

The messy prompt

"Write a LinkedIn post about AI agents and why they matter."

What might happen?

- You get a generic, hype-heavy post.
- It defines agents loosely.
- It contains vague claims such as "agents will transform everything."
- It might include buzzwords you did not ask for.
-

That is not "bad AI". That is a vague brief.

Now rewrite using the four-part prompt.

The reliable prompt

Role: You are a practical AI educator writing for beginners.
Task: Write a LinkedIn post explaining what AI agents are and why they matter.
Context: The audience has no prior knowledge of AI, GenAI, or agentic systems.
Requirements:

- Keep it beginner-friendly and jargon light.
- Use the expedition analogy (guide, map, tools, crew).
- Include one real-world example of an agent in business (like customer support triage or invoice processing).
- Avoid hype. Use clear, grounded language.
- End with one question to encourage comments.

Output format:

- 1 short hook line
- 2 short paragraphs
- 1 example paragraph
- 1 closing question

Now the guide has a route plan. The role shapes tone. The context defines the audience. The requirements define content and style. The format defines structure.

This is the core skill of prompting: you convert "write something about X" into "produce Y, for Z audience, using A constraints, in B format". If you can do that, the rest of the book becomes easier.

Copyright © Blockchain Council | All rights reserved

Chapter 3: Prompting Basics

3.9 Worked Example: Rewriting a Messy Prompt Into a Reliable One

Now the guide has a route plan. The role shapes tone. The context defines the audience. The requirements define content and style. The format defines structure.
This is the core skill of prompting: you convert “write something about X” into “produce Y, for Z audience, using A constraints, in B format”. If you can do that, the rest of the book becomes easier.

Copyright © Blockchain Council | All rights reserved

Chapter 3: Prompting Basics

3.10 Let's Implement: Turn Messy Prompts Into Reliable Prompts

This exercise is deliberately practical. You will rewrite prompts, run them, compare outputs, and learn how to fix predictable failures.

You can do this in any generative AI chat tool.

Step 1: Use the Prompt Rewrite Checklist

Before you touch the examples, keep this mental checklist:

- What is the output supposed to be?
- Who is it for?
- What context does the model need?
- What constraints prevent drift?
- What format makes it easy to use?

Now apply it.

Exercise A: "Make it better"

In this exercise, we see how a precise prompt template can significantly improve the quality of an excerpt.

Messy prompt: "Make this better: [paste your paragraph]."

Your rewrite goal: clarity without changing meaning.

Reliable prompt template:

Role: You are a strict but helpful editor.
Task: Rewrite the paragraph for clarity and readability.
Context:
The target reader is a complete beginner. Keep the original meaning.
Requirements:
- Reduce jargon.
- Keep it the same length or shorter.
- Preserve any numbers or proper nouns exactly as written.
- If a sentence is ambiguous, rewrite it rather than adding new facts.

Output format:
Return two versions:
1) Clean rewrite
2) Clean rewrite with one-sentence explanation of what you changed

Copyright © Blockchain Council | All rights reserved

3.10 Let's Implement: Turn Messy Prompts Into Reliable Prompts

Expected result: the output should feel cleaner, not "more intelligent". If new facts appear, your constraints were not strict enough, or your source paragraph was ambiguous.

Troubleshooting: if it becomes too short and loses meaning, add "Do not remove key details" and define what counts as key.

Exercise B: "Summarize this"

Let's observe how a reliable prompt template outperforms a generic prompt in terms of clarity, precision, and relevance for summarization tasks.

Messy prompt: "Summarize this article."
Your rewrite goal: a summary you can actually use.
Reliable prompt template:

Role: You are a summarizer for busy readers.
Task: Summarize the text I provide.
Context: The reader is new to AI.
Requirements:
- Keep it accurate and grounded to the text.
- Include: main idea, 3 key points, 1 practical takeaway.
- If the text does not support a claim, do not invent it.

Output format:
- 1 sentence main idea
- 3 short bullet-like lines (no long paragraphs)
- 1 practical takeaway sentence

Expected result: you get a structured output that is quick to scan.

Troubleshooting: if it adds ideas not in the text, add "Use only the provided text. If missing, say 'Not in text'."

Copyright © Blockchain Council | All rights reserved

3.10 Let's Implement: Turn Messy Prompts Into Reliable Prompts

Exercise C: "Create a plan"

A proper, meaningful, and reliable plan can be effectively generated if a proper prompt template is followed. Let's see how by implementing this exercise.

Messy prompt: "Create a learning plan for AI."
Your rewrite goal: a plan with scope and pace.
Reliable prompt template:

Role: You are a learning coach.
Task: Create a 14-day beginner learning plan for AI that prepares me for Generative AI and Agentic AI basics.
Context: I have no prior AI knowledge. I can spend 45 minutes per day.
Requirements:

- Use the expedition analogy lightly to keep motivation.
- Each day must include: a concept, a short exercise, and a 'field note' reflection question.
- Avoid governance, ethics, or policy. Focus on fundamentals and practice.

Output format:

- Day-by-day list with:
- Day X: Concept
- Exercise:
- Field note question:

Expected result: the plan feels doable, not encyclopaedic.

Troubleshooting: if it tries to cover too much, add "Limit each day to one concept and one exercise."

Copyright © Blockchain Council | All rights reserved

Step 2: Compare outputs like a navigator

After running each exercise, do not ask "Did I like it?" Ask:

a. Did it follow the format?
b. Did it stay within context?
c. Did it meet the success criteria?
d. Did it avoid making things up?

This is how you start thinking like an expedition leader, not just a passenger.

Copyright © Blockchain Council | All rights reserved

Chapter 3: Prompting Basics

3.11 Analogy Tie-Back: Why Good Prompts Feel Like Good Route Plans

Prompting is navigation.

A vague prompt is like leaving camp with no route plan and announcing you will “figure it out”. You might still reach something interesting, but you are also one unlucky turn away from a swamp.

A structured prompt is like planning the trail, packing for the weather, and agreeing on what “success” means before you walk. You do not eliminate uncertainty, but you reduce preventable mistakes.

This is the real point of prompting basics. You are not learning tricks. You are learning control.

In the next chapters, you will learn stronger tools for control. Few-shot prompting will show you how examples act like trail markers. Reasoning support will show you how to break a hard climb into safe segments. Retrieval will show you how to bring field notes onto the map table. Tool use will give your guide a compass, calculator, and radio. Evaluation will tell you whether you actually arrived or just felt good while walking.

For now, you have a core skill that makes everything else work: you can brief the guide like you mean it.

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

Showing the Route with Examples, and Breaking the Climb into Safe Steps

You have already learned the first survival skill of this expedition: how to brief the guide without accidentally sending them on a poetic stroll into the fog. In the previous chapter, you moved from "wishes" to "briefings", and you saw why structure (role, task, context, constraints) makes the guide calmer and more useful.

Now we upgrade your navigation from *directions* to *demonstrations*.

Because here is the inconvenient truth about language models: they are often better at copying a pattern you show them than obeying a pattern you vaguely describe. If you want a particular style, format, tone, or decision process, the fastest way is usually to show it, not lecture it.

That is what few-shot prompting is.

Once you start doing real work, a second truth shows up: many useful tasks are not one step. They are a chain of steps. If you try to do the whole chain in one breath, the model can drift, skip checks, or confidently sprint past a missing bridge.

That is why we decompose tasks. We turn one scary climb into a set of safe switchbacks.

This chapter blends two skills that will carry you through the rest of Part B and make Part C feel natural later:

a. **Few-shot prompting:** you show the route with examples.

b. **Reasoning support without magic:** you break the route into steps, add check points, and verify you are still on the trail.

The mission stays the same: beginners should become comfortable enough to read intermediate, jargon-heavy materials and translate them into practical behaviour. You do not need to be "an AI person". You need to be a good expedition leader.

Let's get started!

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

Story beat: The Trail gets Real

Base camp was clean. The air was still. The guide was polite and eager. You could ask simple questions and get sensible answers.

Now you are on the actual trail.

The terrain changes. Wind picks up. The path splits into three routes that all look “reasonable”. Your guide is still helpful, but their default behavior is to make the most plausible continuation of the conversation, not to guarantee that you end up where you intended.

This is the moment where you stop saying, “Please explain this in a friendly way”, and start saying, “Here is the format. Here are three examples. Do the next one the same way.”

This is also the moment where you stop asking, “Can you write me a plan?”, and start asking, “First break the goal into steps, then ask me two questions, then produce the plan, then check it against constraints.”

In expedition terms, you have moved from chatting with the guide to running the expedition properly.

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

4.1 Few-Shot Prompting: Why Examples Beat Instructions

Few-shot prompting means you provide a few examples of the input and the output you want, and the model continues the pattern.

It is called “few-shot” because you are not training the model (that is a different thing), you are simply giving it a few “shots” at seeing what good looks like inside the current context.

A very beginner-friendly way to think about it:

- **Zero-shot:** “Do the task.”
- **Few-shot:** “Here are a few demonstrations. Now do the task in the same way.”

In expedition terms, zero-shot is you describing the route using words. Few-shot is you pointing at footprints in the snow and saying, “Follow these.”

Why does it work so well?

Because language models are pattern engines. They are unusually good at picking up a structure from examples: tone, formatting, level of detail, even subtle habits like how you start sentences or how you explain mistakes.

If your instructions are vague, the model has too many plausible styles to choose from. Examples collapse that uncertainty quickly.

A single good example is a trail marker. Three good examples are a paved road.

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

4.2 When is a Few-Shot Approach Worth it, and When is it Overkill

Few-shot prompting is not mandatory for every task. You use it when you care about consistency.

It is especially useful for:

- **Style matching**
  You want a brand voice, a personal writing style, or a “house format”.
- **Structured outputs**
  You want a consistent schema: summaries, checklists, tickets, templates, and lesson sections.
- **Decision patterns**
  You want the model to classify or judge something using your rules.
- **Transformations**
  Rewrite in a specific way: simplify for beginners, convert into a table format, convert notes into a script.

It is less useful when:

- **The task is simple, and the model already does it well**
  Basic explanations, generic drafts, quick brainstorms.
- **You do not care about consistent formatting**
  If you are exploring, not producing.
- **Your examples are weak or inconsistent**
  Bad examples teach bad patterns quickly. The guide is obedient like that.

In expedition terms, you do not set up trail markers for walking from your tent to the water bottle. You set them up for crossings, cliffs, and fog.

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

4.1 Few-Shot Prompting: Why Examples Beat Instructions

Few-shot prompting means you provide a few examples of the input and the output you want, and the model continues the pattern.

It is called “few-shot” because you are not training the model (that is a different thing), you are simply giving it a few “shots” at seeing what good looks like inside the current context.

A very beginner-friendly way to think about it:

- **Zero-shot:** “Do the task.”
- **Few-shot:** “Here are a few demonstrations. Now do the task in the same way.”

In expedition terms, zero-shot is you describing the route using words. Few-shot is you pointing at footprints in the snow and saying, “Follow these.”

Why does it work so well?

Because language models are pattern engines. They are unusually good at picking up a structure from examples: tone, formatting, level of detail, even subtle habits like how you start sentences or how you explain mistakes.

If your instructions are vague, the model has too many plausible styles to choose from. Examples collapse that uncertainty quickly.

A single good example is a trail marker. Three good examples are a paved road.

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

4.3 The Hidden Trick: Examples Create “Soft Rules.”

In Chapter 3, you learned that constraints are explicit rules. Few-shot examples are softer. They do not say “must”. They *imply* must.

This matters because models sometimes ignore explicit rules when the rest of the prompt suggests a different pattern.

If you say, “Keep it under 100 words,” and then you show two examples that are 250 words each, the model is likely to follow the examples.

So few-shot prompting gives you power, but it also gives you responsibility.

Your examples are not decoration. They are instructions in a disguise.

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

4.4 The Anatomy of a Good Few-Shot Prompt

A reliable few-shot prompt usually has these parts:

a. What you are doing and why
b. The examples (input → output pairs)
c. The new input
d. A clear instruction to continue the pattern

You can keep it simple. The main discipline is clarity.

Here is a generic template you can reuse:

Role: (optional)
Task: Do the transformation/classification/writing.
Examples: Provide 3–5 pairs.
Now do this: Provide the new input.
Output format: Match the examples.

In expedition terms, you show the footprints, then point at the next patch of snow.

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

4.5 Choosing Examples: The “Closest Trail” Rule

Your examples should be close to the task you want. If they are too different, the model learns the wrong pattern.

A simple rule: **examples should be the closest trails to your current trail.**

If you want the model to summarize technical content for beginners, do not provide examples of marketing copy. Provide examples of technical summaries written for beginners.

If you want the model to write support replies, show support replies, not product descriptions.

Also: keep examples short. You are not trying to impress the model. You are trying to teach a pattern efficiently.

A beginner-friendly sweet spot is often **3 examples**. Use **5** when the style is subtle, or the output must be consistent over many runs.

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

4.6 Common Few-Shot Failures (and How to Fix them)

Failure 1: The model copies content, not just style

Sometimes it clings too closely to the example's topic.
Fix: diversify topics in examples while keeping format and tone consistent.

Failure 2: The model ignores your constraints

Usually, because the examples contradict them.
Fix: make examples obey the constraints. If length matters, ensure example lengths match.

Failure 3: The model mixes patterns

Your examples are inconsistent.
Fix: rewrite examples so they share the same structure. Consistency beats creativity here.

Failure 4: The model drifts over time

The first output is great, later outputs wobble.
Fix: keep a "golden example" at the top of your working prompt and reuse it. In long sessions, restate the pattern.

In expedition terms, if your trail markers start pointing in different directions, the group will wander. That is not the hikers being "bad". That is the signage being chaotic.

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

4.7 Reasoning Support Without Magic: Why Decomposition Matters

Now we shift to the second skill: decomposition.

A lot of beginner pain comes from trying to do multi-step tasks in one step:

“Write a business plan, include market research, create financial projections, and make it investor-ready.”

That is like asking your guide to cross three mountain passes while also cooking dinner and composing a victory speech.

Even if the model produces something that looks impressive, it might be:

- Missing key assumptions
- Internally inconsistent
- Not aligned to your constraints
- Confident about made-up details

Decomposition means you break the big task into smaller tasks, in a deliberate order. It is not about “making the model think like a human”. It is about reducing drift and increasing control.

In expedition terms, you do not climb a cliff in one leap. You take switchbacks, you stop at safe ledges, and you check the map at each point.

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

4.8 The Decomposition Ladder: From One-Shot to Controlled Workflow

Here are three levels of doing a task.

Level 1: One-shot request

- “Write a lesson plan about prompting.”
- Fast. Often generic. Easy to drift.

Level 2: Plan, then produce

- “First, outline the lesson plan in 6 sections. Then write each section.”
- Better structure. Still can miss constraints.

Level 3: Plan, ask, produce, verify

- “First, propose an outline. Then ask me two clarifying questions. Then write the draft. Then self-check against these requirements and revise once.”
- This level is where reliability starts to feel real.

You are not building an agent yet, but you are already behaving like an expedition leader: plan, check, proceed, verify.

Part C will formalize this into workflows. Part D will formalize it into agents. Right now, you are building the habit.

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

4.9 “Reasoning Prompts” Without Turning It Into Mysticism

Some people treat reasoning prompts like spell casting. They sprinkle magical phrases and hope the model becomes a disciplined thinker.

We are not doing that.

We are doing something simpler: we add structure and checkpoints.

The three most useful checkpoint patterns for beginners are:

a. **Step list**
“Break this into steps before answering.”

b. **Checklist validation**
“After writing, check each requirement and state whether it is satisfied.”

c. **Assumption control**
“List assumptions you made. If unsure, ask a clarifying question instead of guessing.”

This is reasoning support without drama. It is operational.

In expedition terms, you are adding a map check, a safety check, and a weather check.

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

4.10 Combining Few-Shot With Decomposition: The Real Power Move

Here is where things get fun.

Few-shot prompting gives you consistent style and format.

Decomposition gives you a consistent process.

Put them together, and you get outputs that are both:

- formatted the way you want
- produced through a controlled sequence

This is the bridge skill that makes later chapters feel natural. When you read an intermediate concept that says “create a planner step and an executor step”, you will not panic. You will think, “Ah, switchbacks and checkpoints.”

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

4.10 Combining Few-Shot With Decomposition: The Real Power Move

Here is where things get fun.

Few-shot prompting gives you consistent style and format.

Decomposition gives you a consistent process.

Put them together, and you get outputs that are both:

- formatted the way you want
- produced through a controlled sequence

This is the bridge skill that makes later chapters feel natural. When you read an intermediate concept that says “create a planner step and an executor step”, you will not panic. You will think, “Ah, switchbacks and checkpoints.”

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

Worked Example: A “Style Mimic” Plus a “Safe Process.”

We will do a worked example that uses both skills, in a beginner-friendly setting:

Goal: Produce short explanations that teach beginners, in your expedition voice, without slipping into jargon or fluff.

This is a real-world need in your book, your content, and later, your projects.

Step 1: Define the output shape

We want a mini-lesson that always has:

- A simple definition
- An expedition analogy
- A tiny example
- One quick “field note” takeaway

Instead of describing that structure endlessly, we will show it.

Step 2: Provide 3 examples (few-shot)

Example 1 (Input): “Explain token.”
Example 1 (Output):
A token is a small chunk of text that a language model reads and writes, like pieces of words rather than whole sentences. In expedition terms, tokens are the weight units in your backpack. You might think you packed “one paragraph”, but the backpack counts “many tokens”. Tiny example: the word “unbelievable” might be split into multiple chunks, which increases your token count. Field note: if you paste long text or code, your backpack fills faster than you expect.

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

Worked Example: A “Style Mimic” Plus a “Safe Process.”

Example 2 (Input): “Explain context window.”
Example 2 (Output):
The context window is the maximum amount of text the model can consider at one time, measured in tokens. In expedition terms, it is the size of the map table at camp. If the table is full, older notes fall off, even if they mattered earlier. Tiny example: in a long chat, the model may forget a rule you gave at the start because it is no longer “on the table”. Field note: Restate key constraints when the conversation becomes long.

Example 3 (Input): “Explain few-shot prompting.”
Example 3 (Output):
Few-shot prompting gives the model a few examples of the behavior you want, so it can continue the pattern. In expedition terms, it is showing footprints and trail markers instead of only giving verbal directions. Tiny example: if you want a specific email style, show three sample emails and ask for the fourth in the same style. Field note: your examples are soft rules, so make them consistent.

Now we have taught a pattern.

Step 3: Add decomposition (process control)

Now we ask the model to create a new mini-lesson, but we force a process:

- Propose the structure first
- Write the output
- Self-check against constraints

Step 4: The combined prompt

Here is what you would actually type:

“Continue the pattern from the examples above. New input: ‘Explain temperature in language models’. Before writing, outline the four parts you will include. Then write the mini-lesson. Then verify: simple language, expedition analogy present, tiny example present, field note present.”

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

Worked Example: A “Style Mimic” Plus a “Safe Process.”

What output should look like, and why

A good output will match style and structure because of a few-shot examples. It will also be less likely to drift because you forced a checkpoint process.

If it fails, you debug systematically:

- If the style is wrong, your examples were not strong enough.
- If it skipped a required part, your checkpoint instruction was unclear.
- If it became too long, your examples were too long, or you did not set a length expectation.

In expedition terms, you either adjust the trail markers (examples) or you adjust the checkpoints (process).

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

4.11 Let's Implement: Build Your Own "Prompting Kit" for Consistent Outputs

This implementation section is designed to be completed in one sitting.

You will create two reusable assets:

a. "A few-shot style mimic (3–5 examples)"
b. "A decomposition workflow prompt (plan → ask → produce → verify)"

You can run this in any modern GenAI chat tool (ChatGPT, Gemini, Perplexity, etc.)

Implementation Part A: Few-Shot Style Mimic (3–5 examples)

Step A1: Pick one output type you care about.

Choose one of these (or your own):

- Beginner explanation (definition + analogy + example + field note)
- Support reply (polite + concise + next steps)
- Summary (main idea + 3 points + takeaway)
- Lesson section (hook + explanation + example + exercise)

For this book, the first and the fourth options are your best friends.

Step A2: Write 3 examples yourself (short and consistent).

Yes, you write them. You are teaching the model your voice.

Keep them tight. "Think 'trail markers', not 'novels'".

If you do not want to write from scratch, you can take small snippets from your existing chapter text and compress them.

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

4.11 Let's Implement: Build Your Own "Prompting Kit" for Consistent Outputs

Step A3: Test with a new input.

Ask the model to produce the next one.

Then inspect:

- Did it match the structure?
- Did it match the tone?
- Did it keep a beginner-friendly language?
- Did it remain consistent in length?

Expected results

If your examples are consistent, you will see a surprising level of style matching quickly.

Troubleshooting

If the output becomes generic, add one more example that is highly representative of your voice. Make it the clearest trail marker.

If the output copies content themes too closely, diversify the example topics while keeping the format identical.

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

4.11 Let's Implement: Build Your Own "Prompting Kit" for Consistent Outputs

Implementation Part B: Decomposition Workflow (Safe Multi-Step Output)

Now you will create a reusable prompt that forces planning, clarification, and verification.

Step B1: Choose a real task you will repeat

For your ebook creation, likely tasks include:

- Turn rough notes into a clean section
- Expand a short idea into a full chapter segment
- Convert jargon into beginner language
- Produce exercises with expected results and troubleshooting

Step B2: Use this reusable workflow prompt

Paste this as a template and adjust the bracketed parts:

"Role: You are my expedition co-writer for a beginner AI book.
Task: [Describe the output you want].
Context: [Paste any source text or notes].

Requirements: Keep the language beginner-friendly. Use the expedition analogy lightly. Avoid fluff.

Process:
1. Propose an outline with 5–7 short headings.
2. Ask me up to 2 clarifying questions if needed. If not needed, say 'No questions'.
3. Write the draft.
4. Self-check against these constraints: clarity, beginner-friendliness, analogy included, no unnecessary jargon, and that the output follows the outline.
5. Revise once based on the self-check."

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

4.11 Let's Implement: Build Your Own "Prompting Kit" for Consistent Outputs

Expected results

Outputs become more consistent, less rambly, and more aligned to your intent.

Troubleshooting

- If it asks too many questions, cap it: "Ask at most 1 question."
- If it asks no questions but clearly guessed, add: "If unsure, ask rather than assume."
- If it produces a weak outline, strengthen it by saying: "Make headings action-based, not vague."

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

4.12 Analogy Tie-Back: Footprints, Switchbacks, and Arriving on Purpose

Few-shot prompting is how you put footprints on the trail.

Instead of hoping the guide interprets your instructions correctly, you show them exactly how the group walks, talks, and records field notes. Examples create momentum and consistency.

Decomposition is how you stop trying to climb cliffs in one jump.

You break the route into safe segments. You stop at ledges. You check the map. You verify that the crew is still on the right ridge. That is not “extra work”. That is how you avoid expensive mistakes later.

Put together, these two skills are a silent superpower. They let beginners produce work that feels intermediate: consistent outputs, controlled process, and fewer unpleasant surprises.

And when you later read a “jargonish” book that talks about workflows, planners, validators, or task decomposition, you will not feel lost. You will recognize the same expedition logic with fancier labels.

You already know the trail.

Copyright © Blockchain Council | All rights reserved

Chapter 4: Few-Shot Prompting and Decomposing Tasks

Field Notes: What You Should Carry Forward

1. Few-shot prompting is showing examples so the model continues the pattern.
2. Examples act like soft rules, so they must be consistent.
3. Three strong examples often beat ten vague instructions.
4. Decomposition turns a big task into safe steps with checkpoints.
5. Add verification: check requirements, list assumptions, and ask clarifying questions instead of guessing.
6. Combining style examples and process checkpoints produces the most reliable outputs from the model.

Next, we will keep navigating, but with stronger tools: how to supply “field notes” from external sources (retrieval basics), so the guide stops improvising terrain that you could have simply placed on the map table.

Copyright © Blockchain Council | All rights reserved

Chapter 5: Retrieval Basics

Story beat: The Day You Stop Trusting Memory and Start Using a Library

By now, you and your guide have a rhythm.

You know how to brief the guide without accidentally asking for a poetic stroll into the fog. You learned how examples create “trail markers” (few-shot), and how breaking work into steps prevents cliff jumps (decomposition).

Then the trail gets… real.

Someone asks a question that *sounds* simple, but it is actually about facts that are specific to your world, such as your notes, company, study material, course policy, internal docs, and personal project files. Your guide answers smoothly and confidently.

And it’s wrong.

Not maliciously wrong. Not “the guide is trying to ruin your hike” wrong. Just the classic situation you already saw earlier: if the guide does not have the right map on the table, it guesses the terrain anyway, because it has learned how answers usually sound, not what is true in your specific context.

This is the moment you become an expedition leader in a new way.

You stop expecting the guide’s memory to be the truth. You start carrying **field notes** and using a **library.**

That is what “retrieval basics” means.

Copyright © Blockchain Council | All rights reserved

Chapter 5: Retrieval Basics

5.1 The Problem Retrieval Solves

A language model is great at generating text that fits patterns. It is not automatically great at knowing *your* facts.

Even if the model "knows" a lot in general, it does not automatically know:

- The exact wording of your internal policy.
- The details in your PDF.
- The constraints from your spec doc.
- The definitions from your course notes.
- The latest changes in your project.

You already learned the key principle: a model cannot use information it cannot see. If it is not on the map table, it is not usable.

So you have two options:

a. Keep asking anyway and hope the guide guesses correctly.
b. Put the relevant notes on the table, then tell the guide to stick to them.

Option 2 is how you stop getting eaten by wolves.

In the GenAI world, option 2 is usually called **grounding**. In our expedition language: *placing the right field notes on the map table and telling the guide not to invent trails that are not on the map.*

Copyright © Blockchain Council | All rights reserved

Chapter 5: Retrieval Basics

5.2 The Core Idea of RAG (Retrieval-Augmented Generation)

You will see the phrase **RAG** everywhere. It sounds intimidating. It's not.

RAG = Retrieval + Generation

- **Retrieval**: find relevant pieces of information from your documents
- **Generation**: ask the model to answer using those pieces

Think of it like this:

- The model is your guide.
- Your documents are your expedition journal, field manual, and map notes.
- Retrieval is the act of flipping to the right pages.
- Generation is the guide speaking, but only after reading the right pages.

That's the whole trick.

Importantly, RAG does **not** magically make the model smarter. It makes the model **better informed at the moment of answering.**

This is what you actually wanted in the first place.

Copyright © Blockchain Council | All rights reserved

Chapter 5: Retrieval Basics

5.3 Why “Just Paste The Document” Is Not the Same as Retrieval

Beginners often do a sensible thing:

They paste a big blob of text and say, “Answer based on this.”

Sometimes it works.

Sometimes it fails in annoying ways, for reasons you already understand:

- The blob is too long (the map table is full, older things fall off).
- The important part is buried in noise (the guide glances at the wrong sentence).
- The model summarizes or compresses and loses specifics.
- The answer requires connecting two distant parts of the blob.

Retrieval is the “grown-up version” of pasting.

Instead of dumping the whole library onto the table, you:

a. Store the library neatly.
b. Fetch only the 3–8 most relevant snippets for this question.
c. Put those snippets on the table.
d. Ask the guide to answer using those snippets.

That is why RAG feels like a power-up: it is mostly an organization.

Copyright © Blockchain Council | All rights reserved

Chapter 5: Retrieval Basics

5.4 The Minimal RAG Pipeline (What's Actually Happening)

A lightweight RAG system typically has four stages:

Step 1: Collect documents
PDFs, notes, FAQs, wiki pages, policies, manuals.

Step 2: Split documents into chunks
Because searching a thousand small cards is easier than searching one giant scroll.

Step 3: Create a search method
Often this is *semantic search* using embeddings, sometimes keyword search, sometimes a mix.

Step 4: Retrieve top chunks for the question
Grab the most relevant cards based on similarity.

Step 5: Build the answer prompt
Place rules at the top, paste retrieved chunks below, then ask the question.

Step 6: Generate the answer
The model writes using the retrieved text.

Optional Step 7: Cite sources
List chunk IDs or document names so the answer is auditable.

In expedition terms: you gather your maps, you cut them into foldable sections, you label them, you build an index, you pull the right sections for today's route, and only then you decide where to walk.

Copyright © Blockchain Council | All rights reserved

Chapter 5: Retrieval Basics

5.5 Chunks: Why We Split Documents in the First Place

Chunking sounds like a technical detail. It's actually a survival skill.

If you store whole documents as one piece, retrieval becomes dumb:

- A 20-page doc might be "relevant" overall, but only one paragraph matters.
- The model gets too much irrelevant text, which increases confusion and cost.

Chunking is how you turn a library into index cards.

What makes a good chunk (beginner rules)

A good chunk is:

- small enough to fit into the context comfortably.
- big enough to contain a complete thought.
- self-contained when possible (it helps if it includes the heading or topic).

In human terms, you want snippets that make sense even if you read them alone.

Typical beginner chunk sizes

If you need a starting point:

- 200–500 words per chunk work well for many text docs,
- overlap a little (for example, repeat the last 1–2 sentences into the next chunk) so meaning does not get cut in half.

Do not over-optimize this early. You can tune later.

Copyright © Blockchain Council | All rights reserved

Chapter 5: Retrieval Basics

5.6 Embeddings (Semantic Search) Explained Without Pain

If you see the word embedding and feel your brain reaching for the exit, here's the friendly version:

An embedding is a way to convert text into a list of numbers that represents its meaning.

Once you turn text into numbers, you can:

- Turn the question into numbers, too.
- Compare them.
- Find which chunks are "closest" in meaning.

In expedition terms:

- The keyword search is looking for the same words in the journal.
- Embeddings are finding pages that talk about the same idea, even with different words.

Example:

If your question is: "How do refunds work?"
A keyword search might miss a paragraph that says: "We offer returns within 14 days."

Embeddings usually catch that, because "refunds" and "returns" live close together in meaning-space.

That's why embeddings are popular.

That said, do not treat embeddings as magic. They are strong, but they can still retrieve the wrong chunk if:

- Your docs are messy.
- Your chunking is bad.
- The question is ambiguous.
- The top chunk is "near" but not actually answering.

