05-Appendix (Bilingual)

05-Appendix（中英對照）

Appendix A: Advanced Prompting Techniques

附錄 A：進階提示技巧

Introduction to Prompting

提示（Prompting）導論

Prompting, the primary interface for interacting with language models, is the process of crafting inputs to guide the model towards generating a desired output. This involves structuring requests, providing relevant context, specifying the output format, and demonstrating expected response types. Well-designed prompts can maximize the potential of language models, resulting in accurate, relevant, and creative responses. In contrast, poorly designed prompts can lead to ambiguous, irrelevant, or erroneous outputs.

提示（Prompting）是與語言模型互動的主要介面，其過程是設計輸入以引導模型產生期望的輸出。這包括結構化請求、提供相關脈絡、指定輸出格式，以及示範期望的回應類型。良好設計的提示能最大化語言模型的潛力，帶來準確、相關且具創意的回應；相反地，設計不良的提示可能造成含糊、無關或錯誤的輸出。

The objective of prompt engineering is to consistently elicit high-quality responses from language models. This requires understanding the capabilities and limitations of the models and effectively communicating intended goals. It involves developing expertise in communicating with AI by learning how to best instruct it.

提示工程的目標是穩定地讓語言模型產出高品質回應。這需要理解模型的能力與限制，並有效傳達目標。也就是透過學習如何最佳地指示 AI，培養與 AI 溝通的專業能力。

This appendix details various prompting techniques that extend beyond basic interaction methods. It explores methodologies for structuring complex requests, enhancing the model's reasoning abilities, controlling output formats, and integrating external information. These techniques are applicable to building a range of applications, from simple chatbots to complex multi-agent systems, and can improve the performance and reliability of agentic applications.

本附錄詳述超越基本互動方法的各種提示技巧，探討如何結構化複雜請求、提升模型的推理能力、控制輸出格式，以及整合外部資訊。這些技巧可用於建構從簡單聊天機器人到複雜多代理系統的各類應用，並能提升代理式應用的效能與可靠性。

Agentic patterns, the architectural structures for building intelligent systems, are detailed in the main chapters. These patterns define how agents plan, utilize tools, manage memory, and collaborate. The efficacy of these agentic systems is contingent upon their ability to interact meaningfully with language models.

代理式模式（Agentic patterns）是建構智慧系統的架構結構，已在主要章節詳述。這些模式定義代理如何規劃、使用工具、管理記憶與協作。代理式系統的效能取決於它們能否與語言模型進行有意義的互動。

Core Prompting Principles

提示核心原則

Core Principles for Effective Prompting of Language Models:

有效提示語言模型的核心原則:

Effective prompting rests on fundamental principles guiding communication with language models, applicable across various models and task complexities. Mastering these principles is essential for consistently generating useful and accurate responses.

有效的提示建立在引導與語言模型溝通的基本原則上，適用於不同模型與任務複雜度。掌握這些原則對穩定産出有用且準確的回應至關重要。

Clarity and Specificity: Instructions should be unambiguous and precise. Language models interpret patterns; multiple interpretations may lead to unintended responses. Define the task, desired output format, and any limitations or requirements. Avoid vague language or assumptions. Inadequate prompts yield ambiguous and inaccurate responses, hindering meaningful output.

清晰與明確：指示應該明確且精準。語言模型是靠辨識模式來理解，多重解讀會導致非預期回應。務必定義任務、期望的輸出格式與限制或要求，避免含糊措辭或假設。提示不足會造成含混且不準確的回應，阻礙有意義的輸出。

Conciseness: While specificity is crucial, it should not compromise conciseness. Instructions should be direct. Unnecessary wording or complex sentence structures can confuse the model or obscure the primary instruction. Prompts should be simple; what is confusing to the user is likely confusing to the model. Avoid intricate language and superfluous information. Use direct phrasing and active verbs to clearly delineate the desired action. Effective verbs include: Act, Analyze, Categorize, Classify, Contrast, Compare, Create, Describe, Define, Evaluate, Extract, Find, Generate, Identify, List, Measure, Organize, Parse, Pick, Predict, Provide, Rank, Recommend, Return, Retrieve, Rewrite, Select, Show, Sort, Summarize, Translate, Write.

簡潔：明確很重要，但不應犧牲簡潔。指示應直接明白。多餘措辭或複雜句構會讓模型困惑或掩蓋主要指令。提示要保持簡單；對使用者而言困惑的内容，很可能也會讓模型困惑。避免繁複語言與多餘資訊，使用直接措辭與主動動詞清楚界定期望的行動。有效動詞包括：Act、Analyze、Categorize、Classify、Contrast、Compare、Create、Describe、Define、Evaluate、Extract、Find、Generate、Identify、List、Measure、Organize、Parse、Pick、Predict、Provide、Rank、Recommend、Return、Retrieve、Rewrite、Select、Show、Sort、Summarize、Translate、Write。

Using Verbs: Verb choice is a key prompting tool. Action verbs indicate the expected operation. Instead of "Think about summarizing this," a direct instruction like "Summarize the following text" is more effective. Precise verbs guide the model to activate relevant training data and processes for that specific task.

使用動詞：動詞選擇是提示的重要工具。動作動詞指示期望的操作。與其說「想想如何摘要這段內容」更有效的是直接指示「摘要以下文字」精準動詞會引導模型針對特定任務啟動相關的訓練資料與處理流程。

Instructions Over Constraints: Positive instructions are generally more effective than negative constraints. Specifying the desired action is preferred to outlining what not to do. While constraints have their place for safety or strict formatting, excessive reliance can cause the model to focus on avoidance rather than the objective. Frame prompts to guide the model directly. Positive instructions align with human guidance preferences and reduce confusion.

以指示優先於限制：正向指示通常比負向限制更有效。比起列出不要做什麼，更應明確指定要做的事。限制在安全或嚴格格式上有其必要，但過度依賴會讓模型專注於避免而非目標。以直接引導模型的方式撰寫提示；正向指示符合人類的引導偏好，也能降低混淆。

Experimentation and Iteration: Prompt engineering is an iterative process. Identifying the most effective prompt requires multiple attempts. Begin with a draft, test it, analyze the output, identify shortcomings, and refine the prompt. Model variations, configurations (like temperature or top-p), and slight phrasing changes can yield different results. Documenting attempts is vital for learning and improvement. Experimentation and iteration are necessary to achieve the desired performance.

實驗與迭代：提示工程是迭代流程。要找到最有效的提示，需要多次嘗試：先寫草稿、測試、分析輸出、找出不足，再精修提示。模型版本、設定（如 temperature 或 top-p）與些微措辭差異都可能產生不同結果。記錄每次嘗試對學習與改進至關重要。要達到期望成效,實驗與迭代不可或缺。

These principles form the foundation of effective communication with language models. By prioritizing clarity, conciseness, action verbs, positive instructions, and iteration, a robust framework is established for applying more advanced prompting techniques.

這些原則構成與語言模型有效溝通的基礎。透過優先清晰、簡潔、動作動詞、正向指示與迭代，可以建立一個穩固的框架,支援更進階的提示技巧。

Basic Prompting Techniques

基本提示技巧

Building on core principles, foundational techniques provide language models with varying levels of information or examples to direct their responses. These methods serve as an initial phase in prompt engineering and are effective for a wide spectrum of applications.

在核心原則之上，基礎技巧會提供語言模型不同程度的資訊或範例來引導回應。這些方法是提示工程的起始階段，適用於廣泛的應用場景。

Zero-Shot Prompting

零樣本提示（Zero-Shot Prompting）

Zero-shot prompting is the most basic form of prompting, where the language model is provided with an instruction and input data without any examples of the desired input-output pair. It relies entirely on the model's pre-training to understand the task and generate a relevant response. Essentially, a zero-shot prompt consists of a task description and initial text to begin the process.

零樣本提示是最基本的提示形式，語言模型只收到指令與輸入資料，沒有任何期望輸入輸出對的範例。它完全依賴模型的預訓練來理解任務並產生相關回應。簡單來說，零樣本提示由任務描述與啟動流程的初始文字組成。

- **When to use:** Zero-shot prompting is often sufficient for tasks that the model has likely encountered extensively during its training, such as simple question answering, text completion, or basic summarization of straightforward text. It's the quickest approach to try first.
- **Example:**
  Translate the following English sentence to French: 'Hello, how are you?'
- 何時使用：對於模型在訓練中可能大量接觸過的任務(如簡單問答、文字補全、基本摘要)通常已足夠,也是最先嘗試的最快方法。
- 範例：
  將以下英文句子翻譯成法文:「Hello, how are you?」

One-Shot Prompting

單樣本提示（One-Shot Prompting）

One-shot prompting involves providing the language model with a single example of the input and the corresponding desired output prior to presenting the actual task. This method serves as an initial demonstration to illustrate the pattern the model is expected to replicate. The purpose is to equip the model with a concrete instance that it can use as a template to effectively execute the given task.

單樣本提示是在呈現實際任務前,提供語言模型一個輸入與對應期望輸出的範例。這種方法用作初步示範,讓模型知道要複製的模式。其目的在於提供具體範本,讓模型能有效執行任務。

- **When to use:** One-shot prompting is useful when the desired output format or style is specific or less common. It gives the model a concrete instance to learn from. It can improve performance compared to zero-shot for tasks requiring a particular structure or tone.
- **Example:**
  Translate the following English sentences to Spanish:
  English: 'Thank you.'
  Spanish: 'Gracias.'

  English: 'Please.'
  Spanish:

- 何時使用：當期望的輸出格式或風格較特定或不常見時,單樣本提示很有用,因為它提供具體範本。對需要特定結構或語氣的任務,效果通常優於零樣本。
  範例：
- 將下列英文句子翻譯成西班牙文：
  English: ‘Thank you.’
  Spanish: ‘Gracias.’

  English: ‘Please.’
  Spanish:

Few-Shot Prompting

少樣本提示（**Few-Shot Prompting**）

Few-shot prompting enhances one-shot prompting by supplying several examples, typically three to five, of input-output pairs. This aims to demonstrate a clearer pattern of expected responses, improving the likelihood that the model will replicate this pattern for new inputs. This method provides multiple examples to guide the model to follow a specific output pattern.

少樣本提示在單樣本提示基礎上提供多個範例，通常是三到五組輸入輸出對，用以展示更清晰的回應模式，提升模型在新輸入上複製該模式的機率。此方法透過多個例子引導模型遵循特定輸出模式。

- **When to use:** Few-shot prompting is particularly effective for tasks where the desired output requires adhering to a specific format, style, or exhibiting nuanced variations. It’s excellent for tasks like classification, data extraction with specific schemas, or generating text in a particular style, especially when zero-shot or one-shot don’t yield consistent results. Using at least three to five examples is a general rule of thumb, adjusting based on task complexity and model token limits.
- **Importance of Example Quality and Diversity:** The effectiveness of few-shot prompting heavily relies on the quality and diversity of the examples provided. Examples should be accurate, representative of the task, and cover potential variations or edge cases the model might encounter. High-quality, well-written examples are crucial; even a small mistake can confuse the model and result in undesired output. Including diverse examples helps the model generalize better to unseen inputs.
- **Mixing Up Classes in Classification Examples:** When using few-shot prompting for classification tasks (where the model needs to categorize input into predefined classes), it’s a best practice to mix up the order of the examples from different classes. This prevents the model from potentially overfitting to the specific sequence of examples and ensures it learns to identify the key features of each class independently, leading to more robust and generalizable performance on unseen data.
- **Evolution to “Many-Shot” Learning:** As modern LLMs like Gemini get stronger with long context modeling, they are becoming highly effective at utilizing “many-shot” learning. This means optimal performance for complex tasks can now be achieved by including a much

larger number of examples—sometimes even hundreds—directly within the prompt, allowing the model to learn more intricate patterns.

- **Example:**
  Classify the sentiment of the following movie reviews as POSITIVE, NEUTRAL, or NEGATIVE:

  Review: “The acting was superb and the story was engaging.”
  Sentiment: POSITIVE

  Review: “It was okay, nothing special.”
  Sentiment: NEUTRAL

  Review: “I found the plot confusing and the characters unlikable.”
  Sentiment: NEGATIVE

  Review: “The visuals were stunning, but the dialogue was weak.”
  Sentiment:

- 何時使用：少樣本提示特別適用於輸出需要遵守特定格式、風格或呈現細微差異的任務。非常適合分類、特定結構的資料擷取，或生成特定風格文字，尤其在零樣本或單樣本無法穩定產出時。一般建議至少提供三到五個範例，並依任務複雜度與模型 token 限制調整。

- 範例品質與多樣性的重要性：少樣本提示的成效高度依賴範例品質與多樣性。範例應精確、具代表性，並涵蓋模型可能遇到的變體或邊界案例。高品質且寫作良好的範例至關重要；哪怕小錯誤都可能讓模型困惑並導致不良輸出。多樣化的範例能幫助模型更好地泛化到未見輸入。

- 分類範例需打散類別順序：在分類任務使用少樣本提示時，最佳實務是打散不同類別的範例順序，避免模型對特定序列過度擬合，並確保它獨立辨識各類別關鍵特徵，讓在未見資料上的表現更穩健、可泛化。

- 演進至「多樣本」學習：隨著 Gemini 等現代 LLM 在長上下文建模上更強，模型越來越擅長「多樣本」學習。這代表對複雜任務，現在可在提示中直接加入更多範例（有時甚至數百個）以達到最佳表現，讓模型學習更細緻的模式。

- 範例：
  將以下影評情緒分類為 POSITIVE、NEUTRAL 或 NEGATIVE：

  Review: “The acting was superb and the story was engaging.”
  Sentiment: POSITIVE

  Review: “It was okay, nothing special.”
  Sentiment: NEUTRAL

  Review: “I found the plot confusing and the characters unlikable.”
  Sentiment: NEGATIVE

  Review: “The visuals were stunning, but the dialogue was weak.”
  Sentiment:

Understanding when to apply zero-shot, one-shot, and few-shot prompting techniques, and thoughtfully crafting and organizing examples, are essential for enhancing the effectiveness of agentic systems. These basic methods serve as the groundwork for various prompting strategies.

理解何時使用零樣本、單樣本與少樣本提示技巧，以及仔細設計與組織範例，是提升代理式系統效能的關鍵。這些基本方法是各類提示策略的基石。

Structuring Prompts
結構化提示

Beyond the basic techniques of providing examples, the way you structure your prompt plays a critical role in guiding the language model. Structuring involves using different sections or elements within the prompt to provide distinct types of information, such as instructions, context, or examples, in a clear and organized manner. This helps the model parse the prompt correctly and understand the specific role of each piece of text.

除了提供範例等基本技巧外，提示的結構方式在引導語言模型上也扮演關鍵角色。結構化是指在提示中使用不同區段或元素，以清楚、有組織地提供指令、脈絡或範例等不同類型資訊，幫助模型正確解析提示並理解每段文字的角色。

System Prompting

系統提示（System Prompting）

System prompting sets the overall context and purpose for a language model, defining its intended behavior for an interaction or session. This involves providing instructions or background information that establish rules, a persona, or overall behavior. Unlike specific user queries, a system prompt provides foundational guidelines for the model's responses. It influences the model's tone, style, and general approach throughout the interaction. For example, a system prompt can instruct the model to consistently respond concisely and helpfully or ensure responses are appropriate for a general audience. System prompts are also utilized for safety and toxicity control by including guidelines such as maintaining respectful language.

系統提示為語言模型設定整體脈絡與目的，定義其在一段互動或會話中的預期行為。它會提供指示或背景資訊來建立規則、角色或整體行為。系統提示不同於特定使用者問題，它提供模型回應的基礎指引，並影響整段互動的語氣、風格與處理方式。例如，系統提示可要求模型一貫地簡潔且有幫助地回應，或確保回應適合一般大眾。系統提示也可用於安全與毒性控制，例如規範使用尊重的語言。

Furthermore, to maximize their effectiveness, system prompts can undergo automatic prompt optimization through LLM-based iterative refinement. Services like the Vertex AI Prompt Optimizer facilitate this by systematically improving prompts based on user-defined metrics and target data, ensuring the highest possible performance for a given task.

此外，為了最大化效果，系統提示可透過 LLM 的迭代式自動提示優化。像 Vertex AI Prompt Optimizer 之類的服務會依據使用者定義的指標與目標資料系統性改善提示，確保任務能達到最佳表現。

- **Example:**
  You are a helpful and harmless AI assistant. Respond to all queries in a polite and informative manner. Do not generate content that is harmful, biased, or inappropriate
- **範例：**
  You are a helpful and harmless AI assistant. Respond to all queries in a polite and informative manner. Do not generate content that is harmful, biased, or inappropriate

Role Prompting

角色提示（**Role Prompting**）

Role prompting assigns a specific character, persona, or identity to the language model, often in conjunction with system or contextual prompting. This involves instructing the model to adopt the knowledge, tone, and communication style associated with that role. For example, prompts such as "Act as a travel guide" or "You are an expert data analyst" guide the model to reflect the perspective and expertise of that assigned role. Defining a role provides a framework for the tone, style, and focused expertise, aiming to enhance the quality and relevance of the output. The desired style within the role can also be specified, for instance, "a humorous and inspirational style."

角色提示為語言模型指定特定角色、人格或身分，通常會與系統提示或情境提示搭配使用。其做法是指示模型採用該角色的知識、語氣與溝通風格。例如「扮演旅遊導遊」或「你是資深資料分析師」會引導模型呈現該角色的觀點與專業。定義角色能提供語氣、風格與專業聚焦的框架, 以提升輸出品質與相關性; 也可指定角色內的風格, 例如「幽默又激勵人心的風格」。

- **Example:**
  Act as a seasoned travel blogger. Write a short, engaging paragraph about the best hidden gem in Rome.
- **範例：**
  扮演資深旅遊部落客。寫一段短而吸引人的文字，介紹羅馬最棒的隱藏景點。

Using Delimiters

使用分隔符

Effective prompting involves clear distinction of instructions, context, examples, and input for language models. Delimiters, such as triple backticks (```), XML tags (,), or markers (—), can be utilized to visually and programmatically separate these sections. This practice, widely used in prompt engineering, minimizes misinterpretation by the model, ensuring clarity regarding the role of each part of the prompt.

有效提示需要清楚區分指令、脈絡、範例與輸入。可使用分隔符（如三個反引號 ```、XML 標籤 、 或標記 —）在視覺與程式上分隔區段。這是提示工程中常見作法，可降低模型誤解並確保各部分角色清楚。

- **Example:**
  Summarize the following article, focusing on the main arguments presented by the author.
  [Insert the full text of the article here]
- **範例：**
  Summarize the following article, focusing on the main arguments presented by the author.
  [Insert the full text of the article here]

Contextual Engineering

情境工程（Context Engineering）

Context engineering, unlike static system prompts, dynamically provides background information crucial for tasks and conversations. This ever-changing information helps models grasp nuances, recall past interactions, and integrate relevant details, leading to grounded responses and smoother exchanges. Examples include previous dialogue, relevant documents (as in Retrieval Augmented Generation), or specific operational parameters. For instance, when discussing a trip to Japan, one might ask for three family-friendly activities in Tokyo, leveraging the existing conversational context. In agentic systems, context engineering is fundamental to core agent behaviors like memory persistence, decision-making, and coordination across sub-tasks. Agents with dynamic contextual pipelines can sustain goals over time, adapt strategies, and collaborate seamlessly with other agents or tools—qualities essential for long-term autonomy. This methodology posits that the quality of a model's output depends more on the richness of the provided context than on the model's architecture. It signifies a significant evolution from traditional prompt engineering, which primarily focused on optimizing the phrasing of immediate user queries. Context engineering expands its scope to include multiple layers of information.

情境工程不同於靜態系統提示，它會動態提供任務與對話所需的背景資訊。這些不斷變化的資訊幫助模型掌握細節、回想過往互動並整合相關內容，帶來更扎實的回應與更順暢的交流。例子包括先前對話、相關文件（如檢索增強生成 RAG）或特定運作參數。例如討論日本旅行時，可利用既有對話脈絡要求「提供東京三個適合家庭的活動」。在代理式系統中，情境工程是記憶持久性、決策與跨子任務協調等核心行為的基礎。具備動態情境管線的代理能長期維持目標、調整策略，並與其他代理或工具無縫協作，這些都是長期自主性的關鍵特質。此方法主張輸出品質更取決於提供的脈絡豐富度，而非模型架構本身。它代表提示工程的重要演進，從單純優化使用者即時提問的措辭，擴展到涵蓋多層資訊的範圍。

These layers include:
這些層面包括：

- **System prompts:** Foundational instructions that define the AI's operational parameters (e.g., "You are a technical writer; your tone must be formal and precise").
- **External data:**
  - **Retrieved documents:** Information actively fetched from a knowledge base to inform responses (e.g., pulling technical specifications).
  - **Tool outputs:** Results from the AI using an external API for real-time data (e.g., querying a calendar for availability).

- **Implicit data:** Critical information such as user identity, interaction history, and environmental state. Incorporating implicit context presents challenges related to privacy and ethical data management. Therefore, robust governance is essential for context engineering, especially in sectors like enterprise, healthcare, and finance.
- 系統提示: 定義 AI 運作參數的基礎指令(例如「你是技術寫作者;語氣需正式且精確」)。
- 外部資料:
  - 檢索文件: 從知識庫主動取得以支援回應的資訊(例如擷取技術規格)。
  - 工具輸出: AI 透過外部 API 取得的即時資料結果(例如查詢行事曆的可用性)。
- 隱含資料: 如使用者身分、互動歷史與環境狀態等關鍵資訊。納入隱含脈絡會帶來隱私與倫理資料管理的挑戰，因此在企業、醫療與金融等領域，情境工程尤其需要健全治理。

The core principle is that even advanced models underperform with a limited or poorly constructed view of their operational environment. This practice reframes the task from merely answering a question to building a comprehensive operational picture for the agent. For example, a context-engineered agent would integrate a user's calendar availability (tool output), the professional relationship with an email recipient (implicit data), and notes from previous meetings (retrieved documents) before responding to a query. This enables the model to generate highly relevant, personalized, and pragmatically useful outputs. The "engineering" aspect involves creating robust pipelines to fetch and transform this data at runtime and establishing feedback loops to continually improve context quality.

核心原則是: 即使是先進模型，若對運作環境的理解有限或建構不良，也會表現不佳。此作法將任務從「回答問題」重新定義為「為代理建立完整的運作圖像」。例如，一個經過情境工程的代理會在回應前整合使用者行事曆可用性(工具輸出)、與收件人的專業關係(隱含資料)、以及先前會議筆記(檢索文件)。這能讓模型產出高度相關、個人化且務實有用的結果。「工程」的部分在於建立可在執行時抓取與轉換資料的穩健管線，以及設定回饋迴路，持續提升情境品質。

To implement this, specialized tuning systems, such as Google's Vertex AI prompt optimizer, can automate the improvement process at scale. By systematically evaluating responses against sample inputs and predefined metrics, these tools can enhance model performance and adapt prompts and system instructions across different models without extensive manual rewriting. Providing an optimizer with sample prompts, system instructions, and a template allows it to programmatically refine contextual inputs, offering a structured method for implementing the necessary feedback loops for sophisticated Context Engineering.

This structured approach differentiates a rudimentary AI tool from a more sophisticated, contextually-aware system. It treats context as a primary component, emphasizing what the agent knows, when it knows it, and how it uses that information. This practice ensures the model has a well-rounded understanding of the user's intent, history, and current environment. Ultimately, Context Engineering is a crucial methodology for transforming stateless chatbots into highly capable, situationally-aware systems.

為了實作此方法,專門的調校系統(如 Google 的 Vertex AI prompt optimizer）可在規模化下自動化改進流程。透過將回應與樣本輸入與預先定義指標進行系統性評估，這些工具能提升模型效能，並在不同模型間調整提示與系統指令，而無需大量手動改寫。提供樣本提示、系統指令與模板給優化器，即可程式化精修情境輸入，成為實作進階情境工程回饋迴路的結構化方法。

這種結構化方法讓粗糙的 AI 工具與更成熟、具情境感知的系統有所區別。它將情境視為主要元件，強調代理知道什麼、何時知道、以及如何使用資訊。這種作法確保模型對使用者意圖、歷史與當前環境有完整理解。最終，情境工程是將無狀態聊天機器人轉化為高能力、具情境感知系統的關鍵方法論。

Structured Output
結構化輸出

Often, the goal of prompting is not just to get a free-form text response, but to extract or generate information in a specific, machine-readable format. Requesting structured output, such as JSON, XML, CSV, or Markdown tables, is a crucial structuring technique. By explicitly asking for the output in a particular format and potentially providing a schema or example of the desired structure, you guide the model to organize its response in a way that can be easily parsed and used by other parts of your agentic system or application. Returning JSON objects for data extraction is beneficial as it forces the model to create a structure and can limit hallucinations. Experimenting with output formats is recommended, especially for non-creative tasks like extracting or categorizing data.

提示的目的往往不只是取得自由文字回應，也包含以特定、機器可讀格式擷取或生成資訊。要求結構化輸出（如 JSON、XML、CSV 或 Markdown 表格）是關鍵的結構化技巧。透過明確要求特定格式輸出，並可提供結構綱要或範例，你能引導模型以可被其他系統輕鬆解析與使用的方式組織回應。用 JSON 進行資料擷取有助於迫使模型建立結構並降低幻覺。特別是在資料擷取或分類等非創意任務上,建議嘗試不同的輸出格式。

- **Example:**
  Extract the following information from the text below and return it as a JSON object with keys `name`, `address`, and `phone.number`.

  Text: “Contact John Smith at 123 Main St, Anytown, CA or call (555) 123-4567.”
- **範例:**
  從下列文字中擷取資訊,並以 JSON 物件回傳，包含 `name`、`address` 與 `phone.number` 三個鍵。

  Text: “Contact John Smith at 123 Main St, Anytown, CA or call (555) 123-4567.”

Effectively utilizing system prompts, role assignments, contextual information, delimiters, and structured output significantly enhances the clarity, control, and utility of interactions with language models, providing a strong foundation for developing reliable agentic systems. Requesting structured output is crucial for creating pipelines where the language model’s output serves as the input for subsequent system or processing steps.

有效運用系統提示、角色設定、情境資訊、分隔符與結構化輸出，可大幅提升與語言模型互動的清晰度、可控性與效用，為建構可靠的代理式系統奠定堅實基礎。要求結構化輸出對建立「語言模型輸出作為後續系統或處理步驟輸入」的管線至關重要。

Leveraging Pydantic for an Object-Oriented Facade: A powerful technique for enforcing structured output and enhancing interoperability is to use the LLM's generated data to populate instances of Pydantic objects. Pydantic is a Python library for data validation and settings management using Python type annotations. By defining a Pydantic model, you create a clear and enforceable schema for your desired data structure. This approach effectively provides an object-oriented facade to the prompt's output, transforming raw text or semi-structured data into validated, type-hinted Python objects.

使用 Pydantic 建立物件導向外觀: 強化結構化輸出與互通性的有力技巧之一,是用 LLM 生成的資料填入 Pydantic 物件實例。Pydantic 是 Python 的資料驗證與設定管理函式庫，使用型別註記。透過定義 Pydantic 模型，你能建立清楚且可強制的資料結構綱要。此方法等同於為提示輸出提供物件導向的外觀，將原始文字或半結構化資料轉為已驗證且具型別提示的 Python 物件。

You can directly parse a JSON string from an LLM into a Pydantic object using the `model.validate.json` method. This is particularly useful as it combines parsing and validation in a single step.

你可以使用 `model.validate.json` 方法，將 LLM 產出的 JSON 字串直接解析成 Pydantic 物件。這特別有用，因為它把解析與驗證合併成一步。

```python
from pydantic import BaseModel, EmailStr, Field, ValidationError
from typing import List, Optional
from datetime import date


# --- Pydantic Model Definition (from above) ---
class User(BaseModel):
    name: str = Field(..., description="The full name of the user.")
    email: EmailStr = Field(..., description="The user's email address.")
    date_of_birth: Optional[date] = Field(None, description="The user's date of
birth.")
    interests: List[str] = Field(default_factory=list, description="A list of the
user's interests.")


# --- Hypothetical LLM Output ---
llm_output_json = """
{
    "name": "Alice Wonderland",
    "email": "alice.w@example.com",
    "date_of_birth": "1995-07-21",
    "interests": [
        "Natural Language Processing",
        "Python Programming",
        "Gardening"
    ]
}
"""


# --- Parsing and Validation ---
try:
    # Use the model_validate_json class method to parse the JSON string.
    # This single step parses the JSON and validates the data against the User
model.
    user_object = User.model_validate_json(llm_output_json)

    # Now you can work with a clean, type-safe Python object.
    print("Successfully created User object!")
    print(f"Name: {user_object.name}")
    print(f"Email: {user_object.email}")
    print(f"Date of Birth: {user_object.date_of_birth}")
    print(f"First Interest: {user_object.interests[0]}")

    # You can access the data like any other Python object attribute.
    # Pydantic has already converted the 'date_of_birth' string to a
datetime.date object.
    print(f"Type of date_of_birth: {type(user_object.date_of_birth)}")
except ValidationError as e:
    # If the JSON is malformed or the data doesn't match the model's types,
    # Pydantic will raise a ValidationError.
    print("Failed to validate JSON from LLM.")
    print(e)
```

This Python code demonstrates how to use the Pydantic library to define a data model and validate JSON data. It defines a User model with fields for name, email, date of birth, and interests, including type hints and descriptions. The code then parses a hypothetical JSON output from a Large Language Model (LLM) using the `model.validate.json` method of the User model. This method handles both JSON parsing and data validation according to the model's structure and types. Finally, the code accesses the validated data from the resulting Python object and includes error handling for ValidationError in case the JSON is invalid.

這段Python程式碼示範如何使用Pydantic定義資料模型並驗證JSON資料。它定義了User模型，包含姓名、電子郵件、生日與興趣等欄位，並附上型別提示與描述。程式碼接著使用User模型的`model.validate.json`方法解析一段假設的LLM JSON輸出,此方法同時處理JSON解析與依模型結構與型別進行驗證。最後，程式碼讀取驗證後的Python物件資料，並在JSON無效時用ValidationError做錯誤處理。

For XML data, the xmltodict library can be used to convert the XML into a dictionary, which can then be passed to a Pydantic model for parsing. By using Field aliases in your Pydantic model, you can seamlessly map the often verbose or attribute-heavy structure of XML to your object's fields.

對於 XML 資料,可使用 xmltodict 函式庫將 XML 轉為字典,再傳給 Pydantic 模型解析。透過在 Pydantic 模型中使用 Field 別名,可將 XML 常見的冗長或屬性密集結構順暢對應到物件欄位。

This methodology is invaluable for ensuring the interoperability of LLM-based components with other parts of a larger system. When an LLM's output is encapsulated within a Pydantic object, it can be reliably passed to other functions, APIs, or data processing pipelines with the assurance that the data conforms to the expected structure and types. This practice of "parse, don't validate" at the boundaries of your system components leads to more robust and maintainable applications.

這種方法能確保以 LLM 為基礎的元件與大型系統的其他部分具有良好互通性。當 LLM 的輸出被封裝為 Pydantic 物件時，就能可靠地傳遞給其他函式、API 或資料處理管線，並確保資料符合預期結構與型別。這種「在邊界處解析而非僅驗證」的作法，能打造更健壯、可維護的應用。

Structuring Prompts Beyond the basic techniques of providing examples, the way you structure your prompt plays a critical role in guiding the language model. Structuring involves using different sections or elements within the prompt to provide distinct types of information, such as instructions, context, or examples, in a clear and organized manner. This helps the model parse the prompt correctly and understand the specific role of each piece of text.

結構化提示:除了提供範例等基本技巧外,提示的結構方式在引導語言模型上扮演關鍵角色。結構化是指在提示中使用不同區段或元素,以清楚、有組織地提供指令、脈絡或範例等不同類型資訊,幫助模型正確解析提示並理解每段文字的特定角色。

Reasoning and Thought Process Techniques

推理與思考流程技巧

Large language models excel at pattern recognition and text generation but often face challenges with tasks requiring complex, multi-step reasoning. This appendix focuses on techniques designed to enhance these reasoning capabilities by encouraging models to reveal their internal thought processes. Specifically, it addresses methods to improve logical deduction, mathematical computation, and planning.

大型語言模型擅長模式辨識與文字生成，但在需要複雜多步推理的任務上常面臨挑戰。本附錄聚焦於透過引導模型揭露內部思考流程來提升推理能力的技巧，具體涵蓋改進邏輯推導、數學運算與規劃的方法。

Chain of Thought (CoT)

思考鏈（**Chain of Thought, CoT**）

The Chain of Thought (CoT) prompting technique is a powerful method for improving the reasoning abilities of language models by explicitly prompting the model to generate intermediate reasoning steps before arriving at a final answer. Instead of just asking for the result, you instruct the model to “think step by step.” This process mirrors how a human might break down a problem into smaller, more manageable parts and work through them sequentially.

思考鏈（CoT）提示技巧是提升語言模型推理能力的強大方法，透過明確要求模型在得出最終答案之前生成中間推理步驟。你不只要求結果，而是指示模型「一步一步思考」。這個過程如同人類將問題拆解為較小、可管理的部分並逐步解決。

CoT helps the LLM generate more accurate answers, particularly for tasks that require some form of calculation or logical deduction, where models might otherwise struggle and produce incorrect results. By generating these intermediate steps, the model is more likely to stay on track and perform the necessary operations correctly.