So we still need good prompting and good constraints on the generation step.

Copyright © Blockchain Council | All rights reserved

Chapter 5: Retrieval Basics

5.7 The Prompt Pattern That Makes Retrieval Actually Work

Retrieval without the right prompt often fails in a specific way:

You retrieve good notes... and the model ignores them.

So you use a retrieval-aware prompt. Here is the beginner-friendly pattern:

Retrieval-aware answer prompt (template)

Role: You are a careful assistant answering using the provided notes.
Task: Answer the question using only the notes.
Context: Notes are provided below.
Rules:

- If the answer is not in the notes, say “Not in notes.”
- Do not add external facts.
- Quote or cite which note you used.

Output format:

- Answer
- Sources used (note IDs)

This directly matches the grounding rule you already practiced earlier: *do not invent trails that are not on this map.*

If you remember only one thing from this chapter, remember this:

Retrieval improves truthfulness only when you force the model to treat retrieved text as the map.

Copyright © Blockchain Council | All rights reserved

5.8 Worked Example: From “Confident Guessing” to “Library-Backed Answers.”

Let’s make this concrete with a scenario you’ll recognize if you’ve ever supported learners, users, or customers.

Scenario

You have three small internal docs:

- Policy.md
- FAQ.md
- Pricing.md
-

A user asks:

“Can I transfer my paid course to someone else?”

What happens without retrieval
If you ask a model directly, it might say something like:

“Yes, most platforms allow transfers. Contact support.”

That sounds reasonable. It might also be wrong. It might even conflict with your real policy.

The guide is doing what it always does: generating the most plausible continuation.

What happens with retrieval

Now you retrieve the relevant chunk from Policy.md, which says (example):

“Transfers are allowed within 7 days of purchase if the course has not been accessed. Contact support with the new learner’s email.”

Now the answer becomes grounded and specific:

“Yes, transfers are allowed within 7 days of purchase if the course hasn’t been accessed. Contact support with the new learner’s email.”
Sources: Policy.md (Transfers section)

Same model. Same “intelligence.” The difference is that you put the right page on the table.

Copyright © Blockchain Council | All rights reserved

Chapter 5: Retrieval Basics

5.9 Failure Patterns (and Quick Fixes)

Lightweight RAG usually fails in predictable ways. That's good news, because predictable failures are fixable.

Failure 1: Retrieval brings irrelevant chunks

Symptom: The answer feels off because the retrieved notes are off.

Fixes:

- Improve chunking (split cleaner, add headings).
- Retrieve more than one chunk (top 5 instead of top 2).
- Add a re-ranking step later (we'll keep it lightweight for now).

Failure 2: The model ignores the notes

Symptom: It answers from general knowledge.

Fixes:

- Strengthen the rule: "Use only notes. If it's missing, say 'Not in notes'."
- Force citation: "List note IDs used."
- Put rules at the top and keep them high-signal

Failure 3: The notes contain the answer, but the model still misses it

Symptom: The answer exists, but the model says "Not in notes."

Fixes:

- Retrieve more chunks.
- Reduce noise (shorter, cleaner chunks).
- Explicitly ask the model to scan notes first, then answer.

A very practical trick:

"First list which notes look relevant (by ID) and why. Then answer using only those notes." That forces attention onto the notes before writing.

Failure 4: Notes conflict with each other

Symptom: Two docs say different things.

Fixes:

- Ask the model to report conflict instead of choosing.
- Add "priority rules" (Policy overrides FAQ, newer docs override older).

In expedition terms: if two maps disagree, you don't sprint. You stop and verify.

Copyright © Blockchain Council | All rights reserved

Chapter 5: Retrieval Basics

5.10 Let's Implement: Simple RAG With a Few Local Documents

This implementation is designed to be doable in one sitting, even if you are a beginner. You are going to play two roles:

- You are the retriever (you bring the right notes)
- The model is the generator (it writes the answer)

You will run the exercise in two modes:

- Mode A: File upload RAG experience (if your tool supports uploads)
- Mode B: Note-card RAG experience (works everywhere, even without uploads)

Pick one. If you do both, the concept will click.

Mode A: "Upload the docs and answer from them" (No code, beginner-friendly)

Step A1: Create a folder on your computer, for example, rag_demo

Step A2: Create three simple documents

Inside **rag_demo**, create:

- Policy.txt
- FAQ.txt
- Pricing.txt

Sample content you can paste:

Policy.txt

- Transfers: Transfers are allowed within 7 days of purchase if the course has not been accessed. Contact support with the new learner's email.
- Refunds: Refunds are available only within 7 days of purchase. After that, no refunds are provided.
- Access: Course access remains active for 12 months from the date of enrollment.

Copyright © Blockchain Council | All rights reserved

5.10 Let's Implement: Simple RAG With a Few Local Documents

FAQ.txt

- Q: Do I get recordings?
A: Yes, recordings are available after each session.

- Q: Can I pause my course?
A: You can pause once for up to 14 days.

- Q: Do you provide certificates?
A: Yes, after completion and passing the criteria.

Pricing.txt

- Standard pricing applies unless a discount is explicitly approved.
- Discounts: One-time discounts may be considered for multi-year renewals.
- Renewals: Renewals are billed annually unless otherwise stated.

Use your own documents if you prefer. The only rule: keep the documents short for the first run.

Step A3: Open your chat tool and start a fresh chat

Start a new conversation so unrelated context does not bleed in.

Step A4: Upload the files (if available)

Look for an attachment icon (often a paperclip) or a button like Upload, Attach, or Add files.

Upload:

- Policy.txt
- FAQ.txt
- Pricing.txt

If your tool does not support upload, skip to Mode B.

Copyright © Blockchain Council | All rights reserved

5.10 Let's Implement: Simple RAG With a Few Local Documents

Step A5: Paste the "retrieval rules" prompt (copy exactly)

Paste this instruction block:
"Role: You are a careful assistant.
Task: Answer questions using only the uploaded documents.

Rules:
1. If the answer is not explicitly stated in the documents, say: 'Not in notes.'
2. Do not add external facts. Do not guess.
3. Quote the exact sentence(s) you used before you explain.
4. After answering, list the document name(s) you used as Sources."

This is the key. This is the part that trains the guide to behave like a grounded assistant.

Step A6: Ask three test questions

Ask these, one by one:

1. "Can I transfer my paid course to someone else?"
2. "How long is my access active?"
3. "Do you offer refunds after 30 days?"

Expected results:

- For Q1: a specific answer with "within 7 days" and "if not accessed," plus a quote from Policy.txt.
- For Q2: "12 months," plus a quote.
- For Q3: "Not in notes" or "No refunds after 7 days," depending on how you phrased your policy.

If the tool tries to guess, tighten the rules further:

"If it's not explicitly written, respond only with: Not in notes."

That is strict on purpose. You are training reliability.

Copyright © Blockchain Council | All rights reserved

5.10 Let's Implement: Simple RAG With a Few Local Documents

Mode B: "Note cards with IDs" (Works in any chat tool)

If your chat tool cannot upload files, you can still do real retrieval behavior by bringing note cards into the conversation.

Step B1: Turn your documents into labeled chunks

Open your Policy.txt and copy it into the chat, but break it into chunks like note cards.

Use this exact format:

[Policy-1 | Transfers]
(paste only the transfer sentence)

[Policy-2 | Refunds]
(paste only the refunds sentence)

[Policy-3 | Access]
(paste only the access sentence)

Do the same for FAQ and Pricing:

[FAQ-1 | Recordings]

[Pricing-1 | Discounts]

This feels manual because it is manual. You are learning the mechanism.

Step B2: Ask the retrieval-aware question prompt

Now ask:

"Answer the question using only the note cards provided.
If the answer is not explicitly stated, say 'Not in notes.'
After your answer, list the card IDs you used under Sources."

Copyright © Blockchain Council | All rights reserved

5.10 Let's Implement: Simple RAG With a Few Local Documents

Step B3: Test the same three questions

Use the same three questions from Mode A.

Expected results: (the "click" moment)

If you get:

- Specific answers
- No guessing
- Sources listed by ID

Then congratulations: you just built the human version of RAG. You are now thinking like a system designer, not just a chat user.

Troubleshooting:

If it invents facts

Your rule is not strict enough. Make it binary:

"If not in notes, say only: Not in notes."

If it answers but does not cite

Force a template:

"Output format:
Answer:
Quote:
Sources:"

If it cites the wrong thing

Your chunks are unclear. Make chunks smaller and cleaner. One card, one idea.

If it says "Not in notes" too often

Your docs might not contain the info, or your chunking removed the key line. Add the missing line to the notes.

In expedition terms: you are improving the quality of your field notebook. Better notes create better guidance.

Copyright © Blockchain Council | All rights reserved

Chapter 5: Retrieval Basics

5.11 Analogy Tie-Back: Why Retrieval Feels Like “Expert Supervision” For The Guide

Without retrieval, your guide is improvising based on general trail stories. Sometimes that’s fine. Sometimes it’s a disaster. With retrieval, you do something simple but powerful:

You hand the guide your field notebook, open to the right page, and say:

“Use this. If it’s not here, say so.”

That is grounding.

That is why retrieval is the bridge between “fun chat” and “reliable system.”

It fits perfectly with what you already learned:

- Prompting is briefing the guide.
- Few-shot is leaving footprints.
- Decomposition is using switchbacks.
- Retrieval is bringing the right field notes onto the map table.
- Later, tools and agents will be about giving the expedition a radio, compass, and crew.

You’re not learning tricks. You’re learning control.

Copyright © Blockchain Council | All rights reserved

Chapter 5: Retrieval Basics

5.12 Field Notes: What You Should Carry Forward

- Retrieval is how you make answers depend on your documents, not vibes.
- RAG is just Retrieval + Generation.
- Chunking turns a library into usable note cards.
- Retrieval works only when prompting forces the model to treat notes as the map.
- “Not in notes” is a feature, not a failure. It prevents confident nonsense.
- Lightweight RAG can start manual, then become code, then become production.

Copyright © Blockchain Council | All rights reserved

Chapter 5: Retrieval Basics

Chapter Wrap: You Built a Library, Not a Bigger Imagination

In the early chapters, you learned that fluent answers are not guaranteed truths. You learned that context is a map table with limited space. You learned how to brief the guide, how to leave footprints with examples, and how to climb safely with steps and checkpoints.

Now you added the missing piece that makes GenAI practical in real work:

A way to bring the right notes at the right time.

You and your guide end the day not with more opinions, but with a small camp library: organized, searchable, and useful.

In the next chapter, we give the guide tools: a compass, a calculator, and a radio.

Copyright © Blockchain Council | All rights reserved

Chapter 6: Tool Use

Giving the Guide a Compass, Calculator, and Radio

You've done something important in the last chapter: you stopped treating the guide's memory like reality, and you built a small library. That alone upgrades your expedition from "enthusiastic wandering" to "reliable navigation."

Now comes the next upgrade: you stop asking your guide to *pretend* it can do things, and you actually give it equipment.

A language model, on its own, is a very talented talker. It can explain, draft, summarize, and structure. It can even sound like it is "doing" math or "checking" the web. But unless you connect it to tools, it is still just generating text.

In expedition terms, you have a guide who can read maps and tell stories, but they are still standing at camp with no compass, no calculator, and no radio. They can estimate. They can guess. They can act confidently. But... they cannot actually *measure*.

This chapter is where you hand your guide three pieces of gear:

A **compass**, so it can make crisp directional decisions (tool choice and routing).
A **calculator**, so it can compute instead of improvising numbers.
A **radio**, so it can check information outside the conversation (even if we start with a simple stub).

By the end, you will understand what "tool use" really means, why it changes reliability so dramatically, and how to build a simple function-calling loop with a calculator plus a web-like stub you can run locally.

And yes, we will keep the story moving forward in a way that feels like actual progress, not an endless hike in circles.

Copyright © Blockchain Council | All rights reserved

Chapter 6: Tool Use

Story Beat: The Gear Table at Camp

The expedition has a new routine now.

In the morning, you lay out your route plan. You pull up your field notes (retrieval). You decide what the day's goal is and what "done" looks like (prompting). You place trail markers so the guide matches your format (few-shot). You break the climb into safe switchbacks (decomposition).

So far, so good.

Then the day throws a very normal problem at you. You need to answer a question like:

"How much water do we need for this section if we walk 14 miles, climb 2,500 feet, and it's hot?"

Or:
"Is the bridge ahead open today?"

Or:
"What's the total cost if we buy supplies for three people for four days, with a 12% tax?"

The guide can *sound* like it knows. It can give you "reasonable" numbers. It can give you a "likely" status. It can write a confident plan. But you do not want a confident storyteller right now. You want a guide who can do two very specific things:

a. **Compute.**
b. **Check.**

So you set up a gear table. You place a calculator on it. You place a radio on it. You label them clearly. Then you tell your guide: "When you need these, use them." That is tool use. It is not mystical. It is not "the model got smarter." It is simply that your guide now has equipment that produces real outputs instead of vibes.

Copyright © Blockchain Council | All rights reserved

Chapter 6: Tool Use

6.1 The Core Problem Tools Solve: “Plausible” Is Not the Same as “Performed.”

Let’s name the failure mode we are fixing.

A model will often produce an answer that is plausible. That means it fits the pattern of what an answer should look like. It does not guarantee the answer was computed.

If you ask, “What is 17,493 × 48?” a model might confidently provide you with a number. Sometimes it’s right. Sometimes it isn’t. The danger is that the wrong one still looks clean.

If you ask, “What’s the weather tomorrow in Denver?” the model can write a nice forecast-shaped paragraph. But unless it can check, it is guessing.

This is why tool use matters. It turns certain categories of tasks from “best-effort generation” into “call the thing that actually knows.”

In expedition terms, tools are how you stop debating the direction and start using the compass.

Copyright © Blockchain Council | All rights reserved

Chapter 6: Tool Use

6.2 Tools vs Retrieval: They Sound Similar, but They Solve Different Problems

Chapter 5 was about **bringing trusted notes to the table**. That reduces hallucinations by grounding the model in your documents.

Tools are different. Tools are about **performing actions or queries** that produce results the model should not invent.

Retrieval answers: “What does our library say?”
Tools answer: “What does the world (or a system) return right now?”

A quick distinction:

Retrieval is your **field notebook.**
Tools are your **gear**.

Sometimes you use both. In fact, many practical systems do:

You retrieve a policy chunk (field notes), then use a calculator (gear), then respond with a grounded answer (guide speaking with evidence).

That combination is the point of this whole Part B arc: you are turning a talkative guide into an operational expedition partner.

Copyright © Blockchain Council | All rights reserved

Chapter 6: Tool Use

6.3 What “Function Calling” Really Means

“Function calling” is a fancy label for a simple workflow:

1. The model reads your request.
2. The model decides that a tool should be used.
3. The model outputs a structured tool call (usually JSON).
4. Your program runs the tool.
5. The tool returns a result.
6. The model uses the result to write the final answer.

That is it.
The critical idea is this: the model is not doing the tool’s work. It is only choosing and formatting the request to the tool, then interpreting the tool output.

In expedition terms:

The guide does not become the compass.
The guide decides when to use the compass and how to read it.

Copyright © Blockchain Council | All rights reserved

Chapter 6: Tool Use

6.4 Tools Need Contracts: Name, Purpose, Inputs, Outputs

A tool is only useful if the guide understands:

- What is it called?
- When it should be used.
- What arguments it expects.
- What it returns.

This is a contract.

A beginner-friendly tool contract looks like this:

Tool name: `calculator`
What it does: Performs arithmetic.
Inputs: operation and numbers.
Outputs: numeric result (and maybe error messages).

Tool name: `web_search_stub`
What it does: Returns canned “search results” from a local dataset.
Inputs: query string.
Outputs: list of results with titles/snippets/URLs (or “not found”).

Notice something: the tool contract is not about the internal code. It’s about the interface. The guide does not need to know how the radio works. It needs to know how to use it.

Copyright © Blockchain Council | All rights reserved

Chapter 6: Tool Use

6.5 The Tool Loop: The Basic Pattern You Will Keep Reusing

Most tool-using assistants follow a simple loop:

Step A: Decide whether to call a tool.
Step B: If yes, call it with structured inputs.
Step C: Receive output.
Step D: Produce the final answer grounded in the tool output.

This loop is the early ancestor of agentic behavior, but we are not calling it an agent yet. It's still a single-purpose navigation aid: "Use the equipment when needed."

If you build only this correctly, you already surpass a huge amount of beginner frustration.

Copyright © Blockchain Council | All rights reserved

Chapter 6: Tool Use

6.6 When Tools Help Most (and When They Don’t)

Tools shine when the task is:

- Math, counting, and unit conversion.
- Database lookups.
- Searching documents or websites.
- Calling internal APIs.
- Running code.
- Validating formats.
- Checking availability or status.

Tools are less helpful when the task is:

- Drafting narrative text.
- Explaining concepts.
- Rewriting tone and structure.
- Summarizing general information (unless you must quote sources).

In expedition terms: you don’t use a calculator to tell a story. You use it to stop lying to yourself about distances.

Copyright © Blockchain Council | All rights reserved

Chapter 6: Tool Use

6.7 Tool Selection: Teaching the Guide “When to Reach for Gear.”

This is where many people stumble. They define tools, but the model doesn’t use them consistently.

The fix is to give the guide a decision rule that is easy to follow.

For beginners, a clean rule is:

- If the question needs exact arithmetic, use the calculator.
- If the question needs outside information, use the radio.
- If the answer is in the notes, use retrieval (the library from the previous chapter).
- If none of these apply, answer directly, but label uncertainty when needed.

You are not trying to make the guide “smarter.” You are making the guide disciplined.

Copyright © Blockchain Council | All rights reserved

Chapter 6: Tool Use

6.8 Tool Output Is Not the Answer Yet

A tool returns raw output. The model still has to:

- Interpret it.
- Explain it.
- Fit it to the user's question.
- Format it.

This is where the model still adds value. You do not want a tool to dump raw data to a beginner user. You want the guide to translate.

In expedition terms: the compass points north. The guide decides which trail to take.

Copyright © Blockchain Council | All rights reserved

Chapter 6: Tool Use

6.9 Errors, Edge Cases, and Tool Failure (The Reality Check)

Tools are not perfect. They fail in predictable ways:

- Bad input format.
- Missing parameters.
- Time-outs.
- No results found.
- Conflicting results.

A beginner mistake is building a tool system that assumes tools always succeed. A practical system treats tool failure as normal.

So you give your guide a failure protocol:

If the tool errors, explain the error in plain language. Try one correction if obvious. If still failing, ask a clarifying question or suggest an alternative.

That alone makes your tool system feel professional.

Copyright © Blockchain Council | All rights reserved

Chapter 6: Tool Use

6.10 The “Web-like Stub”: Why We Start With a Fake Radio

You might wonder why we are building a stub instead of “real web access.” Because the goal here is not internet coverage. The goal is to learn the *pattern*:

Ask → decide tool → call tool → use result → answer.

A stub lets you practice the mechanics locally with zero external dependencies. It makes the loop visible. It makes debugging easier. And it keeps this chapter beginner-friendly.

Later, swapping the stub for a real search API is just replacing the radio hardware. The expedition procedure stays the same.

Copyright © Blockchain Council | All rights reserved

Chapter 6: Tool Use

Worked Example: Planning a Day's Route With Tools

Let's run a scenario where tool use is obviously better than pure text generation.

Scenario: You're planning tomorrow's hike

You have three facts:

- Distance: 12.6 miles
- Elevation gain: 2,900 feet
- Start time: 7:30 AM

You want:

- An estimated hiking time.
- A water recommendation.
- A check of "bridge status" from your "radio."

We will use two tools:

- Calculator (for estimated time).
- Web-like stub (for bridge status).

Step 1: The question you ask the guide

"Tomorrow we are hiking 12.6 miles with 2,900 feet of gain. We start at 7:30 AM. Estimate total time on trail and suggest how much water to carry. Also, check if the Pine Creek bridge is open."

Step 2: What a tool-aware guide does differently

Instead of inventing numbers, the guide says:

"I should compute a time estimate."
"I should check the bridge status using the radio."

It then calls:

- Tool call: calculator
  Inputs: use a time estimate formula.

- Tool call: web_search_stub
  Inputs: "Pine Creek bridge open."

Copyright © Blockchain Council | All rights reserved

Worked Example: Planning a Day's Route With Tools

Step 3: The tool outputs

Calculator output: 6.8 hours (example)
Stub output: "Pine Creek bridge: OPEN as of last update."

Step 4: The guide's final answer (grounded)

Now the guide writes:

Estimated time: ~6 hours 45 minutes (based on the calculation).
Water: recommend 2.5–3.5 liters depending on heat and pace.
Bridge: OPEN (per stub).

Notice what changed. The model didn't become a hiker. It became a better expedition partner because it stopped improvising the measurable parts.

Copyright © Blockchain Council | All rights reserved

Chapter 6: Tool Use

6.11 Let's Implement: Tool Use in Gemini AI Studio

Up to this point, we have been training the guide's words. Now we train the guide's hands.

In this exercise, you will use built-in tools inside Gemini AI Studio so the model can actually do things instead of "sounding like it did them." The key difference from the older Playground-style demos is simple: you will not paste tool outputs yourself. When these tools are enabled, the platform runs them and shows you what happened. AI Studio is designed for toggling tools like code execution and grounding right from the interface.

This is the goal: you see a clean sequence on screen.

User question → tool runs → grounded final answer.

That is "tool use" in one shot.

Step 1: Open Gemini AI Studio (Chat)

1. Open this page: aistudio.google.com/prompts/new_chat
2. Sign in with your Google account.
3. Make sure you are in a **chat-style prompt** (a normal conversation view).

Step 2: Find the Run settings panel

AI Studio includes a **Run settings** area where you can choose model parameters and **toggle tools**. Look for a panel on the side (often the right side) labelled something like **Run settings** or **Settings**.

If you do not see it immediately:

- Look for a small settings icon or a collapsible side panel.
- Expand it until you see a section named Tools.

Copyright © Blockchain Council | All rights reserved

6.11 Let's Implement: Tool Use in Gemini AI Studio

Step 3: Turn on the two tools we need (Calculator + Radio)

In the Tools section, switch on:

1. Code execution (this will be our "calculator")
2. Grounding with Google Search (this will be our "radio")

Important: you might also see "function calling." That is powerful, but it usually needs an external app to actually run your functions. For this hands-on demo, we want tools that run inside the playground without you building anything.

Step 4: Run a "Calculator" mission (no guessing allowed)

Paste this prompt exactly:

Prompt A (Calculator drill):
"You are my expedition guide. Do not estimate. Use code execution for all arithmetic. We have 3 people, hiking for 4 days. Each person needs 2.75 liters of water per day.

a. Total liters for the whole trip

b. If we carry 1.5-liter bottles, how many bottles do we need (round up) Return the answer as a short table and one plain-English sentence."

What you should see:

- A small **code execution** step appears (the guide "uses the calculator").
- Then you get a final answer that looks calm and exact, not vibe-based.

Why this matters in the expedition story:

- This is the moment the guide stops doing "trail math theatre."
- You did not make the guide smarter. You gave it gear, then required it to use it.

Copyright © Blockchain Council | All rights reserved

6.11 Let's Implement: Tool Use in Gemini AI Studio

Step 5: Run a "Radio" mission (fresh facts with sources)

Now paste:

Prompt B (Radio drill):

"You are my expedition guide. Use Google Search grounding.
Question: What is the latest Node.js LTS version today?
Give the version, the codename if available, and cite sources."

What you should see:

- The model triggers **grounding with Google Search** if it judges that search will improve the answer.
- The final response should include **supporting links/citations** in the UI when grounding is used.

Why this matters in the expedition story:

- This is the radio check before you commit to a route.
- Instead of "I think it's open," you get "Here's what the outside world says, with receipts."

Step 6: The one prompt that proves "tool choice" (gear table in action)
Now we make the guide choose the right gear in a single turn. Paste:

Prompt C (Gear table drill):

"You have two tools available:

- Code execution for exact math
- Google Search grounding for up-to-date facts
  Task: We're planning supplies.

1. Calculate the total food cost if meals cost $7.95 each, we need 3 meals per day, for 3 people, for 4 days, plus 8.25% tax.
2. Then, using Google Search grounding, tell me whether Yosemite National Park requires reservations for entry this month, and cite sources.

Keep the final answer short and clearly split into Part 1 and Part 2."

Copyright © Blockchain Council | All rights reserved

Chapter 6: Tool Use

6.11 Let's Implement: Tool Use in Gemini AI Studio

Step 6: The one prompt that proves "tool choice" (gear table in action)
Now we make the guide choose the right gear in a single turn. Paste:

Prompt C (Gear table drill):

"You have two tools available:

- Code execution for exact math
- Google Search grounding for up-to-date facts
  Task: We're planning supplies.

1. Calculate the total food cost if meals cost $7.95 each, we need 3 meals per day, for 3 people, for 4 days, plus 8.25% tax.
2. Then, using Google Search grounding, tell me whether Yosemite National Park requires reservations for entry this month, and cite sources.

Keep the final answer short and clearly split into Part 1 and Part 2."

What this teaches (in one screenshot-worthy moment):

- Code execution handles what must be computed.
- Search grounding handles what must be checked.
- The guide stops pretending both are "just writing."

Step 7: Tiny troubleshooting (because reality likes to bite)

If the model does not use the tool:

- Add one line to your prompt: "You must use the tool, not estimate."
- Make the question tool-worthy:
  - For code execution: use non-trivial numbers and ask for rounding.
  - For search grounding: ask "latest," "today," "this month," or "current," and request citations.

If the UI doesn't show citations:

- Confirm Grounding with Google Search is enabled in Tools.

Copyright © Blockchain Council | All rights reserved

Chapter 6: Tool Use

6.12 The Real Lesson: Tools Turn Confidence Into Evidence

Here is the clean principle you just practiced: **a model is allowed to be fluent, but it is not allowed to pretend.**

- When the task is measurable, you want measurable outputs.
- When the task depends on the outside world, you want checked information with sources.
- When both appear in one request, you want the guide to pick the right gear without you micromanaging every step.

In production, your application would run tools through APIs and systems. The model still does the same job you saw today: it decides when to use a tool, formats the request internally, and then explains the result clearly.
So the learning carries forward cleanly.

You did not make the guide smarter. You gave it the equipment, and you required it to use it.

Copyright © Blockchain Council | All rights reserved

Chapter 6: Tool Use

Analogy Tie-Back: A Guide With Gear Stops Guessing About Measurable Things

At the start of this chapter, the guide had a familiar bad habit: sounding right.

Now you have a better habit: being verifiably right whenever the task can be measured or checked.

Compass logic is coming soon (tool routing and selection at scale). The radio will become real (actual search, internal systems, and retrieval, plus tools together). For now, you ended the day with a tested gear table and a repeatable drill.

The expedition does not “go on forever” because each chapter ends with a new capability you can point at and say: We can do this now, reliably.

Next, we ask the most uncomfortable but useful question in the entire journey:

Did we actually arrive, or did we just enjoy the hike?
That is evaluation.

Copyright © Blockchain Council | All rights reserved

Chapter 7: Evaluation for Beginners (Did We Actually Arrive, or Just Feel Like It?)

Story Beat: The Sunset Signpost Problem (Why "Feels Right" Isn't a Compass)

By now, your expedition has momentum.

You can brief your guide with clearer prompts. You can add examples to show the "shape" of what you want. You can decompose messy requests into smaller steps. You can bring field notes (retrieval), so the guide stops guessing your specific facts. You can even hand the guide gear (tools) so it computes and checks instead of improvising.

So you'd think the hard part is over. Then comes the moment that quietly ruins most beginner projects.

It's late afternoon. The trail has been long. Everyone's tired. And you see a signpost. It says: **"Summit: 2 miles".**

Half the team cheers. Someone starts celebrating early. Someone else says, "Two miles is nothing." Another person says, "Wait, are those trail miles or straight-line miles?" The guide confidently says, "We're basically there."

And this is the key problem: confidence is not an arrival.

In applied AI work, especially with generative systems, you can get outputs that *sound right*, *look right*, and *feel right*, while being wrong in ways that matter.

A lot of beginners build something, see three good outputs, and decide: "It works."

That is the "sunset signpost" trap.

This chapter is your antidote. You are going to learn how to answer one boring but powerful question:

Did we actually arrive, or did we just feel like it because the output sounded confident?

You will learn it in a beginner-friendly way, without turning your project into a PhD thesis. No complicated dashboards. No heavy math. Just a simple, repeatable method you can apply to prompts, few-shot examples, retrieval systems, and tool-using assistants.

By the end, you will have your own tiny evaluation kit: a test set, a rubric, and a scoring loop. In expedition terms, this chapter is where you stop relying on "vibes" and start using **checkpoints, bearings, and a logbook.**

Copyright © Blockchain Council | All rights reserved

Chapter 7: Evaluation for Beginners (Did We Actually Arrive, or Just Feel Like It?)

7.1 What “Evaluation” Means

Evaluation is not about proving your AI is perfect. Evaluation is about answering two practical questions:

a. **Is this system reliable enough for the task I want?**

b. **If I change something, did it actually improve, or did I just get lucky with a few examples?**

Think of evaluation as **a way to measure progress without lying to yourself**. If you are building a chatbot, evaluation helps you see if it answers correctly across many questions, not just the ones you tried. If you are building a summarizer, evaluation helps you see if it stays accurate, not just “nice.”

If you are building a retrieval system (RAG), evaluation helps you see whether answers are grounded in your documents or drifting into plausible fiction. If you are building a tool-using assistant, evaluation helps you see whether it calls tools correctly, uses results properly, and stops making things up.

Without evaluation, your project becomes a campfire story:

- entertaining
- confident
- and occasionally dangerous

With evaluation, your project becomes a route plan backed by evidence.

Copyright © Blockchain Council | All rights reserved

Chapter 7: Evaluation for Beginners (Did We Actually Arrive, or Just Feel Like It?)

7.2 Why Beginners Skip Evaluation (And Why That Backfires)

Most beginners skip evaluation for understandable reasons:

- It feels like extra work when you just want to build.
- It sounds technical and intimidating.
- The model outputs are fuzzy, so you assume scoring must be fuzzy too.
- You do not know what “good” even means yet.