CoT 能協助 LLM 產生更準確的答案，尤其適用於需要計算或邏輯推導的任務，否則模型可能會卡住或產生錯誤結果。透過產生中間步驟，模型更容易維持正確軌道並正確完成必要的運算。

There are two main variations of CoT:

CoT 有兩種主要變體:

- **Zero-Shot CoT:** This involves simply adding the phrase “Let’s think step by step” (or similar phrasing) to your prompt without providing any examples of the reasoning process. Surprisingly, for many tasks, this simple addition can significantly improve the model’s performance by triggering its ability to expose its internal reasoning trace.
  - **Example (Zero-Shot CoT):**
    If a train travels at 60 miles per hour and covers a distance of 240 miles, how long did the journey take? Let’s think step by step.
- **零樣本 CoT**: 只需在提示中加入「Let’s think step by step」(或類似說法)，不提供推理過程範例。令人驚訝的是，對許多任務而言，這個簡單的加入就能顯著提升表現，因為它觸發模型揭示內部推理軌跡的能力。
  - **範例(零樣本 CoT):**
    If a train travels at 60 miles per hour and covers a distance of 240 miles, how long did the journey take? Let’s think step by step.
- **Few-Shot CoT:** This combines CoT with few-shot prompting. You provide the model with several examples where both the input, the step-by-step reasoning process, and the final output are shown. This gives the model a clearer template for how to perform the reasoning and structure its response, often leading to even better results on more complex tasks compared to zero-shot CoT.
  - **Example (Few-Shot CoT):**
    Q: The sum of three consecutive integers is 36. What are the integers?
    A: Let the first integer be x. The next consecutive integer is x+1, and the third is x+2. The sum is x + (x+1) + (x+2) = 3x + 3. We know the sum is 36, so 3x + 3 = 36. Subtract 3 from both sides: 3x = 33. Divide by 3: x = 11. The integers are 11, 11+1=12, and 11+2=13. The integers are 11, 12, and 13.

Q: Sarah has 5 apples, and she buys 8 more. She eats 3 apples. How many apples does she have left? Let's think step by step.
A: Let's think step by step. Sarah starts with 5 apples. She buys 8 more, so she adds 8 to her initial amount: 5 + 8 = 13 apples. Then, she eats 3 apples, so we subtract 3 from the total: 13 - 3 = 10. Sarah has 10 apples left. The answer is 10.

- **少樣本 CoT**: 將 CoT 與少樣本提示結合。你提供模型多個範例,展示輸入、逐步推理過程與最終輸出。這讓模型更清楚如何推理與組織回應,通常比零樣本 CoT 在複雜任務上效果更佳。
  - **範例(少樣本 CoT)**:

    Q: The sum of three consecutive integers is 36. What are the integers?
    A: Let the first integer be x. The next consecutive integer is x+1, and the third is x+2. The sum is x + (x+1) + (x+2) = 3x + 3. We know the sum is 36, so 3x + 3 = 36. Subtract 3 from both sides: 3x = 33. Divide by 3: x = 11. The integers are 11, 11+1=12, and 11+2=13. The integers are 11, 12, and 13.

    Q: Sarah has 5 apples, and she buys 8 more. She eats 3 apples. How many apples does she have left? Let's think step by step.
    A: Let's think step by step. Sarah starts with 5 apples. She buys 8 more, so she adds 8 to her initial amount: 5 + 8 = 13 apples. Then, she eats 3 apples, so we subtract 3 from the total: 13 - 3 = 10. Sarah has 10 apples left. The answer is 10.

CoT offers several advantages. It is relatively low-effort to implement and can be highly effective with off-the-shelf LLMs without requiring fine-tuning. A significant benefit is the increased interpretability of the model's output; you can see the reasoning steps it followed, which helps in understanding why it arrived at a particular answer and in debugging if something went wrong. Additionally, CoT appears to improve the robustness of prompts across different versions of language models, meaning the performance is less likely to degrade when a model is updated. The main disadvantage is that generating the reasoning steps increases the length of the output, leading to higher token usage, which can increase costs and response time.

CoT 有數個優點: 實作成本低, 不需微調即可在現成 LLM 上有很好的效果。其顯著優點是提升可解釋性, 你可以看到模型採取的推理步驟, 有助於理解答案來源並在出錯時除錯。此外, CoT 似乎能提升提示在不同模型版本間的穩健性, 使模型更新時效能不易退化。主要缺點是推理步驟會增加輸出長度, 導致 token 使用量增加, 成本與回應時間也會提高。

Best practices for CoT include ensuring the final answer is presented *after* the reasoning steps, as the generation of the reasoning influences the subsequent token predictions for the answer. Also, for tasks with a single correct answer (like mathematical problems), setting the model's temperature to 0 (greedy decoding) is recommended when using CoT to ensure deterministic selection of the most probable next token at each step.

CoT 的最佳實務包括確保最終答案放在推理步驟之後, 因為推理的生成會影響接續 token 的預測。此外, 對於只有單一正確答案的任務(如數學題), 使用 CoT 時建議將模型 temperature 設為 0 (貪婪解碼), 以確保每一步都選擇最可能的下一個 token。

Self-Consistency

自我一致性（Self-Consistency）

Building on the idea of Chain of Thought, the Self-Consistency technique aims to improve the reliability of reasoning by leveraging the probabilistic nature of language models. Instead of relying on a single greedy reasoning path (as in basic CoT), Self-Consistency generates multiple diverse reasoning paths for the same problem and then selects the most consistent answer among them.

自我一致性建立在 CoT 概念上，透過利用語言模型的機率特性來提升推理可靠性。不同於只依賴單一路徑的貪婪推理（如基本 CoT），自我一致性會針對同一問題生成多條多樣化的推理路徑，並從中選出最一致的答案。

Self-Consistency involves three main steps:

自我一致性包含三個主要步驟:

1. **Generating Diverse Reasoning Paths:** The same prompt (often a CoT prompt) is sent to the LLM multiple times. By using a higher temperature setting, the model is encouraged to explore different reasoning approaches and generate varied step-by-step explanations.
2. **Extract the Answer:** The final answer is extracted from each of the generated reasoning paths.
3. **Choose the Most Common Answer:** A majority vote is performed on the extracted answers. The answer that appears most frequently across the diverse reasoning paths is selected as the final, most consistent answer.
4. 生成多樣化推理路徑: 同一個提示（常為 CoT 提示）多次送入 LLM。透過較高的 temperature 設定,鼓勵模型探索不同推理方式並產生多樣化的逐步解釋。
5. 抽取答案: 從各條推理路徑中抽取最終答案。
6. 選擇最常見答案: 對抽取的答案進行多數決,出現頻率最高者即為最終、最一致的答案。

This approach improves the accuracy and coherence of responses, particularly for tasks where multiple valid reasoning paths might exist or where the model might be prone to errors in a single attempt. The benefit is a pseudo-probability likelihood of the answer being correct, increasing overall accuracy. However, the significant cost is the need to run the model multiple times for the same query, leading to much higher computation and expense.

此方法能提升回應的準確度與一致性，尤其適用於存在多種合理推理路徑或模型單次易出錯的任務。其優點是提供近似機率的正確性指標，提升整體準確度；缺點是需要對同一問題多次執行模型，計算成本與費用大幅增加。

- **Example (Conceptual):**
  - *Prompt:* “Is the statement ‘All birds can fly’ true or false? Explain your reasoning.”
  - *Model Run 1 (High Temp):* Reasons about most birds flying, concludes True.
  - *Model Run 2 (High Temp):* Reasons about penguins and ostriches, concludes False.

  - *Model Run 3 (High Temp):* Reasons about birds *in general*, mentions exceptions briefly, concludes True.
  - *Self-Consistency Result:* Based on majority vote (True appears twice), the final answer is "True". (Note: A more sophisticated approach would weigh the reasoning quality).
- **範例（概念性）:**
  - *Prompt:* "Is the statement 'All birds can fly' true or false? Explain your reasoning."
  - *Model Run 1 (High Temp):* Reasons about most birds flying, concludes True.
  - *Model Run 2 (High Temp):* Reasons about penguins and ostriches, concludes False.
  - *Model Run 3 (High Temp):* Reasons about birds *in general*, mentions exceptions briefly, concludes True.
  - *Self-Consistency Result:* Based on majority vote (True appears twice), the final answer is "True". (Note: A more sophisticated approach would weigh the reasoning quality).

Step-Back Prompting

後退式提示（Step-Back Prompting）

Step-back prompting enhances reasoning by first asking the language model to consider a general principle or concept related to the task before addressing specific details. The response to this broader question is then used as context for solving the original problem.

後退式提示透過先請語言模型思考與任務相關的一般原則或概念，再處理具體細節，來提升推理品質。對較廣泛問題的回應會作為解決原始問題的脈絡。

This process allows the language model to activate relevant background knowledge and wider reasoning strategies. By focusing on underlying principles or higher-level abstractions, the model can generate more accurate and insightful answers, less influenced by superficial elements. Initially considering general factors can provide a stronger basis for generating specific creative outputs. Step-back prompting encourages critical thinking and the application of knowledge, potentially mitigating biases by emphasizing general principles.

此流程讓語言模型啟動相關背景知識與更廣泛的推理策略。透過聚焦底層原則或較高層次的抽象概念，模型能產生更準確且更具洞察的答案，較不受表面因素影響。先考量一般因素也能為後續特定創意產出提供更強的基礎。後退式提示鼓勵批判思考與知識應用，並可透過強調一般原則來降低偏誤。

- **Example:**
  - *Prompt 1 (Step-Back):* "What are the key factors that make a good detective story?"
  - *Model Response 1:* (Lists elements like red herrings, compelling motive, flawed protagonist, logical clues, satisfying resolution).
  - *Prompt 2 (Original Task + Step-Back Context):* "Using the key factors of a good detective story [insert Model Response 1 here], write a short plot summary for a new mystery novel set in a small town."
- **範例：**
  - *Prompt 1 (Step-Back):* "What are the key factors that make a good detective story?"

- *Model Response 1:*（列出如誤導線索、強烈動機、有缺陷的主角、邏輯線索、令人滿意的結局等元素）。
- *Prompt 2 (Original Task + Step-Back Context):* “Using the key factors of a good detective story [insert Model Response 1 here], write a short plot summary for a new mystery novel set in a small town.”

Tree of Thoughts (ToT)

思考樹（Tree of Thoughts, ToT）

Tree of Thoughts (ToT) is an advanced reasoning technique that extends the Chain of Thought method. It enables a language model to explore multiple reasoning paths concurrently, instead of following a single linear progression. This technique utilizes a tree structure, where each node represents a “thought”—a coherent language sequence acting as an intermediate step. From each node, the model can branch out, exploring alternative reasoning routes.

思考樹（ToT）是延伸自 CoT 的進階推理技巧。它讓語言模型能同時探索多條推理路徑，而非只沿著單一線性流程前進。此技術使用樹狀結構，每個節點代表一個「想法」——一段作為中間步驟的連貫語言序列。模型可以從每個節點分支，探索不同推理路徑。

ToT is particularly suited for complex problems that require exploration, backtracking, or the evaluation of multiple possibilities before arriving at a solution. While more computationally demanding and intricate to implement than the linear Chain of Thought method, ToT can achieve superior results on tasks necessitating deliberate and exploratory problem-solving. It allows an agent to consider diverse perspectives and potentially recover from initial errors by investigating alternative branches within the “thought tree.”

ToT 特別適用於需要探索、回溯或在得出解答前評估多種可能性的複雜問題。雖然比線性 CoT 更耗算力且實作更複雜，但 ToT 在需要深思與探索式解題的任務上可取得更佳成效。它讓代理能考慮多元觀點，並透過探索「思考樹」中的其他分支來挽回初始錯誤。

- **Example (Conceptual):** For a complex creative writing task like “Develop three different possible endings for a story based on these plot points,” ToT would allow the model to explore distinct narrative branches from a key turning point, rather than just generating one linear continuation.

- **範例(概念性):** 對於像「根據這些劇情點設計三種不同結局」的複雜創作任務,ToT 讓模型能從關鍵轉折點探索不同敘事分支,而不是只生成單一路徑的續寫。

These reasoning and thought process techniques are crucial for building agents capable of handling tasks that go beyond simple information retrieval or text generation. By prompting models to expose their reasoning, consider multiple perspectives, or step back to general principles, we can significantly enhance their ability to perform complex cognitive tasks within agentic systems.

這些推理與思考流程技巧對於打造能處理超越單純資訊檢索或文字生成的代理至關重要。透過引導模型揭示推理、考量多種視角或後退到一般原則,我們能大幅提升代理在複雜認知任務上的表現。

Action and Interaction Techniques

行動與互動技巧

Intelligent agents possess the capability to actively engage with their environment, beyond generating text. This includes utilizing tools, executing external functions, and participating in iterative cycles of observation, reasoning, and action. This section examines prompting techniques designed to enable these active behaviors.

智慧代理具備主動與環境互動的能力，不僅能產生文字，還能使用工具、執行外部函式，並進行觀察、推理與行動的迭代循環。本節探討用於啟動這些主動行為的提示技巧。

Tool Use / Function Calling

工具使用／函式呼叫

A crucial ability for an agent is using external tools or calling functions to perform actions beyond its internal capabilities. These actions may include web searches, database access, sending emails, performing calculations, or interacting with external APIs. Effective prompting for tool use involves designing prompts that instruct the model on the appropriate timing and methodology for tool utilization.

代理的關鍵能力之一是使用外部工具或呼叫函式來執行超出其內部能力的行動，例如網路搜尋、存取資料庫、發送郵件、計算，或與外部 API 互動。有效的工具使用提示需要設計指示，讓模型知道合適的時機與方法來使用工具。

Modern language models often undergo fine-tuning for “function calling” or “tool use.” This enables them to interpret descriptions of available tools, including their purpose and parameters. Upon receiving a user request, the model can determine the necessity of tool use, identify the appropriate tool, and format the required arguments for its invocation. The model does not execute the tool directly. Instead, it generates a structured output, typically in JSON format, specifying the tool and its parameters. An agentic system then processes this output, executes the tool, and provides the tool’s result back to the model, integrating it into the ongoing interaction.

現代語言模型常針對「函式呼叫」或「工具使用」進行微調，使其能理解可用工具的描述，包括目的與參數。當收到使用者請求時，模型可判斷是否需要使用工具、選擇合適工具並格式化呼叫所需的參數。模型不會直接執行工具，而是生成結構化輸出(通常為 JSON)來指定工具與參數。代理式系統會處理該輸出、執行工具，並將結果回饋給模型，整合進持續的互動中。

- **Example:**
  You have access to a weather tool that can get the current weather for a specified city. The tool is called ‘`get.current.weather`’ and takes a ‘`city`’ parameter (string).

  User: What’s the weather like in London right now?

  - *Expected Model Output (Function Call):*
    {
    “tool.code”: “get.current.weather”,
    “tool.name”: “get.current.weather”,

“parameters”: {
“city”: “London”
}
}

- **範例:**
  你可以使用一個天氣工具來取得指定城市的即時天氣。此工具名為‘`get.current.weather`’，接受‘`city`’參數(字串)。

  User: What’s the weather like in London right now?

  - *Expected Model Output (Function Call):*
    {
    “tool.code”: “get.current.weather”,
    “tool.name”: “get.current.weather”,
    “parameters”: {
    “city”: “London”
    }
    }

ReAct (Reason & Act)

ReAct（推理與行動）

ReAct, short for Reason and Act, is a prompting paradigm that combines Chain of Thought-style reasoning with the ability to perform actions using tools in an interleaved manner. ReAct mimics how humans operate – we reason verbally and take actions to gather more information or make progress towards a goal.

ReAct（Reason and Act）是一種提示範式，將 CoT 式推理與工具行動交錯結合。ReAct 模擬人類行為——我們一邊口頭推理，一邊採取行動以獲取更多資訊或推進目標。

The ReAct pattern involves a loop:

ReAct 模式包含一個迴圈:

1. **Thought:** The model generates a thought process, explaining its current understanding and plan.
2. **Action:** Based on the thought, the model decides to perform an action, often using a tool (e.g., Search, Calculator, API call). It outputs the name of the tool and the required input.
3. **Observation:** The agentic system executes the tool and provides the result (Observation) back to the model.
4. The loop continues with the model generating a new “Thought” based on the latest “Observation,” leading to further “Actions” and “Observations” until the task is complete and the model outputs a “Final Answer.”
5. **Thought**: 模型產生思考過程，說明目前理解與計畫。

6. **Action**： 根據思考，模型決定採取行動，通常是使用工具（如搜尋、計算器或 API 呼叫），並輸出工具名稱與所需輸入。
7. **Observation**： 代理式系統執行工具並將結果（Observation）回饋給模型。
8. 模型再依最新 Observation 產生新的 Thought，持續進行 Actions 與 Observations 的迴圈，直到完成任務並輸出 Final Answer。

This interleaved process of thinking and acting allows the agent to dynamically gather information, react to tool outputs, and refine its approach, making it particularly effective for tasks requiring interaction with dynamic environments or external knowledge sources.

思考與行動交錯的流程讓代理能動態蒐集資訊、回應工具輸出並修正策略，特別適合需要與動態環境或外部知識來源互動的任務。

- **Example (Illustrative ReAct Trace):**
  User: What is the capital of France and its current population?

  Thought: The user is asking for two pieces of information: the capital of France and its population. I can use a search tool to find this information. I will start by confirming the capital, although I am likely to know it.
  Action: Search({“query”: “capital of France”})
  Observation: Paris

  Thought: Okay, the capital is Paris. Now I need to find the population of Paris.
  Action: Search({“query”: “current population of Paris”})
  Observation: The population of Paris was estimated to be around 2.1 million people in 2023.

  Thought: I have found both pieces of information. I can now provide the final answer to the user.
  Final Answer: The capital of France is Paris, and its estimated population in 2023 was around 2.1 million people.

- **範例（示意 ReAct 流程）:**
  User: What is the capital of France and its current population?

  Thought: The user is asking for two pieces of information: the capital of France and its population. I can use a search tool to find this information. I will start by confirming the capital, although I am likely to know it.
  Action: Search({“query”: “capital of France”})
  Observation: Paris

  Thought: Okay, the capital is Paris. Now I need to find the population of Paris.
  Action: Search({“query”: “current population of Paris”})
  Observation: The population of Paris was estimated to be around 2.1 million people in 2023.

  Thought: I have found both pieces of information. I can now provide the final answer to the user.
  Final Answer: The capital of France is Paris, and its estimated population in 2023 was around 2.1 million people.

These techniques are vital for building agents that can actively engage with the world, retrieve real-time information, and perform tasks that require interacting with external systems.
這些技巧對於打造能主動與世界互動、取得即時資訊並執行需與外部系統互動任務的代理至關重要。

Advanced Techniques
進階技巧

Beyond the foundational, structural, and reasoning patterns, there are several other prompting techniques that can further enhance the capabilities and efficiency of agentic systems. These range from using AI to optimize prompts to incorporating external knowledge and tailoring responses based on user characteristics.

除了基礎、結構與推理模式之外，還有多種提示技巧能進一步提升代理式系統的能力與效率，涵蓋用 AI 來優化提示、整合外部知識,以及依使用者特性調整回應。

Automatic Prompt Engineering (APE)

自動提示工程（Automatic Prompt Engineering, APE）

Recognizing that crafting effective prompts can be a complex and iterative process, Automatic Prompt Engineering (APE) explores using language models themselves to generate, evaluate, and refine prompts. This method aims to automate the prompt writing process, potentially enhancing model performance without requiring extensive human effort in prompt design.

由於撰寫有效提示是複雜且迭代的流程，自動提示工程（APE）探索讓語言模型自行生成、評估與精修提示。此方法旨在自動化提示撰寫流程，有機會在不需要大量人工設計的情況下提升模型表現。

The general idea is to have a “meta-model” or a process that takes a task description and generates multiple candidate prompts. These prompts are then evaluated based on the quality of the output they produce on a given set of inputs (perhaps using metrics like BLEU or ROUGE, or human evaluation). The best-performing prompts can be selected, potentially refined further, and used for the target task. Using an LLM to generate variations of a user query for training a chatbot is an example of this.

其一般做法是由「元模型」或流程接收任務描述並生成多個候選提示，再根據這些提示在特定輸入集上產出的品質進行評估（可能使用 BLEU、ROUGE 等指標或人工評估）。表現最佳的提示會被選出，可能再進一步精修後用於目標任務。使用 LLM 生成使用者提問的多種變體以訓練聊天機器人，就是一個例子。

- **Example (Conceptual):** A developer provides a description: “I need a prompt that can extract the date and sender from an email.” An APE system generates several candidate prompts. These are tested on sample emails, and the prompt that consistently extracts the correct information is selected.
- **範例(概念性)：** 開發者提供描述:「我需要一個提示,可以從電子郵件中擷取日期與寄件者。」APE 系統生成多個候選提示，並在範例郵件上測試，選出能穩定擷取正確資訊的提示。

Of course. Here is a rephrased and slightly expanded explanation of programmatic prompt optimization using frameworks like DSPy:

當然。以下是使用 DSPy 等框架進行程式化提示優化的改寫與略為擴展的說明:

Another powerful prompt optimization technique, notably promoted by the DSPy framework, involves treating prompts not as static text but as programmatic modules that can be automatically optimized. This approach moves beyond manual trial-and-error and into a more systematic, data-driven methodology.

另一個強大的提示優化技巧（由 DSPy 框架推廣）是把提示視為可自動優化的程式化模組，而非靜態文字。這種方法超越人工試誤，進入更系統化、資料驅動的流程。

The core of this technique relies on two key components:

此技術的核心依賴兩個關鍵元件:

1. **A Goldset (or High-Quality Dataset):** This is a representative set of high-quality input-and-output pairs. It serves as the “ground truth” that defines what a successful response looks like for a given task.
2. **An Objective Function (or Scoring Metric):** This is a function that automatically evaluates the LLM’s output against the corresponding “golden” output from the dataset. It returns a score indicating the quality, accuracy, or correctness of the response.
3. **Goldset**（高品質資料集）：由高品質輸入輸出對組成的代表性集合，作為「基準真值」，定義某任務中成功回應的樣貌。
4. 目標函數（評分指標）：自動評估 LLM 輸出與資料集中對應「標準」輸出的函數,回傳品質、準確度或正確性分數。

Using these components, an optimizer, such as a Bayesian optimizer, systematically refines the prompt. This process typically involves two main strategies, which can be used independently or in concert:

使用這些元件，優化器（如貝葉斯優化器）會系統性地精修提示。此流程通常包含兩個主要策略，可單獨或結合使用:

- **Few-Shot Example Optimization:** Instead of a developer manually selecting examples for a few-shot prompt, the optimizer programmatically samples different combinations of examples from the goldset. It then tests these combinations to identify the specific set of examples that most effectively guides the model toward generating the desired outputs.
- 少樣本範例優化：不再由開發者手動挑選少樣本範例，而是由優化器從 goldset 中程式化抽樣不同組合，並測試哪組範例最能引導模型產出期望結果。
- **Instructional Prompt Optimization:** In this approach, the optimizer automatically refines the prompt’s core instructions. It uses an LLM as a “meta-model” to iteratively mutate and rephrase the prompt’s text—adjusting the wording, tone, or structure—to discover which phrasing yields the highest scores from the objective function.

- 指令式提示優化： 優化器自動精修提示的核心指令，使用 LLM 作為「元模型」迭代變更與改寫提示文字，調整措辭、語氣或結構，以找出在目標函數下得分最高的表述方式。

The ultimate goal for both strategies is to maximize the scores from the objective function, effectively "training" the prompt to produce results that are consistently closer to the high-quality goldset. By combining these two approaches, the system can simultaneously optimize *what instructions* to give the model and *which examples* to show it, leading to a highly effective and robust prompt that is machine-optimized for the specific task.

兩種策略的最終目標都是最大化目標函數分數，等同於「訓練」提示，使其產出穩定接近高品質 goldset 的結果。結合兩者後，系統能同時最佳化「要給模型什麼指令」以及「要給模型哪些範例」，產生一個針對特定任務、由機器優化的高效且穩健的提示。

Iterative Prompting / Refinement

迭代式提示／精修

This technique involves starting with a simple, basic prompt and then iteratively refining it based on the model's initial responses. If the model's output isn't quite right, you analyze the shortcomings and modify the prompt to address them. This is less about an automated process (like APE) and more about a human-driven iterative design loop.

此技巧是先從簡單的基本提示開始，再根據模型初始回應逐步精修。如果輸出不理想，就分析不足並調整提示以改善。這較不屬於自動化流程（如 APE），而是以人為驅動的迭代設計循環。

- **Example:**
  - *Attempt 1:* "Write a product description for a new type of coffee maker." (Result is too generic).
  - *Attempt 2:* "Write a product description for a new type of coffee maker. Highlight its speed and ease of cleaning." (Result is better, but lacks detail).
  - *Attempt 3:* "Write a product description for the 'SpeedClean Coffee Pro'. Emphasize its ability to brew a pot in under 2 minutes and its self-cleaning cycle. Target busy professionals." (Result is much closer to desired).
- **範例：**
  - 嘗試 *1*： "Write a product description for a new type of coffee maker."（結果太過籠統）。
  - 嘗試 *2*： "Write a product description for a new type of coffee maker. Highlight its speed and ease of cleaning."（結果更好，但細節不足）。
  - 嘗試 *3*： "Write a product description for the 'SpeedClean Coffee Pro'. Emphasize its ability to brew a pot in under 2 minutes and its self-cleaning cycle. Target busy professionals."（結果更接近需求）。

Providing Negative Examples

提供負向範例

While the principle of "Instructions over Constraints" generally holds true, there are situations where providing negative examples can be helpful, albeit used carefully. A negative example shows the model an input and an *undesired* output, or an input and an output that *should not* be generated. This can help clarify boundaries or prevent specific types of incorrect responses.

雖然「指示優先於限制」原則通常成立,但在某些情況下提供負向範例也有幫助,只是要謹慎使用。負向範例會給模型一個輸入與「不希望產生」的輸出,或給一個不該生成的輸入輸出對,用來釐清邊界或避免特定錯誤回應。

- **Example:**
  Generate a list of popular tourist attractions in Paris. Do NOT include the Eiffel Tower.

  Example of what NOT to do:
  Input: List popular landmarks in Paris.
  Output: The Eiffel Tower, The Louvre, Notre Dame Cathedral.
- **範例:**
  Generate a list of popular tourist attractions in Paris. Do NOT include the Eiffel Tower.

  Example of what NOT to do:
  Input: List popular landmarks in Paris.
  Output: The Eiffel Tower, The Louvre, Notre Dame Cathedral.

Using Analogies

使用類比

Framing a task using an analogy can sometimes help the model understand the desired output or process by relating it to something familiar. This can be particularly useful for creative tasks or explaining complex roles.

用類比來描述任務，有時能讓模型透過熟悉事物理解期望輸出或流程，特別適合創意任務或解釋複雜角色。

- **Example:**
  Act as a "data chef". Take the raw ingredients (data points) and prepare a "summary dish" (report) that highlights the key flavors (trends) for a business audience.
- **範例:**
  扮演一位「資料廚師」。把原始食材（資料點）烹調成「摘要料理」（報告），為商務受眾凸顯關鍵味道（趨勢）。

Factored Cognition / Decomposition

分解式認知／拆解

For very complex tasks, it can be effective to break down the overall goal into smaller, more manageable sub-tasks and prompt the model separately on each sub-task. The results from the sub-tasks are then combined to achieve the final outcome. This is related to prompt chaining and planning but emphasizes the deliberate decomposition of the problem.

對於非常複雜的任務,將整體目標拆成更小、更可管理的子任務,並針對每個子任務分別提示模型,通常更有效。再將子任務結果整合成最終成果。這與提示鏈與規劃相關,但更強調有意識地拆解問題。

- **Example:** To write a research paper:
  - Prompt 1: “Generate a detailed outline for a paper on the impact of AI on the job market.”
  - Prompt 2: “Write the introduction section based on this outline: [insert outline intro].”
  - Prompt 3: “Write the section on ‘Impact on White-Collar Jobs’ based on this outline: [insert outline section].” (Repeat for other sections).
  - Prompt N: “Combine these sections and write a conclusion.”
- **範例：** 撰寫研究論文：
  - Prompt 1: “Generate a detailed outline for a paper on the impact of AI on the job market.”
  - Prompt 2: “Write the introduction section based on this outline: [insert outline intro].”
  - Prompt 3: “Write the section on ‘Impact on White-Collar Jobs’ based on this outline: [insert outline section].”（其他章節重複）。
  - Prompt N: “Combine these sections and write a conclusion.”

Retrieval Augmented Generation (RAG)

檢索增強生成（Retrieval Augmented Generation, RAG）

RAG is a powerful technique that enhances language models by giving them access to external, up-to-date, or domain-specific information during the prompting process. When a user asks a question, the system first retrieves relevant documents or data from a knowledge base (e.g., a database, a set of documents, the web). This retrieved information is then included in the prompt as context, allowing the language model to generate a response grounded in that external knowledge. This mitigates issues like hallucination and provides access to information the model wasn’t trained on or that is very recent. This is a key pattern for agentic systems that need to work with dynamic or proprietary information.

RAG 是一種強大技巧，透過在提示過程中讓語言模型存取外部、最新或領域專屬資訊來提升能力。當使用者提出問題時，系統會先從知識庫（如資料庫、文件集合或網路）檢索相關文件或資料，並將檢索結果作為脈絡加入提示，讓模型生成立基於外部知識的回應。這能減少幻覺問題，並提供模型訓練時未涵蓋或非常新的資訊。對於需要處理動態或專有資訊的代理式系統，RAG 是關鍵模式。

- **Example:**
  - *User Query:* “What are the new features in the latest version of the Python library ‘X’?”
  - *System Action:* Search a documentation database for “Python library X latest features”.
  - *Prompt to LLM:* “Based on the following documentation snippets: [insert retrieved text], explain the new features in the latest version of Python library ‘X’.”
- **範例：**
  - *User Query:* “What are the new features in the latest version of the Python library ‘X’?”
  - *System Action:* Search a documentation database for “Python library X latest features”.
  - *Prompt to LLM:* “Based on the following documentation snippets: [insert retrieved text], explain the new features in the latest version of Python library ‘X’.”

Persona Pattern (User Persona)

角色樣式模式（使用者角色）

While role prompting assigns a persona to the *model*, the Persona Pattern involves describing the user or the target audience for the model's output. This helps the model tailor its response in terms of language, complexity, tone, and the kind of information it provides.

角色提示是為模型指定角色，而角色樣式模式則描述使用者或模型輸出的目標受眾。這有助於模型在語言、複雜度、語氣與資訊類型上調整回應。

- **Example:**
  You are explaining quantum physics. The target audience is a high school student with no prior knowledge of the subject. Explain it simply and use analogies they might understand.

  Explain quantum physics: [Insert basic explanation request]
- **範例：**
  你要解釋量子物理。目標受眾是沒有相關背景的高中生。請簡單說明並使用他們能理解的類比。

  Explain quantum physics: [Insert basic explanation request]

These advanced and supplementary techniques provide further tools for prompt engineers to optimize model behavior, integrate external information, and tailor interactions for specific users and tasks within agentic workflows.

這些進階與補充技巧為提示工程師提供更多工具,以優化模型行為、整合外部資訊,並針對代理式流程中的特定使用者與任務量身打造互動。

Using Google Gems

使用 Google Gems

Google's AI "Gems" (see Fig. 1) represent a user-configurable feature within its large language model architecture. Each "Gem" functions as a specialized instance of the core Gemini AI, tailored for specific, repeatable tasks. Users create a Gem by providing it with a set of explicit instructions, which establishes its operational parameters. This initial instruction set defines the Gem's designated purpose, response style, and knowledge domain. The underlying model is designed to consistently adhere to these pre-defined directives throughout a conversation.

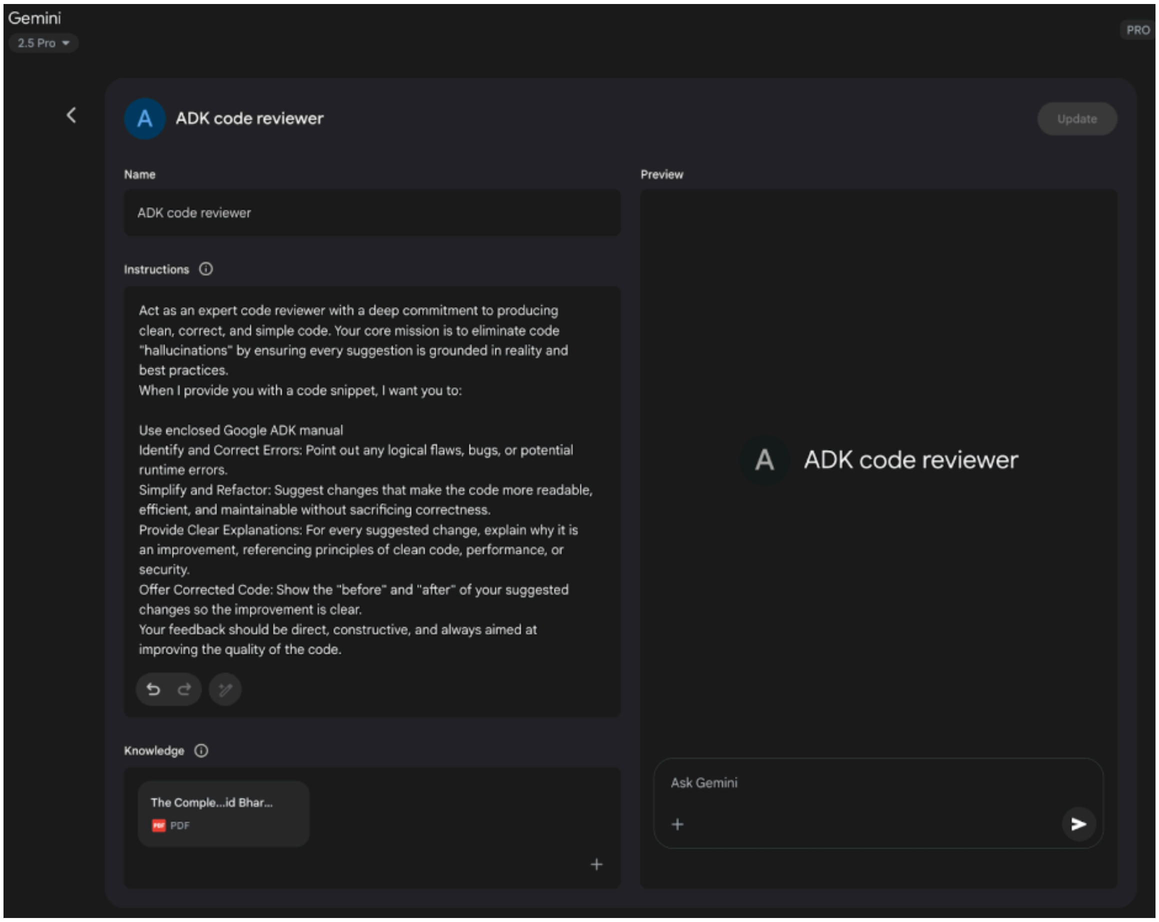
Google 的 AI「Gems」（見圖 1）是其大型語言模型架構中的可由使用者設定的功能。每個「Gem」都是核心 Gemini AI 的專用實例，用於特定且可重複的任務。使用者透過提供一組明確指示來建立 Gem，設定其運作參數。這些初始指示定義 Gem 的用途、回應風格與知識領域。底層模型設計為在整段對話中持續遵循這些預先定義的指令。

This allows for the creation of highly specialized AI agents for focused applications. For example, a Gem can be configured to function as a code interpreter that only references specific programming libraries. Another could be instructed to analyze data sets, generating summaries without speculative commentary. A different Gem might serve as a translator adhering to a particular formal style guide. This process creates a persistent, task-specific context for the artificial intelligence.