But skipping evaluation creates a predictable failure pattern:

a. You build something.
b. You test a few cases.
c. It looks good.
d. You change prompts, add retrieval, and add tools.
e. Something breaks.
f. You do not know what improved, what got worse, or why.
g. You start “prompt thrashing,” changing random things, and hoping.
h. You lose confidence and stop.

Evaluation is how you avoid that spiral. It gives you a calm, repeatable way to say:

- “This version is better than the last.”
- “This change fixed accuracy but hurt conciseness.”
- “Retrieval improved factuality but introduced citation mistakes.”
- “Tool use improved math but increased latency and failure cases.”

In expedition terms, evaluation is how you avoid walking in circles because every valley looks the same at dusk.

Copyright © Blockchain Council | All rights reserved

Chapter 7: Evaluation for Beginners (Did We Actually Arrive, or Just Feel Like It?)

7.3 What Should You Measure? (Quality Is Not One Thing)

Beginners often ask: “How do I score AI outputs when they’re not like normal software outputs?”

The trick is to stop treating “quality” as one blob. Quality is a bundle of different dimensions. Depending on your task, some matter more than others.

Here are the most useful beginner-friendly dimensions.

Correctness
Is the answer factually correct or logically correct?

For math, this is straightforward. For policy questions, it means aligning with the policy text. For summaries, it means no invented facts.

Grounding

Does the answer come from your provided documents (in a retrieval system), or is it free-floating?

Grounding matters when you want the model to use your notes, not its general training.

Completeness

Did it cover the key parts of the question?

A correct but incomplete answer can still fail the user.

Clarity

Is it readable and structured for the audience?

A correct answer that confuses the reader is still a bad outcome.

Format Compliance

Did it follow the required format: JSON, bullet list, steps, table, email, whatever you asked for?

This matters a lot in automation.

Copyright © Blockchain Council | All rights reserved

Chapter 7: Evaluation for Beginners (Did We Actually Arrive, or Just Feel Like It?)

7.3 What Should You Measure? (Quality Is Not One Thing)

Safety of Assumptions (Practical, Not "Ethics Chapter")

This is not a governance lecture. This is a practical check: **did the model assume details it should have asked for?**

For example: "Book a meeting tomorrow at 2" is risky if tomorrow means different time zones. Or "Use the newest policy" is risky if the assistant did not retrieve it.

Tool Use Correctness (When Tools Exist)

- Did it call the right tool?
- Did it send the right arguments?
- Did it use the tool result properly?
- Did it stop tool-calling when done?

These are measurable.

Important: You do not need to measure everything. Start with 2 to 4 dimensions that match your task.

That keeps evaluation small and realistic.

Copyright © Blockchain Council | All rights reserved

Chapter 7: Evaluation for Beginners (Did We Actually Arrive, or Just Feel Like It?)

7.4 The Tiny Test Set (Your First Real Evaluation Superpower)

A “test set” sounds fancy. It is not. A beginner test set is simply:

A small list of inputs you care about, along with what “good” looks like for each.

That’s it.

In expedition terms, a test set is a list of **checkpoints:**

- “At this creek, the water should be cold.”
- “At this ridge, the view should be east.”
- “At this junction, we should see the red marker.”

If you do not have checkpoints, you cannot know if you are on the right trail.

How big should it be?

Start with 10 to 20 cases. That is enough to reveal patterns without exhausting you.

Where do you get test cases?

Use these sources:

a. Real questions people ask (if you have them)

b. The tricky edge cases that broke your system

c. The “sounds easy but isn’t” questions

d. Boundary cases: short inputs, long inputs, ambiguous inputs

e. Counterexamples: questions the assistant should refuse or should ask clarifying questions for (not policy-based refusal, just “not enough info” cases)

Make the test set balanced

A good beginner test set usually includes:

- 60% normal cases
- 30% tricky cases
- 10% nasty edge cases

This keeps you honest without making you miserable.

Copyright © Blockchain Council | All rights reserved

Chapter 7: Evaluation for Beginners (Did We Actually Arrive, or Just Feel Like It?)

7.5 Gold Answers vs Rubrics (Two Ways to Define “Good”)

When you build a test set, you need a way to score outputs. There are two beginner-friendly methods.

Method A: Gold Answers (Reference Outputs)

You write the “correct answer” (or a target answer) for each input.

This works best when:

- The task is deterministic (math, extraction, classification)
- There is a clear expected response
- Formatting matters a lot

Example: “What is 17,493 × 48?” has a single correct answer.

Method B: Rubrics (Score by Criteria)

You define what “good” means using criteria, then score the output against that.
This works best when:

- The output can vary (summaries, explanations, writing)
- There is no single perfect answer
- You care about multiple dimensions (correctness + clarity + grounding)

A rubric can be simple:

- 2 = good
- 1 = partial
- 0 = bad

Or slightly richer:

- correctness (0–2)
- grounding (0–2)
- clarity (0–2)
- format (0–2)

Rubrics are powerful because they let you measure improvement without pretending there is one perfect phrasing.

Copyright © Blockchain Council | All rights reserved

Chapter 7: Evaluation for Beginners (Did We Actually Arrive, or Just Feel Like It?)

7.6 Manual Scoring Is Not “Unprofessional” (It’s the Right Start)

Beginners often feel they must automate evaluation immediately. You do not.

Manual scoring is how you learn what matters. It also teaches you something deeper:

Most AI quality problems are not random. They are patterns.

When you score outputs manually, you start noticing patterns like:

- It fails on long questions
- It ignores the second constraint
- It answers confidently when it should ask a clarifying question
- Retrieval is pulling irrelevant chunks
- Tool calls are correct, but the final answer misuses the tool result

Once you see patterns, automation becomes easier and more meaningful.

In expedition terms, manual scoring is the first time you stop, look at your map, and mark where the trail actually went.

Copyright © Blockchain Council | All rights reserved

Chapter 7: Evaluation for Beginners (Did We Actually Arrive, or Just Feel Like It?)

7.7 Lightweight Automatic Scoring (Beginner-Friendly Options)

Once you have a tiny test set and rubric, you can add a bit of automation. Here are beginner-safe options.

Exact match (for structured outputs)

If the answer should be JSON or a specific label, you can check exact match or schema validity.

Example: output must include "total": <number> and "currency": "USD".

Numeric checks (for math and tool results)

If the answer includes a number, parse it and check.

Regex checks (for format)

Example: must contain “Step 1”, “Step 2”, “Step 3”.

Retrieval checks (for grounding)

If you require citations, check if the output includes citations and whether cited chunks match the retrieved context.

This can be lightweight at first:

- “Did it cite something?”
- “Did it cite at least one correct chunk?”

Copyright © Blockchain Council | All rights reserved

Chapter 7: Evaluation for Beginners (Did We Actually Arrive, or Just Feel Like It?)

7.7 Lightweight Automatic Scoring (Beginner-Friendly Options)

“LLM as judge” (carefully, as training wheels)

You can ask a model to score another model’s output against a rubric. This can work surprisingly well when:

- Your rubric is specific.
- You give the judge the same context documents.
- You keep the scoring scale small (0–2).

But you must treat it as a *helper*, not the truth. A judge model can be biased, inconsistent, or overly generous.

So the beginner rule is:

Use LLM-judging to speed up scoring, not to replace your understanding.

In expedition terms, it’s like asking a second guide to check your compass reading. Helpful, but you still look at the terrain.

Copyright © Blockchain Council | All rights reserved

Chapter 7: Evaluation for Beginners (Did We Actually Arrive, or Just Feel Like It?)

7.8 Evaluation Changes When You Add Retrieval (RAG)

In a normal prompt-only system, evaluation is mostly about output quality. In a retrieval system, you are evaluating two things:

a. **Did we retrieve the right notes?**
b. **Did we answer correctly using those notes?**

This matters because RAG can fail in two different ways:

- Retrieval failure: You pulled the wrong chunks, so the model had no chance.
- Generation failure: you pulled good chunks, but the model ignored or misused them.

A beginner-friendly way to score RAG is to split your rubric:

- Retrieval relevance (0–2)
- Answer grounding (0–2)
- Answer correctness (0–2)

Even a simple split like that makes debugging dramatically easier. Because when the output is wrong, you can say:

- “Retrieval was fine, generation failed.”
  or
- “Generation tried, retrieval failed.”

That is real progress.

Copyright © Blockchain Council | All rights reserved

Chapter 7: Evaluation for Beginners (Did We Actually Arrive, or Just Feel Like It?)

7.9 Evaluation Changes When You Add Tools

Tools introduce a new kind of failure: the model can be conceptually correct but *operationally* wrong.

Examples:

- It should call the calculator, but does not.
- It calls the calculator with wrong arguments.
- It calls the correct tool but ignores the result.
- It keeps calling tools in a loop when it should stop.

So your evaluation should include a small “tool behavior” score.

Beginner-friendly scoring:

- **2 = tool used correctly and result used correctly.**
- **1 = tool used but with a mistake (wrong args or sloppy integration).**
- **0 = tool not used when needed, or tool result ignored.**

This turns tool use from magic into measurable behavior. And measurable behavior is what makes you confident enough to build larger systems later.

Copyright © Blockchain Council | All rights reserved

Chapter 7: Evaluation for Beginners (Did We Actually Arrive, or Just Feel Like It?)

Worked Example: The Checkpoint Logbook (A Tiny Evaluation in Action)

Let's make this concrete with a simple scenario that matches where your expedition is right now.

You have built:

- A small library of documents (retrieval)
- A couple of tools (calculator, search stub)
- A prompt that tells the guide to be structured and grounded

Now you want to know: **is it actually reliable?**

The Setup: A Mini "Trail Operations Assistant"

Imagine you have three local documents:

a. **Trail Rules** (camp rules, safety guidelines, what to do in storms)
b. **Supply List** (water, food, weights, quantities)
c. **Route Notes** (mileage between checkpoints, altitude gain, expected time)

Your assistant should answer questions like:

- "How much water should we carry for a 12-mile day?"
- "What do we do if lightning starts?"
- "What is the mileage from Pine Ridge to Lake Junction?"
- "What is 17,493 × 48?" (calculator tool)

Now here is the problem: if you just test three questions and get three good answers, you will be tempted to declare victory.

Instead, we create a tiny test set.

Copyright © Blockchain Council | All rights reserved

Chapter 7: Evaluation for Beginners (Did We Actually Arrive, or Just Feel Like It?)

Worked Example: The Checkpoint Logbook (A Tiny Evaluation in Action)

A Tiny Test Set (10 Cases)

Here is what 10 cases might look like:

a. Lightning protocol (doc-based)
b. Water per person per mile (doc-based)
c. Mileage between checkpoints (doc-based)
d. Food weight calculation (doc + calculator)
e. Taxed supply cost (calculator)
f. An ambiguous question that should trigger a clarifying question
g. Question outside the docs (should say it does not know)
h. Long multi-constraint question (format + correctness)
i. Retrieval trap (similar words, wrong chunk risk)
j. Tool trap (math embedded in text)

The Rubric (Simple 0–2 Scores)

Score each answer on:

- Correctness (0–2)
- Grounding (0–2)
- Format and clarity (0–2)
- Tool behavior (0–2, only when tools apply)

Now, even with just 10 cases, you can compare versions:

- Version A: prompt only
- Version B: prompt + retrieval
- Version C: prompt + retrieval + tools

And you will see something real:

- Retrieval improves grounding.
- Tools improve numeric accuracy.
- Some prompts improve clarity but reduce completeness.
- Some changes “feel better” but score worse.

That is the entire point.

Copyright © Blockchain Council | All rights reserved

Chapter 7: Evaluation for Beginners (Did We Actually Arrive, or Just Feel Like It?)

7.10 Let's Implement: Build a Tiny Test Set and Score Outputs (Handheld Mode)

You are going to build a tiny "arrival check" you can repeat anytime. Not a whole evaluation programme. Just a quick checkpoint that answers one question honestly: *Did we actually arrive, or did the output just sound confident?*

Step 1: Open Gemini AI Studio (Chat)

a. Go to aistudio.google.com/prompts/new_chat
b. Sign in.
c. Start a fresh chat (new conversation).

Step 2: Turn on the two tools we'll test

In the right-side panel (often called **Run settings** or **Settings**), find **Tools** and switch on:

- **Code execution** (our "calculator")
- **Grounding with Google Search** (our "radio")

If you can't find Tools immediately, look for a small gear icon or a collapsible side panel and expand it until you see **Tools**.

Step 3: Paste the "Checkpoint Guide" instruction (one-time setup)

In the same settings panel, find System instructions (or any "Instructions" box) and paste this:

Checkpoint Guide (System Instructions)

You are my expedition guide. Follow these rules:

a. If a question requires math, use code execution. Do not estimate.

b. If a question asks for "latest", "today", or "this month", use Google Search grounding and cite sources.

c. If I give you a "Field Notes Pack", treat it as the only trusted local source for that question. If the answer is not in it, say: "Not in the field notes."

d. Keep answers short and structured. No motivational speeches. We're hiking.

This mirrors the vibe of Chapter 6: *User question → tool runs (when needed) → grounded answer.*

Copyright © Blockchain Council | All rights reserved

Chapter 7: Evaluation for Beginners (Did We Actually Arrive, or Just Feel Like It?)

7.10 Let's Implement: Build a Tiny Test Set and Score Outputs (Handheld Mode)

Step 4: Create your mini "Field Notes Pack" (copy-paste once)

Copy this into a separate note (Notepad, Notes app, anywhere). You will paste it only for the field-notes questions.

Field Notes Pack (Local Notes) :

Trail Rules (excerpt)

- Lightning: If thunder is heard, move to lower ground, avoid isolated trees and ridgelines, and wait at least 30 minutes after the last thunder before continuing.
- Water: Default is 2.5–3.5 liters per person per day. Increase in heat or steep climbs.
- Fires: Only in designated rings. Never during high-wind conditions. Fully extinguish until cold to touch.

Route Notes (excerpt)

- Day 2 route: Pine Ridge → Lake Junction is listed as 6.2 miles on the route card.
- Lake Junction bridge status is not included in these notes.

Supply Notes (excerpt)

- Meals: Plan 3 meals/day/person.
- Emergency: carry 1 spare meal/person/day for safety margin on unfamiliar terrain.

Copyright © Blockchain Council | All rights reserved

Chapter 7: Evaluation for Beginners (Did We Actually Arrive, or Just Feel Like It?)

7.10 Let's Implement: Build a Tiny Test Set and Score Outputs (Handheld Mode)

Step 5: Make a tiny test set (only 6 cases)

Open a simple sheet (Google Sheets or Excel). Create columns:

- ID
- Question
- Expected check (1 line)
- Model output
- Score (0–2)
- What went wrong (short note)

Now add these 6 rows:

1. **ID**: Q1
   Question: (paste Field Notes Pack) Then ask: "What should we do if lightning starts during the hike?"
   Expected check: Must mention lower ground, avoid isolated trees/ridges, 30-minute rule.

2. **ID**: Q2
   Question: (paste Field Notes Pack) Then ask: "How many miles is Pine Ridge to Lake Junction?"
   Expected check: Must answer 6.2 miles.

3. **ID**: Q3
   Question: (paste Field Notes Pack) Then ask: "Is the bridge at Lake Junction open today?"
   Expected check: Must say "Not in the field notes." (No guessing.)

4. **ID**: Q4
   Question: "Calculate 17,493 × 48. Use the tool."
   Expected check: Must use code execution; must be exact.

5. **ID**: Q5
   Question: "What is the latest Node.js LTS version today? Cite sources."
   Expected check: Must use Search grounding; must include citations.

6. **ID**: Q6
   Question: "Give tomorrow's plan in under 90 words. Include water guidance."
   Expected check: Must stay under 90 words; must include water range.

Copyright © Blockchain Council | All rights reserved

Chapter 7: Evaluation for Beginners (Did We Actually Arrive, or Just Feel Like It?)

7.10 Let's Implement: Build a Tiny Test Set and Score Outputs (Handheld Mode)

Step 6: Run the 6 cases in Gemini AI Studio

For each row:

a. Paste the question into the chat.
b. Copy the model's answer into Model output (paste raw, do not edit).
c. Score it.

Step 7: Score with a beginner-proof rubric (0–2)

Use the same scoring logic every time:

- 2 = Pass: Meets the expected check cleanly. No invented facts.
- 1 = Partial: Mostly right, but missing a key constraint or slightly sloppy.
- 0 = Fail: Wrong, invented, ignored tools, or broke the rule (like guessing bridge status).

You're not trying to be a strict examiner. You're trying to be a reliable navigator.

Step 8: Make one improvement and rerun only 2 cases (not all 6)

Change **one** line in your System Instructions:

Add:

- "If you use Field Notes, quote the exact sentence you relied on."

Now rerun **Q1** and **Q2** only. Rescore.

If scores improve, you've learned something real: tighter grounding instructions reduce "confident wandering."

Copyright © Blockchain Council | All rights reserved

Chapter 7: Evaluation for Beginners (Did We Actually Arrive, or Just Feel Like It?)

7.11 Troubleshooting (Because Something Will Be Weird)

“My scores are inconsistent.”

Good. That means your rubric is too vague.

Fix: rewrite rubric notes to be specific.
Bad: “Good answer.”
Better: “Must mention X and Y. Must not invent Z. Must be under 120 words.”

“My model is sometimes great, sometimes awful.”

That is normal. Generative models have variance.

Fix: evaluate across multiple cases, not single examples.
If needed later: run each test case twice and average.

“Retrieval answers are wrong even with the right docs.”

Two common causes:

- The retrieved chunk is not actually relevant
- The model ignored the chunk because your prompt did not force grounding

Fix: tighten instruction: “Use only the provided context. If missing, say you don’t know.”

“Tool calls look correct, but final answers are wrong.”

Classic integration failure: the model “forgets” to use the tool result.
Fix: instruct: “After tool result, restate key values and use them in the final answer.”

Copyright © Blockchain Council | All rights reserved

Chapter 7: Evaluation for Beginners (Did We Actually Arrive, or Just Feel Like It?)

Analogy Tie-Back: Summit Photos vs Coordinates (What You Just Gained)

A beginner's AI project often runs on summit photos.

You ask a few questions. The model answers nicely. You take a mental screenshot. You tell yourself: "We made it."

But photos are not proof of location. Coordinates are proof. Evaluation is how you get coordinates.

It gives you a trail log:

- Where you succeeded
- Where you failed
- What changed between versions
- What improved and what got worse
- Which failures are retrieval problems, prompt problems, or tool problems

And most importantly, evaluation gives you something that makes the rest of this book feel possible:

Confidence that comes from measurement, not hope.

So tonight, the expedition ends properly. You reach a real checkpoint camp, not just a pretty viewpoint. You hang the logbook on a hook inside the tent. And when you wake up tomorrow for the next chapter, you will not be wandering. You will be navigating.

Since now you can answer the only question that matters at the end of any day on the trail:

Did we actually arrive, or did we just feel like it?

Copyright © Blockchain Council | All rights reserved

Part C:
The Bridge
(From Guide to Crew)



Copyright © Blockchain Council | All rights reserved

Chapter 8: From Prompts to Workflows (Repeatable Routes)

How to Stop Winning “Once” and Start Winning “On Purpose”

You’ve reached a real checkpoint camp now. You have a map reader (the model), you know how to pack your backpack (tokens and context), and you’ve learned to give directions that don’t get you eaten by wolves (prompting). You’ve used examples to show the route (few-shot), you can break a climb into safe switchbacks (decomposition), you’ve built a small library of field notes (retrieval), you’ve handed your guide equipment (tools), and you’ve started logging proof of arrival instead of trusting summit photos (evaluation).

So why do we need workflows?

Because a single great prompt is like a single great day on the trail, it feels amazing. It also does not automatically repeat the next day.

A workflow is how you turn “we got lucky” into “we can do this every time.”
This is the bridge to Agentic AI. Before you give a system autonomy, you teach it a routine. Before you build a crew, you define the handoffs. Before you run an expedition for weeks, you write down the route cards.

This chapter is about one simple shift: **stop treating the model like a vending machine** (“insert prompt, receive magic”), and start treating it like a capable teammate who thrives on a **repeatable process.**

Copyright © Blockchain Council | All rights reserved

Chapter 8: From Prompts to Workflows (Repeatable Routes)

8.1 What a Workflow Is

A prompt is one instruction. A workflow is a series of prompts where each step has a job, and the output of one step becomes the input to the next.

That sounds basic because it is basic. It is also the difference between “occasionally impressive” and “dependably useful.”

A workflow has:

- A start state (what you begin with).
- A sequence of steps (each step has a narrow purpose).
- A handoff format (how one step passes information to the next).
- A finish state (what “done” looks like).
- Optional checkpoints (quality gates, scoring, or validation).

In expedition terms:

- A prompt is “Take me to the lake.”
- A workflow is “Take the ridge trail, stop at the stream, check the weather at the saddle, and if visibility drops, turn back.”

Copyright © Blockchain Council | All rights reserved

Chapter 8: From Prompts to Workflows (Repeatable Routes)

8.2 Why One-Prompt Solutions Break in Real Life

Beginners love one-shot prompts because they sometimes work brilliantly. Then they fail in ways that feel random, unfair, and personal.

It's not personal. Its structure.

One-prompt solutions break because:

The task is secretly multi-stage.

Most useful outcomes are not a single action. Writing a good email often requires: understanding intent, picking tone, drafting, checking for missing details, revise. The one-prompt approach tries to do all of that at once.

Quality depends on "hidden steps."

When you read a great response, you don't see the steps the model did "in its head." Workflows pull those steps out into the open so you can control them.

You cannot reliably self-correct mid-flight.

A one-shot response is like committing to a route before you see the weather. A workflow is checking the sky at the saddle and adjusting.

You need repeatability, not entertainment.

If the output needs to be used for work, learning, or a product, you want the same quality on Tuesday that you got on Sunday.

Workflows are repeatable.

Copyright © Blockchain Council | All rights reserved

Chapter 8: From Prompts to Workflows (Repeatable Routes)

8.2 Why One-Prompt Solutions Break in Real Life

Beginners love one-shot prompts because they sometimes work brilliantly. Then they fail in ways that feel random, unfair, and personal.

It's not personal. Its structure.

One-prompt solutions break because:

The task is secretly multi-stage.

Most useful outcomes are not a single action. Writing a good email often requires: understanding intent, picking tone, drafting, checking for missing details, revise. The one-prompt approach tries to do all of that at once.

Quality depends on "hidden steps."

When you read a great response, you don't see the steps the model did "in its head." Workflows pull those steps out into the open so you can control them.

You cannot reliably self-correct mid-flight.

A one-shot response is like committing to a route before you see the weather. A workflow is checking the sky at the saddle and adjusting.

You need repeatability, not entertainment.

If the output needs to be used for work, learning, or a product, you want the same quality on Tuesday that you got on Sunday.

Workflows are repeatable.

Copyright © Blockchain Council | All rights reserved

Chapter 8: From Prompts to Workflows (Repeatable Routes)

8.3 The Three Upgrades Workflows Give You

Upgrade 1: Separation of concerns

Each step does one thing. That reduces confusion and improves consistency.

Upgrade 2: Quality gates

You can add a critique step that checks for missing items, hallucinations, weak reasoning, or style mismatch.

Upgrade 3: Better controllability

You can change one step without rewriting everything. If your critique is too strict, you tune that step. If your draft is too long, you tune the draft step.

That is how real systems evolve.

Copyright © Blockchain Council | All rights reserved

Chapter 8: From Prompts to Workflows (Repeatable Routes)

8.4 Draft → Critique → Revise (The Smallest Workflow That Changes Everything)

The simplest workflow that produces dramatic improvements is:

1. **Draft**: Generate a first version quickly.
2. **Critique**: Evaluate it against clear criteria.
3. **Revise**: Rewrite using the critique, not vibes.

Think of this as your first route card.

It works because it mimics how competent humans produce quality: first pass, feedback, second pass.

It also sets you up for later chapters. Multi-agent systems often look fancy, but many are just this pattern with roles split across multiple agents. The idea starts here.

Copyright © Blockchain Council | All rights reserved

Chapter 8: From Prompts to Workflows (Repeatable Routes)

8.5 Why Critique Works (Even When the Model Wrote the Draft)

A fair question: if the same model writes the draft and critiques it, isn't it just talking to itself?

Sometimes. But the trick is that the critique step has a different job. You are forcing the model to shift modes.

The draft step is rewarded (by your prompt) for producing content. The critique step is rewarded for catching problems.

Those are different targets. When you separate them, you often get better results than asking for "a perfect answer" in one go.

In expedition terms, the scout and the builder can be the same person, but not at the same time. First scout. Then build. Don't try to hammer nails while you're still arguing with the map.

Copyright © Blockchain Council | All rights reserved

Chapter 8: From Prompts to Workflows (Repeatable Routes)

8.6 The Handoff Note (The Most Underrated Part of Workflows)

Workflows fail most often at the handoff.

The model drafts something. Then you ask it to critique. Then you ask it to revise.

If you do not manage the handoff, you get these classic failures:

- The critique is vague (“looks good!”).
- The critique invents new requirements.
- The revision ignores the critique.
- The revision rewrites the whole thing in a different tone.
- The revision adds new facts that were never in the input.

A good workflow includes a handoff note that says:

- What is the next step to change.
- What it must preserve.
- What it must output.

This is where beginners start thinking like builders.

Copyright © Blockchain Council | All rights reserved

Chapter 8: From Prompts to Workflows (Repeatable Routes)

8.7 Workflow Design Rules (Simple, Strong, Beginner-Friendly)

Rule 1: Each step gets a single mission

Draft is not allowed to be critiqued.
Critique is not allowed to rewrite. Revise is not allowed to invent new goals.

Rule 2: Make the output format stable

If you want repeatability, do not let the model choose the shape every time.

Rule 3: Use “inputs only” boundaries

If the model doesn’t have a fact in the input, it must not invent it. This matters more as you approach agents.

Rule 4: Make “done” explicit

A workflow is not a vibe. It’s a finish line.

Rule 5: Keep the workflow small

Workflows can grow. Start tiny. Earn complexity.

Copyright © Blockchain Council | All rights reserved

Chapter 8: From Prompts to Workflows (Repeatable Routes)

8.8 Workflows vs Agents (The Bridge, Not the Destination)

A workflow is:

- Predefined steps
- A known order
- A known finish line
- Mostly predictable behavior

An agent is:

- Goal-driven
- May loop
- May choose tools
- May adapt steps based on results
- Needs stronger guardrails

This chapter is not about autonomy. It's about routine.

Routine is how you cross the canyon safely.

Copyright © Blockchain Council | All rights reserved

Chapter 8: From Prompts to Workflows (Repeatable Routes)

Worked Example: Turning Messy Notes into a One-Page Brief (Using a Workflow)

Let's use a real-world style task that beginners can understand immediately:

You have messy notes from a meeting, and you need a clean one-page brief.

Here are "field notes" (messy on purpose):

"Client wants faster onboarding. Complaints: too many steps, unclear docs. They also asked about pricing discounts for the annual plan and whether training can be extended beyond 15 hours. Need to propose a call next week. Risks: support load, expectations. Suggest: create an onboarding checklist, add a short video walkthrough, and update the FAQ. Owner: me. Deadline: Friday draft."

Step 1: Draft (first pass)

A good workflow draft prompt does not ask for perfection. It asks for a structured first version.

Expected draft outcome: a one-page brief with sections like Goal, Current issues, Proposed actions, Open questions, Next steps.

Step 2: Critique (quality gate)

A strong critique checks:

- Missing items from the notes
- Overconfident assumptions
- Clarity and structure
- Actionability

Expected critique outcome: a short list of specific changes.

Step 3: Revise (second pass)

Revision uses critique as a checklist and produces the final brief. Expected revision outcome: clearer, tighter, less invented content, better structure.

This is workflow thinking: you don't demand a perfect first pass. You build a process that reliably produces a good final pass.

Copyright © Blockchain Council | All rights reserved

Chapter 8: From Prompts to Workflows (Repeatable Routes)

8.9 The “Workflow Pack” Mindset (How to Reuse This Forever)

Once you have one workflow, you can reuse it across tasks:

- Draft → Critique → Revise for emails, proposals, study notes, blog outlines
- Draft → Fact-check → Rewrite for grounded content (especially after you add retrieval)
- Plan → Execute → Verify for problem solving

In the expedition, this is your set of route cards. You don’t improvise every day. You pick the card that matches the terrain.

Copyright © Blockchain Council | All rights reserved

Chapter 8: From Prompts to Workflows (Repeatable Routes)

8.10 Let's Implement: Draft → Critique → Revise in Gemini AI Studio (Handheld Mode, No Code)

This exercise is designed to be short, practical, and very obvious in its results.

Goal

You will run a three-step workflow in Gemini AI Studio and watch how the output improves across steps.

What you need

- A Google account
- Gemini AI Studio open in your browser
- A short "messy input" (I'll give you one)

Step 1: Open Gemini AI Studio

a. Go to: aistudio.google.com/prompts/new_chat/

b. Sign in.

c. Make sure you are in a chat-style view (a normal conversation experience).

Step 2: Create a fresh chat for the workflow

a. Start a New Chat (if you see a New Chat button/icon, use it).

b. In the side panel, find Run settings if it is visible. AI Studio commonly exposes a Run settings panel where tools and parameters can be toggled.

c. For this exercise, you do not need tools. Keep it simple.

Copyright © Blockchain Council | All rights reserved

Chapter 8: From Prompts to Workflows (Repeatable Routes)

8.10 Let's Implement: Draft → Critique → Revise in Gemini AI Studio (Handheld Mode, No Code)

Step 4: Run the DRAFT step

Now send this message (you can copy/paste):

DRAFT prompt:

"DRAFT
Input notes:
Client wants faster onboarding. Complaints: too many steps, unclear docs. They asked about pricing discounts for the annual plan and whether training can be extended beyond 15 hours.
Need to propose a call next week. Risks: support load, expectations. Suggest: onboarding checklist, short video walkthrough, update FAQ. Owner: me. Deadline: Friday draft."

Read the result. Don't judge it too hard yet. First drafts are allowed to be a little messy. That's the point.