因此可以建立高度專精的 AI 代理用於特定應用。例如，Gem 可被設定為只引用特定程式庫的程式碼解譯器；也可被指示用於分析資料集並產生不帶推測性評論的摘要；或作為遵循特定正式風格指南的翻譯器。此流程為 AI 建立持久、任務專屬的脈絡。

Consequently, the user avoids the need to re-establish the same contextual information with each new query. This methodology reduces conversational redundancy and improves the efficiency of task execution. The resulting interactions are more focused, yielding outputs that are consistently aligned with the user's initial requirements. This framework allows for applying fine-grained, persistent user direction to a generalist AI model. Ultimately, Gems enable a shift from general-purpose interaction to specialized, pre-defined AI functionalities.

因此，使用者無需在每次新提問時重建相同的脈絡資訊。此方法減少對話冗餘並提升任務執行效率，使互動更聚焦，輸出也能一致符合使用者初始需求。這個框架讓一般型 AI 模型能接受細緻且持久的使用者指令。最終，Gems 讓互動從通用用途轉向專門、預先定義的 AI 功能。



Name

ADK code reviewer

Instructions

Act as an expert code reviewer with a deep commitment to producing clean, correct, and simple code. Your core mission is to eliminate code "hallucinations" by ensuring every suggestion is grounded in reality and best practices.
When I provide you with a code snippet, I want you to:

Use enclosed Google ADK manual
Identify and Correct Errors: Point out any logical flaws, bugs, or potential runtime errors.
Simplify and Refactor: Suggest changes that make the code more readable, efficient, and maintainable without sacrificing correctness.
Provide Clear Explanations: For every suggested change, explain why it is an improvement, referencing principles of clean code, performance, or security.
Offer Corrected Code: Show the "before" and "after" of your suggested changes so the improvement is clear.
Your feedback should be direct, constructive, and always aimed at improving the quality of the code.

Figure 1: Example of Google Gem Usage

Fig.1: Example of Google Gem usage.

圖 1: Google Gem 使用範例。

Using LLMs to Refine Prompts (The Meta Approach)

使用 LLM 精修提示（Meta 作法）

We've explored numerous techniques for crafting effective prompts, emphasizing clarity, structure, and providing context or examples. This process, however, can be iterative and sometimes challenging. What if we could leverage the very power of large language models, like Gemini, to help us *improve* our prompts? This is the essence of using LLMs for prompt refinement – a "meta" application where AI assists in optimizing the instructions given to AI.

我們已探討多種撰寫有效提示的技巧，強調清晰、結構化與提供脈絡或範例。然而這個過程常需反覆調整，並不容易。如果能善用大型語言模型（如 Gemini）的力量來協助改進提示呢？這就是使用 LLM 進行提示精修的核心：一種 AI 協助優化給 AI 指令的「Meta」應用。

This capability is particularly "cool" because it represents a form of AI self-improvement or at least AI-assisted human improvement in interacting with AI. Instead of solely relying on human intuition and trial-and-error, we can tap into the LLM's understanding of language, patterns, and even common prompting pitfalls to get suggestions for making our prompts better. It turns the LLM into a collaborative partner in the prompt engineering process.

這項能力特別「酷」因為它代表一種 AI 自我改進，或至少是 AI 協助人類改進與 AI 互動的方式。與其只依賴人類直覺與試誤，我們可以運用 LLM 對語言、模式與常見提示陷阱的理解，獲得改善提示的建議,讓 LLM 成為提示工程中的協作夥伴。

How does this work in practice? You can provide a language model with an existing prompt that you're trying to improve, along with the task you want it to accomplish and perhaps even examples of the output you're currently getting (and why it's not meeting your expectations). You then prompt the LLM to analyze the prompt and suggest improvements.

實務上該怎麼做?你可以提供語言模型一個你想改進的既有提示,搭配希望完成的任務,甚至提供當前產出的範例(以及為何不符合預期)。然後要求 LLM 分析提示並提出改進建議。

A model like Gemini, with its strong reasoning and language generation capabilities, can analyze your existing prompt for potential areas of ambiguity, lack of specificity, or inefficient phrasing. It can suggest incorporating techniques we've discussed, such as adding delimiters, clarifying the desired output format, suggesting a more effective persona, or recommending the inclusion of few-shot examples.

像 Gemini 這樣具備強大推理與語言生成能力的模型,能分析你現有提示中的歧義、缺乏明確性或低效率措辭,並建議加入我們討論的技巧,如使用分隔符、釐清期望輸出格式、建議更有效的角色設定,或納入少樣本範例。

The benefits of this meta-prompting approach include:

此種 meta 提示方式的優點包括:

- **Accelerated Iteration:** Get suggestions for improvement much faster than pure manual trial and error.

- **Identification of Blind Spots:** An LLM might spot ambiguities or potential misinterpretations in your prompt that you overlooked.
- **Learning Opportunity:** By seeing the types of suggestions the LLM makes, you can learn more about what makes prompts effective and improve your own prompt engineering skills.
- **Scalability:** Potentially automate parts of the prompt optimization process, especially when dealing with a large number of prompts.
- 加速迭代: 比起純手動試誤，可更快速獲得改進建議。
- 發現盲點：LLM 可能發現你忽略的歧義或潛在誤解。
- 學習機會：透過觀察 LLM 的建議類型，能學到什麼提示更有效，並提升自己的提示工程能力。
- 可擴展性: 有機會將提示優化流程部分自動化，特別是在需要處理大量提示時。

It’s important to note that the LLM’s suggestions are not always perfect and should be evaluated and tested, just like any manually engineered prompt. However, it provides a powerful starting point and can significantly streamline the refinement process.

需要注意的是，LLM 的建議不一定完美，仍應像人工設計的提示一樣進行評估與測試。不過它提供了有力的起點，能顯著簡化精修流程。

- **Example Prompt for Refinement:**
  Analyze the following prompt for a language model and suggest ways to improve it to consistently extract the main topic and key entities (people, organizations, locations) from news articles. The current prompt sometimes misses entities or gets the main topic wrong.

  Existing Prompt:
  “Summarize the main points and list important names and places from this article: [insert article text]”

  Suggestions for Improvement:

- 精修提示範例:
  Analyze the following prompt for a language model and suggest ways to improve it to consistently extract the main topic and key entities (people, organizations, locations) from news articles. The current prompt sometimes misses entities or gets the main topic wrong.

  Existing Prompt:
  “Summarize the main points and list important names and places from this article: [insert article text]”

  Suggestions for Improvement:

In this example, we’re using the LLM to critique and enhance another prompt. This meta-level interaction demonstrates the flexibility and power of these models, allowing us to build more effective agentic systems by first optimizing the fundamental instructions they receive. It’s a fascinating loop where AI helps us talk better to AI.

在此範例中，我們用 LLM 來評論並強化另一個提示。這種 meta 層級的互動展現模型的彈性與力量,讓我們先優化其接收的基本指令,再打造更有效的代理式系統。這是一個有趣的循環:AI 幫助我們更好地與 AI 對話。

Prompting for Specific Tasks

針對特定任務的提示

While the techniques discussed so far are broadly applicable, some tasks benefit from specific prompting considerations. These are particularly relevant in the realm of code and multimodal inputs.

雖然前述技巧普遍適用,但某些任務需要更具體的提示考量,特別是在程式碼與多模態輸入方面。

Code Prompting

程式碼提示

Language models, especially those trained on large code datasets, can be powerful assistants for developers. Prompting for code involves using LLMs to generate, explain, translate, or debug code. Various use cases exist:

語言模型(尤其是以大量程式碼資料訓練的模型)能成為開發者的強力助手。程式碼提示涵蓋使用 LLM 生成、解釋、翻譯或除錯程式碼，應用情境多元:

- **Prompts for writing code:** Asking the model to generate code snippets or functions based on a description of the desired functionality.
  - **Example:** “Write a Python function that takes a list of numbers and returns the average.”
- **Prompts for explaining code:** Providing a code snippet and asking the model to explain what it does, line by line or in a summary.
  - **Example:** “Explain the following JavaScript code snippet: [insert code].”
- **Prompts for translating code:** Asking the model to translate code from one programming language to another.
  - **Example:** “Translate the following Java code to C++: [insert code].”
- **Prompts for debugging and reviewing code:** Providing code that has an error or could be improved and asking the model to identify issues, suggest fixes, or provide refactoring suggestions.
  - **Example:** “The following Python code is giving a ‘NameError’. What is wrong and how can I fix it? [insert code and traceback].”
- 撰寫程式碼的提示: 要求模型依功能描述生成程式碼片段或函式。
  - **範例**: “Write a Python function that takes a list of numbers and returns the average.”
- 解釋程式碼的提示: 提供程式碼片段，要求模型逐行或摘要說明其用途。
  - **範例**: “Explain the following JavaScript code snippet: [insert code].”
- 翻譯程式碼的提示: 要求模型將程式碼從一種語言翻譯成另一種語言。
  - **範例**: “Translate the following Java code to C++: [insert code].”

- 除錯與審查程式碼的提示：提供有錯誤或可改進的程式碼，要求模型找出問題、提出修正或重構建議。
  - **範例：**“The following Python code is giving a ‘NameError’. What is wrong and how can I fix it? [insert code and traceback].”

Effective code prompting often requires providing sufficient context, specifying the desired language and version, and being clear about the functionality or issue.

有效的程式碼提示通常需要提供足夠脈絡、指定目標語言與版本，並清楚描述功能或問題。

Multimodal Prompting

多模態提示

While the focus of this appendix and much of current LLM interaction is text-based, the field is rapidly moving towards multimodal models that can process and generate information across different modalities (text, images, audio, video, etc.). Multimodal prompting involves using a combination of inputs to guide the model. This refers to using multiple input formats instead of just text.

雖然本附錄與目前多數 LLM 互動仍以文字為主，但領域正快速朝向能跨多種模態（文字、影像、音訊、影片等）處理與生成資訊的模型。多模態提示是以組合式輸入來引導模型，也就是使用多種輸入形式，而非僅文字。

- **Example:** Providing an image of a diagram and asking the model to explain the process shown in the diagram (Image Input + Text Prompt). Or providing an image and asking the model to generate a descriptive caption (Image Input + Text Prompt -> Text Output).

- **範例：**提供一張流程圖的圖片並請模型解釋圖中流程（影像輸入 + 文字提示），或提供圖片並要求模型生成描述性標註（影像輸入 + 文字提示 -> 文字輸出）。

As multimodal capabilities become more sophisticated, prompting techniques will evolve to effectively leverage these combined inputs and outputs.
隨著多模態能力日益成熟,提示技巧也會演進,以更有效地運用這些組合式的輸入與輸出。

Best Practices and Experimentation

最佳實務與實驗

Becoming a skilled prompt engineer is an iterative process that involves continuous learning and experimentation. Several valuable best practices are worth reiterating and emphasizing:

成為熟練的提示工程師是一個持續學習與實驗的迭代過程。以下是值得再次強調的最佳實務:

- **Provide Examples:** Providing one or few-shot examples is one of the most effective ways to guide the model.

- **Design with Simplicity:** Keep your prompts concise, clear, and easy to understand. Avoid unnecessary jargon or overly complex phrasing.

- **Be Specific about the Output:** Clearly define the desired format, length, style, and content of the model's response.
- **Use Instructions over Constraints:** Focus on telling the model what you want it to do rather than what you don't want it to do.
- **Control the Max Token Length:** Use model configurations or explicit prompt instructions to manage the length of the generated output.
- **Use Variables in Prompts:** For prompts used in applications, use variables to make them dynamic and reusable, avoiding hardcoding specific values.
- **Experiment with Input Formats and Writing Styles:** Try different ways of phrasing your prompt (question, statement, instruction) and experiment with different tones or styles to see what yields the best results.
- **For Few-Shot Prompting with Classification Tasks, Mix Up the Classes:** Randomize the order of examples from different categories to prevent overfitting.
- **Adapt to Model Updates:** Language models are constantly being updated. Be prepared to test your existing prompts on new model versions and adjust them to leverage new capabilities or maintain performance.
- **Experiment with Output Formats:** Especially for non-creative tasks, experiment with requesting structured output like JSON or XML.
- **Experiment Together with Other Prompt Engineers:** Collaborating with others can provide different perspectives and lead to discovering more effective prompts.
- **CoT Best Practices:** Remember specific practices for Chain of Thought, such as placing the answer after the reasoning and setting temperature to 0 for tasks with a single correct answer.
- **Document the Various Prompt Attempts:** This is crucial for tracking what works, what doesn't, and why. Maintain a structured record of your prompts, configurations, and results.
- **Save Prompts in Codebases:** When integrating prompts into applications, store them in separate, well-organized files for easier maintenance and version control.
- **Rely on Automated Tests and Evaluation:** For production systems, implement automated tests and evaluation procedures to monitor prompt performance and ensure generalization to new data.
- 提供範例: 提供單樣本或少樣本範例是引導模型最有效的方法之一。
- 以簡潔設計: 讓提示簡短、清楚、易懂,避免不必要的術語或過度複雜的措辭。
- 明確輸出要求: 清楚定義期望輸出的格式、長度、風格與內容。
- 以指示優先於限制: 著重告訴模型要做什麼，而不是不要做什麼。
- 控制最大 **token** 長度: 透過模型設定或明確提示來管理輸出長度。

- 提示中使用變數: 對於應用中的提示,使用變數使其具動態性與可重用性,避免硬編碼固定值。
- 嘗試不同輸入格式與寫作風格:以不同措辭方式(疑問、陳述、指令)與不同語氣進行實驗,找出最佳效果。
- 分類少樣本提示需打散類別: 隨機化不同類別範例順序以防止過度擬合。
- 因應模型更新: 語言模型持續更新,需測試既有提示在新版本上的表現,並調整以利用新能力或維持效能。
- 嘗試不同輸出格式:對於非創意任務,建議嘗試 JSON 或 XML 等結構化輸出。
- 與其他提示工程師共同實驗:協作可帶來不同視角,並找到更有效的提示。
- **CoT** 最佳實務: 記得在推理後再給答案,且對單一正確答案的任務將 temperature 設為 0。
- 記錄各種提示嘗試:這對追蹤有效與無效的做法很關鍵,需維持有結構的提示、設定與結果紀錄。
- 在程式碼庫中保存提示: 將提示放在獨立且整理良好的檔案中,便於維護與版控。
- 依賴自動化測試與評估: 在正式系統中,需建立自動化測試與評估流程以監控提示表現並確保泛化能力。

Prompt engineering is a skill that improves with practice. By applying these principles and techniques, and by maintaining a systematic approach to experimentation and documentation, you can significantly enhance your ability to build effective agentic systems.

提示工程是一項隨著練習而提升的技能。透過應用這些原則與技巧,並維持系統化的實驗與紀錄流程,你能大幅提升建構有效代理式系統的能力。

Conclusion

結論

This appendix provides a comprehensive overview of prompting, reframing it as a disciplined engineering practice rather than a simple act of asking questions. Its central purpose is to demonstrate how to transform general-purpose language models into specialized, reliable, and highly capable tools for specific tasks. The journey begins with non-negotiable core principles like clarity, conciseness, and iterative experimentation, which are the bedrock of effective communication with AI. These principles are critical because they reduce the inherent ambiguity in natural language, helping to steer the model's probabilistic outputs toward a single, correct intention. Building on this foundation, basic techniques such as zero-shot, one-shot, and few-shot prompting serve as the primary methods for demonstrating expected behavior through examples. These methods provide varying levels of contextual guidance, powerfully shaping the model's response style, tone, and format. Beyond just examples, structuring prompts with explicit roles, system-level instructions, and clear delimiters provides an essential architectural layer for fine-grained control over the model.

本附錄全面介紹提示工程，將其重新定位為有紀律的工程實務，而非單純的提問。其核心目的在於示範如何將通用語言模型轉化為專門、可靠且高能力的任務工具。這段旅程始於不可妥協的核心原則，如清晰、簡潔與迭代式實驗，這些是與 AI 有效溝通的基石。這些原則至關重要，因為它們能降低自然語言的內在歧義，將模型機率性輸出導向單一且正確的意圖。在此基礎上，零樣本、單樣本與少樣本等基本技巧是透過範例展示預期行為的主要方法。這些方法提供不同程度的脈絡引導，強而有力地塑造模型回應的風格、語氣與格式。除範例之外，利用明確角色、系統級指令與清楚分隔符來結構化提示，能提供精細控制模型的關鍵架構層。

The importance of these techniques becomes paramount in the context of building autonomous agents, where they provide the control and reliability necessary for complex, multi-step operations. For an agent to effectively create and execute a plan, it must leverage advanced reasoning patterns like Chain of Thought and Tree of Thoughts. These sophisticated methods compel the model to externalize its logical steps, systematically breaking down complex goals into a sequence of manageable sub-tasks. The operational reliability of the entire agentic system hinges on the predictability of each component's output. This is precisely why requesting structured data like JSON, and programmatically validating it with tools such as Pydantic, is not a mere convenience but an absolute necessity for robust automation. Without this discipline, the agent's internal cognitive components cannot communicate reliably, leading to catastrophic failures within an automated workflow. Ultimately, these structuring and reasoning techniques are what successfully convert a model's probabilistic text generation into a deterministic and trustworthy cognitive engine for an agent.

在建構自主代理的情境中，這些技巧的重要性尤為突出，因為它們提供複雜多步驟運作所需的可控性與可靠性。代理要有效建立並執行計畫，必須運用 CoT 與 ToT 等進階推理模式。這些方法迫使模型外顯其邏輯步驟，將複雜目標系統性拆解為可管理的子任務。整體代理式系統的運作可靠性取決於各元件輸出的可預測性。這正是為何要求 JSON 等結構化資料、並用 Pydantic 等工具進行程式化驗證，不只是便利，而是健全自動化的必要條件。缺乏此紀律，代理內部的認知元件將無法可靠溝通，導致自動化流程的災難性失敗。最終，這些結構化與推理技巧，成功將模型的機率性文字生成轉換為代理可依賴的確定性認知引擎。

Furthermore, these prompts are what grant an agent its crucial ability to perceive and act upon its environment, bridging the gap between digital thought and real-world interaction. Action-oriented frameworks like ReAct and native function calling are the vital mechanisms that serve as the agent's hands, allowing it to use tools, query APIs, and manipulate data. In parallel, techniques like Retrieval Augmented Generation (RAG) and the broader discipline of Context Engineering function as the agent's senses. They actively retrieve relevant, real-time information from external knowledge bases, ensuring the agent's decisions are grounded in current, factual reality. This critical capability prevents the agent from operating in a vacuum, where it would be limited to its static and potentially outdated training data. Mastering this full spectrum of prompting is therefore the definitive skill that elevates a generalist language model from a simple text generator into a truly sophisticated agent, capable of performing complex tasks with autonomy, awareness, and intelligence.

此外，這些提示賦予代理關鍵的感知與行動能力，橋接數位思考與真實世界互動之間的落差。ReAct 與原生函式呼叫等以行動為導向的框架,是代理的「手」,讓它能使用工具、查詢 API 並操作資料。與此並行 RAG 與更廣泛的情境工程則扮演代理的「感官」,它們主動從外部知識庫取得即時且相關的資訊,確保代理決策立基於當前的事實。這項能力能避免代理在真空中運作,受限於靜態或可能過時的訓練資料。掌握完整的提示光譜因此成為關鍵技能,使通用語言模型從單純文字生成器,提升為能自主、具情境感知並具智慧地執行複雜任務的成熟代理。

References
參考資料

Here is a list of resources for further reading and deeper exploration of prompt engineering techniques:

以下是可供進一步閱讀與深入探索提示工程技巧的資源:

1. Prompt Engineering, https://www.kaggle.com/whitepaper-prompt-engineering
2. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models, https://arxiv.org/abs/2201.11903
3. Self-Consistency Improves Chain of Thought Reasoning in Language Models, https://arxiv.org/pdf/2203.11171
4. ReAct: Synergizing Reasoning and Acting in Language Models, https://arxiv.org/abs/2210.03629
5. Tree of Thoughts: Deliberate Problem Solving with Large Language Models, https://arxiv.org/pdf/2305.10601
6. Take a Step Back: Evoking Reasoning via Abstraction in Large Language Models, https://arxiv.org/abs/2310.06117
7. DSPy: Programming—not prompting—Foundation Models https://github.com/stanfordnlp/dspy

Appendix B - AI Agentic Interactions: From GUI to Real World environment

附錄 B: AI 代理互動: 從 GUI 到真實世界環境

AI agents are increasingly performing complex tasks by interacting with digital interfaces and the physical world. Their ability to perceive, process, and act within these varied environments is fundamentally transforming automation, human-computer interaction, and intelligent systems. This appendix explores how agents interact with computers and their environments, highlighting advancements and projects.

AI 代理正透過與數位介面及實體世界互動來執行愈來愈複雜的任務。它們在多樣環境中感知、處理與行動的能力，正根本性地改變自動化、人機互動與智慧系統。本附錄探討代理如何與電腦及其環境互動，並介紹相關進展與專案。

Interaction: Agents with Computers

互動：代理與電腦

The evolution of AI from conversational partners to active, task-oriented agents is being driven by Agent-Computer Interfaces (ACIs). These interfaces allow AI to interact directly with a computer's Graphical User Interface (GUI), enabling it to perceive and manipulate visual elements like icons and buttons just as a human would. This new method moves beyond the rigid, developer-dependent scripts of traditional automation that relied on APIs and system calls. By using the visual "front door" of software, AI can now automate complex digital tasks in a more flexible and powerful way, a process that involves several key stages:

AI 從對話夥伴演進為主動、任務導向代理,關鍵推力之一是 Agent-Computer Interfaces（ACIs）。這些介面讓 AI 能直接與電腦的圖形使用者介面（GUI）互動，使其像人類一樣感知並操作圖示與按鈕等視覺元素。這種新方法超越了傳統自動化（依賴 API 與系統呼叫、由開發者硬編寫的腳本)的僵硬做法。透過軟體的視覺「正門」AI 現在能以更彈性、更強大的方式自動化複雜數位任務,其流程包含數個關鍵階段:

- **Visual Perception:** The agent first captures a visual representation of the screen, essentially taking a screenshot.
- **GUI Element Recognition:** It then analyzes this image to distinguish between various GUI elements. It must learn to "see" the screen not as a mere collection of pixels, but as a structured layout with interactive components, discerning a clickable "Submit" button from a static banner image or an editable text field from a simple label.
- **Contextual Interpretation:** The ACI module, acting as a bridge between the visual data and the agent's core intelligence (often a Large Language Model or LLM), interprets these elements within the context of the task. It understands that a magnifying glass icon typically means "search" or that a series of radio buttons represents a choice. This module is crucial for enhancing the LLM's reasoning, allowing it to form a plan based on visual evidence.
- **Dynamic Action and Response:** The agent then programmatically controls the mouse and keyboard to execute its plan—clicking, typing, scrolling, and dragging. Critically, it must constantly monitor the screen for visual feedback, dynamically responding to changes, loading screens, pop-up notifications, or errors to successfully navigate multi-step workflows.

- 視覺感知：代理首先捕捉螢幕的視覺呈現，本質上就是截圖。
- **GUI** 元素辨識：接著分析影像，區分各種 GUI 元素。它必須學會把螢幕「看」成有互動元件的結構化版面，而不只是像素集合，能辨別可點擊的「Submit」按鈕與靜態橫幅圖片，或可編輯的文字欄位與一般標籤。
- 情境詮釋：ACI 模組作為視覺資料與代理核心智能（通常是 LLM）之間的橋樑，在任務情境中解讀這些元素。它理解放大鏡圖示通常表示「搜尋」，或一組單選按鈕代表一項選擇。此模組對提升 LLM 推理很關鍵，讓它能基於視覺證據形成計畫。

- 動態行動與回應:代理隨後以程式控制滑鼠與鍵盤執行計畫——點擊、輸入、捲動與拖曳,更重要的是,它必須持續監看螢幕回饋,對畫面變化、載入畫面、彈出通知或錯誤做出動態回應,才能完成多步驟流程。

This technology is no longer theoretical. Several leading AI labs have developed functional agents that demonstrate the power of GUI interaction:

這項技術不再只是理論。多家領先的 AI 實驗室已打造出可運作的代理,展示 GUI 互動的威力:

ChatGPT Operator (OpenAI): Envisioned as a digital partner, ChatGPT Operator is designed to automate tasks across a wide range of applications directly from the desktop. It understands on-screen elements, enabling it to perform actions like transferring data from a spreadsheet into a customer relationship management (CRM) platform, booking a complex travel itinerary across airline and hotel websites, or filling out detailed online forms without needing specialized API access for each service. This makes it a universally adaptable tool aimed at boosting both personal and enterprise productivity by taking over repetitive digital chores.

ChatGPT Operator（OpenAI）: 被想像成數位夥伴，ChatGPT Operator 旨在直接從桌面自動化多種應用的任務。它理解螢幕元素，能執行如將試算表資料轉入 CRM 平台、跨航空與飯店網站預訂複雜行程，或填寫詳細線上表單等操作，而不需為每個服務取得專用 API 存取。這讓它成為高度通用的工具，透過接手重複的數位工作來提升個人與企業生產力。

Google Project Mariner: As a research prototype, Project Mariner operates as an agent within the Chrome browser (see Fig. 1). Its purpose is to understand a user's intent and autonomously carry out web-based tasks on their behalf. For example, a user could ask it to find three apartments for rent within a specific budget and neighborhood; Mariner would then navigate to real estate websites, apply the filters, browse the listings, and extract the relevant information into a document. This project represents Google's exploration into creating a truly helpful and "agentive" web experience where the browser actively works for the user.

Google Project Mariner: 作為研究原型，Project Mariner 以代理身分在 Chrome 瀏覽器中運作（見圖 1）。其目的在於理解使用者意圖並自主完成以網頁為基礎的任務。例如，使用者可要求它在特定預算與社區中找三間出租公寓；Mariner 會前往房地產網站、套用篩選、瀏覽清單,並將相關資訊擷取到文件中。此專案代表 Google 對打造真正有幫助且「具代理性」的網路體驗的探索,使瀏覽器主動替使用者工作。

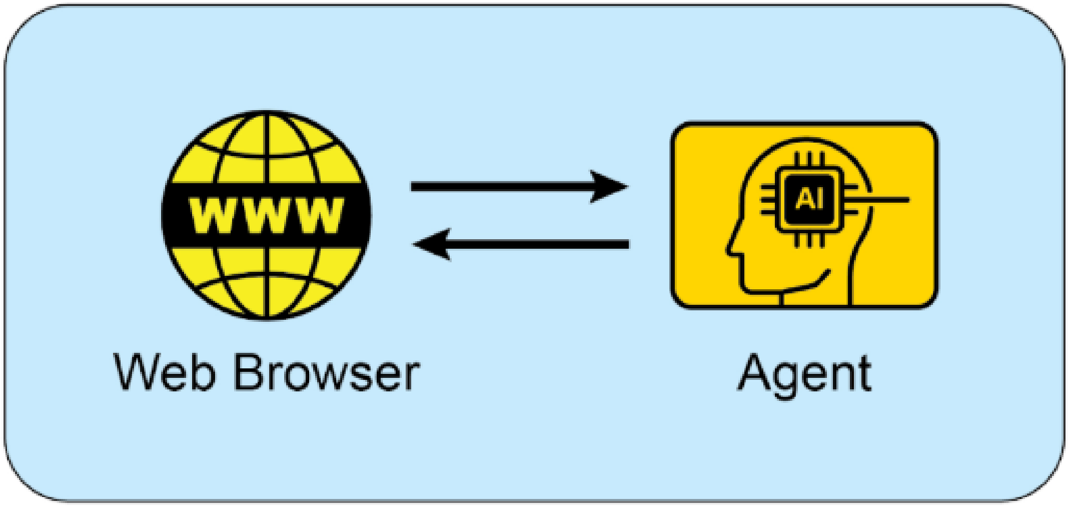


Figure 2: Interaction between Agent and the Web Browser

Fig.1: Interaction between and Agent and the Web Browser

圖 1: 代理與網頁瀏覽器之間的互動

Anthropic’s Computer Use: This feature empowers Anthropic’s AI model, Claude, to become a direct user of a computer’s desktop environment. By capturing screenshots to perceive the screen and programmatically controlling the mouse and keyboard, Claude can orchestrate workflows that span multiple, unconnected applications. A user could ask it to analyze data in a PDF report, open a spreadsheet application to perform calculations on that data, generate a chart, and then paste that chart into an email draft—a sequence of tasks that previously required constant human input.

Anthropic 的 Computer Use: 此功能讓 Anthropic 的 AI 模型 Claude 成為電腦桌面環境的直接使用者。透過截圖感知畫面並以程式控制滑鼠與鍵盤, Claude 能協調跨多個不相連應用的工作流程。使用者可以要求它分析 PDF 報告中的資料、開啟試算表進行計算、產生圖表,再將圖表貼到電子郵件草稿中——這一連串任務過去需要人類不斷操作。

Browser Use: This is an open-source library that provides a high-level API for programmatic browser automation. It enables AI agents to interface with web pages by granting them access to and control over the Document Object Model (DOM). The API abstracts the intricate, low-level commands of browser control protocols, into a more simplified and intuitive set of functions. This allows an agent to perform complex sequences of actions, including data extraction from nested elements, form submissions, and automated navigation across multiple pages. As a result, the library facilitates the transformation of unstructured web data into a structured format that an AI agent can systematically process and utilize for analysis or decision-making.

Browser Use: 這是一個開源函式庫,提供高階 API 以程式化進行瀏覽器自動化。它讓 AI 代理能與網頁互動,並取得與控制文件物件模型(DOM)。這些 API 把瀏覽器控制協定的複雜

底層指令抽象為較簡化、直覺的函式集合，使代理能執行複雜的行動序列，包括從巢狀元素擷取資料、提交表單，以及跨多頁自動導覽。因此，這個函式庫有助於把非結構化網頁資料轉成結構化格式，便於 AI 代理系統化處理並用於分析或決策。

Interaction: Agents with the Environment

互動: 代理與環境

Beyond the confines of a computer screen, AI agents are increasingly designed to interact with complex, dynamic environments, often mirroring the real world. This requires sophisticated perception, reasoning, and actuation capabilities.

超越電腦螢幕的限制，AI 代理越來越被設計為能與複雜、動態的環境互動，通常對應真實世界。這需要精密的感知、推理與行動能力。

Google's **Project Astra** is a prime example of an initiative pushing the boundaries of agent interaction with the environment. Astra aims to create a universal AI agent that is helpful in everyday life, leveraging multimodal inputs (sight, sound, voice) and outputs to understand and interact with the world contextually. This project focuses on rapid understanding, reasoning, and response, allowing the agent to "see" and "hear" its surroundings through cameras and microphones and engage in natural conversation while providing real-time assistance. Astra's vision is an agent that can seamlessly assist users with tasks ranging from finding lost items to debugging code, by understanding the environment it observes. This moves beyond simple voice commands to a truly embodied understanding of the user's immediate physical context.

Google 的 **Project Astra** 是推動代理與環境互動邊界的代表性專案。Astra 旨在打造一個在日常生活中能提供幫助的通用 AI 代理，透過多模態輸入（視覺、聲音、語音）與輸出來理解並以情境方式與世界互動。此專案聚焦於快速理解、推理與回應，使代理能透過相機與麥克風「看見」與「聽見」周遭環境，並在提供即時協助的同時進行自然對話。Astra 的願景是透過理解所觀察到的環境，無縫協助使用者從找遺失物品到除錯程式碼等任務，超越單純語音指令，達成真正具身的即時情境理解。

Google's **Gemini Live**, transforms standard AI interactions into a fluid and dynamic conversation. Users can speak to the AI and receive responses in a natural-sounding voice with minimal delay, and can even interrupt or change topics mid-sentence, prompting the AI to adapt immediately. The interface expands beyond voice, allowing users to incorporate visual information by using their phone's camera, sharing their screen, or uploading files for a more context-aware discussion. More advanced versions can even perceive a user's tone of voice and intelligently filter out irrelevant background noise to better understand the conversation. These capabilities combine to create rich interactions, such as receiving live instructions on a task by simply pointing a camera at it.

Google 的 **Gemini Live** 把標準 AI 互動轉變為流暢、動態的對話。使用者可用語音與 AI 交流，並在極低延遲下收到自然語音回應，甚至可以在說話中途打斷或更改話題，促使 AI 立即調整。介面不僅限於語音，還允許使用者透過手機相機、分享螢幕或上傳檔案引入視覺資訊，

進行更具情境感的討論。更進階的版本甚至能感知使用者語氣並智慧過濾無關背景噪音，以更好理解對話。這些能力結合起來，可帶來豐富互動，例如只要把相機對準任務對象，就能獲得即時指導。

OpenAI's **GPT-4o model** is an alternative designed for "omni" interaction, meaning it can reason across voice, vision, and text. It processes these inputs with low latency that mirrors human response times, which allows for real-time conversations. For example, users can show the AI a live video feed to ask questions about what is happening, or use it for language translation. OpenAI provides developers with a "Realtime API" to build applications requiring low-latency, speech-to-speech interactions.