Step 5: Run the CRITIQUE step

Now send:

CRITIQUE prompt:

"CRITIQUE
Evaluate the DRAFT you just wrote using the contract. Be strict. List what is missing, unclear, risky, or misformatted. Do not rewrite."

Read the critique. You should see a very practical list.

Step 6: Run the REVISE step

Now send:

REVISE prompt:

"REVISE
Rewrite the brief using your critique as a checklist.
Rules: keep facts limited to the input notes. Match the required format. Under 220 words."

Read the revised result.

Copyright © Blockchain Council | All rights reserved

Chapter 8: From Prompts to Workflows (Repeatable Routes)

8.10 Let's Implement: Draft → Critique → Revise in Gemini AI Studio (Handheld Mode, No Code)

Step 7: What you should notice (the "aha" moment)

You should see three visible improvements:

a. Structure stabilizes: the final has a repeatable shape.

b. Missing items get recovered: the critique forces coverage.

c. Hallucination pressure drops: the workflow pushes the model to stay inside boundaries.

If your REVISE step still invents details, you will fix that in Troubleshooting.

Step 8: Make it "repeatable routes" for your own use

Now repeat the same workflow with your own messy notes (a meeting summary, a rough paragraph, a project idea). Keep the contract the same. Only change the input.

That is the whole point: a workflow you can reuse.

Copyright © Blockchain Council | All rights reserved

Chapter 8: From Prompts to Workflows (Repeatable Routes)

8.11 Troubleshooting (Because the Wind Still Exists)

“The critique is too polite and useless.”

Fix: add one line to your CRITIQUE prompt:
“Assume this will be sent to an executive, and mistakes are costly. Be direct.”

“The revise step ignores the critique.”

Fix: force explicit checklist usage. Add this to the REVISE prompt:
“Before the final answer, list the critique items you are addressing in one line each, then produce the final.”

If that feels too long, ask the model to do the checklist silently but still follow it.

“The revise step adds new facts.”

Fix: tighten the boundary:
“Any fact not in the input notes must be marked as [NEEDS CONFIRMATION] instead of being stated.”

“Everything becomes longer instead of clearer.”

Fix: add compression constraints:
“Total length under 180 words. Prefer short sentences. Remove filler.”

“The model changes tone between steps.”

Fix: lock tone in the contract:
“Tone: clear, professional, neutral. No hype.”

Copyright © Blockchain Council | All rights reserved

Chapter 8: From Prompts to Workflows (Repeatable Routes)

Analogy Tie-Back: Route Cards Are How Expeditions Scale

You can cross the canyon once by luck, bravery, or a questionable jump.

You cross it repeatedly by building a rope bridge and writing route cards.

That is what you did here.

You turned prompting from “say the right magic words” into “run the route.”

And the story ends cleanly for today: you don’t wander off into an endless forest of prompts. You build a repeatable crossing method, you pack it away, and you camp on the far side of the gap with a new capability.

Tomorrow’s terrain changes again, because once you have workflows, the next question appears naturally:

If the guide can run a route, can the guide run a routine?

That’s the doorway into agentic thinking. Not “one prompt, one answer,” but “a goal, a loop, a few tools, and a stop condition.”

That is the next chapter: **What “Agentic” Actually Means (Goals, Tools, Loops).**

Copyright © Blockchain Council | All rights reserved

Part D: Agentic AI (Running the Expedition)

Agentic AI



Copyright © Blockchain Council | All rights reserved

Chapter 9: What “Agentic” Actually Means (Goals, Tools, Loops)

Story Beat: The Day the Guide Stops Waiting for You

In the last chapter, you built route cards. Not “magic prompts.” Not vibes. Actual repeatable routes: draft, critique, revise. You crossed the canyon, wrote down the steps, and camped on the far side with something rare in beginner land: a method you can reuse.

Now morning shows up with a small inconvenience. The terrain has changed again.

A workflow is a route you can run. But someone still has to *run it*. Someone has to look at the weather, notice the rope is fraying, decide whether to take the ridge or the valley, and stop the group before they march confidently into a swamp.

This is the moment the expedition stops being “me and a clever guide” and becomes “a crew that can operate over time.”

That shift has a name: **agentic**.

Not because the model suddenly becomes alive. Not because it grows free will. But because you move from single-turn instructions to goal pursuit across multiple steps, you use tools, loops, and a memory of what has already happened.

If generative AI is the guide who answers you when you radio in, agentic AI is what happens when you give the guide a job to finish, a kit of tools, a logbook, and permission to take multiple steps until the job is done, then stop.

Today’s focus is simple: by the end of this chapter, you will be able to explain agentic AI without mysticism, and you will be able to “see” an agent loop in your head the same way you can now “see” tokens and workflows.

We’ll keep the expedition story tight. We’ll make the technical pieces real. And we’ll keep the ending clean so it doesn’t feel like we’re wandering into an endless forest.

Copyright © Blockchain Council | All rights reserved

Chapter 9: What “Agentic” Actually Means (Goals, Tools, Loops)

9.1 The Small Leap From Workflows to “Agents”

A workflow is structured, prompting over multiple steps. An agent is structured prompting plus a loop. That sounds minor until you feel what it changes.

A workflow says:

Do step 1, then step 2, then step 3.

An agent says:

Keep going until the goal is achieved (or we hit a stop condition), and adapt if reality changes.

In expedition terms:

- A workflow is a laminated route card: “Cross the canyon safely.”
- An agent is the expedition lead who uses route cards, checks conditions, and makes decisions over time.

This matters because the real world is not a single prompt.

Real tasks have:

- Missing information
- Changing requirements
- Unreliable sources
- Tool failures
- Decisions that depend on intermediate results
- And the most annoying constraint of all: you do not get infinite tries.

So the natural next step after workflows is not “make prompts longer.”

It’s: **build a loop that can run a plan, observe results, update state, and stop.**

Copyright © Blockchain Council | All rights reserved

Chapter 9: What “Agentic” Actually Means (Goals, Tools, Loops)

9.2 A Precise Definition (Without the Hype)

Here’s a definition that holds up under pressure:

Agentic AI is a system pattern where a model helps pursue a goal across multiple steps by running a loop that chooses actions (often tool calls), observes outcomes, updates state, and decides whether to continue or stop.

Notice what that definition avoids:

- It doesn’t claim the model “wants” anything.
- It doesn’t claim autonomy as a personality trait.
- It doesn’t pretend that the model becomes a robot.

It’s a *pattern*.

And that pattern has parts you can name and design.

Copyright © Blockchain Council | All rights reserved

Chapter 9: What “Agentic” Actually Means (Goals, Tools, Loops)

9.3 The Four Parts You Must See (Or You’ll Get Confused Later)

Most agentic systems, from simple to advanced, have the same core ingredients:

a. **Goal**
b. **State**
c. **Actions (Tools)**
d. **Loop + Stop Conditions**

If you can point to these four in a system, you can understand it. If you can’t, you’re probably looking at marketing.

Let’s unpack each part using the expedition lens.

Copyright © Blockchain Council | All rights reserved

Chapter 9: What “Agentic” Actually Means (Goals, Tools, Loops)

9.4 Goals: “What Are We Trying to Finish?”

A goal is not a prompt. A goal is a destination with success criteria.

Bad goal (vague, unfinished):
“Help me with my business.”

Better goal (bounded, finishable):
“Draft a one-page onboarding brief for a new customer, using my notes, with sections: Goal, Steps, Risks, Timeline, and Open Questions.”

Best goal (finishable plus measurable):
“Draft a one-page onboarding brief from my notes. Include the five sections listed. Must not invent facts. If something is missing, list it as a question. Keep it under 350 words.”

In expedition terms:

- A bad goal is “let’s explore.”
- A good goal is “reach camp by sunset using the ridge route; if visibility drops, stop at the lower shelter.”

Goals need **constraints** because constraints are what make “done” real.

A beginner-friendly rule:

If you cannot say what “done” looks like, you do not have a goal yet. You have a hope.

Copyright © Blockchain Council | All rights reserved

Chapter 9: What “Agentic” Actually Means (Goals, Tools, Loops)

9.5 State: “What Does the Expedition Remember Between Steps?”

State is what the system carries forward, so it doesn’t keep resetting to zero.

This is where many beginners get tangled because they confuse three different things:

a. **Context window** (what fits in the model’s input right now)
b. **Session memory** (what we’ve done so far in this run)
c. **Long-term memory** (what we store across days, projects, or users)

You already met context windows back in the backpack chapter. You learned the hard truth: the guide can only “see” what’s on the table.

State is how you stop re-reading the entire notebook every time you take a step.

In expedition terms:

- Context window is what’s visible in the tent right now.
- Session memory is today’s running log: “We tried the ridge, it was iced, we turned back.”
- Long-term memory is the expedition archive: “In this region, afternoon fog is common. Pack extra layers.”

For this chapter, we only need the simplest version:

State = the running log of what we did, what we learned, and what remains.

Even a tiny state makes an agent feel dramatically more capable because it stops repeating work.

Copyright © Blockchain Council | All rights reserved

Chapter 9: What “Agentic” Actually Means (Goals, Tools, Loops)

9.6 Tools: “What Can We Do Besides Talk?”

Tools are how an agent affects the world or retrieves truth from outside the model’s head.

If the model is the guide, tools are the gear:

- Compass
- Calculator
- Radio
- Map archive
- Checklist
- Calendar
- Search
- Database
- Code runner
- Document reader

You already learned tool use earlier: the key idea was that the model shouldn’t pretend it computed something if a calculator exists.

In an agent loop, tools become more than helpful extras. They become actions the system can choose.

A tool call is not just “using a feature.”

It is a decision:

- “I should calculate this.”
- “I should look this up.”
- “I should verify.”
- “I should write a draft.”
- “I should critique what I wrote.”
- “I should stop and ask the user a question.”

This is why agentic AI is inherently *systems-y*. The model is only one part.

Copyright © Blockchain Council | All rights reserved

Chapter 9: What “Agentic” Actually Means (Goals, Tools, Loops)

9.7 The Loop: The Small Machine That Creates “Agency”

A loop is what turns a capable assistant into a goal pursuer.

The simplest useful loop looks like this:

a. **Observe** (read the goal + current state)
b. **Decide** (choose the next action)
c. **Act** (use a tool or produce an output)
d. **Record** (update state with what happened)
e. **Check stop conditions** (done? stuck? unsafe to continue? out of budget?)
f. **Repeat or stop**

That’s it, no magic. Just iteration with memory.

In expedition terms:

- Observe: check the map, weather, and where you are.
- Decide: pick the next move.
- Act: walk, climb, radio, measure.
- Record: log what worked and what didn’t.
- Stop: camp, reroute, or abort if conditions are bad.

If you understand this loop, you understand the heart of agentic AI.

Copyright © Blockchain Council | All rights reserved

Chapter 9: What “Agentic” Actually Means (Goals, Tools, Loops)

9.8 Stop Conditions: The Most Underrated Skill

A stop condition is the rule that prevents your agent from taking “just one more step” forever. If workflows taught you repeatability, stop conditions teach you **control**.

Good stop conditions fall into a few categories:

Success stops

- “We produced the final answer in the required format.”
- “The rubric score is above threshold.”
- “All checklist items are satisfied.”

Budget stops

- “We hit max steps.”
- “We hit the token/cost budget.”
- “We hit the time limit.”

Quality stops

- “We are repeating ourselves.”
- “We are guessing too much.”
- “We can’t verify critical facts.”

Escalation stops

- “We need user input to continue.”
- “We found conflicting constraints.”

In expedition terms:
Stop conditions are how you avoid hiking at night because you got excited.

Beginners often miss this and accidentally build the worst kind of agent: a system that can keep going confidently even when it should pause.

Your future self will thank you for learning to stop early.

Copyright © Blockchain Council | All rights reserved

Chapter 9: What “Agentic” Actually Means (Goals, Tools, Loops)

9.9 What Agentic Is Not (So You Don’t Get Tricked)

To stay grounded, keep these distinctions sharp:

Agentic is not:

- “a chatbot that writes long answers”
- “a prompt with extra steps”
- “a model that sounds confident”
- “a UI feature called ‘Agent Mode’”

Agentic is:

- A loop
- Plus state
- Plus actions
- Plus stopping rules
- Aimed at a goal

If you can’t point to the loop and the stops, you are not looking at an agent. You are looking at a conversation.

And conversations can be excellent. They’re just not the same thing.

Copyright © Blockchain Council | All rights reserved

Chapter 9: What “Agentic” Actually Means (Goals, Tools, Loops)

9.10 Why This Chapter Is the Real Bridge Into Part D

To stay grounded, keep these distinctions sharp:

Agentic is not:

- “a chatbot that writes long answers”
- “a prompt with extra steps”
- “a model that sounds confident”
- “a UI feature called ‘Agent Mode’”

Agentic is:

- A loop
- Plus state
- Plus actions
- Plus stopping rules
- Aimed at a goal

If you can’t point to the loop and the stops, you are not looking at an agent. You are looking at a conversation.

And conversations can be excellent. They’re just not the same thing.

Copyright © Blockchain Council | All rights reserved

Chapter 9: What “Agentic” Actually Means (Goals, Tools, Loops)

Worked Example: The Camp Quartermaster (A Tiny Agent Loop in the Wild)

Let’s build an agent in your head first, before you build one in a notebook.

You are running a small expedition camp. Your goal:

“Prepare a complete ‘tomorrow plan’ for the crew.”

Constraints:

- Must include: route plan, supplies checklist, risk notes, and a 2-sentence briefing.
- Must not invent weather data. If the weather is unknown, state what you assume and what you would check.
- Must stop once all sections are completed and checked.

Tools available:

- A checklist tool (to verify required sections)
- A calculator tool (to estimate time blocks)
- A notes tool (your field notes library: yesterday’s observations)

State starts empty:

- Completed sections: none
- Known facts: yesterday’s trail was muddy; one person had low water; the map shows two routes (ridge and valley)

Copyright © Blockchain Council | All rights reserved

Chapter 9: What “Agentic” Actually Means (Goals, Tools, Loops)

Worked Example: The Camp Quartermaster (A Tiny Agent Loop in the Wild)

Now watch the loop:

Step 1: Observe
Goal + constraints + current state.

Step 2: Decide
First action should be to outline the required sections so we don’t forget anything.

Step 3: Act
Create a structured draft skeleton with four sections.

Step 4: Record
State updates: “Skeleton drafted.”

Step 5: Stop check
Done? No. We have structure, not content.

Loop again.

Step 1: Observe
We need a route plan, supplies, risks, and a briefing.

Step 2: Decide
We should choose a route. But we do not have weather. We should decide based on known facts and make assumptions.

Step 3: Act
Draft route plan:

- Prefer the valley if the ridge is slippery and visibility is unknown.
- Estimate time: use calculator assumptions (distance and pace).

Step 4: Record
State: “Route plan drafted with assumptions.”

Step 5: Stop check
Done? No.

Loop again.

Copyright © Blockchain Council | All rights reserved

Chapter 9: What “Agentic” Actually Means (Goals, Tools, Loops)

Worked Example: The Camp Quartermaster (A Tiny Agent Loop in the Wild)

Decide
The supplies checklist should reflect yesterday’s issues (mud, low water). Use the notes tool to pull relevant reminders.

Act
Supplies checklist:

- Extra water
- Traction support
- Dry socks
- Simple first aid
- Headlamp (stop condition guard in the real world)

Record
State: supplies complete.

Loop again.

Decide
Write risk notes based on known facts and unknowns.

Act
Risks:

- Dehydration
- Muddy trail slows pace
- Unknown weather changes route safety
- Mitigation: start earlier, carry extra water, confirm weather before departure

Record
Risks complete.

Loop again.

Copyright © Blockchain Council | All rights reserved

Chapter 9: What “Agentic” Actually Means (Goals, Tools, Loops)

Worked Example: The Camp Quartermaster (A Tiny Agent Loop in the Wild)

Decide
Write the 2-sentence briefing that uses the plan and keeps the tone calm.

Act
Briefing written.

Stop check
Now we run the checklist tool:

- Route plan present? yes
- Supplies present? yes
- Risks present? yes
- Briefing present? yes
- Invented weather? no

Stop.

That was an agent loop.

Not because the system was alive. Because it:

- Pursued a goal across steps
- Used a state to track what was done
- Used tools to support decisions
- And stopped when conditions were met

Copyright © Blockchain Council | All rights reserved

Chapter 9: What “Agentic” Actually Means (Goals, Tools, Loops)

9.10 Demonstration First, Implementation Optional

Up to this point in the expedition, you have been practising commands (prompts), constraints (rules), and checks (evaluation). In the Agentic section, the skill shifts: you are no longer “asking the guide a question.” You are giving the guide a goal and watching it run a loop until it reaches a stop condition. That makes implementation more setup-heavy (libraries, keys, runtime quirks) and more failure-prone (structured outputs breaking, retries, tool timeouts).

To understand this chapter’s core idea, it is strongly recommended to go through this Google Colab Notebook:

https://colab.research.google.com/drive/1zWKE-USjyFf9bFN5N-dtXIWJTDcYqzT5?usp=sharing

To implement it yourself (optional)

You will need a **Gemini API key**, and you should use **gemini-2.5-flash** for this exercise.

Step A: Get your Gemini API key (Google AI Studio)

1. Open the API key page in Google AI Studio and sign in with your Google account.
2. Create a new API key and copy it somewhere safe.

Step B: Use the preferred model

Inside def_llm() in candidates paste: gemini-2.5-flash. For best results.

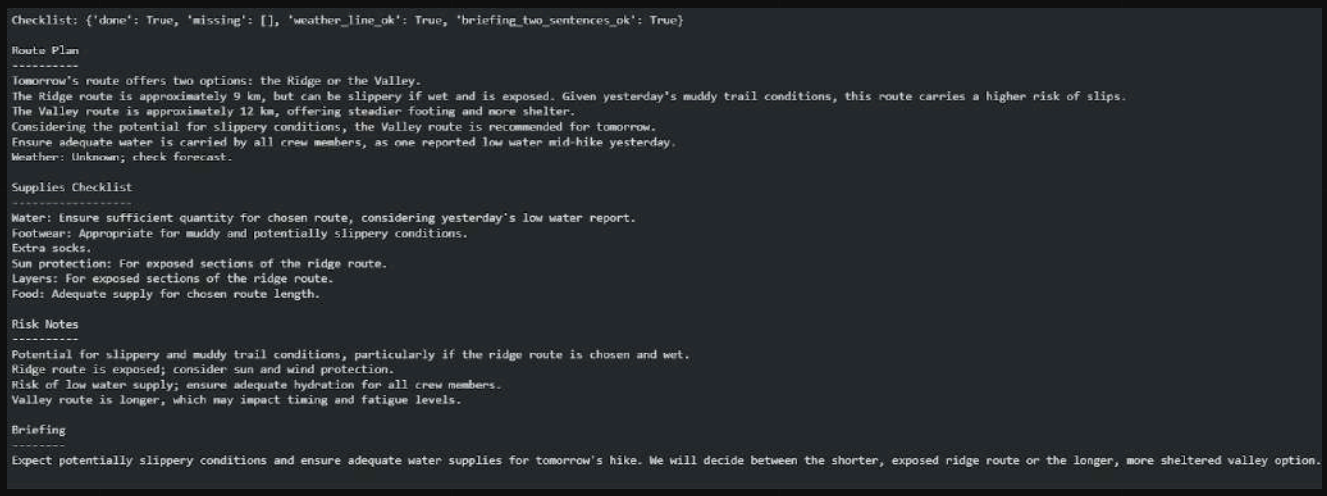
Copyright © Blockchain Council | All rights reserved

Chapter 9: What “Agentic” Actually Means (Goals, Tools, Loops)

9.11 Demonstration Walkthrough

What you’re looking at (and why it proves “agentic”)

Below is the kind of output you saw in your run (the “Route Plan / Supplies / Risks / Briefing” format with a checklist at the top). This output is not just a nicely formatted answer. It’s evidence of an agentic loop being executed.



```
Checklist: {'done': True, 'missing': [], 'weather_line_ok': True, 'briefing_two_sentences_ok': True}

Route Plan
----------
Tomorrow's route offers two options: the Ridge or the Valley.
The Ridge route is approximately 9 km, but can be slippery if wet and is exposed. Given yesterday's muddy trail conditions, this route carries a higher risk of slips.
The Valley route is approximately 12 km, offering steadier footing and more shelter.
Considering the potential for slippery conditions, the Valley route is recommended for tomorrow.
Ensure adequate water is carried by all crew members, as one reported low water mid-hike yesterday.
Weather: Unknown; check forecast.

Supplies Checklist
------------------
Water: Ensure sufficient quantity for chosen route, considering yesterday's low water report.
Footwear: Appropriate for muddy and potentially slippery conditions.
Extra socks.
Sun protection: For exposed sections of the ridge route.
Layers: For exposed sections of the ridge route.
Food: Adequate supply for chosen route length.

Risk Notes
----------
Potential for slippery and muddy trail conditions, particularly if the ridge route is chosen and wet.
Ridge route is exposed; consider sun and wind protection.
Risk of low water supply; ensure adequate hydration for all crew members.
Valley route is longer, which may impact timing and fatigue levels.

Briefing
--------
Expect potentially slippery conditions and ensure adequate water supplies for tomorrow's hike. We will decide between the shorter, exposed ridge route or the longer, more sheltered valley option.
```

Here’s what happened under the hood, in plain English:

1) A goal was set.

Something like: “*Prepare tomorrow’s hike plan and safety notes based on what we know*.” This matters because agentic work starts with a destination, not a single question.

2) The goal got decomposed into sections (task decomposition).

Instead of one big response, the system breaks the job into smaller deliverables, such as:

- Route recommendation
- Supplies checklist
- Risk notes
- Briefing summary

That decomposition is the “route plan that survives reality.” When the world changes, you can re-run only the part that depends on the changed fact.

Copyright © Blockchain Council | All rights reserved

Chapter 9: What “Agentic” Actually Means (Goals, Tools, Loops)

9.11 Demonstration Walkthrough

3) The agent gathered “state inputs” (notes) before writing.

In the notebook, small “notes” like:

- “yesterday was muddy”
- “someone reported low water”
- “ridge vs valley tradeoffs”

These notes act like **field notes**: they keep the writing anchored.

4) The agent produced a draft plan.

The model wrote content for each section. This is still generative behaviour, but it is now directed by the system’s structure.

5) A checker validated the plan against a rubric (the checklist).

That top line you see, something like:

- `done: True`
- `missing: []`
- `weather_line_ok: True`
- `briefing_two_sentences_ok: True`

…is the system saying: “We required specific properties. We checked them. The output passed.”

This is the key agentic upgrade: **the system doesn’t just “generate.” It generates, checks, and either stops or retries.**

6) Stop condition triggered, loop ended.

When the checklist returns “done: True,” the agent stops. That is the difference between a chat response and an agent run. A chat answer ends when the model feels like it. An agent ends when the **system rules** say it may end.

Copyright © Blockchain Council | All rights reserved

Chapter 9: What “Agentic” Actually Means (Goals, Tools, Loops)

9.12 Why the plan looks the way it looks (observations → causes)

Let’s map visible output to the mechanisms that caused it.

Observation: It compares two route options before recommending one.
Cause: the system prompt (or planner instruction) required comparing alternatives, not picking the first plausible answer. In expedition terms, you forced the guide to *look at the map before marching*.

Observation: It explicitly calls out uncertainty (“Weather: Unknown; check forecast”).
Cause: your current setup does not truly fetch live weather. The agent is doing the correct behaviour: flagging missing information instead of hallucinating it. This is a feature, not a weakness.

Observation: It mentions yesterday’s muddy trail and low water.
Cause: those were injected as “notes/state,” so the output is grounded in the expedition log rather than purely invented.

Observation: The briefing is short, controlled, and formatted.
Cause: constraints like “two sentences” were part of the checker rubric, so the system enforced discipline.

This is the pattern you want the reader to internalize:
Good agentic planning is not a talent. It is a system of constraints plus a loop that enforces them.

Copyright © Blockchain Council | All rights reserved

Chapter 9: What “Agentic” Actually Means (Goals, Tools, Loops)

9.13 Troubleshooting

1) The plan is coherent but “feels generic”

That’s usually because your notes/state are too thin. Your agent can’t invent reality responsibly, so it writes safe generalities.

Upgrade the inputs: add 3–5 more concrete facts (distance windows, crew fitness constraint, water sources, time cut-off).

2) The plan is long and rambly

Your decomposition is correct, but your constraints are weak.

Tighten the rubric: maximum lines per section, required bullets (or required short paragraphs), and a strict briefing limit.

3) The agent “hallucinates” tool results

This happens when you *describe* tools but don’t actually execute them.

Rule of thumb: if the tool didn’t run, the plan must say “unknown” and request a check. Your current output is already moving in that direction, which is good.

4) Gemini API errors (key, quota, or model name) stop the run

If you see auth failures or “quota/rate” style errors, it usually means the API key is missing/incorrect, the model name is mistyped, or you’re hitting limits. Recheck that the key is set in the notebook, confirm the model string is exactly what you intended (for this chapter: `gemini-2.5-flash`), restart the runtime, then retry with slower pacing if you’re rate-limited.

Copyright © Blockchain Council | All rights reserved

Chapter 9: What “Agentic” Actually Means (Goals, Tools, Loops)

9.14 Analogy Tie-Back

In expedition terms, this chapter is the moment you stop treating the guide like a clever conversationalist and start treating the guide like a crew leader running a daily routine:

- The goal is tomorrow’s camp-to-camp objective.
- The decomposition is assigning jobs: route scout, quartermaster, safety lead, briefing lead.
- The checker is the laminated safety card you read every morning before you move.
- The stop condition is the final “all checks green” before you step onto the trail.

And here’s the important shift : You are no longer hoping the guide says something smart. You are designing a system that makes “smart” the default by forcing structure, verifying outputs, and refusing to move forward until the basics are met.

That sets up the next chapter perfectly, because routines alone aren’t enough once reality throws surprises. Tomorrow, we move from “a loop that works” to **Planning and task decomposition: route planning that survives reality.** You’ll learn how to split a messy objective into resilient sub-tasks, run a planner and an executor as separate roles, and use retries when conditions change mid-hike.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

How to Stop “Looping” and Start “Getting Things Done”

You now know what an agent is: not a magical creature, but a system pattern that can pursue a goal across multiple steps. You can picture the loop: decide, act, observe, update, stop.

Here’s the catch: a loop without a plan is just a very determined way to wander.

In the last chapter, we gave the guide permission to take multiple steps. In this chapter, we teach the guide how to choose which steps make sense, in what order, and what to do when reality refuses to cooperate. This is where agentic AI stops feeling like “multi-step prompting” and starts feeling like “a workflow that can survive turbulence.”

We’re going to keep the expedition analogy present but practical. Think of planning as the route plan. Think of decomposition as breaking the hike into navigable segments. And think of retries as what you do when the bridge is out but you still need to reach camp before dark.

By the end of this chapter, you will be able to:

- Explain why agents need planning (not just looping).
- Break a goal into tasks that can be executed safely.
- Understand the planner → executor pattern (and why it is so common).
- Design retries that help instead of creating chaos.
- Recognize when to retry, when to re-plan, and when to stop.

We’ll also keep the ending clean. No endless forest. Just a solid new tool in your pack, ready for the next chapter where tool orchestration becomes the main event.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

Story Beat: The Fog That Makes Good Maps Useless

On paper, today's route looks easy. You have a map. You have a compass. You even have yesterday's notes. The kind of setup that makes people overconfident.

Then fog rolls in.

Not the poetic kind. The annoying kind. The kind that turns a "simple ridge walk" into "I cannot see ten feet ahead and I have zero interest in discovering a cliff by accident."

This is the moment planning becomes real.

A map is not a plan. A plan is what you do with the map, given your constraints, your supplies, your team's condition, and the fact that weather exists purely to humble you.

In agent terms: a model can generate answers, and a loop can keep it moving, but without explicit planning and decomposition, your "agent" will often do one of two things:

- Keep producing plausible text that does not move the work forward
- Get stuck the moment anything changes, because nothing was modular enough to adapt

So we plan. Not because we love bureaucracy. Because we love arriving.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.1 Planning Is Not Optional Once Reality Is Involved

A lot of beginner agent demos look like this:

a. Ask the model what to do
b. Do the first thing
c. Ask the model what to do next
d. Repeat

That can work for tiny tasks. It also collapses under real constraints.
Real tasks have:

- Missing information (you do not know what you do not know)
- Dependencies (you cannot build step 6 before step 2 exists)
- Verification needs (you must check outputs, not just admire them)
- Tool failures (APIs time out, files are missing, permissions block you)
- Changing requirements (stakeholders do that thing where they remember new needs late)

In expedition terms, you cannot just walk forward and hope the mountain respects your optimism.

So planning becomes the stabilizer. It turns “keep looping” into “take deliberate steps toward a goal.”

A simple but durable definition:

Planning is the act of choosing a sequence of actions and checkpoints that can reach a goal under constraints, with a way to adapt when conditions change.

Notice what is inside that definition:

- Sequence of actions
- Checkpoints
- Constraints
- Adaptation

That is the whole game.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.2 Task Decomposition: The Skill That Makes Plans Executable

Decomposition is the difference between:

“Build a cabin.”
and
“Clear the ground, place the foundation, raise the frame, add the roof, seal it, inspect it.”

The model can generate either. The question is whether the second version is designed in a way that your system can actually run.

In agentic AI, decomposition is not just “splitting into smaller tasks.” It is splitting into tasks that are:

- **Independent when possible** (so you can rerun one part without rerunning everything).
- **Checkable** (so you can validate progress).
- **Tool-aligned** (so each task maps to the tools you actually have).
- **Bounded** (so tasks do not become infinite essays).

Here’s the decomposition mistake beginners make:

They break a goal into “sections of a response” rather than “units of work.”

Those are different.

Decomposition Type A: Deliverables

This is about what you will produce.

- Outline
- Summary
- Plan
- Code
- Test results
- Report

Deliverables matter because they give you artifacts to inspect.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.2 Task Decomposition: The Skill That Makes Plans Executable

Decomposition Type B: Actions

This is about what the agent will do.