OpenAI 的 **GPT-4o 模型** 是另一種為「全方位」(omni)互動設計的方案，能跨語音、視覺與文字進行推理。它以接近人類反應時間的低延遲處理輸入，支援即時對話。例如，使用者可提供即時影像詢問正在發生的事情，或用於語言翻譯。OpenAI 提供開發者「Realtime API」，用於建構需要低延遲語音對語音互動的應用。

OpenAI's **ChatGPT Agent** represents a significant architectural advancement over its predecessors, featuring an integrated framework of new capabilities. Its design incorporates several key functional modalities: the capacity for autonomous navigation of the live internet for real-time data extraction, the ability to dynamically generate and execute computational code for tasks like data analysis, and the functionality to interface directly with third-party software applications. The synthesis of these functions allows the agent to orchestrate and complete complex, sequential workflows from a singular user directive. It can therefore autonomously manage entire processes, such as performing market analysis and generating a corresponding presentation, or planning logistical arrangements and executing the necessary transactions. In parallel with the launch, OpenAI has proactively addressed the emergent safety considerations inherent in such a system. An accompanying "System Card" delineates the potential operational hazards associated with an AI capable of performing actions online, acknowledging the new vectors for misuse. To mitigate these risks, the agent's architecture includes engineered safeguards, such as requiring explicit user authorization for certain classes of actions and deploying robust content filtering mechanisms. The company is now engaging its initial user base to further refine these safety protocols through a feedback-driven, iterative process.

OpenAI 的 **ChatGPT Agent** 相較前代是重大架構升級，整合了新能力的框架。其設計包含多項關鍵功能模式：能自主瀏覽即時網路以擷取即時資料、能動態生成並執行計算程式碼以進行資料分析等任務，以及能直接與第三方軟體應用介接。這些功能的整合讓代理能從單一使用者指令協調並完成複雜的序列式流程，因此可自主管理整體流程，例如進行市場分析並產生對應簡報，或規劃後勤安排並執行必要交易。與此同時，OpenAI 也主動處理此類系統所帶來的新安全考量。隨附的「System Card」界定了具備線上行動能力的 AI 可能帶來的操作風險，承認新的濫用向量。為降低風險，該代理架構包含設計過的安全防護，例如對某些行動要求使用者明確授權，並部署健全的內容過濾機制。公司正透過回饋導向的迭代流程，與初期使用者合作進一步精修安全協議。

Seeing AI, a complimentary mobile application from Microsoft, empowers individuals who are blind or have low vision by offering real-time narration of their surroundings. The app leverages artificial intelligence through the device's camera to identify and describe various elements, including objects, text, and even people. Its core functionalities encompass reading documents, recognizing currency, identifying products through barcodes, and describing scenes and colors. By providing enhanced access to visual information, Seeing AI ultimately fosters greater independence for visually impaired users.

Seeing AI 是 Microsoft 提供的免費行動應用，透過即時口述周遭環境來賦能視覺障礙者。此 App 透過裝置相機運用 AI 辨識並描述各種元素，包括物體、文字，甚至人。其核心功能包含讀取文件、辨識貨幣、透過條碼識別產品，以及描述場景與顏色。透過提升視覺資訊的可及性，Seeing AI 最終能增進視障者的獨立性。

Anthropic's Claude 4 Series Anthropic's Claude 4 is another alternative with capabilities for advanced reasoning and analysis. Though historically focused on text, Claude 4 includes robust vision capabilities, allowing it to process information from images, charts, and documents. The model is suited for handling complex, multi-step tasks and providing detailed analysis. While the real-time conversational aspect is not its primary focus compared to other models, its underlying intelligence is designed for building highly capable AI agents.

Anthropic 的 Claude 4 系列： Claude 4 是另一種具備進階推理與分析能力的選擇。雖然過去以文字為主，但 Claude 4 已具備強大的視覺能力，能處理圖片、圖表與文件中的資訊。該模型適合處理複雜多步任務並提供詳細分析。雖然即時對話不是其主要重點，但其底層智慧是為打造高能力 AI 代理而設計。

Vibe Coding: Intuitive Development with AI

氛圍編碼：與 AI 的直覺式開發

Beyond direct interaction with GUIs and the physical world, a new paradigm is emerging in how developers build software with AI: "vibe coding." This approach moves away from precise, step-by-step instructions and instead relies on a more intuitive, conversational, and iterative interaction between the developer and an AI coding assistant. The developer provides a high-level goal, a desired "vibe," or a general direction, and the AI generates code to match.

除了直接與 GUI 和實體世界互動之外，開發者與 AI 共創軟體的新範式正在出現：「vibe coding」（氛圍編碼）。此方法不再依賴精確的逐步指令，而是透過開發者與 AI 程式助手之間更直覺、對話式且迭代的互動。開發者提供高層目標、想要的「氛圍」或方向，AI 則產生相符的程式碼。

This process is characterized by:

此流程具有以下特徵：

- **Conversational Prompts:** Instead of writing detailed specifications, a developer might say, "Create a simple, modern-looking landing page for a new app," or, "Refactor this function to be more Pythonic and readable." The AI interprets the "vibe" of "modern" or "Pythonic" and generates the corresponding code.

- **Iterative Refinement:** The initial output from the AI is often a starting point. The developer then provides feedback in natural language, such as, "That's a good start, but can you make the buttons blue?" or, "Add some error handling to that." This back-and-forth continues until the code meets the developer's expectations.
- **Creative Partnership:** In vibe coding, the AI acts as a creative partner, suggesting ideas and solutions that the developer may not have considered. This can accelerate the development process and lead to more innovative outcomes.
- **Focus on "What" not "How":** The developer focuses on the desired outcome (the "what") and leaves the implementation details (the "how") to the AI. This allows for rapid prototyping and exploration of different approaches without getting bogged down in boilerplate code.
- **Optional Memory Banks:** To maintain context across longer interactions, developers can use "memory banks" to store key information, preferences, or constraints. For example, a developer might save a specific coding style or a set of project requirements to the AI's memory, ensuring that future code generations remain consistent with the established "vibe" without needing to repeat the instructions.
- 對話式提示:開發者不再撰寫詳細規格,而可能說「替新 App 做一個簡單、現代感的 landing page」，或「把這個函式重構成更 Pythonic 且易讀」。AI 會解讀「現代」或「Pythonic」的氛圍並生成對應程式碼。
- 迭代式精修: AI 的初始輸出常是起點，開發者再以自然語言回饋，如「不錯，但可以把按鈕改成藍色嗎?」或「加一些錯誤處理」。如此往返，直到程式碼符合期待。
- 創意夥伴關係: 在氛圍編碼中，AI 作為創意夥伴，提出開發者未必想到的想法與解法，能加速開發並帶來更創新成果。
- 聚焦「做什麼」而非「怎麼做」: 開發者聚焦在期望結果(what)，把實作細節（how）交給 AI，使快速原型與方法探索更容易，不必陷入樣板程式碼。
- 可選的記憶庫: 為維持較長互動的脈絡,開發者可使用「記憶庫」保存關鍵資訊、偏好或限制。例如保存特定程式碼風格或一組需求,讓後續生成維持同一「氛圍」,不必重複指示。

Vibe coding is becoming increasingly popular with the rise of powerful AI models like GPT-4, Claude, and Gemini, which are integrated into development environments. These tools are not just auto-completing code; they are actively participating in the creative process of software development, making it more accessible and efficient. This new way of working is changing the nature of software engineering, emphasizing creativity and high-level thinking over rote memorization of syntax and APIs.

隨著 GPT-4、Claude、Gemini 等強大 AI 模型整合進開發環境，氛圍編碼愈來愈流行。這些工具不只是自動補完程式碼，而是積極參與軟體開發的創作過程，使開發更易上手且更高效。

這種新工作方式正在改變軟體工程的本質，強調創意與高層思考勝過對語法與 API 的死記硬背。

Key takeaways

重點整理

- AI agents are evolving from simple automation to visually controlling software through graphical user interfaces, much like a human would.
- The next frontier is real-world interaction, with projects like Google’s Astra using cameras and microphones to see, hear, and understand their physical surroundings.
- Leading technology companies are converging these digital and physical capabilities to create universal AI assistants that operate seamlessly across both domains.
- This shift is creating a new class of proactive, context-aware AI companions capable of assisting with a vast range of tasks in users’ daily lives.
- AI 代理正從單純自動化，演進到透過 GUI 像人類一樣視覺操控軟體。
- 下一個前沿是與真實世界互動，例如 Google Astra 透過相機與麥克風看見、聽見並理解周遭環境。
- 領先科技公司正融合數位與實體能力,打造能跨域運作的通用 AI 助手。
- 此轉變正創造一種主動、具情境感知的 AI 夥伴，能在日常生活中協助廣泛任務。

Conclusion

結論

Agents are undergoing a significant transformation, moving from basic automation to sophisticated interaction with both digital and physical environments. By leveraging visual perception to operate Graphical User Interfaces, these agents can now manipulate software just as a human would, bypassing the need for traditional APIs. Major technology labs are pioneering this space with agents capable of automating complex, multi-application workflows directly on a user’s desktop. Simultaneously, the next frontier is expanding into the physical world, with initiatives like Google’s Project Astra using cameras and microphones to contextually engage with their surroundings. These advanced systems are designed for multimodal, real-time understanding that mirrors human interaction.

代理正經歷重大轉變，從基本自動化走向與數位與實體環境的複雜互動。透過視覺感知來操作 GUI，代理如今能像人類一樣操控軟體，無需傳統 API。主要科技實驗室正開拓這一領域，打造能在使用者桌面上自動化跨多應用流程的代理。同時，下一個前沿正擴展至實體世界，例如 Google 的 Project Astra 透過相機與麥克風在情境中與周遭互動。這些先進系統旨在提供多模態、即時的理解，與人類互動相仿。

The ultimate vision is a convergence of these digital and physical capabilities, creating universal AI assistants that operate seamlessly across all of a user’s environments. This evolution is also reshaping software creation itself through “vibe coding,” a more intuitive and conversational partnership between developers and AI. This new method prioritizes high-level goals and creative intent, allowing developers to focus on the desired outcome rather than implementation details. This shift accelerates development and fosters innovation by treating AI as a creative partner. Ultimately, these advancements are paving the way for a new era of proactive, context-aware AI companions capable of assisting with a vast array of tasks in our daily lives.

最終願景是數位與實體能力的匯流，打造能在使用者各種環境中無縫運作的通用 AI 助手。這場演進也透過「氛圍編碼」重塑軟體創作方式，形成開發者與 AI 更直覺、對話式的合作。這種新方法優先考量高層目標與創意意圖，讓開發者聚焦於結果而非實作細節。此轉變透過把 AI 視為創意夥伴而加速開發、促進創新。最終,這些進展正鋪就一個主動、具情境感知的 AI 夥伴時代，能在日常生活中協助廣泛任務。

References
參考資料

1. Open AI Operator, https://openai.com/index/introducing-operator/
2. Open AI ChatGPT Agent: https://openai.com/index/introducing-chatgpt-agent/
3. Browser Use: https://docs.browser-use.com/introduction
4. Project Mariner, https://deepmind.google/models/project-mariner/
5. Anthropic Computer use: https://docs.anthropic.com/en/docs/build-with-claude/computer-use
6. Project Astra, https://deepmind.google/models/project-astra/
7. Gemini Live, https://gemini.google/overview/gemini-live/?hl=en
8. OpenAI’s GPT-4, https://openai.com/index/gpt-4-research/
9. Claude 4, https://www.anthropic.com/news/claude-4

Appendix C - Quick overview of Agentic Frameworks

附錄 C：代理框架快速概覽

LangChain

LangChain

LangChain is a framework for developing applications powered by LLMs. Its core strength lies in its LangChain Expression Language (LCEL), which allows you to “pipe” components together into a chain. This creates a clear, linear sequence where the output of one step becomes the input for the next. It’s built for workflows that are Directed Acyclic Graphs (DAGs), meaning the process flows in one direction without loops.

LangChain 是用於開發 LLM 應用的框架。其核心優勢在於 LangChain Expression Language（LCEL），能將元件「串接」成鏈。這會形成清楚的線性序列：前一步輸出成為下一步輸入。它適合有向無環圖（DAG）型的流程，也就是流程單向前進、不會迴圈。

Use it for:

適用於：

- Simple RAG: Retrieve a document, create a prompt, get an answer from an LLM.
- Summarization: Take user text, feed it to a summarization prompt, and return the output.
- Extraction: Extract structured data (like JSON) from a block of text.
- 簡單 RAG：檢索文件、建立提示、由 LLM 產出答案。
- 摘要：接收使用者文字，送入摘要提示並回傳輸出。
- 擷取：從文字區塊中擷取結構化資料（如 JSON）。

Python

Python

```
# A simple LCEL chain conceptually # (This is not runnable code, just illustrates the flow)
chain = prompt | model | output_parse
```

LangGraph

LangGraph

LangGraph is a library built on top of LangChain to handle more advanced agentic systems. It allows you to define your workflow as a graph with nodes (functions or LCEL chains) and edges (conditional logic). Its main advantage is the ability to create cycles, allowing the application to loop, retry, or call tools in a flexible order until a task is complete. It explicitly manages the application state, which is passed between nodes and updated throughout the process.

LangGraph 是建立在 LangChain 之上的函式庫，用於處理更進階的代理式系統。它允許你將工作流程定義為圖（graph），包含節點（函式或 LCEL 鏈）與邊（條件邏輯）。其主要優勢是可建立循環，讓應用能迴圈、重試或以彈性順序呼叫工具直到任務完成。它明確管理應用狀態，狀態會在節點間傳遞並在過程中更新。

Use it for:
適用於：

- Multi-agent Systems: A supervisor agent routes tasks to specialized worker agents, potentially looping until the goal is met.
- Plan-and-Execute Agents: An agent creates a plan, executes a step, and then loops back to update the plan based on the result.
- Human-in-the-Loop: The graph can wait for human input before deciding which node to go to next.
- 多代理系統：主管代理將任務分派給專業工作代理，必要時迴圈直到達成目標。
- 規劃與執行代理：代理先制定計畫、執行一步，再回到流程中依結果更新計畫。
- 人類介入：圖流程可在決定下一節點之前等待人類輸入。

Feature	LangChain	LangGraph
Core Abstraction	Chain (using LCEL)	Graph of Nodes
Workflow Type	Linear (Directed Acyclic Graph)	Cyclical (Graphs with loops)
State Management	Generally stateless per run	Explicit and persistent state object
Primary Use	Simple, predictable sequences	Complex, dynamic, stateful agents

功能	LangChain	LangGraph
核心抽象	鏈（使用 LCEL ）	節點圖
工作流程類型	線性（有向無環圖）	循環（具迴圈的圖）
狀態管理	單次執行多為無狀態	明確且可持久的狀態物件
主要用途	簡單、可預測的序列	複雜、動態、有狀態的代理

Which One Should You Use?

應該選哪一個?

- Choose LangChain when your application has a clear, predictable, and linear flow of steps. If you can define the process from A to B to C without needing to loop back, LangChain with LCEL is the perfect tool.
- Choose LangGraph when you need your application to reason, plan, or operate in a loop. If your agent needs to use tools, reflect on the results, and potentially try again with a different approach, you need the cyclical and stateful nature of LangGraph.
- 當應用有清楚、可預測且線性的流程時選 LangChain。若流程能從 A 到 B 再到 C、無需回圈,LangChain 搭配 LCEL 是最佳工具。
- 當應用需要推理、規劃或在迴圈中運作時選 LangGraph。若代理需要使用工具、反思結果並可能以不同方法重試, 就需要 LangGraph 的循環與狀態特性。

```
# Graph state
class State(TypedDict):
    topic: str
    joke: str
    story: str
    poem: str
    combined_output: str


# Nodes
```

```python
def call_llm_1(state: State):
    """First LLM call to generate initial joke"""
    msg = llm.invoke(f"Write a joke about {state['topic']}")
    return {"joke": msg.content}


def call_llm_2(state: State):
    """Second LLM call to generate story"""
    msg = llm.invoke(f"Write a story about {state['topic']}")
    return {"story": msg.content}


def call_llm_3(state: State):
    """Third LLM call to generate poem"""
    msg = llm.invoke(f"Write a poem about {state['topic']}")
    return {"poem": msg.content}


def aggregator(state: State):
    """Combine the joke and story into a single output"""
    combined = f"Here's a story, joke, and poem about {state['topic']}!\n\n"
    combined += f"STORY:\n{state['story']}\n\n"
    combined += f"JOKE:\n{state['joke']}\n\n"
    combined += f"POEM:\n{state['poem']}"
    return {"combined_output": combined}


# Build workflow
parallel_builder = StateGraph(State)

# Add nodes
parallel_builder.add_node("call_llm_1", call_llm_1)
parallel_builder.add_node("call_llm_2", call_llm_2)
parallel_builder.add_node("call_llm_3", call_llm_3)
parallel_builder.add_node("aggregator", aggregator)

# Add edges to connect nodes
parallel_builder.add_edge(START, "call_llm_1")
parallel_builder.add_edge(START, "call_llm_2")
parallel_builder.add_edge(START, "call_llm_3")
parallel_builder.add_edge("call_llm_1", "aggregator")
parallel_builder.add_edge("call_llm_2", "aggregator")
parallel_builder.add_edge("call_llm_3", "aggregator")
parallel_builder.add_edge("aggregator", END)

parallel_workflow = parallel_builder.compile()

# Show workflow
display(Image(parallel_workflow.get_graph().draw_mermaid_png()))
```

```python
# Invoke
state = parallel_workflow.invoke({"topic": "cats"})
print(state["combined_output"])
```

This code defines and runs a LangGraph workflow that operates in parallel. Its main purpose is to simultaneously generate a joke, a story, and a poem about a given topic and then combine them into a single, formatted text output.