- Search
- Read a file
- Call an API
- Run a calculation
- Draft a response
- Verify output against criteria
- Retry or replan

Actions matter because they define the actual run loop.

A strong agent plan usually includes both: deliverables you can inspect, and actions you can execute.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.3 The Route Plan That Survives Reality

A brittle plan looks perfect until something changes. Then it breaks like cheap plastic in cold weather.

A resilient plan is different. It expects trouble and still holds shape.

In expedition terms, a resilient route plan has:

- A primary route
- Known hazards
- Alternative options
- Decision points
- Stopping rules

In agent terms, a resilient plan has:

- A main sequence of tasks
- Explicit assumptions
- Defined inputs and outputs for each step
- Validation checkpoints
- Retry and replan triggers

The simplest upgrade: write assumptions down

Assumptions are hidden dependencies. If the plan assumes “the API will respond quickly,” then tool timeout is not an edge case. It is a violated assumption.

If the plan assumes “the source documents exist,” then missing files are not random misfortune. They are a known failure mode.

So a planner should produce assumptions as first-class outputs.

This is the expedition equivalent of saying:

“We assume the river is crossable. If it is not, we take the southern bridge.”

Not glamorous. Extremely useful.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.4 Planning vs Scheduling vs Deciding

These words get mixed up. Let's separate them cleanly.

- Planning: Deciding what to do and why, with checkpoints and alternatives.
- Scheduling: Deciding when to do it (order, time, resource allocation).
- Deciding: Choosing at a fork when new information arrives.

In early agent systems, you often do planning and scheduling together because the environment is small. As systems grow, you may separate them, but you do not need to do that yet.

For beginners building intermediate systems, one robust approach is:

- Do a planning pass that produces ordered steps
- Execute step-by-step
- Allow decisions to update the plan when reality changes

That is the planner → executor pattern in plain language.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.5 The Planner → Executor Pattern (Why Everyone Uses It)

This pattern is popular for a simple reason: it reduces confusion.

Instead of one model prompt doing everything, you assign roles:

The Planner

Produces a structured plan.

- Steps
- Dependencies
- Expected outputs
- Success criteria
- Risks
- Stop conditions

The planner is allowed to think broadly. It is encouraged to be systematic. It should be relatively cheap compared to execution, because you might replan.

The Executor

Runs one step at a time.

- Performs tool calls.
- Produces intermediate artifacts.
- Reports results back into state.
- Asks for validation.
- Retries a step when needed.
- Escalates to replan when needed.

The executor is where reality lives. It is where failures happen. It is where the system either moves forward or gets stuck.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.5 The Planner → Executor Pattern (Why Everyone Uses It)

Why separating them helps

If you combine planning and execution into a single prompt, you usually get:

- Plans that ignore tool constraints.
- Steps that are not checkable.
- A messy blend of “what we should do” and “what we did”.
- Long, expensive outputs that are hard to debug.

Separating planner and executor gives you cleaner logs, clearer reasoning boundaries, and a system you can adjust without rewriting everything.

In expedition terms: the planner draws the route card. The executor walks it.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.6 Plans Need Structure, Not Poetry

A plan that only exists as prose is hard to execute reliably.

Humans can read a paragraph and infer structure. Systems cannot, unless you build a lot of extra parsing and interpretation. So we usually make plans structured.

Beginner-friendly plan structures include:

Option 1: A checklist plan

Each step is a line item with a clear done/not-done state.

- Step 1: gather inputs
- Step 2: draft output
- Step 3: validate output
- Step 4: revise if needed
- Step 5: final deliverable

This is the simplest and often enough.

Option 2: A task object plan

Each step has fields like:

- id
- description
- input_needed
- output_expected
- tool_to_use
- validation_rule
- retry_policy

This is a little more technical, but it scales better.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.6 Plans Need Structure, Not Poetry

Option 3: A dependency graph plan

Steps have dependencies (a DAG, if you like the formal term).

- You cannot do step 5 until steps 2 and 4 are complete.

This becomes valuable once tasks can branch or run in parallel, but you do not need to jump there immediately.

A good beginner rule:

If you cannot point to a step and say "I will know it is done because X," then it is not a step yet.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.7 Execution Is a Conversation With the World

Planning happens in your head. Execution happens in the mud. This is why agent systems need a concept called state.

State is what the system knows so far:

- The plan, versioned
- Which steps are complete
- Intermediate outputs (artifacts)
- Tool results
- Errors encountered
- Decisions made
- Retries attempted

Without state, execution becomes repetitive. The agent forgets what it tried. It loops in place. It keeps “re-discovering” the same failure.

With state, the executor can say:

“We tried the API call. It timed out. We retried twice. We should switch approach or replan.”

That is maturity.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.8 The Three Failure Types You Must Handle

Not all failures are equal. Treating them the same makes your agent either fragile or reckless.

Failure Type 1: Tool failure

The tool did not work.

- Timeout
- Rate limit
- Permission denied
- Network error
- Invalid response format

These often deserve retries, but not infinite retries.

Failure Type 2: Output failure

The model produced output that does not meet requirements.

- Malformed JSON
- Missing required fields
- Violates formatting constraints
- Incomplete reasoning steps

These often deserve a targeted retry with stronger constraints and a checker.

Failure Type 3: Plan failure

The plan itself is wrong.

- Missing prerequisite step
- Wrong assumptions
- Wrong tool choice
- Dependency overlooked

This deserves re-planning, not retries.

A clean agent design distinguishes these and reacts differently.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.8 The Three Failure Types You Must Handle

In expedition terms:

- Tool failure is “the radio is down”
- Output failure is “the packing list forgot water”
- Plan failure is “we planned a ridge route during fog season”

You do not solve fog season by yelling at the radio.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.9 Retries: The Art of Not Making Things Worse

Retries sound simple: try again.

In practice, retries can create chaos if you do not design them carefully.

A retry policy should answer:

- What counts as retryable.
- How many retries are allowed.
- Whether you change anything between retries.
- When to escalate to replan.
- When to stop and surface failure.

The three retry modes

A surprisingly helpful way to think about retries:

a. **Same attempt, same inputs**
   Useful for transient tool failures (timeouts, network hiccups).

b. **Same attempt, improved constraints**
   Useful for output failures (formatting, missing fields).
   You do not need new data, you need stricter structure.

c. **New attempt with modified plan**
   Useful when repeated failures show the plan is wrong.
   This is replan territory.

A practical beginner retry ladder

When something fails:

- Retry once with the same inputs (if tool-related).
- Retry once with stronger constraints (if output-related).
- If still failing, replan that step or switch tools.
- If the plan is collapsing, replan the whole route.
- If constraints make the goal impossible, stop.

This ladder prevents infinite loops and avoids the common beginner tragedy: “the agent kept trying the same broken thing forever.”

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.9 Retries: The Art of Not Making Things Worse

The hidden villain: non-idempotent actions

Some actions are safe to retry. Some are not.

- Reading a file is usually safe.
- Posting a payment is not safe.
- Sending an email is not safe.
- Creating a new record in a database may not be safe.

In agent systems, retries are safest when actions are idempotent, meaning you can repeat them without causing unintended extra effects.

You do not need to memorize the term. You just need to remember the rule:

If repeating a step can cause damage or duplication, add safeguards before you allow retries.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.10 Checkpoints: How You Know You Are Not Just “Generating”

Without checkpoints, an agent can produce pages of confident text while making zero progress.

Checkpoints are validation moments.

They can be:

- Structural checks (did we produce valid JSON?).
- Completeness checks (did we fill all required fields?).
- Quality checks (does it meet the rubric?).
- Sanity checks (are we contradicting ourselves?).

A common agent pattern is:

- Planner creates plan plus acceptance criteria.
- Executor runs a step
- Checker validates output
- If fail: retry or replan
- If pass: mark done and continue

This is how you turn “generation” into “work.”

In expedition terms, checkpoints are the trail markers and safety checks:

- “Are we still on the ridge?”
- “Do we have water?”
- “Is the weather worsening?”
- “Do we stop here or push forward?”

The point is not to slow down. The point is to avoid walking fast in the wrong direction.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.11 A Worked Example (Conceptual, No Code Yet): Planning a Route That Can Adapt

Let's use a clean example that mirrors real agentic work: not too small, not too huge.

Goal: Create a mini learning plan for someone to go from beginner to intermediate on a topic, with checkpoints and a final deliverable.

If you ask a model in one prompt, you might get a nice plan. But if you want a system that can adapt, you do planning plus execution.

Planner output (what "good" looks like)

A planner does not just list steps. It makes them executable.

It might produce something like:

- Step 1: confirm scope and constraints
    - input: topic, time budget, learner background.
    - output: scope summary, constraints list.
    - validation: all fields present.
- Step 2: propose module structure
    - input: scope summary.
    - output: module list with learning goals.
    - validation: modules cover scope, time fits budget.
- Step 3: draft a week-by-week schedule
    - input: module list, time budget.
    - output: schedule with checkpoints.
    - validation: each week has measurable outcome.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.11 A Worked Example (Conceptual, No Code Yet): Planning a Route That Can Adapt

- Step 4: produce final deliverable
  - input: schedule, modules
  - output: formatted plan
  - validation: final format matches template

Plus assumptions and stop conditions, like:

- assumption: learner has 5 hours/week
- stop: if required input missing, request it (do not hallucinate)
- stop: if time budget is too small, propose a reduced plan

That “stop” part matters more than it sounds. It is how you prevent the agent from inventing facts and pretending it is fine.

Executor behavior (how the system runs it)

The executor takes Step 1:

- Asks for missing inputs or reads them from state.
- Writes the scope summary.
- Hands it to a checker.

If the checker says “missing constraints,” the executor retries with a stricter structure.

Then it moves to Step 2, and so on.

Where retries show up

Imagine the executor tries Step 3 and the schedule exceeds the time budget.

That is not a tool failure. That is not a formatting issue.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.11 A Worked Example (Conceptual, No Code Yet): Planning a Route That Can Adapt

That is either:

- An output failure (schedule does not meet criteria), which can be retried.
- Or a plan failure (module list too large), which may require going back and adjusting Step 2.

This is the heart of “route planning that survives reality.”

You do not keep walking the same trail when you realize it does not reach camp. You adjust the route card.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.12 Demonstration: Planner → Executor → Checker → Router (with retry + replan)

To understand this chapter's core idea it is strongly recommended to go through this Google Colab Notebook :

https://colab.research.google.com/drive/1xuig65o7ICNNhJUPTQyR2hGMG2Mgs_B3?usp=sharing

To implement it yourself you can copy-paste the code from the shared notebook in another notebook with your dedicated API key and run it for yourself.

What you need (before opening the Colab link)

To run the notebook yourself (optional), you'll need a Gemini API key and the preferred model for this demo is `gemini-2.5-flash` (same expectation as Chapter 9's setup guidance).

Step A: Get your Gemini API key (Google AI Studio)

a. Open Google AI Studio and go to the API key page (sign in with your Google account).

b. Create a new API key and copy it somewhere safe.

c. Treat the key like a toothbrush: personal, not shareable, and not something you paste into screenshots.

Step B: Add the key in Colab and pick the model

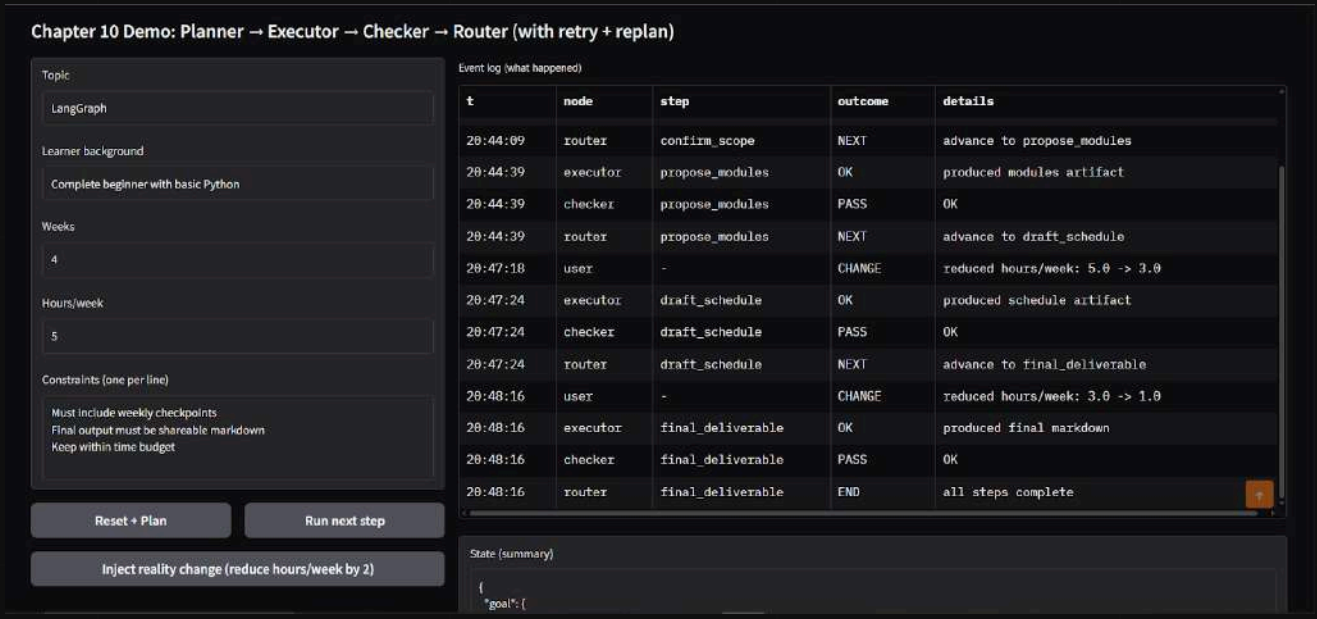
In the notebook, you'll paste the key where the environment variable / secret is read, and ensure the model selection is set to `gemini-2.5-flash`

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.12 Demonstration: Planner → Executor → Checker → Router (with retry + replan)

What you're seeing in the Gradio interface (and why it matters)



The image shows a simple UI, but it's actually a "workflow dashboard":

Left panel (Inputs): This is your **start state.** You provide the goal context (Topic), constraints (hours/week, weeks, constraints list), and the learner background. A workflow always begins with a *known start state*.

Buttons (Controls):

- **Reset + Plan:** Creates the initial "route card" (the plan) for what happens next.
- **Run next step:** Advances the workflow one node at a time (so you can observe it).
- **Inject reality change (reduce hours/week by 2):** Simulates what happens when the world changes mid-route (and forces the workflow to adapt rather than pretend nothing happened).

Right panel (Event log): this is the expedition logbook. It records what actually happened, step-by-step: which node ran, what outcome it produced, and why the router chose the next step.

Bottom panel (Final deliverable markdown): This is the finish state. It proves the workflow didn't just talk about doing work. It produced an artifact you can actually use.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.13 The recommended interaction (designed for screenshot-worthy output)

Use this sequence because it makes the workflow behaviour obvious and produces a clean “story” in the event log.

Run 1: A clean, normal route

a. Enter:

- Topic: LangGraph
- Learner background: Complete beginner with basic Python
- Weeks: 4
- Hours/week: 5
- Constraints:
  - Must include weekly checkpoints
  - Final output must be shareable markdown
  - Keep within the time budget

b. Click Reset + Plan.

c. Click Run next step a few times until you see the workflow progress through steps like:
- confirm_scope
- propose_modules
- draft_schedule
- final_deliverable

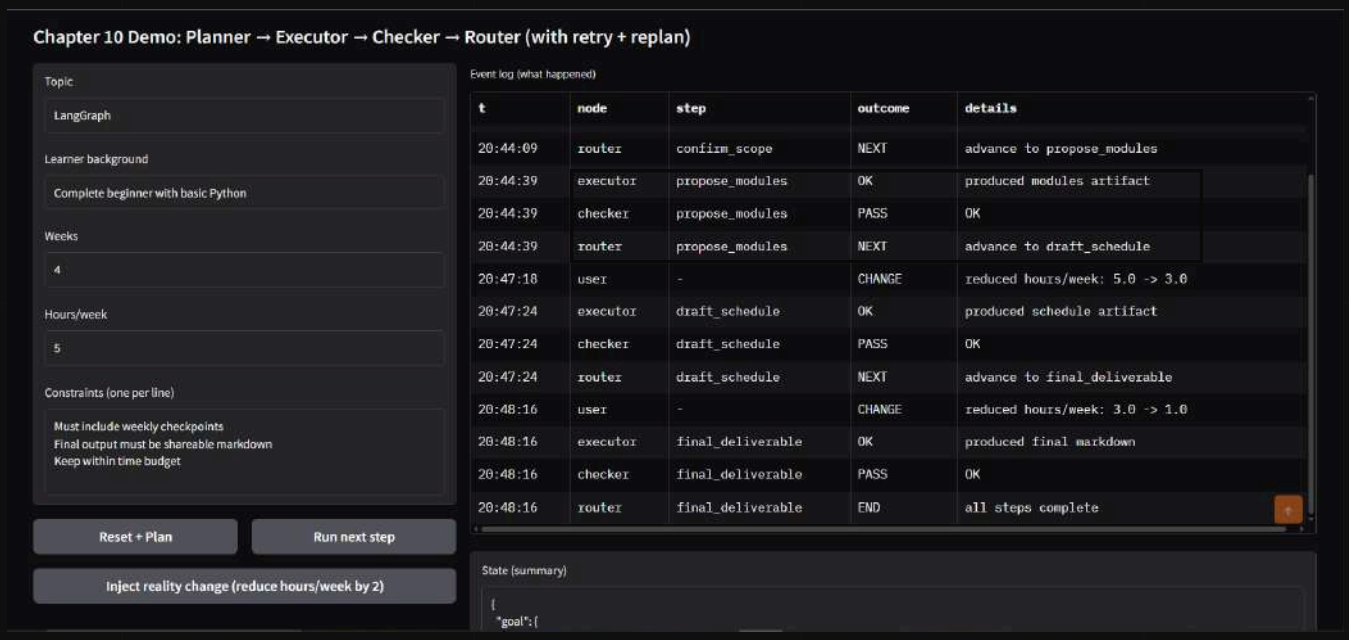
d. Look for the pattern in the event log:

- The executor produces an artifact
- checker validates it (PASS/FAIL)
- The router decides NEXT or END

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.13 The recommended interaction (designed for screenshot-worthy output)



If your run matches the image, you'll see event log rows like:

- executor `propose_modules` OK
- checker `propose_modules` PASS
- router `propose_modules` NEXT
- and finally: `router final_deliverable END`

Copyright © Blockchain Council | All rights reserved

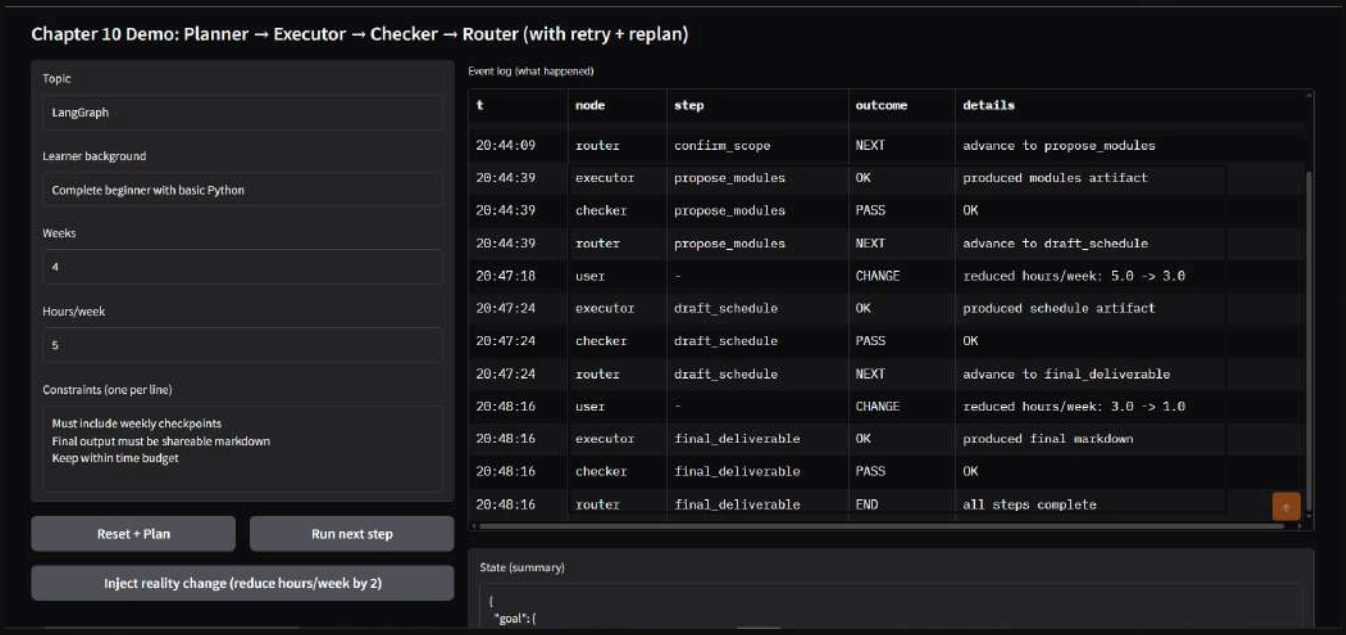
Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.13 The recommended interaction (designed for screenshot-worthy output)

Run 2: "Reality changes" mid-way (this is the money shot)

a. After the workflow has started (or even after you've drafted a schedule once), click:
Inject reality change (reduce hours/week by 2)
It is done twice in the demonstration (5 → 3 → 1). That's perfect.

b. Continue clicking Run next step until completion.



Copyright © Blockchain Council | All rights reserved

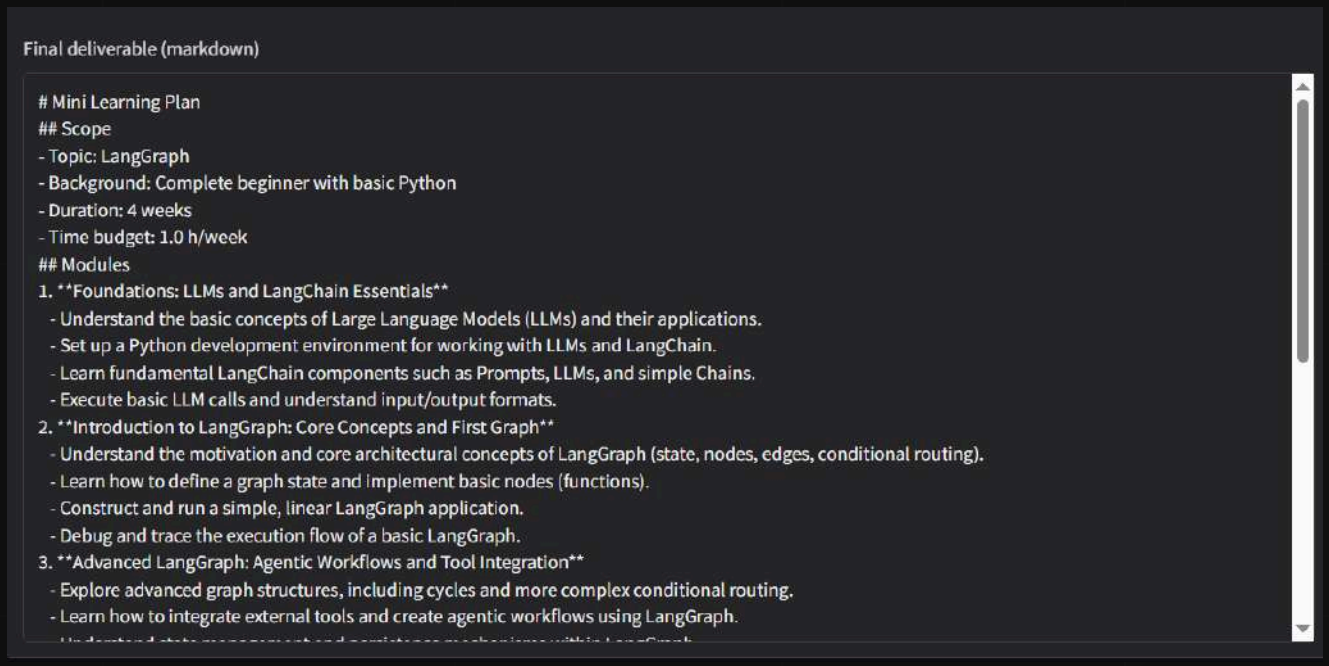
Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.13 The recommended interaction (designed for screenshot-worthy output)

What you should notice (and what your readers should screenshot):

- The event log contains explicit **CHANGE** entries (user-driven state update).
- The workflow still reaches **final_deliverable.**
- The markdown output reflects the new time budget (for example, “Time budget: 1.0 h/week”).

This is exactly the “repeatable routes” shift: the system adapts because the workflow has structure, not because the model is magically wise.



Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.14 What do we deduce from the Demonstration

This chapter claims that **one-prompt solutions break in real life because useful work is secretly multi-stage**, and workflows make those hidden steps explicit and controllable.

This demo is “real” (not theatre) for three reasons:

1) Separation of concerns (each step has a job)

The planner is not also the writer. The checker is not also the planner. Each node has a narrow purpose, which is the first workflow upgrade Chapter 10 talks about: you stop asking for perfection in one go, and you build a process that reliably produces quality.

2) Quality gates are enforced, not suggested

In your earlier failing runs, you saw missing fields and a “FAIL confirm_scope” outcome. That’s actually a good failure because it shows the checker is doing real work: it blocks progress when the handoff is incomplete. In your successful run, you see PASS on key steps, which proves the gates are working.

3) Controllability under change (the workflow survives reality)

The “inject reality change” button is the expedition version of weather turning or a teammate losing water. A one-shot answer would happily ignore the change. Your workflow doesn’t. It records the change, re-routes through the remaining steps, and produces a new final artifact that matches the new constraints.

This is the rope-bridge lesson from the book: you don’t cross by leaping harder, you cross by using a repeatable method.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.15 Troubleshooting (Because the wind still exists)

1) Gemini API failure (most common)

Symptoms

- Authentication error, permission denied, invalid key
- Requests fail instantly
- Output is empty, or the run stops early

Fixes

- Confirm the API key is correctly pasted into the notebook secret/env var (no extra spaces).
- Restart the runtime after setting the key (Colab can be stubborn).
- Ensure the selected model is exactly gemini-2.5-flash.
- If you hit rate limits, wait a bit and rerun (free tiers can throttle).

2) The workflow "runs" but the final output is empty or {}

Likely cause: A strict parser expects structured output, but the model returned something slightly off-format.

Fixes

- Lower the "strictness" of the required schema for intermediate steps (keep strictness highest only for the final deliverable).
- Add a repair step: if JSON parse fails, ask the model to re-emit only valid JSON (no extra prose).
- Reduce output size (shorter artifacts produce fewer formatting failures).

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.15 Troubleshooting (Because the wind still exists)

3) It loops or feels stuck

Likely cause: router stop conditions are too vague, or the “NEXT” logic doesn’t have a clean terminal check.
Fixes

- Make stop conditions explicit: “END only when final_deliverable exists and checker PASS.”
- Add a max-step guard (stop after N steps and print state summary + last failure).
- Log the router decision inputs (why it chose NEXT).

4) The checker is too strict (everything FAILs)

Fixes

- Improve the handoff format: make each node output a clean, minimal structure.
- Relax the checker for early steps (scope can be “good enough”), and tighten at the final artifact.

Copyright © Blockchain Council | All rights reserved

Chapter 10: Planning and Task Decomposition (Route Planning That Survives Reality)

10.16 Analogy tie-back: Route cards, not vibes

If prompts are language, workflows are the route cards. You pick the card that matches the terrain, and you reuse it.

The event log in your demo is the expedition journal: it proves you crossed by procedure, not by luck. And the “reality change” button is the canyon wind showing up right on schedule.

Notice what the router is already doing: deciding what happens next based on state and checks. In the next chapter (Tool orchestration), that router grows up. Instead of routing between “propose_modules” and “draft_schedule”, it routes between tools: which tool to call, what to do when a tool fails, and how confident the system should be before it commits.

So keep this mental picture: today you built route cards. Next chapter, you choose the right gear for the terrain and learn what happens when the gear lies, breaks, or returns something weird.

Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

How to Run the Right Gear at the Right Time Without Pretending It Never Breaks

Chapter 10 taught your expedition how to move in repeatable steps. You stopped doing everything in one breath and started using a planner, an executor, checkpoints, retries, and sensible stopping rules. That was the moment the journey stopped being "chatting with a model" and started being "running a system".

Now we add a new kind of realism: your system has more than one tool, and the world does not politely cooperate.

Tool orchestration is what happens after "tool use" becomes normal. In Chapter 6, you handed the guide a compass, a calculator, and a radio. In this chapter, you teach the guide how to choose which one to use, how to feed it clean inputs, how to recognize when it failed, and how to act responsibly when confidence drops.

By the end of this chapter, you will be able to:

- Explain what tool orchestration is (and what it is not).
- Understand tool choice as a routing problem with constraints.
- Design a tool router that reliably picks between 2–3 tools.
- Separate tool failure from model uncertainty (and handle each differently).
- Use confidence as a practical control dial, not a decorative number.
- Build a small orchestration loop that is observable and debuggable.

And just like the previous chapters, we'll build intuition first. The demonstration later is not the lesson itself. It's the proof that the lesson behaves the way we claim when the trail gets messy.

Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

Story Beat: The Gear Closet and the Wrong Turn

Chapter 10 taught your expedition how to move in repeatable steps. You stopped doing everything in one breath and started using a planner, an executor, checkpoints, retries, and sensible stopping rules. That was the moment the journey stopped being "chatting with a model" and started being "running a system".

Now we add a new kind of realism: your system has more than one tool, and the world does not politely cooperate.

Tool orchestration is what happens after "tool use" becomes normal. In Chapter 6, you handed the guide a compass, a calculator, and a radio. In this chapter, you teach the guide how to choose which one to use, how to feed it clean inputs, how to recognize when it failed, and how to act responsibly when confidence drops.

By the end of this chapter, you will be able to:

- Explain what tool orchestration is (and what it is not).
- Understand tool choice as a routing problem with constraints.
- Design a tool router that reliably picks between 2–3 tools.
- Separate tool failure from model uncertainty (and handle each differently).
- Use confidence as a practical control dial, not a decorative number.
- Build a small orchestration loop that is observable and debuggable.

And just like the previous chapters, we'll build intuition first. The demonstration later is not the lesson itself. It's the proof that the lesson behaves the way we claim when the trail gets messy.

Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

11.1 From Tool Use to Tool Orchestration

Tool use is the moment you connect a model to external capabilities. You stop asking it to “pretend” it can compute or search, and you give it something real.

Orchestration is what happens next, when you have multiple tools and the system must behave intelligently across them.

A clean way to separate the two:

- Tool use: A model calls a tool and uses the result.
- Tool orchestration: A system decides which tool to call, shapes the input, executes safely, interprets results, handles failure, updates confidence, logs what happened, and decides what to do next.

That extra list is not “enterprise fluff”. It is the minimum required for reliability once tools matter.

Because tools introduce two realities that pure text generation does not:

- They can fail in ways that are not subtle.
- They can have side effects.

A model generating a wrong answer is bad. A model triggering the wrong tool call can be worse. Orchestration is the discipline that keeps tool use from turning into a chaotic remote control.

Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

11.2 Tools Are Capabilities With Contracts

In beginner demos, a “tool” is often treated like a magic function: you call it, and it returns the truth.

In real systems, tools are capabilities with contracts. A contract answers questions like:

- What inputs does the tool accept?
- What output shape does it return?
- What counts as success?
- What are the known failure modes?
- How long does it usually take?
- What does it cost?
- Does it have side effects?
- Is it safe to retry?

This contract mindset is foundational because orchestration depends on predictable interfaces. The router can’t make a good choice if it doesn’t understand the tool’s affordances and limitations.

Think about a calculator tool versus a search tool:

- A calculator wants a strict expression. It either evaluates or errors.
- A search tool wants a query string. It returns candidates, not certainty.

If you treat both as “call tool, get answer”, you will build a system that looks functional until a user asks something slightly off-script.

So before you even build a router, you want a tiny “tool shelf” that describes each tool’s role.

Tool name: CALC
Best for: numeric expressions, arithmetic, simple math
Input: a sanitized expression string
Output: numeric result or error
Failure modes: invalid characters, divide-by-zero, unsupported operations
Retry safety: usually safe, low risk

Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

11.2 Tools Are Capabilities With Contracts

Tool name: SEARCH
Best for: retrieving snippets and references from a corpus
Input: a query string
Output: a list of matches or a best match
Failure modes: empty results, low relevance
Retry safety: safe, but repeated retries rarely help unless the query changes

Tool name: NOTES
Best for: known definitions and internal chapter concepts
Input: a key or topic label
Output: stored text
Failure modes: missing key
Retry safety: safe, but better to fallback than to retry

Notice the pattern: orchestration becomes easier when each tool has a clear “best for” scope. Tools that do everything are hard to orchestrate. Tools that do one job well are orchestration-friendly.

Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

11.3 Tool Choice Is a Routing Problem

Tool choice feels like intuition when you are a human. You look at a query and your brain instantly says “math” or “lookup” or “explain”.

In a system, tool choice is routing. Routing means: given an input, choose an action from a fixed menu.

A router is a classifier with constraints.

At the beginner scale, the router can be simple:

- If the user asks for arithmetic, route to CALC.
- If the user asks “define X” and X is in our internal notes, route to NOTES.
- If the user asks a general question that needs a lookup, route to SEARCH.
- If the request is purely conversational or can be answered without tools, route to NONE.

At the intermediate scale, you add realism:

- Some queries match multiple tools.
- Some tools can answer, but one is cheaper or more reliable.
- Some tools are blocked or failing right now.
- Some tools return partial results, and you need to decide whether that’s good enough.

A good router does not just pick a tool. It picks a tool for a reason, under constraints.

Here are the three constraints that dominate practical orchestration:

a. Reliability constraint: which tool is most likely to produce a usable result?
b. Cost constraint: which tool should we use first when multiple can work?
c. Latency constraint: which tool keeps the system responsive?

Even with only 2–3 tools, you can teach good habits by making the router’s decision explicit instead of implicit.

Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

11.4 The Router Output Should Be Structured, Not Vague

In the early chapters, we used structured prompts and checklists to reduce ambiguity. The same logic applies here, but more sharply.

If the router returns a paragraph, your system now has to interpret a paragraph. That makes orchestration fragile and hard to debug.

Instead, routers should return a small, structured decision record. A practical minimum looks like this:

- tool: one of [CALC, NOTES, SEARCH, NONE]
- tool_input: the exact input to pass to the tool
- confidence: a number between 0 and 1
- reason: a short justification for logging

This is not bureaucracy. It is a safety rail.

It also creates the clean separation you want in a good agentic system:

- The router decides.
- The tool executes.
- The orchestrator manages the flow and policies.

When you keep those roles distinct, your system becomes easier to test, easier to debug, and easier to extend.

If you later add a new tool, you add it to the menu and update the routing logic. You don’t rewrite the whole agent.

Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

11.5 Input Shaping: The Quiet Work That Prevents Loud Failures

A large percentage of “tool failures” are not real failures. They are input failures.

Meaning: the tool did exactly what it was designed to do, and the orchestrator handed it garbage.

So orchestration includes input shaping, which usually has three steps:

- Extract: pull out the piece of the user request the tool actually needs.
- Normalize: remove irrelevant wrappers and standardize formatting.
- Validate: ensure the input matches the tool’s contract.

For a calculator tool, input shaping might mean stripping words and keeping only the expression. For a search tool, it might mean keeping the query short and removing noise. For a notes tool, it might mean mapping “tool confidence” to a canonical key like “tool_confidence”.

This is the unglamorous part of orchestration. It is also where reliability is won.

One beginner-friendly rule that scales surprisingly far:

- Tools like clean, minimal inputs.
- Models like expressive, human inputs.
- Orchestration is the bridge between the two.

Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

11.6 Tool Failure: A Taxonomy That Keeps You Calm

When tools fail, beginners often respond with one of two extremes:

- Extreme A: retry forever
- Extreme B: stop instantly

The correct behavior depends on what kind of failure occurred.

A practical taxonomy:

- Hard failure: the tool errors or throws an exception
  Examples: timeout, permission denied, rate limited, invalid JSON

- Soft failure: the tool returns nothing usable
  Examples: empty search results, missing key, blank output

- Semantic failure: the tool returns output, but it does not solve the task
  Examples: search returns an irrelevant snippet, calculator returns a number, but the expression was wrong

- Partial failure: the tool returns some useful output, but not enough
  Examples: search returns one partial match but misses key context

This taxonomy matters because it changes what you should do next:

- Hard failure often deserves a retry if it might be transient.
- Soft failure usually deserves a fallback (a different tool or a rephrased query).
- Semantic failure often deserves re-routing or clarification.
- Partial failure often deserves “continue cautiously” with lower confidence.

Tool orchestration is not about avoiding failure. It’s about responding to failure like a system that expects reality.

Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

11.7 Retries, Fallbacks, and Reroutes Are Different Moves

When Chapter 10 introduced retries, the goal was resilience without spirals. The same applies here, but with sharper boundaries.

- Retry: call the same tool again
- Fallback: switch to a different tool or strategy
- Reroute: re-run the router decision based on new evidence

These moves are not interchangeable.

Retries are best when:

- The failure was likely transient (timeout, intermittent error).
- You can change something meaningful (sanitized input, smaller request).

Fallbacks are best when:

- The tool's failure suggests the tool is wrong for this case.
- The tool returned empty output, and repeating won't help.
- You have an alternate tool that can answer the same intent.

Reroutes are best when:

- The evidence suggests the router misclassified the request.
- The user query was ambiguous, and the tool outcome clarified the intent.
- The system learned a constraint (tool unavailable, permission blocked).

There is also a hidden rule that matters a lot once tools have side effects:

Not every tool call is safe to retry.

- A calculation is usually safe.
- A search is safe.
- A "send email" tool is not safe.
- A "place order" tool is definitely not safe.

So orchestration policies must consider idempotence, even if you don't use the term. The simple version is: If repeating the tool call can cause duplication or harm, do not retry without safeguards. In this chapter's scope, we keep tools intentionally safe and controlled so you can learn orchestration mechanics without stepping into side-effect land too soon.

Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

11.8 Confidence: A Control Dial, Not a Trophy

A lot of systems slap a confidence score onto outputs as decoration. That's not what we want.

In this book, confidence is a practical control variable used for three things:

a. To decide whether to proceed
b. To decide whether to fallback
c. To decide whether to stop gracefully

There are two confidence moments you care about:

a. Pre-call confidence: how confident the router is that the chosen tool is appropriate

b. Post-call confidence: how confident the system is after seeing the tool's result

- If the tool succeeds cleanly, post-call confidence should increas If the tool fails, post-call confidence should decrease.

- If the system needed a fallback, confidence should decrease somewhat because reliability just took a hit, even if you recovered.

A crucial beginner distinction: Tool confidence is not the same as answer truth. You can be highly confident that CALC is the right tool for "(48/6) + 13" without yet knowing the result.

That's tool selection confidence.

After the tool runs and returns a clean numeric output, your answer confidence increases because you now have evidence.

Confidence should track evidence, not vibes.

Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

11.9 Confidence-Driven Stopping: How Not to Hallucinate Under Pressure

One of the most common orchestration failures looks like this:

The tool fails. System still returns an answer anyway, because it “doesn’t want to be unhelpful”.

That is exactly how hallucinations sneak into tool-based systems. A good orchestrator dares to stop when reliability collapses.

Confidence-driven stopping is a simple policy:

- If confidence remains high, continue and answer.
- If confidence drops to medium, continue cautiously and log what happened.
- If confidence drops low, stop gracefully, explain the limitation, and suggest a safer next step (rephrase, provide missing info, try again later).

This is not about being pessimistic. It’s about being honest.

In expedition terms, if the bridge is out and the weather is turning, you don’t “confidently continue” by inventing a bridge. You stop, re-route, or set camp.

In tool systems, that translates to:

- Do not fabricate tool results.
- Do not pretend that a failed tool succeeded.
- Do not turn missing data into confident conclusions.

Stopping gracefully is a feature.

Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

11.10 Orchestration State: The Small Memory That Keeps You Sane

Tool orchestration is a multi-step process. Multi-step processes need a state.

State includes:

- The user query.
- The router decision (tool, tool_input, confidence, reason).
- Tool outputs (raw and parsed).
- Failures encountered.
- Retries attempted.
- Fallbacks used.
- Current confidence.
- Stop reason (if stopping).

Without state, the system will repeat itself and lose track of what happened. With state, you can enforce policies:

- Only one retry per tool.
- Only one fallback per run.
- Maximum total steps.
- Stop if confidence drops below threshold.

This is how you prevent infinite loops in tool orchestration, the same way Chapter 10 prevented infinite loops in planning.

Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

11.11 Observability: The Event Log Is Part of the System, Not an Optional Extra

A tool orchestrator that doesn't log is a black box. A black box cannot be debugged, improved, or trusted.

So you want a simple event log, even in beginner demos.

Each event should capture:

- Step number
- What made the decision (router or tool)
- Which tool was chosen
- Tool input
- Outcome (OK, FAIL, EMPTY, RETRY, FALLBACK)
- Confidence before and after
- A short detail message

This is not overkill. It's the system's travel journal.

When something goes wrong, you want to know whether the failure was:

- Bad routing
- Bad tool input
- Tool outage
- Bad fallback policy
- Bad stopping rule

Logs turn "it didn't work" into "here's why it didn't work". And once you can see why, you can fix it.

Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

11.12 Demonstration: Tool Orchestration (Choice, Failure, Confidence)

To understand this chapter's core idea it is strongly recommended to go through this Google Colab Notebook :

https://colab.research.google.com/drive/1ObQHiDf50L-wDzmBRTOhQlWX1ei33IAK?usp=sharing

To implement it yourself you can copy-paste the code from the shared notebook in another notebook with your dedicated API key and run it for yourself.

What you need (before opening the Colab link)

To run the notebook yourself (optional), you'll need a Gemini API key and the preferred model for this demo is gemini-2.5-flash (same expectation as Chapter 9's setup guidance).

Step A: Get your Gemini API key (Google AI Studio)

a. Open Google AI Studio and go to the API key page (sign in with your Google account).

b. Create a new API key and copy it somewhere safe.

c. Treat the key like a toothbrush: personal, not shareable, and not something you paste into screenshots.

Step B: Add the key in Colab and pick the model

In the notebook, you'll paste the key where the environment variable / secret is read, and ensure the model selection is set to gemini-2.5-flash

Step C: Run the notebook cells in order

The notebook is intentionally structured into blocks:

- Install dependencies (Gradio, Pydantic, LangChain Gemini wrapper)
- Imports + Gemini setup
- Tool definitions (CALC, NOTES, SEARCH) + failure injection
- Router prompt + strict JSON parser + self-heal
- Orchestrator loop (choice, failure, confidence updates, fallback)
- Gradio UI

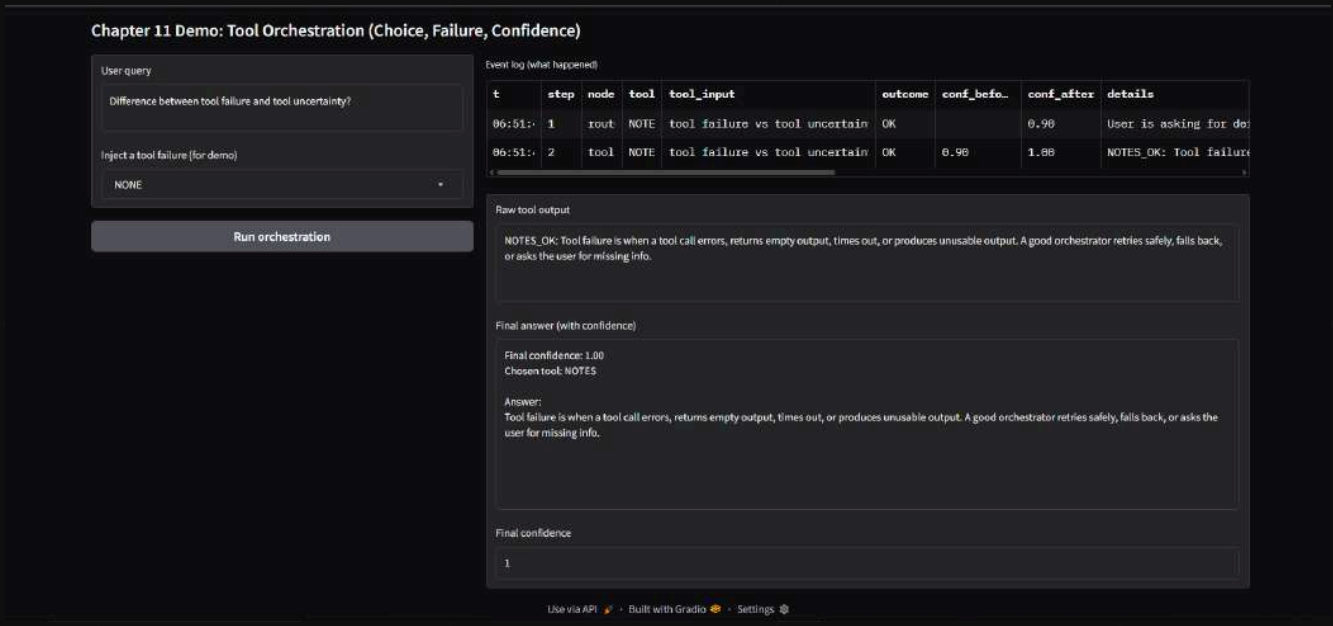
Once the Gradio app launches, you will see a small interface that is basically a "tool orchestration dashboard".

Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

11.12 Demonstration: Tool Orchestration (Choice, Failure, Confidence)

What you are seeing in the Gradio interface (and why it matters)



The UI has four main parts:

- User query (left): the natural language request.
- Inject a tool failure (left): a dropdown that deliberately breaks a chosen tool.
- Run orchestration (button): triggers the orchestrator.
- Results panel (right): event log, raw tool output, final answer with confidence, and final confidence as a number.

The key learning artifact is the event log.

It shows, step-by-step:

- What the router chose.
- What tool was called.
- Whether it succeeded.
- Whether a retry or fallback happened.
- How confidence changed after each step

That log is the clearest way to understand orchestration without hand-waving.

Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

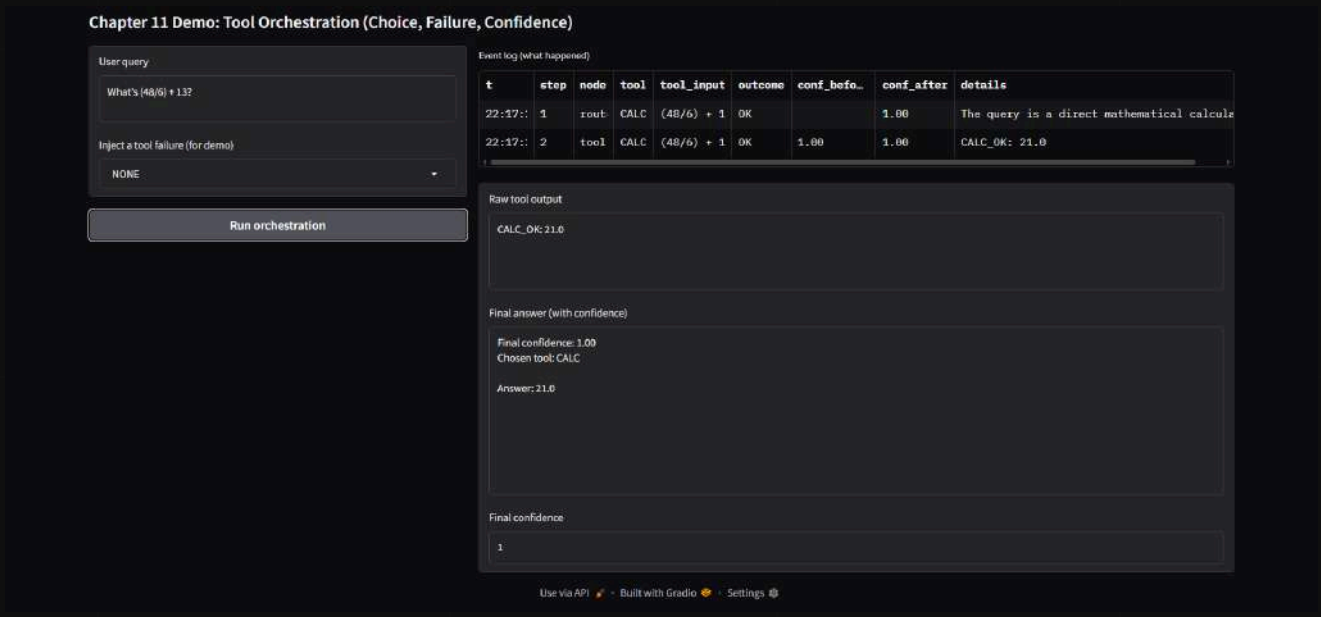
11.12 Demonstration: Tool Orchestration (Choice, Failure, Confidence)

The recommended interactions (designed for clean, teachable runs)

Use the following runs because they make the orchestration behaviour obvious.

Run 1: Clean tool choice (CALC succeeds)

User query: What's (48/6) + 13?
Inject failure: NONE
Click Run orchestration.



What you should notice:

- The router chooses CALC.
- The tool output shows CALC_OK with a numeric result.
- Confidence stays high (often reaches 1.00).
- Final answer is the numeric value, not a "model guess".

This is the happy path: correct tool, correct input, clean output.

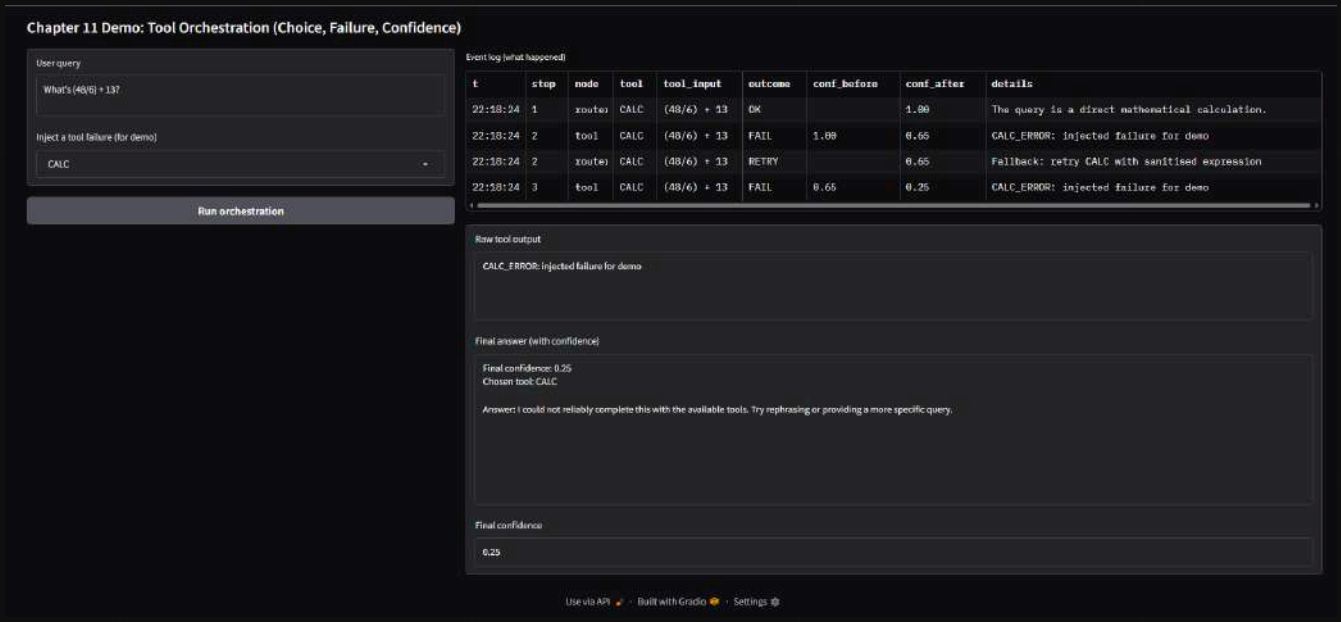
Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

11.12 Demonstration: Tool Orchestration (Choice, Failure, Confidence)

Run 2: Failure + retry fallback (CALC fails)

Same user query: What's (48/6) + 13?
Inject failure: CALC
Click Run orchestration.



What you should notice:

- The tool fails with an injected error.
- Confidence drops after the failure.
- The router applies a fallback policy: retry CALC with a sanitised expression.
- If the failure is still injected, the retry fails again.
- Final confidence becomes low.
- Final answer becomes a graceful stop, not a hallucinated number.

This run teaches the most important orchestration lesson in one go:

When tools fail, the system should not pretend. It should adapt or stop.

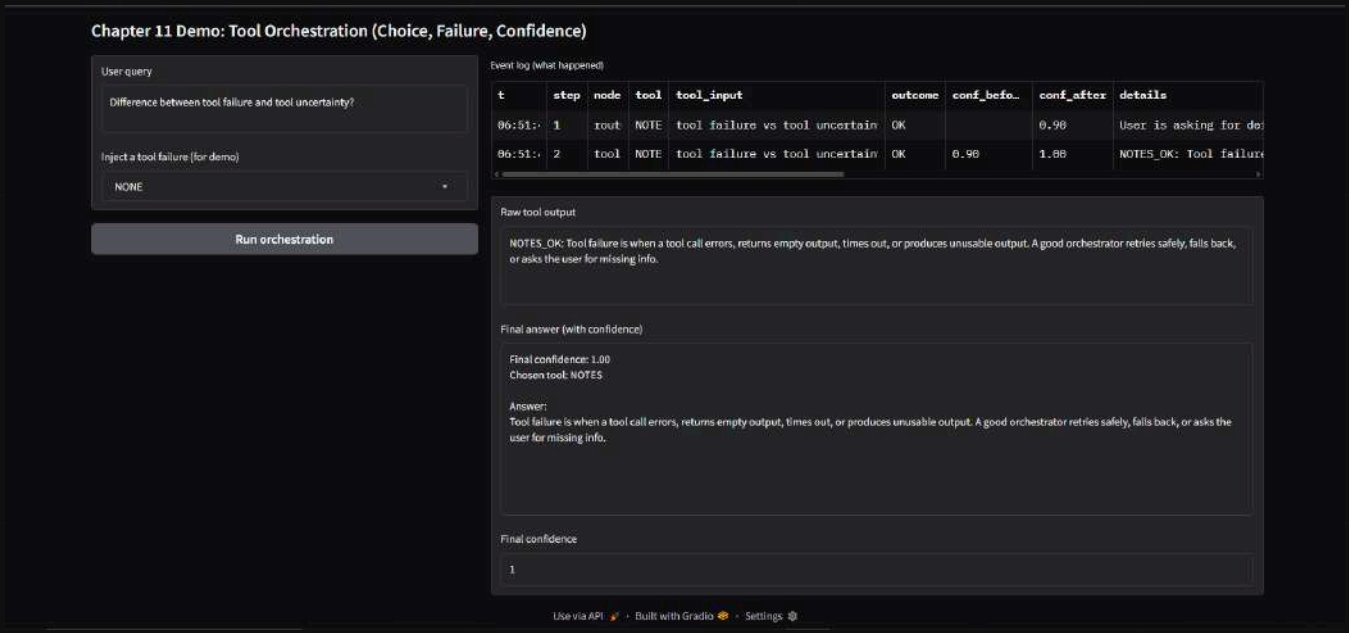
Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

11.12 Demonstration: Tool Orchestration (Choice, Failure, Confidence)

Run 3: Snippet lookup route (SEARCH)

User query: Difference between tool failure and tool uncertainty?
Inject failure: NONE
Click Run orchestration.



What you should notice:

- Router selects NOTES.
- NOTES returns the closest matching snippet from the offline corpus.
- Confidence reflects that this is a lookup, not a guaranteed truth source.

This run is important because it hints at the real world: you will often route to retrieval or search, and you will need to treat results as useful but not sacred.

Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

11.13 What Do We Deduce From the Demonstration?

If you only take a few ideas from this chapter, take these.

1) Tool choice is a routing decision, not a prompt trick

- The router behaves like a dispatcher.
- It classifies the request and chooses a capability.

That is agentic thinking: the model is coordinating work, not doing all work.

2) Failure is not exceptional. It is expected

- The injected failure toggle is not a toy feature.
- It represents reality: APIs fail, tools break, outputs come back unusable.

A system that cannot handle tool failure cannot be called “agentic” in any practical sense, because it cannot survive the world it operates in.

3) Retries must change something meaningful

- A retry with the same input is often just repetition.
- In the demo, the retry sanitizes the expression. That is a meaningful change. It is a small example of a big principle: Retries should be policy-driven, not hope-driven.

4) Confidence is a state variable that should move

- Confidence starts as the router’s belief.
- Then it updates based on evidence.
- When a tool fails, confidence drops.
- When a tool succeeds, confidence rises.
- When fallback is required, confidence takes a small hit because reliability decreased.

That is exactly how real systems should treat confidence: not as decoration, but as a live indicator that can influence stop conditions.

5) The event log is the difference between learning and guessing

- You can read the event log and understand what happened without mind-reading the model.
- That is a beginner superpower. It is also a production necessity.

Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

11.14 Troubleshooting (Because Tools Still Have Opinions)

If your notebook run does not match the expected behaviour, here are the most common fixes.

1) Gemini API key issues

Symptoms:

- Errors during router_decide
- Authentication failures
- Nothing happens when orchestration runs

Fixes:

- Ensure GOOGLE_API_KEY is set correctly in Colab
- Prefer Colab Secrets over hardcoding
- Confirm the model name is correct: gemini-2.5-flash

2) Router returns non-JSON or messy output

Symptoms:

- Pydantic validation errors
- JSON extraction failures
- Router falls back to heuristic routing more often than expected

Fixes:

- Keep the router system prompt strict and short
- Do not add extra “helpful” wording that invites explanation
- Remember: routers should be boring

Even if the model occasionally misbehaves, the notebook’s heuristic_router is there to keep the system running. That is intentional.

Copyright © Blockchain Council | All rights reserved

11.14 Troubleshooting (Because Tools Still Have Opinions)

3) CALC rejects the expression

Symptoms:

- CALC_ERROR about unsupported characters
- No numeric output even for simple maths

Fixes:

- Use only digits, operators, parentheses, decimal points.
- Avoid fancy punctuation.
- If the input includes extra text, the sanitised retry should usually rescue it.
- The calculator is designed to be safe, not permissive. That is a good trade.

4) SEARCH returns empty results

Symptoms:

- SEARCH_EMPTY: no relevant snippet found.

Fixes:

- Try phrasing the query closer to the snippet text.
- Remember: this SEARCH is an offline stub with a tiny corpus.
- In real systems, SEARCH would connect to a real retrieval layer.

This is a teaching feature, not a full search engine.

5) Gradio does not display or the link does not open

Fixes:

- Make sure demo.launch() runs.
- If Colab blocks popups, open the provided Gradio link manually.
- Try rerunning the UI cell after restarting the runtime.

Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

11.15 Analogy Tie-Back: The Quartermaster and the Gear Check

In expedition terms, tool orchestration is the quartermaster's job.

The guide does not carry every tool in their hands at once. The quartermaster decides:

- Which gear comes out now
- What to do if the gear breaks
- Whether the team is still on a reliable path forward

Chapter 10 gave you route cards, this chapter gave you gear discipline.

Now you are ready for the next step in the Agentic AI section: not just one guide with tools, but a small crew where different agents specialise, hand off work, and depend on each other's outputs.

In the next chapter (Multi-agent basics), the challenge becomes bigger:

- It is not only "which tool do we call?"
- It becomes "which specialist do we ask, and how do they coordinate without stepping on each other?"

For now, close this chapter with a simple win:

You can build a system that chooses tools deliberately, survives failure, updates confidence honestly, and leaves a trace of what happened.

That is not a storybook ending. That is a real engineering milestone.

Copyright © Blockchain Council | All rights reserved

Chapter 11: Tool Orchestration (Choice, Failure, Confidence)

11.15 Analogy Tie-Back: The Quartermaster and the Gear Check

In expedition terms, tool orchestration is the quartermaster's job.

The guide does not carry every tool in their hands at once. The quartermaster decides:

- Which gear comes out now
- What to do if the gear breaks
- Whether the team is still on a reliable path forward

Chapter 10 gave you route cards, this chapter gave you gear discipline.

Now you are ready for the next step in the Agentic AI section: not just one guide with tools, but a small crew where different agents specialise, hand off work, and depend on each other's outputs.

In the next chapter (Multi-agent basics), the challenge becomes bigger:

- It is not only "which tool do we call?"
- It becomes "which specialist do we ask, and how do they coordinate without stepping on each other?"

For now, close this chapter with a simple win:

You can build a system that chooses tools deliberately, survives failure, updates confidence honestly, and leaves a trace of what happened.

That is not a storybook ending. That is a real engineering milestone.

Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

How Two Focused Specialists Beat One Overworked "Do-Everything" Agent

Chapter 11 gave your expedition "gear discipline": choosing the right tool, feeding it clean inputs, handling failures without drama, and tracking confidence honestly. That's already a big step up from "the model said something, so we shipped it".

Now we make the next realistic upgrade: we stop pretending one single agent should do everything.

In the real world, important work is rarely done by one person doing research, planning, writing, checking, and revising in one sitting. It is done by a small crew with clear roles, clear handoffs, and clear checkpoints. Agentic AI becomes much more reliable when you copy that pattern instead of chasing the fantasy of a single "genius agent" that never gets tired, never drifts, and never forgets constraints.

By the end of this chapter, you will be able to:

- Explain what a "multi-agent system" actually is (in practical, engineering terms).
- Understand why specialization reduces errors and improves speed of iteration.
- Design a simple two-agent workflow: Researcher → Builder, with an explicit handoff.
- Treat a handoff like a contract: structured, validated, and easy to debug.
- Handle mid-run reality changes without restarting the world.
- Use a deterministic checker as a gate, so quality is not based on vibes.

And just like the previous chapters, we'll build intuition first. The demonstration later is not the lesson itself. It's the proof that the lesson holds when the trail shifts under your boots.

Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

Story Beat: The Day the Guide Tried to Be the Cook, Scout, Engineer, and Medic

Early in the journey, one guide works fine. You ask a question, you get an answer. You ask for a plan, you get a plan. Simple.

Then the expedition grows up.

Now the “one guide” is expected to: research the terrain, choose the route, pack the gear list, write the instructions, check them for errors, revise them for clarity, and still keep an eye on the weather. That is not intelligence. That is workload.

In real expeditions, you split roles because the mountain does not care that you are busy. You split roles because specialization reduces the chances of a single mistake turning into a chain reaction.

Multi-agent systems apply the same idea: not “many models talking”, but “clear roles, clear handoffs, and a coordinator that keeps the process honest”.

Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

12.1 What “Multi-Agent” Means in This Book (No Mysticism, Just Structure)

A multi-agent system is not automatically “multiple AIs chatting in a group”. That can be entertaining, but it can also be noisy and unaccountable.

In this book, a multi-agent system means something narrower and more useful:

A set of specialized agent roles, each with a defined job, connected by explicit handoffs, run under an orchestrated workflow with checks.

That definition hides three important ideas.

a. First, roles are not vibes. A role is a constraint. The Researcher is not allowed to “just write the final answer”. The Builder is not allowed to “do research from scratch”. Each agent’s job is bounded.

b. Second, handoffs are not casual conversation. A handoff is a structured package of information that the next agent can reliably use. If you cannot describe what is handed off, you do not have a real system. You have a chat.

c. Third, orchestration is the adult supervision. Something decides the order of steps, stores state, logs events, and handles exceptions. Without orchestration, a multi-agent setup is just a room full of enthusiastic interns and no manager.

In Chapter 10, you learned “workflow thinking”: a repeatable route. In Chapter 11, you learned “tool discipline”. In this chapter, you learn “crew discipline”.

Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

12.2 Why Specialization Works: Less Cognitive Load, Less Drift

Large language models are flexible, but that flexibility is a double-edged sword. When you ask a single agent to do everything, you increase the chance of:

- Context drift: it starts with the goal, then wanders into a different interpretation.
- Constraint loss: it forgets one of the requirements halfway through.
- Shallow reasoning: it tries to do research and writing at once, so both become thin.
- Untraceable mistakes: you cannot tell whether the error came from research, writing, or checking.

Specialization helps because it reduces cognitive load per step.

The Researcher can spend all their attention producing a clean structure: goals, assumptions, risks, and a blueprint. The Builder can spend all its attention producing a high-quality deliverable from that blueprint. And a checker can enforce minimum quality without needing to “feel confident”.

This is the same reason real teams use specs, tickets, design docs, and acceptance criteria. It is not bureaucracy. It is survivability.

Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

12.3 The Two-Agent Pattern We'll Use: Researcher → Builder

There are many multi-agent designs, but beginners need something that is simple, visible, and testable. So we start with a two-agent pattern that appears constantly in real work:

- Researcher: turns a messy request into a structured plan.
- Builder: turns the structured plan into a deliverable.

If you work in software, this mirrors how a product requirement becomes a technical spec, and then becomes an implementation. If you work in content, it mirrors how research notes become an outline, and then become the final piece.

The crucial point is the handoff. The handoff is the bridge that prevents the Builder from guessing what the Researcher "probably meant".

So the two-agent pattern is not just "two calls to a model". It is two roles connected by a contract.

Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

12.3 The Two-Agent Pattern We'll Use: Researcher → Builder

There are many multi-agent designs, but beginners need something that is simple, visible, and testable. So we start with a two-agent pattern that appears constantly in real work:

- Researcher: turns a messy request into a structured plan.
- Builder: turns the structured plan into a deliverable.

If you work in software, this mirrors how a product requirement becomes a technical spec, and then becomes an implementation. If you work in content, it mirrors how research notes become an outline, and then become the final piece.

The crucial point is the handoff. The handoff is the bridge that prevents the Builder from guessing what the Researcher "probably meant".

So the two-agent pattern is not just "two calls to a model". It is two roles connected by a contract.

Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

12.4 The Handoff Pack: Your System's Most Important Document

If you remember one phrase from this chapter, make it this:

Handoffs are where systems either become reliable or become improved.

A handoff pack should answer, clearly:

- What is the goal?
- What is the scope, including constraints?
- What assumptions are we making?
- What risks should we watch?
- What is the blueprint of the output?
- What open questions remain?

Notice what is missing: the handoff pack is not the final deliverable. It is the recipe, the checklist, and the shopping list. It is what the Builder can trust.

A strong handoff pack has three properties:

a. It is structured. Not "a paragraph summary", but a predictable shape.
b. It is validated. You can detect if it is broken before it causes damage.
c. It is complete enough to build with. The Builder should not need telepathy.

When you do this well, you get a nice side effect: debugging becomes simple. If the final deliverable is wrong, you can ask, "Was the handoff wrong, or was the build wrong?" That is already a huge upgrade over "The model messed up".

Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

12.5 Contracts, Not Conversations: Why Structure Matters

A conversational handoff sounds like this:

“Here’s what I found. It should probably include these points. Maybe add a weekly plan. Make it beginner friendly.”

A contract handoff sounds like this:

- Goal: X
- Scope: topic, learner background, weeks, hours per week, constraints
- Blueprint: sections, each with must-include items
- Assumptions: list
- Risks: list
- Open questions: list

The second one is boring in the best way. It is boring like a seatbelt.

Structure prevents two classic multi-agent failures:

a. Interpretation mismatch: Builder guesses a different meaning than the researcher intended.
b. Hidden requirements: a constraint exists in someone’s head but not in the handoff.

In this chapter’s demo, the handoff pack is a strict JSON validated by a schema. That is not an arbitrary choice. It is the simplest way to make handoffs machine-checkable.

Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

12.6 The Quiet Power Move: Schema Validation

Beginners often think reliability comes from “a better prompt”.

Prompts matter, but systems become reliable when they can detect breakage.

Schema validation is a clean example. If the Researcher is required to output a JSON object with specific fields, you can validate it. If it fails validation, you can retry or tighten instructions. You do not blindly hand garbage to the Builder and hope for miracles.

This is the same idea as type checking, API contracts, and database schema constraints. You are not trying to insult the model. You are building guardrails because the real world is messy.

In the demo, the ResearcherPack schema includes fields like: goal, scope, key points, assumptions, risks, open questions, blueprint, and recommended tools. If any of those are missing or malformed, validation fails early, when it is cheap to fix.

That is the practical heart of “multi-agent reliability”: early detection, not late regret.

Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

12.7 The Orchestrator: The Crew Needs a Coordinator

A two-agent system still needs something that runs it.

That “something” is the orchestrator. It does not need to be fancy. It just needs to be honest.

At a minimum, an orchestrator should:

- Store the current inputs and state.
- Run the Researcher to produce the handoff pack.
- Run the Builder using the handoff pack.
- Record an event log of what happened.
- Optionally run a checker to gate the result.

This turns your system from “a sequence of prompts” into “a process”.

And once you have a process, you can do engineering. You can measure, debug, and improve.

Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

12.8 The Checker: A Deterministic Gate, Not a Third Opinion

Here is a trap that catches a lot of people:

When the output is weak, they add another model call to “review it”.

Sometimes that helps. Often, it just creates a new pile of text.

A checker is different from a reviewer. A checker is a gate with explicit rules.
Instead of asking, “Does this look good?”, you ask:

- Does it contain the required sections?
- Does it mention weekly checkpoints?
- Does it include headings so it is shareable markdown?
- Does it reference the time budget?
- Does it acknowledge constraints?

These checks can be simple string checks. They can be deterministic. They can run fast.
And they give you a pass/fail report that is easy to act on.

In the demo, the checker is intentionally “not vibes”. It produces a small validation report that says what is missing.

This is a big conceptual step for beginners: quality control can be a function, not a feeling.

Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

12.10 Common Multi-Agent Failure Modes (So You Recognize Them Early)

Multi-agent systems are powerful, but they introduce new failure modes. It is better to meet them here, in the safe practice zone, than later in production.

- Handoff rot: the pack is incomplete or unclear, so the Builder guesses.
- Role leakage: the Researcher starts writing the final output, or the Builder starts inventing research.
- Compounding errors: one wrong assumption becomes the foundation for the rest.
- Coordination loops: agents keep bouncing the task back and forth without progress.
- Silent failure: an agent fails, but the orchestrator does not record it clearly.

The antidote to most of these is not “smarter prompts”. It is contracts, state, logs, and gates.

Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

12.11 When Should You Use Multi-Agent at All?

You do not need multi-agent systems for everything.

Use a single workflow when:

- The task is small and stable.
- The constraints are minimal.
- The cost of errors is low.

Use a multi-agent pattern when:

- The task has multiple distinct skills (research, writing, checking).
- The task has meaningful constraints that must be respected.
- You need traceability, so you can debug what went wrong.
- Reality changes are likely.

In expedition terms, you do not hire a full crew to walk to the nearest viewpoint. You hire a crew when you are crossing terrain that can punish mistakes.

Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

12.11 When Should You Use Multi-Agent at All?

You do not need multi-agent systems for everything.

Use a single workflow when:

- The task is small and stable.
- The constraints are minimal.
- The cost of errors is low.

Use a multi-agent pattern when:

- The task has multiple distinct skills (research, writing, checking).
- The task has meaningful constraints that must be respected.
- You need traceability, so you can debug what went wrong.
- Reality changes are likely.

In expedition terms, you do not hire a full crew to walk to the nearest viewpoint. You hire a crew when you are crossing terrain that can punish mistakes.

Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

12.12 Demonstration: Multi-Agent (Researcher → Builder) With Explicit Handoff

To understand this chapter's core idea, it is strongly recommended to go through this Google Colab Notebook:

https://colab.research.google.com/drive/1HuE_2nZEduPNnwKjzI7NLPxQuhyvRorw?usp=sharing

To run it yourself (optional), you'll need a Gemini API key and the preferred model for this demo is `gemini-2.5-flash`.

What you need (before opening the Colab link)

To run the notebook yourself (optional), you'll need a Gemini API key and the preferred model for this demo is gemini-2.5-flash (same expectation as Chapter 9's setup guidance).

Step A: Get your Gemini API key (Google AI Studio)

a. Open Google AI Studio and go to the API key page (sign in with your Google account).

b. Create a new API key and copy it somewhere safe.

Step B: Add the key in Colab and pick the model

In the notebook, you'll paste the key where the environment variable / secret is read or where it says "Your API Key", and ensure the model selection is set to gemini-2.5-flash

Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

12.12 Demonstration: Multi-Agent (Researcher → Builder) With Explicit Handoff

Step C: Run the notebook blocks in order

The notebook is organized like a small expedition route with clear camps:

- Block 1 installs dependencies (Gradio, Pydantic, LangChain for Gemini).
- Block 2 sets up the Gemini client (gemini-2.5-flash).
- Block 3 defines the strict handoff schema (ResearcherPack).
- Block 4 adds robust JSON extraction plus the Researcher and Builder runners.
- Block 5 defines the checker gate (deterministic checks).
- Block 6 builds the orchestrator and event log, including a reality-change injection.
- Block 7 launches a clean Gradio UI so you can run the system like a small control panel.

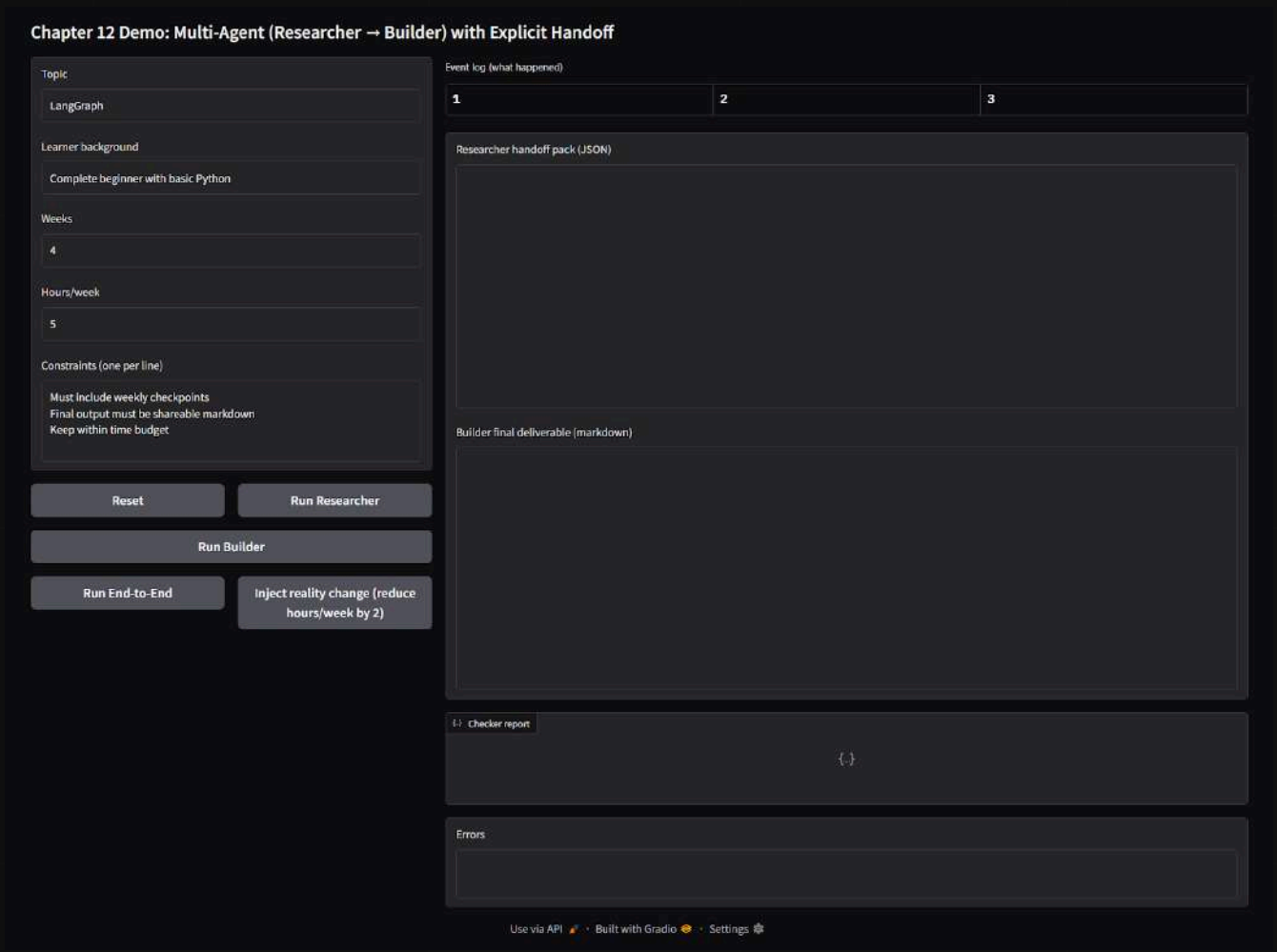
The important concept: you are not “using a multi-agent framework” here. You are building the pattern directly, so you can see every moving part.

Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

12.12 Demonstration: Multi-Agent (Researcher → Builder) With Explicit Handoff

What you are seeing in the interface (and why it matters)



When the Gradio UI loads, you'll see inputs like:

- Topic
- Learner background
- Weeks
- Hours per week
- Constraints (one per line)

These inputs are not about the multi-agent system itself. They are the user request the system is going to process. In the demo, the default topic is "LangGraph", but that is just an example topic for the learning plan output.

Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

12.12 Demonstration: Multi-Agent (Researcher → Builder) With Explicit Handoff

On the right side, you'll see three key areas:

- An event log: what happened, step by step.
- A Researcher handoff pack (JSON): the structured contract produced by the Researcher.
- A Builder final deliverable (markdown): the final output built from the handoff. And a checker report: pass/fail plus details.

This layout is deliberate. It trains you to think like a system builder: inputs, state, steps, outputs, and verification.

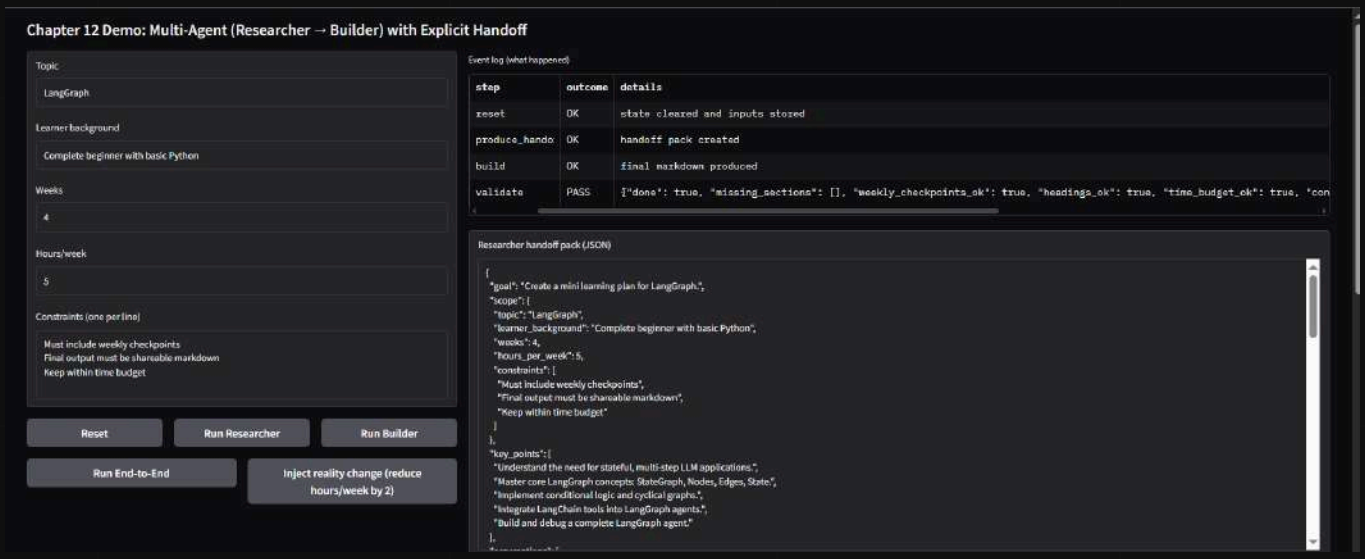
Copyright © Blockchain Council | All rights reserved

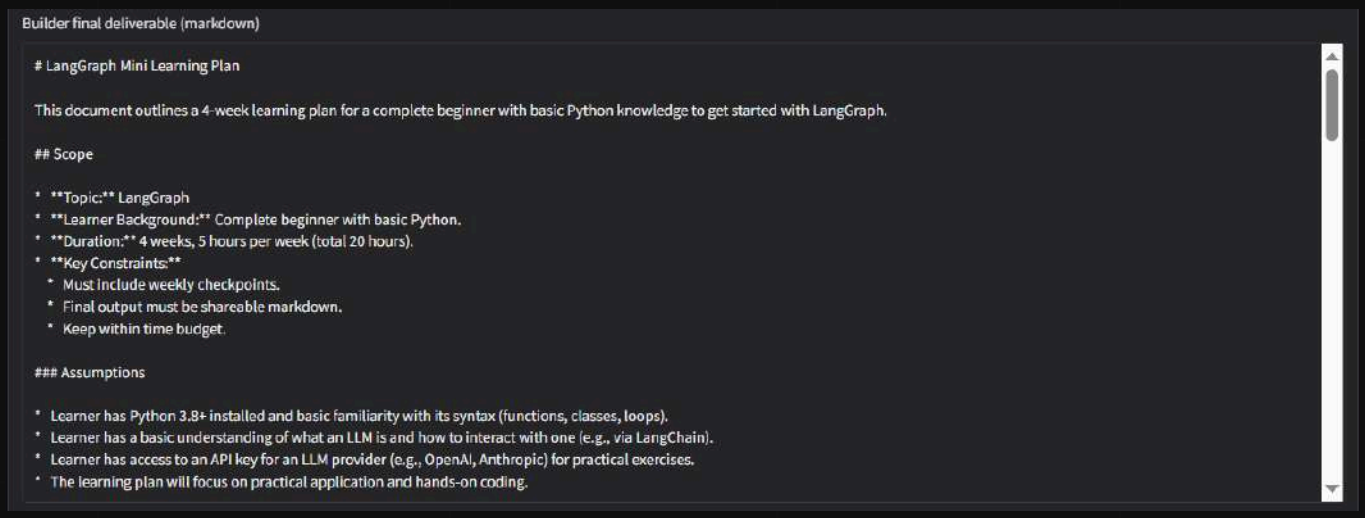
Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

The recommended interactions (designed for clean, teachable runs)

Interaction 1: Clean end-to-end handoff

a. Click Reset (clears state and stores inputs).
b. Click Run End-to-End.





Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

The recommended interactions (designed for clean, teachable runs)

What you should observe:

- In the event log, you should see a clean sequence such as:
- system reset OK
- researcher produce_handoff OK
- builder build OK
- checker validate PASS

You should also see:

a. A Researcher handoff pack JSON that includes goal, scope, blueprint, assumptions, risks, and constraints.

b. A Builder output in markdown that is shareable and structured, usually with headings like Scope, Assumptions, Modules, Weekly Plan, and Checkpoints.

Why this matters:

This is the simplest proof of the multi-agent contract. The Builder is not inventing structure. It is using the structure handed to it. The checker is not “judging quality”. It is verifying basic requirements.

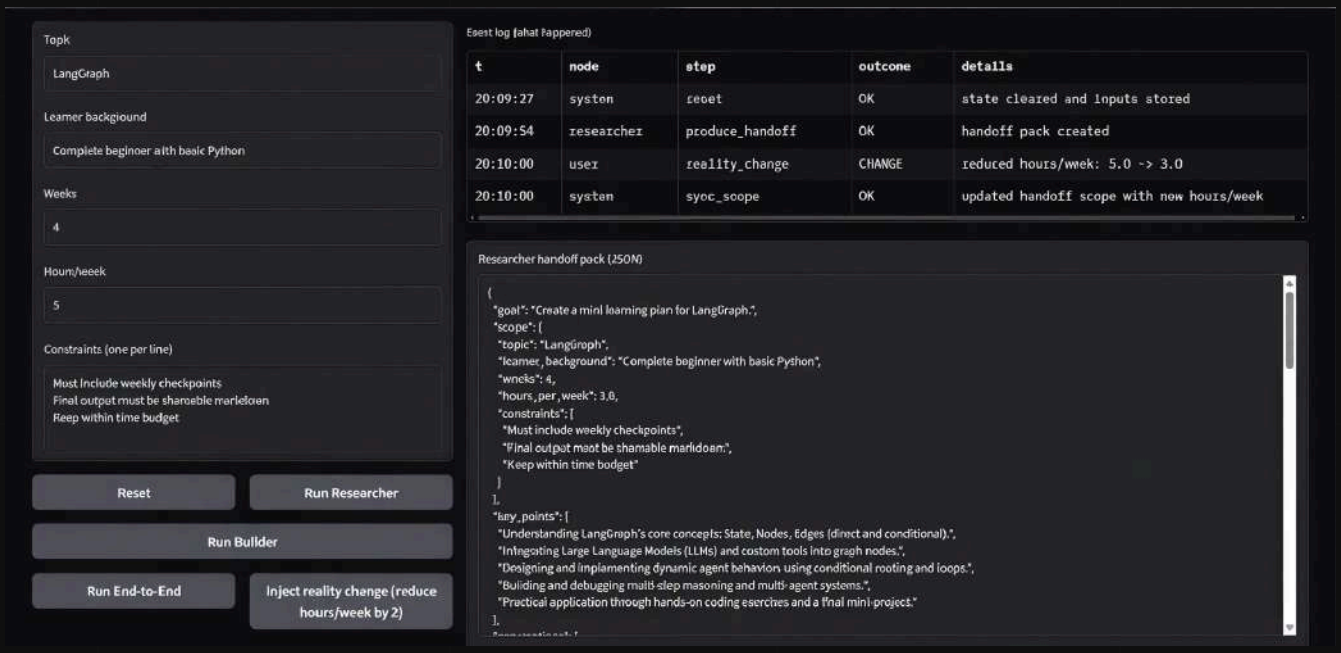
Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

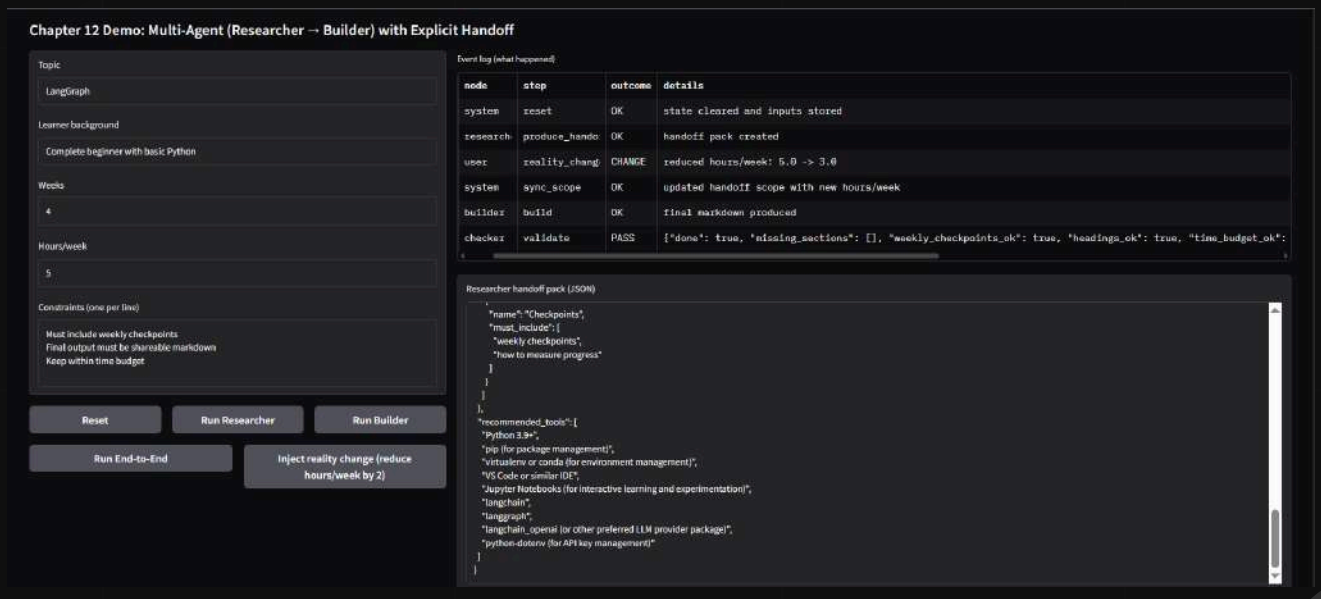
The recommended interactions (designed for clean, teachable runs)

Interaction 2: Reality change mid-way (hours reduced)

a. Click Reset.
b. Click Run Researcher.
c. Click Inject reality change (reduce hours/week by 2).
d. Click Run Builder.



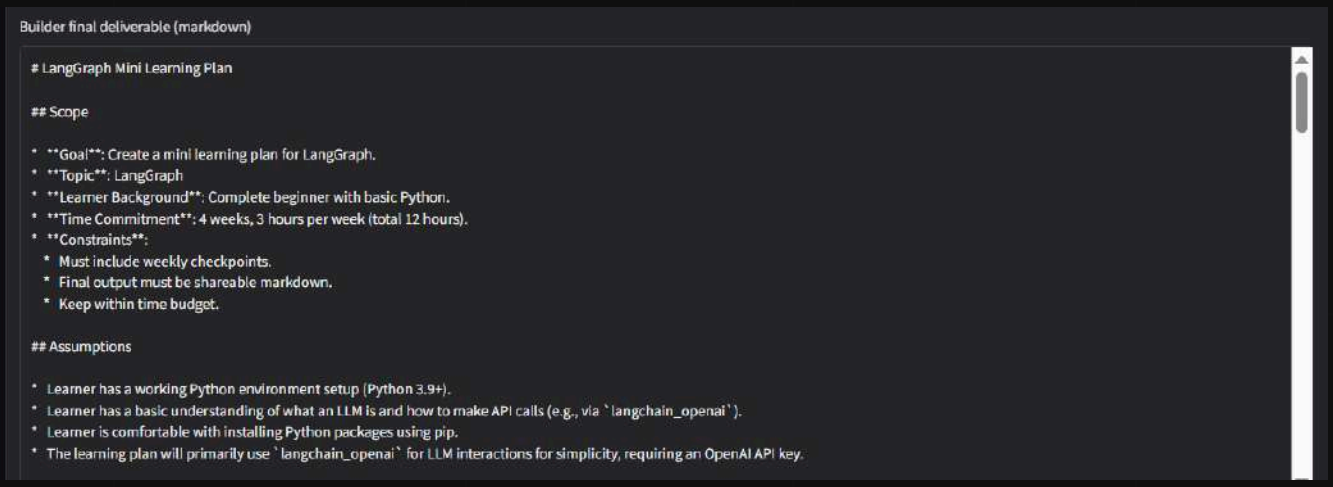
After Running the builder after introducing the reality change:



Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

The recommended interactions (designed for clean, teachable runs)



What you should observe:

- In the event log, you should see a user-driven change recorded, such as:
- user reality_change CHANGE (hours/week reduced)
- system sync_scope OK (handoff scope updated)
- builder build OK
- checker validate PASS (or FAIL if something slipped)

You should also notice in the Builder output that the time budget is updated. If the system started at 5 hours/week for 4 weeks (20 hours total), and you reduce hours/week by 2, the output should reflect the new reality (3 hours/week, 12 hours total).

Why this matters:
This is the “survives reality” test, but applied to coordination. The system did not restart from scratch. It updated state, synced the handoff contract, and continued.

That is what agentic systems are for.

Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

12.13 What Do We Deduce From the Demonstration?

1. First, role separation is not decorative. The Researcher produces a contract. The Builder produces a deliverable. This separation makes it easier to trust, inspect, and improve each part.

2. Second, explicit handoffs reduce ambiguity. The handoff pack is not a casual note. It is a structured object that the Builder can consume without guessing. That is why it is validated against a schema.

3. Third, orchestration plus logging turns “LLM behavior” into “system behavior”. The event log gives you a timeline. If something fails, you can locate where it failed. That makes debugging possible.

4. Fourth, deterministic checking is a practical quality gate. Instead of hoping the output is good, you check for required structure. In real systems, you can expand these checks: word count bounds, formatting requirements, presence of required sections, even unit tests for code output.

Finally, reality changes are manageable when state is explicit. When inputs shift mid-run, the orchestrator updates the stored scope, syncs the handoff pack, and the Builder adapts. This is the difference between “chat that reacts” and “workflow that adapts”.

Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

12.14 Troubleshooting (Because Agents Still Have Opinions)

1. If the Researcher fails with JSON errors:
   The system retries with tighter instructions, but you can also reduce complexity. Shorten constraints, simplify the topic, or reduce the required blueprint detail.

2. If the Builder output ignores constraints:
   Check whether the constraints are clearly present in the handoff scope. If the handoff pack is missing a constraint, the Builder cannot respect it reliably.

3. If the checker fails:
   Look at the checker report. It will usually tell you what is missing (for example, “missing_sections” or time budget not referenced). This is exactly why the checker exists: it gives you an actionable diagnosis.

4. If your API key errors or you get empty responses:
   Verify that GOOGLE_API_KEY is set in Colab Secrets and that you are using the intended model name. Also confirm you have network access in the runtime.

5. If results vary too much run to run:
   Lower temperature for the model calls. In this notebook, the temperature is already set low to encourage stability. Multi-agent reliability gets harder when you dial up randomness.

Copyright © Blockchain Council | All rights reserved

Chapter 12: Multi-Agent Basics (Specialist Crew, Not One Stressed-Out Hero)

12.15 Analogy Tie-Back: The Scout, the Builder, and the Clipboard

In expedition terms, the Researcher is your scout. They go ahead, study the terrain, and come back with a route brief that's structured enough to act on. The Builder is your camp engineer. They take that brief and turn it into something real and usable. And the checker is the person with the clipboard, which sounds dull until it saves you from leaving behind the one thing you'll desperately need at 2 a.m.

What you've built in this chapter is not "a group chat of AIs". It's a crew with division of labour, a shared contract, and a clear way to verify that the work actually meets the constraints. That's why the explicit handoff matters. It's your crew's written agreement about what "done" means, before anyone starts building.

Now here's the honest twist: once you have a crew, the next thing you discover is that crews can still get lost.

Not because they're incompetent, but because the world is messy. Requirements shift. Tools fail. One step quietly outputs something malformed. A constraint falls off the back of the truck. Suddenly the final result is wrong, and you're staring at the system thinking, "Where did it go off the rails?"

That is exactly why the next chapter exists.

In Chapter 13, we're going to treat debugging as a first-class expedition skill. You'll learn how to trace what happened using logs, how to label failures with an error taxonomy so you're not diagnosing everything as "the model is weird", and how to apply fix patterns that actually work in practice: tightening schemas, improving routing, strengthening checkers, adding guardrails, and reducing drift.

A good expedition doesn't end by pretending nothing went wrong. It ends with a debrief: what happened, what failed, what we learned, and how we update the playbook so the next run is smoother, safer, and faster. Agent debugging is that debrief, turned into a repeatable engineering method.

So yes, you now have a crew. Next, we teach you how to find the moment the crew took the wrong turn, and how to make sure they don't take it again.

Copyright © Blockchain Council | All rights reserved

Chapter 13: Agent Debugging (When the Crew Gets Lost)

Tracing Logs, Naming Failures, and Fixing Systems Without Guesswork

By now, your expedition has grown up.

In the early chapters, you learned how to talk to the guide. Then you learned how to give better directions, how to carry the right amount of context, and how to ground the guide in field notes instead of imagination. You added tools. You learned to evaluate outputs with simple tests. Then you crossed the bridge into agentic AI: loops, plans, tool orchestration, and a specialist crew with explicit handoffs.

That is a real progression.

It also creates a new reality: when something goes wrong, it can go wrong in more than one place.

A single prompt can be fixed by rewriting a sentence. A workflow can be fixed by moving a step. A tool call can be fixed by sanitizing input. A multi-agent system can be wrong because the Researcher handed off the wrong scope, because the Builder misread a constraint, because the router chose the wrong tool, because the memory carried stale state, or because a checker silently passed something it should have rejected.

Agent debugging is the skill of finding the real cause, not the most visible symptom.

And because this is the final chapter, it is also the skill that turns everything you have learned into something you can actually use without fear. If you can debug an agentic system, you can improve it. If you can't, you will eventually stop trusting it, no matter how clever it looked on good days.

This chapter is pure theory on purpose. Not because debugging is boring, but because debugging is real. Earlier chapters used demonstrations to make concepts tangible. You could press a button, see a run, inspect an event log, and connect the theory to something you can observe.

Debugging is different.

In the real world, you do not debug an agent with one dramatic button press that magically repairs the system. That kind of "press here to fix" demo is an overglorified lollipop: sweet for a moment, misleading for much longer.

Real debugging happens in the messy middle. You read logs, inspect prompts, review schemas. trace tool inputs and outputs, reproduce failures, tighten validators, adjust stop conditions, add regression tests, rerun, compare traces and repeat.

So instead of giving you a theatrical demo that risks teaching the wrong mental model, we will do something more valuable: build your intuition and your method. The best outcome here is not that you remember a specific interface. It is that you remember a reliable way of thinking when your system fails on a Tuesday afternoon with a stakeholder waiting.

Copyright © Blockchain Council | All rights reserved

Chapter 13: Agent Debugging (When the Crew Gets Lost)

Story Beat: The Fog on Day Twelve

Imagine the expedition is moving well. The route cards are working. The crew is cooperating. The quartermaster is choosing tools responsibly. The checklist passes. The output looks correct.

Then a fog rolls in.

Not dramatic, not cinematic. Just enough to make landmarks unreliable.

The crew still walks. The system still produces output. Nothing “crashes”. And yet, somewhere in the fog, the expedition drifts off the trail.

This is the most dangerous kind of failure in agentic AI: quiet wrongness.

Debugging is how you regain the trail without pretending the fog does not exist.

Copyright © Blockchain Council | All rights reserved

Chapter 13: Agent Debugging (When the Crew Gets Lost)

13.1 What Agent Debugging Actually Is

Traditional debugging usually assumes a program with deterministic behavior. Given the same input, the program does the same thing. Your job is to find the wrong line of code, fix it, and confirm the fix.

Agentic systems are not purely deterministic. They involve probabilistic model behavior, external tools, asynchronous failure modes, and multi-step decision-making. Even when you set temperature low, the system can still vary based on tool results, retrieval ordering, timing, and hidden state.

So agent debugging is not just “find the bug”.

Agent debugging is:

Tracing a multi-step run to identify where the system’s behavior diverged from its intended behavior, then applying a fix that prevents recurrence, and finally adding evidence that the fix worked.

That definition hides three important shifts:

First, you debug behavior, not just code. The “bug” might be a prompt, a schema, a retry policy, a tool router rule, or a missing invariant.

Second, you debug across layers. The same symptom can arise from multiple layers, and the first suspicious thing is not always the cause.

Third, you debug with artifacts. You cannot rely on memory or vibes. You need logs, traces, intermediate outputs, and contracts you can inspect.

If Chapter 9 taught you what an agent loop is, and Chapters 11 and 12 taught you how to make it structured, this chapter will teach you how to diagnose it when it goes sideways.

Copyright © Blockchain Council | All rights reserved

Chapter 13: Agent Debugging (When the Crew Gets Lost)

13.2 The Four-Layer Mental Model (So You Don’t Chase Ghosts)

When beginners debug agentic systems, they often jump straight to “the model is wrong”. Sometimes that is true. More often, it is incomplete.

A more reliable way to debug is to separate the system into layers:

Layer 1: The model layer
Prompts, system instructions, role definitions, output formatting rules.

Layer 2: The orchestration layer
Routers, planners, stop conditions, retry logic, fallback policy, confidence updates, step limits.

Layer 3: The tool layer
APIs, calculators, search, retrieval, databases, rate limits, timeouts, schema mismatches.

Layer 4: The state layer
Memory, session state, handoff packs, cached results, previous run residue, user constraints, environment configuration.

Most failures you will see are interactions between layers.

For example: a tool returns empty output (tool layer), the router misinterprets it as success (orchestration layer), confidence stays too high (orchestration layer), the Builder proceeds with missing facts (model layer), and the final answer sounds confident but is wrong.

If you only look at the final answer, you will blame the model. If you trace the run, you will find the real cause.

Debugging starts with choosing the right layer to inspect first.

Copyright © Blockchain Council | All rights reserved

Chapter 13: Agent Debugging (When the Crew Gets Lost)

13.2 The Four-Layer Mental Model (So You Don’t Chase Ghosts)

When beginners debug agentic systems, they often jump straight to “the model is wrong”. Sometimes that is true. More often, it is incomplete.

A more reliable way to debug is to separate the system into layers:

Layer 1: The model layer
Prompts, system instructions, role definitions, output formatting rules.

Layer 2: The orchestration layer
Routers, planners, stop conditions, retry logic, fallback policy, confidence updates, step limits.

Layer 3: The tool layer
APIs, calculators, search, retrieval, databases, rate limits, timeouts, schema mismatches.

Layer 4: The state layer
Memory, session state, handoff packs, cached results, previous run residue, user constraints, environment configuration.

Most failures you will see are interactions between layers.

For example: a tool returns empty output (tool layer), the router misinterprets it as success (orchestration layer), confidence stays too high (orchestration layer), the Builder proceeds with missing facts (model layer), and the final answer sounds confident but is wrong.

If you only look at the final answer, you will blame the model. If you trace the run, you will find the real cause.

Debugging starts with choosing the right layer to inspect first.

Copyright © Blockchain Council | All rights reserved

Chapter 13: Agent Debugging (When the Crew Gets Lost)

13.3 Observability: Your Expedition Journal Becomes Your Black Box Recorder

In Chapter 11, you saw an event log as a teaching tool. It helped you understand what happened in a tool orchestration run.

Here is the difference now: the event log is no longer just educational. It is forensic.

When the crew gets lost, the event log is your map of footprints in the snow.

Observability, in plain terms, means your system leaves enough evidence for you to answer:

- What happened?
- In what order?
- With what inputs?
- With what outputs?
- Under what constraints?
- With what confidence?
- And where did it start going wrong?

For agentic systems, you want four kinds of evidence:

a. Structured logs
   A timeline of decisions and actions. Router choices, tool calls, retries, handoffs, check results.

b. Traces
   A way to connect events that belong to the same run. A run ID, step IDs, parent-child relationships between “planner decided” and “executor acted”.

c. Artifacts
   The actual intermediate outputs that matter: handoff packs, parsed tool outputs, retrieved snippets, validation reports.

d. Metrics
   Counts and rates: tool failure rate, retry rate, average steps per run, time per tool, fraction of runs that stop due to low confidence.

You can build a beginner-friendly system using only logs and artifacts. Traces and metrics become more important as you scale.

But the principle is the same: if you cannot see it, you cannot fix it.

Copyright © Blockchain Council | All rights reserved

Chapter 13: Agent Debugging (When the Crew Gets Lost)

13.4 The Anatomy of a Run (What to Capture, Step by Step)

A reliable debugging habit is to treat each agent run like a flight.

A flight has phases: takeoff, climb, cruise, descent, landing. An agent run does too.

Here is a practical run anatomy you can use:

Phase A: Input capture
User query, constraints, context, any configuration flags.

Phase B: Planning or routing
Planner output, router tool choice, role assignments, confidence before action.

Phase C: Execution
Tool inputs, tool outputs, parsing results, retries, fallbacks, timeouts.

Phase D: Assembly
Builder output or answer synthesis, citations or grounding references, formatting.

Phase E: Validation and stopping
Checker reports, stop condition triggered, final confidence, final output.

If you log only the final answer, you are debugging blind.

If you log the phases, you can spot where divergence begins. Debugging becomes “find the first wrong step”, not “argue with the last output”.

A small but powerful mindset: the first wrong step is usually earlier than you think.

Copyright © Blockchain Council | All rights reserved

Chapter 13: Agent Debugging (When the Crew Gets Lost)

13.5 Error Taxonomy: Naming Failures So You Can Fix Them

In early projects, everything is “it failed”.

In mature projects, failures have names.

Naming matters because fixes depend on the failure type. A timeout fix is not the same as a schema fix. A hallucination fix is not the same as a routing fix.

Here is a practical error taxonomy for agentic systems. You do not need to memorize it as a list. You need to recognize the shapes.

Category 1: Input and scope errors

What it looks like: the system solves the wrong problem, or ignores a constraint. Common causes: ambiguous user input, missing constraint capture, handoff pack incomplete, scope not stored in state.

Common fixes: tighten input schema, enforce mandatory constraints, echo scope back internally before acting, validate scope in the checker.

Category 2: Prompt and instruction errors

What it looks like: the agent behaves in an unintended style, misses format requirements, or oversteps role boundaries.

Common causes: prompt too vague, conflicting instructions, role definitions not strict enough, output formatting not enforced.

Common fixes: rewrite role prompts, add “must not” constraints, require structured outputs, reduce prompt length and ambiguity.

Category 3: Schema and parsing errors

What it looks like: malformed JSON, missing required fields, validation fails, “it worked yesterday” formatting drift.

Common causes: model output not constrained, parser too permissive, schema not enforced before downstream use.

Common fixes: strict schema validation, JSON extraction robustification, retry with stricter prompt, reduce degrees of freedom.

Copyright © Blockchain Council | All rights reserved

Chapter 13: Agent Debugging (When the Crew Gets Lost)

13.5 Error Taxonomy: Naming Failures So You Can Fix Them

Category 4: Tool input errors

What it looks like: tool errors that are actually caused by bad inputs, not tool outages.

Common causes: unsanitized expressions, wrong parameter names, missing required fields, unsafe characters.

Common fixes: input sanitizers, validators, canonicalization, smaller tool input templates.

Category 5: Tool execution errors

What it looks like: timeouts, rate limits, transient failures, empty responses.

Common causes: network instability, API quota, tool downtime, slow endpoints.

Common fixes: retries with backoff, timeouts, fallbacks to alternate tools, caching, graceful stopping with low confidence.

Category 6: Tool output interpretation errors

What it looks like: tool output is fine, but the system misreads it.

Common causes: brittle parsing, assuming a field exists, confusing “empty result” with “success”.

Common fixes: explicit output schemas, defensive parsing, “empty is a signal” rules, checker assertions.

Category 7: Planning errors

What it looks like: wrong order of steps, missing steps, looping, over-planning, under-planning.

Common causes: planner prompt unclear, stop conditions weak, planner not constrained to available tools, plan not validated.

Common fixes: plan schema, plan validation, step limits, “must include” steps for certain tasks, planner-executor separation.

Copyright © Blockchain Council | All rights reserved

Category 8: Retry and fallback policy errors

What it looks like: infinite loops, repeated failures, unnecessary retries, wrong fallback chain.

Common causes: retry without change, fallback not limited, no step budget, wrong mapping from failure type to policy.

Common fixes: retry budgets, fallback budgets, “retry must change something” rule, error-type-specific policy.

Category 9: Confidence and stopping errors

What it looks like: the system confidently answers after tool failure, or stops too early while it could recover.

Common causes: confidence not updated by evidence, thresholds too lax or too strict, confidence treated as decoration.

Common fixes: evidence-based confidence updates, minimum confidence for answering, low-confidence stop messages, ask-for-clarification triggers.

Category 10: State and memory errors

What it looks like: stale constraints, mixing sessions, using old results, ignoring updated reality.

Common causes: state not reset, memory not scoped, cached artifacts reused incorrectly, handoff pack not synced after changes.

Common fixes: explicit reset, session IDs, clear scoping rules, “source of truth” rules for state.

Category 11: Multi-agent coordination errors

What it looks like: role leakage, handoff corruption, inconsistent assumptions between agents, deadlocks.

Common causes: handoff pack incomplete, roles not enforced, agents acting on different versions of state.

Common fixes: strict handoff contracts, role prompts with boundaries, orchestrator-controlled state sync, checker validation on handoff.

Copyright © Blockchain Council | All rights reserved

Chapter 13: Agent Debugging (When the Crew Gets Lost)

13.5 Error Taxonomy: Naming Failures So You Can Fix Them

Category 12: Evaluation blind spots

What it looks like: system seems fine until it hits a particular edge case.

Common causes: no test set, no regression tests, no golden traces, relying on manual spot checks.

Common fixes: small targeted test suite, saved failing examples, regression checks at the workflow level. This taxonomy is not meant to overwhelm you. It is meant to stop the most common debugging failure of all: fixing the wrong thing because everything looks like the same problem.

Copyright © Blockchain Council | All rights reserved

Chapter 13: Agent Debugging (When the Crew Gets Lost)

13.6 The Debugging Loop (A Reliable Method That Beats Guessing)

Debugging agentic systems can feel slippery because there are many moving parts. A method keeps you grounded.

Here is a practical loop you can run every time.

Step 1: Freeze the evidence

Save the inputs, configuration, and the full event log for the failing run. Capture artifacts: handoff pack, tool outputs, validation reports.

Why this matters: if you cannot reproduce or compare, you cannot be sure you fixed anything.

Step 2: Identify the symptom precisely

Write the failure in one sentence without interpreting the cause.

Good: “Builder output violates the time budget constraint.”
Bad: “The model is lazy.”

Precision prevents you from fixing stories instead of systems.

Step 3: Find the first divergence

Scan the run timeline and identify the earliest step that looks wrong relative to intended behavior.

This is the single most valuable debugging habit in agentic systems. The final output is late. The cause is often early.

Step 4: Classify the failure type

Use the taxonomy. Is this a tool failure, a parsing failure, a scope capture failure, a role leakage failure?

Classification narrows the fix options.

Step 5: Form a small hypothesis

Pick one cause that explains the divergence and can be tested.

Example: “The handoff pack lost the updated hours per week, so the Builder used stale scope.”

Copyright © Blockchain Council | All rights reserved

Chapter 13: Agent Debugging (When the Crew Gets Lost)

13.6 The Debugging Loop (A Reliable Method That Beats Guessing)

Step 6: Apply the smallest fix that blocks recurrence

Do not “improve everything”. Fix the cause.

- If the issue is schema drift, enforce schema validation earlier.
- If the issue is bad tool input, add sanitization.
- If the issue is role leakage, tighten the role prompt and enforce it with a checker gate.

Step 7: Add a regression guard

Save this failing case as a test. Add a checker rule if appropriate. Store a golden trace.

A fix that is not defended tends to regress.

Step 8: Rerun and compare traces

A good fix changes behavior where it matters, and leaves other behavior stable.

If your fix causes the system to behave wildly differently, you might have “fixed” it by breaking something else.

This loop is not glamorous, but it is how real systems become reliable.

Copyright © Blockchain Council | All rights reserved

Chapter 13: Agent Debugging (When the Crew Gets Lost)

13.7 Fix Patterns (Common Repairs That Actually Work)

Now we move from diagnosis to treatment. These are fix patterns you will use repeatedly.

Pattern A: Make structure mandatory

If downstream steps depend on an output, make that output structured and validated.

Examples:

- Handoff pack must validate against a schema before Builder runs.
- Router decision must validate before tool execution begins.
- Plan must validate before executor starts.

This turns “maybe” into “checkable”.

Pattern B: Reduce degrees of freedom

If outputs vary too much, remove room for interpretation.

Examples:

- Shorter prompts with fewer optional behaviors.
- Fixed section headings for deliverables.
- Explicit constraints echoed in the agent’s internal scope.

You are not limiting intelligence. You are limiting chaos.

Pattern C: Turn “empty” into a first-class signal

Empty output is not “nothing happened”. It is a meaningful event.

Examples:
SEARCH_EMPTY triggers fallback or a clarifying question.
NOTES_MISSING triggers a different path.
TOOL_TIMEOUT triggers retry with backoff and then stop.

A system that treats empty as success will silently drift into wrongness.

Copyright © Blockchain Council | All rights reserved

13.7 Fix Patterns (Common Repairs That Actually Work)

Pattern D: Enforce role boundaries

In multi-agent systems, role leakage is common and subtle.

Examples:

- Researcher must not write the final deliverable.
- Builder must not invent external facts.
- Checker must not rewrite content, only validate.

Role boundaries reduce cognitive collision and make failures easier to isolate.

Pattern E: Make retries smarter, not more frequent

Retries work when they change something meaningful.

Examples:

- Sanitize input on retry.
- Reduce scope on retry.
- Switch tools on fallback.
- Do not retry a side-effect tool without idempotency safeguards.

A retry is a policy move, not a hope ritual.

Pattern F: Add stop conditions that preserve trust

A system that cannot solve reliably should stop rather than hallucinate.

Examples:

- If confidence drops below threshold, stop with guidance.
- If max steps exceeded, stop with a summary of what failed.
- If repeated tool failures occur, stop and request missing inputs.

Trust is built as much by honest stopping as by successful answering.

Copyright © Blockchain Council | All rights reserved

Chapter 13: Agent Debugging (When the Crew Gets Lost)

13.7 Fix Patterns (Common Repairs That Actually Work)

Pattern G: Anchor the system in explicit invariants

An invariant is a rule that must always hold.

Examples:

- Planner must never reference tools that do not exist.
- Tool outputs must be parsed before being used.
- A constraint must be mentioned in scope before building.
- A handoff pack must include time budget fields.

Invariants turn debugging into “which invariant broke?” instead of “why is it weird?”

Pattern H: Build a small regression suite of real failures

The fastest way to improve is to keep your failures.

Save failing cases. Turn them into checks. Rerun them after changes.

This is how you prevent “fix of the week” whiplash.

Copyright © Blockchain Council | All rights reserved

Chapter 13: Agent Debugging (When the Crew Gets Lost)

13.8 Debugging Multi-Agent Systems (Where the Confusion Often Lives)

Debugging a multi-agent system is not harder because there are more agents. It is harder because responsibility is distributed.

A reliable approach is to debug the handoffs first.

If the Researcher handoff pack is wrong, the Builder can be perfectly obedient and still produce the wrong result. If the handoff is correct and the output is wrong, the Builder becomes the suspect. If both are correct but the system still fails, the orchestrator or checker logic is likely at fault.

So, when debugging multi-agent runs:

- First, validate the handoff pack against the schema and against the original user constraints.
- Second, check whether the Builder output respects the handoff pack.
- Third, check whether the checker rules match what you actually care about.
- Fourth, check state synchronization if reality changed mid-run.

This layered approach prevents the most common multi-agent debugging mistake: blaming the last agent in the chain because it is closest to the final output.

In expedition terms, you do not blame the person pitching the tent if the map handed to them was wrong.

Copyright © Blockchain Council | All rights reserved

Chapter 13: Agent Debugging (When the Crew Gets Lost)

13.9 The Debugging Handoff Pack (Between the System and You)

Earlier, “handoff” meant Researcher to Builder.

In debugging, there is another handoff: system to human.
When a run fails, you need a compact packet that lets you investigate without rereading everything.

A useful debugging handoff includes:

- Run identity: run ID, timestamp, version of prompts and code.
- Inputs: user request, constraints, configuration values.
- Outcome: what failed, how it failed, and at which step.
- Artifacts: router output, plan, handoff pack, tool outputs, checker report.
- Stop reason: max steps, low confidence, tool failure, validation fail.
- Hypothesis notes: what you suspect and what you ruled out.

This “handoff to human” is what makes debugging efficient. Without it, you waste time re-creating context instead of fixing causes.

It also creates institutional memory. Future you will thank past you, which is one of the rare situations where time travel is actually possible.

Copyright © Blockchain Council | All rights reserved

Chapter 13: Agent Debugging (When the Crew Gets Lost)

13.10 A Realistic Debugging Walkthrough (No Demo, Just the Method in Motion)

Let's run a realistic scenario in your head. This is not an implementation. It is the sequence of thinking you will use when the system fails.

Scenario: a two-agent system produces a 4-week learning plan. The user reduces available hours mid-run. The final plan still reflects the old time budget.

Symptom statement: "Final deliverable does not reflect updated hours per week."

Now the method:

- Freeze the evidence: save the event log and artifacts from the run.
- Find first divergence: locate the step where the reality change occurred and confirm whether state was updated.
- Classify: this looks like a state sync failure or a handoff pack mismatch.
- Hypothesis: "The scope update happened in state, but the handoff pack used by the Builder was not regenerated or synced."
- Small fix: after a reality change, enforce a sync step that updates the handoff pack and validates it before Builder runs.
- Regression guard: add a test where hours change mid-run and assert the final output reflects the new time budget.
- Rerun and compare: confirm the event log shows sync_scope OK and the output reflects the new hours.

Notice what did not happen: you did not rewrite every prompt. You did not "just ask the model harder". You changed the system so the correct information is guaranteed to be present where it is needed.

That is agent debugging in its most practical form.

Copyright © Blockchain Council | All rights reserved

Chapter 13: Agent Debugging (When the Crew Gets Lost)

13.11 The Difference Between Evaluation and Debugging (Why We Needed Both)

Chapter 7 introduced evaluation: did we arrive, or did we just feel like it?

Evaluation is outcome-focused. It tells you whether the system is performing. Debugging is cause-focused. It tells you why the system behaved the way it did and what to change to improve it.

Evaluation without debugging leads to frustration. You know it is bad, but not how to fix it.

Debugging without evaluation leads to aimless tuning. You can change things, but you do not know whether you improved.

They work together.

Evaluation is the compass that tells you you drifted. Debugging is the method that gets you back on trail and teaches you why the drift happened.

Copyright © Blockchain Council | All rights reserved

Chapter 13: Agent Debugging (When the Crew Gets Lost)

13.12 Building Your “Debugging Kit” (What You Should Carry Forward)

As you move beyond this book, your debugging kit should include:

- A standard event log format, with step IDs and clear outcomes.
- A habit of saving artifacts that matter: plans, handoffs, tool outputs, checker reports.
- A simple error taxonomy that your team uses consistently.
- A small regression set of real failures you rerun after changes.
- A few deterministic checkers that enforce non-negotiables.
- A clear set of stop conditions that prevent confident nonsense.

This is not heavy process. It is the minimum structure required for systems that interact with reality.

Copyright © Blockchain Council | All rights reserved

Chapter 13: Agent Debugging (When the Crew Gets Lost)

13.13 Closing the Expedition

You started at Base Camp with a simple question: what is AI, and what is it not?

You met the map reader inside the machine and learned why tokens and context matter like a backpack's weight. You learned to give directions that do not get you eaten by wolves. You learned how examples shape behavior. You learned how to decompose tasks without pretending the model has magic. You learned to carry field notes with retrieval, not to inflate imagination. You learned to hand the guide a compass and a radio, then to check whether you actually arrived.

Then you crossed the bridge.

You stopped treating the model as a single answer machine and started treating it as a component in a workflow. You learned what "agentic" actually means: goals, loops, and stopping rules. You learned to plan routes that survive reality. You learned to orchestrate tools responsibly, including failure and confidence. You learned to build a specialist crew with explicit handoffs.

And now, at the end, you learned the skill that makes all of that durable: agent debugging.

Because the truth is simple.

Systems that cannot be debugged cannot be trusted for long.

If you take one final idea from this book, let it be this: the goal was never to make you memorize prompts. Prompts change. Tools change. Frameworks change. Model names change. What lasts is your ability to think clearly about a system, observe its behavior, test it, and improve it.

That is what turns "AI that looks impressive" into "AI that actually helps".

So pack up your gear with intention.

Keep your route cards, your field notes, your logs, your checkers, your handoff discipline. And when the fog rolls in, do not argue with the mountain. Open the expedition journal, find the first wrong step, name the failure, and fix the system so it does not repeat the mistake.

That is how beginners become intermediate builders.

Not by never getting lost, but by knowing exactly how to find the trail again.

Copyright © Blockchain Council | All rights reserved

THANK YOU FOR READING !

BROUGHT TO YOU BY **BLOCKCHAIN COUNCIL** ⟶

blockchain-council.org

Copyright © Blockchain Council | All rights reserved