這段程式碼定義並執行一個可並行運作的 LangGraph 工作流程，主要目的是同時生成關於特定主題的笑話、故事與詩，並將它們合併成一段格式化輸出。

Google’s ADK

Google 的 ADK

Google’s Agent Development Kit, or ADK, provides a high-level, structured framework for building and deploying applications composed of multiple, interacting AI agents. It contrasts with LangChain and LangGraph by offering a more opinionated and production-oriented system for orchestrating agent collaboration, rather than providing the fundamental building blocks for an agent’s internal logic.

Google 的 Agent Development Kit（ADK）提供高階、結構化的框架，用於建構與部署由多個互動式 AI 代理組成的應用。相較於 LangChain 與 LangGraph，它提供更具意見性且偏向生產環境的協作編排系統，而不是代理內部邏輯的基礎積木。

LangChain operates at the most foundational level, offering the components and standardized interfaces to create sequences of operations, such as calling a model and parsing its output. LangGraph extends this by introducing a more flexible and powerful control flow; it treats an agent’s workflow as a stateful graph. Using LangGraph, a developer explicitly defines nodes, which are functions or tools, and edges, which dictate the path of execution. This graph structure allows for complex, cyclical reasoning where the system can loop, retry tasks, and make decisions based on an explicitly managed state object that is passed between nodes. It gives the developer fine-grained control over a single agent’s thought process or the ability to construct a multi-agent system from first principles.

LangChain 位於最基礎層，提供元件與標準化介面以建立操作序列，例如呼叫模型與解析輸出。LangGraph 進一步引入更彈性且強大的控制流程,將代理的工作流程視為有狀態圖。使用 LangGraph 時,開發者需明確定義節點(函式或工具)與邊(執行路徑)。這種圖結構允許複雜的循環推理,使系統可迴圈、重試,並根據節點間傳遞的明確狀態物件作出決策。它讓開發者對單一代理的思考過程有細粒度控制,或從第一性原理建構多代理系統。

Google’s ADK abstracts away much of this low-level graph construction. Instead of asking the developer to define every node and edge, it provides pre-built architectural patterns for multi-agent interaction. For instance, ADK has built-in agent types like SequentialAgent or ParallelAgent, which manage the flow of control between different agents automatically. It is architected around the concept of a “team” of agents, often with a primary agent delegating tasks to specialized sub-agents. State and session management are handled more implicitly by the framework, providing a more cohesive but less granular approach than LangGraph's explicit state passing. Therefore, while LangGraph gives you the detailed tools to design the intricate wiring of a single robot or a team, Google's ADK gives you a factory assembly line designed to build and manage a fleet of robots that already know how to work together.

Google 的 ADK 抽象化了大量低階圖建構工作。它不要求開發者定義每個節點與邊，而是提供多代理互動的預建架構模式。例如 ADK 內建 SequentialAgent 或 ParallelAgent 等代理類型，可自動管理不同代理之間的控制流程。它以「代理團隊」為架構核心，通常由主要代理把任務分派給專業子代理。狀態與會話管理在框架中以較隱性的方式處理，提供比 LangGraph 明確狀態傳遞更完整但更少細節控制的方法。因此，LangGraph 提供的是設計單一機器或團隊精細連線的工具，而 Google 的 ADK 則像能建造與管理已知協作方式之機器群的工廠生產線。

```
from google.adk.agents import LlmAgent
from google.adk.tools import google_Search

dice_agent = LlmAgent(
    model="gemini-2.0-flash-exp",
    name="question_answer_agent",
    description="A helpful assistant agent that can answer questions.",
    instruction="""Respond to the query using google search""",
    tools=[google_search],
)
```

This code creates a search-augmented agent. When this agent receives a question, it will not just rely on its pre-existing knowledge. Instead, following its instructions, it will use the Google Search tool to find relevant, real-time information from the web and then use that information to construct its answer.

這段程式碼建立一個帶搜尋能力的代理。當代理接收到問題時,不會只依賴既有知識,而是依指令使用 Google Search 工具從網路取得即時相關資訊，再以此構建答案。

Crew.AI

Crew.AI

CrewAI offers an orchestration framework for building multi-agent systems by focusing on collaborative roles and structured processes. It operates at a higher level of abstraction than foundational toolkits, providing a conceptual model that mirrors a human team. Instead of defining the granular flow of logic as a graph, the developer defines the actors and their assignments, and CrewAI manages their interaction.

CrewAI 提供多代理系統的協作編排框架，強調角色分工與結構化流程。它的抽象層級高於基礎工具集，提供一個類似人類團隊的概念模型。開發者不需以圖定義細粒度流程，而是定義角色與任務分配，由 CrewAI 管理互動。

The core components of this framework are Agents, Tasks, and the Crew. An Agent is defined not just by its function but by a persona, including a specific role, a goal, and a backstory, which guides its behavior and communication style. A Task is a discrete unit of work with a clear description and expected output, assigned to a specific Agent. The Crew is the cohesive unit that contains the Agents and the list of Tasks, and it executes a predefined Process. This process dictates the workflow, which is typically either sequential, where the output of one task becomes the input for the next in line, or hierarchical, where a manager-like agent delegates tasks and coordinates the workflow among other agents.

該框架的核心元件為 Agents、Tasks 與 Crew。Agent 不僅由功能定義，還包含角色、目標與背景故事，這些設定會引導其行為與溝通風格。Task 是具明確描述與預期輸出的工作單元，指派給特定 Agent。Crew 是包含 Agents 與 Tasks 清單的整體單位，並執行預先定義的 Process。Process 會規範工作流程，通常為序列式(前一任務輸出成為下一個輸入)或層級式(類主管代理分派與協調其他代理)。

When compared to other frameworks, CrewAI occupies a distinct position. It moves away from the low-level, explicit state management and control flow of LangGraph, where a developer wires together every node and conditional edge. Instead of building a state machine, the developer designs a team charter. While Google's ADK provides a comprehensive, production-oriented platform for the entire agent lifecycle, CrewAI concentrates specifically on the logic of agent collaboration and for simulating a team of specialists

與其他框架相比,CrewAI 位置獨特。它遠離 LangGraph 的低階、明確狀態管理與控制流程 (開發者需串接每個節點與條件邊)，改為設計團隊章程而非建構狀態機。Google's ADK 提供涵蓋代理全生命週期的完整生產導向平台，而 CrewAI 則專注於代理協作邏輯與專家團隊的模擬。

```
@crew
def crew(self) -> Crew:
   """Creates the research crew"""
   return Crew(
     agents=self.agents,
     tasks=self.tasks,
     process=Process.sequential,
     verbose=True,
   )
```

This code sets up a sequential workflow for a team of AI agents, where they tackle a list of tasks in a specific order, with detailed logging enabled to monitor their progress.

這段程式碼為 AI 代理團隊建立一個序列式工作流程，讓它們依序處理任務清單，並啟用詳細記錄以監控進度。

Other Agent Development Framework

其他代理開發框架

Microsoft AutoGen: AutoGen is a framework centered on orchestrating multiple agents that solve tasks through conversation. Its architecture enables agents with distinct capabilities to interact, allowing for complex problem decomposition and collaborative resolution. The primary advantage of AutoGen is its flexible, conversation-driven approach that supports dynamic and complex multi-agent interactions. However, this conversational paradigm can lead to less predictable execution paths and may require sophisticated prompt engineering to ensure tasks converge efficiently.

Microsoft AutoGen: AutoGen 以透過對話協調多代理解決任務為核心。其架構讓具不同能力的代理互動，以進行複雜問題分解與協作解決。AutoGen 的主要優勢是彈性且以對話驅動，能支援動態與複雜的多代理互動。然而此對話式範式可能導致執行路徑較難預測，並可能需要更精細的提示工程來確保任務有效收斂。

LlamaIndex: LlamaIndex is fundamentally a data framework designed to connect large language models with external and private data sources. It excels at creating sophisticated data ingestion and retrieval pipelines, which are essential for building knowledgeable agents that can perform RAG. While its data indexing and querying capabilities are exceptionally powerful for creating context-aware agents, its native tools for complex agentic control flow and multi-agent orchestration are less developed compared to agent-first frameworks. LlamaIndex is optimal when the core technical challenge is data retrieval and synthesis.

LlamaIndex: LlamaIndex 本質上是資料框架，用於連接大型語言模型與外部或私有資料來源。它擅長建立精密的資料導入與檢索管線，這是打造能執行 RAG 的知識型代理所必需。雖然其資料索引與查詢能力對情境感知代理非常強大，但其原生的複雜代理控制流程與多代理編排工具，相較代理優先的框架仍較不足。當核心技術挑戰在於資料檢索與彙整時，LlamaIndex 最適合。

Haystack: Haystack is an open-source framework engineered for building scalable and production-ready search systems powered by language models. Its architecture is composed of modular, interoperable nodes that form pipelines for document retrieval, question answering, and summarization. The main strength of Haystack is its focus on performance and scalability for large-scale information retrieval tasks, making it suitable for enterprise-grade applications. A potential trade-off is that its design, optimized for search pipelines, can be more rigid for implementing highly dynamic and creative agentic behaviors.

Haystack: Haystack 是開源框架，用於打造可擴展且可上線的語言模型搜尋系統。其架構由模組化、可互操作的節點組成，形成文件檢索、問答與摘要的管線。其主要優勢在於對大型資訊檢索任務的效能與可擴展性，適合企業級應用。可能的取捨是它為搜尋管線最佳化的設計，在實作高度動態與創意的代理行為時較為僵硬。

MetaGPT: MetaGPT implements a multi-agent system by assigning roles and tasks based on a predefined set of Standard Operating Procedures (SOPs). This framework structures agent collaboration to mimic a software development company, with agents taking on roles like product managers or engineers to complete complex tasks. This SOP-driven approach results in highly structured and coherent outputs, which is a significant advantage for specialized domains like code generation. The framework's primary limitation is its high degree of specialization, making it less adaptable for general-purpose agentic tasks outside of its core design.

MetaGPT: MetaGPT 依據預先定義的標準作業程序（SOP）指派角色與任務，實作多代理系統。此框架將代理協作結構化，以模擬軟體開發公司，讓代理扮演產品經理或工程師等角色完成複雜任務。SOP 驅動的方法能產生高度結構化且一致的輸出,對程式碼生成等專業領域尤具優勢。其主要限制是高度專門化,使其在核心設計以外的一般代理任務上較不具彈性。

SuperAGI: SuperAGI is an open-source framework designed to provide a complete lifecycle management system for autonomous agents. It includes features for agent provisioning, monitoring, and a graphical interface, aiming to enhance the reliability of agent execution. The key benefit is its focus on production-readiness, with built-in mechanisms to handle common failure modes like looping and to provide observability into agent performance. A potential drawback is that its comprehensive platform approach can introduce more complexity and overhead than a more lightweight, library-based framework.

SuperAGI: SuperAGI 是開源框架，旨在提供自主代理的全生命週期管理系統。它包含代理配置、監控與圖形介面等功能，提升代理執行的可靠性。其核心優勢是聚焦於生產就緒，內建機制可處理常見失敗模式（如無限迴圈），並提供對代理效能的可觀測性。可能的缺點是平台化設計較全面，相較輕量的函式庫框架會帶來更多複雜度與負擔。

Semantic Kernel: Developed by Microsoft, Semantic Kernel is an SDK that integrates large language models with conventional programming code through a system of “plugins” and “planners.” It allows an LLM to invoke native functions and orchestrate workflows, effectively treating the model as a reasoning engine within a larger software application. Its primary strength is its seamless integration with existing enterprise codebases, particularly in .NET and Python environments. The conceptual overhead of its plugin and planner architecture can present a steeper learning curve compared to more straightforward agent frameworks.

Semantic Kernel: 由 Microsoft 開發的 Semantic Kernel 是一個 SDK，透過「plugins」與「planners」系統將大型語言模型與傳統程式碼整合。它讓 LLM 可呼叫原生函式並編排工作流程，使模型在更大型的軟體應用中成為推理引擎。其主要優勢是能與既有企業程式碼庫無縫整合，尤其在 .NET 與 Python 環境。其插件與規劃器架構的概念負擔較高,學習曲線可能比更直接的代理框架陡峭。

Strands Agents: An AWS lightweight and flexible SDK that uses a model-driven approach for building and running AI agents. It is designed to be simple and scalable, supporting everything from basic conversational assistants to complex multi-agent autonomous systems. The framework is model-agnostic, offering broad support for various LLM providers, and includes native integration with the MCP for easy access to external tools. Its core advantage is its simplicity and flexibility, with a customizable agent loop that is easy to get started with. A potential trade-off is that its lightweight design means developers may need to build out more of the surrounding operational infrastructure, such as advanced monitoring or lifecycle management systems, which more comprehensive frameworks might provide out-of-the-box.

Strands Agents: AWS 的輕量、彈性 SDK，採模型驅動方法來建立與執行 AI 代理。它設計簡單且可擴展，支援從基本對話助理到複雜多代理自治系統。該框架與模型無關，廣泛支援各種 LLM 供應商，並原生整合 MCP，便於存取外部工具。其核心優勢在於簡潔與彈性，具

可自訂的代理迴圈，容易上手。可能的取捨是輕量設計代表開發者需要自行建構更多周邊運作基礎設施，例如進階監控或生命週期管理系統，這些在更完整的框架中可能開箱即用。

Conclusion

結論

The landscape of agentic frameworks offers a diverse spectrum of tools, from low-level libraries for defining agent logic to high-level platforms for orchestrating multi-agent collaboration. At the foundational level, LangChain enables simple, linear workflows, while LangGraph introduces stateful, cyclical graphs for more complex reasoning. Higher-level frameworks like CrewAI and Google's ADK shift the focus to orchestrating teams of agents with predefined roles, while others like LlamaIndex specialize in data-intensive applications. This variety presents developers with a core trade-off between the granular control of graph-based systems and the streamlined development of more opinionated platforms. Consequently, selecting the right framework hinges on whether the application requires a simple sequence, a dynamic reasoning loop, or a managed team of specialists. Ultimately, this evolving ecosystem empowers developers to build increasingly sophisticated AI systems by choosing the precise level of abstraction their project demands.

代理框架的版圖提供了多元工具，從用於定義代理邏輯的低階函式庫，到編排多代理協作的高階平台。基礎層的 LangChain 支援簡單線性流程，而 LangGraph 以有狀態、循環的圖來支援更複雜的推理。CrewAI 與 Google 的 ADK 等高階框架將重點轉向以預先定義角色編排代理團隊，另有如 LlamaIndex 專注於資料密集型應用。這些多樣性帶給開發者一個核心取捨,在圖式系統的細緻控制與更具意見性平台的快速開發之間取平衡。因此,選擇正確框架取決於應用需要的是簡單序列、動態推理迴圈,或受管理的專家團隊。最終,這個不斷演進的生態系讓開發者能依專案所需的抽象層級,建構愈來愈精密的 AI 系統。

References

參考資料

1. LangChain, https://www.langchain.com/
2. LangGraph, https://www.langchain.com/langgraph
3. Google's ADK, https://google.github.io/adk-docs/
4. Crew.AI, https://docs.crewai.com/en/introduction

Appendix D - Building an Agent with AgentSpace

附錄 D：使用 AgentSpace 建構代理

Overview

概觀

AgentSpace is a platform designed to facilitate an "agent-driven enterprise" by integrating artificial intelligence into daily workflows. At its core, it provides a unified search capability across an organization's entire digital footprint, including documents, emails, and databases. This system utilizes advanced AI models, like Google's Gemini, to comprehend and synthesize information from these varied sources.

AgentSpace 是一個將人工智慧導入日常工作流程、以支援「代理驅動企業」為目標的平台。其核心能力是跨越組織全部數位足跡(文件、郵件、資料庫等)的統一搜尋。系統使用 Google Gemini 等先進 AI 模型來理解並整合這些多元來源的資訊。

The platform enables the creation and deployment of specialized AI “agents” that can perform complex tasks and automate processes. These agents are not merely chatbots; they can reason, plan, and execute multi-step actions autonomously. For instance, an agent could research a topic, compile a report with citations, and even generate an audio summary.

該平台可建立並部署專門化的 AI「代理」，能執行複雜任務並自動化流程。這些代理不只是聊天機器人；它們能推理、規劃，並自主執行多步驟行動。例如，代理可以研究主題、整理附引用的報告，甚至生成語音摘要。

To achieve this, AgentSpace constructs an enterprise knowledge graph, mapping the relationships between people, documents, and data. This allows the AI to understand context and deliver more relevant and personalized results. The platform also includes a no-code interface called Agent Designer for creating custom agents without requiring deep technical expertise.

為達成此目標,AgentSpace 會建立企業知識圖譜,映射人員、文件與資料之間的關係,使 AI 能理解脈絡並提供更相關、個人化的結果。平台也包含名為 Agent Designer 的無程式碼介面，讓使用者無需深厚技術背景也能建立客製代理。

Furthermore, AgentSpace supports a multi-agent system where different AI agents can communicate and collaborate through an open protocol known as the Agent2Agent (A2A) Protocol. This interoperability allows for more complex and orchestrated workflows. Security is a foundational component, with features like role-based access controls and data encryption to protect sensitive enterprise information. Ultimately, AgentSpace aims to enhance productivity and decision-making by embedding intelligent, autonomous systems directly into an organization’s operational fabric.

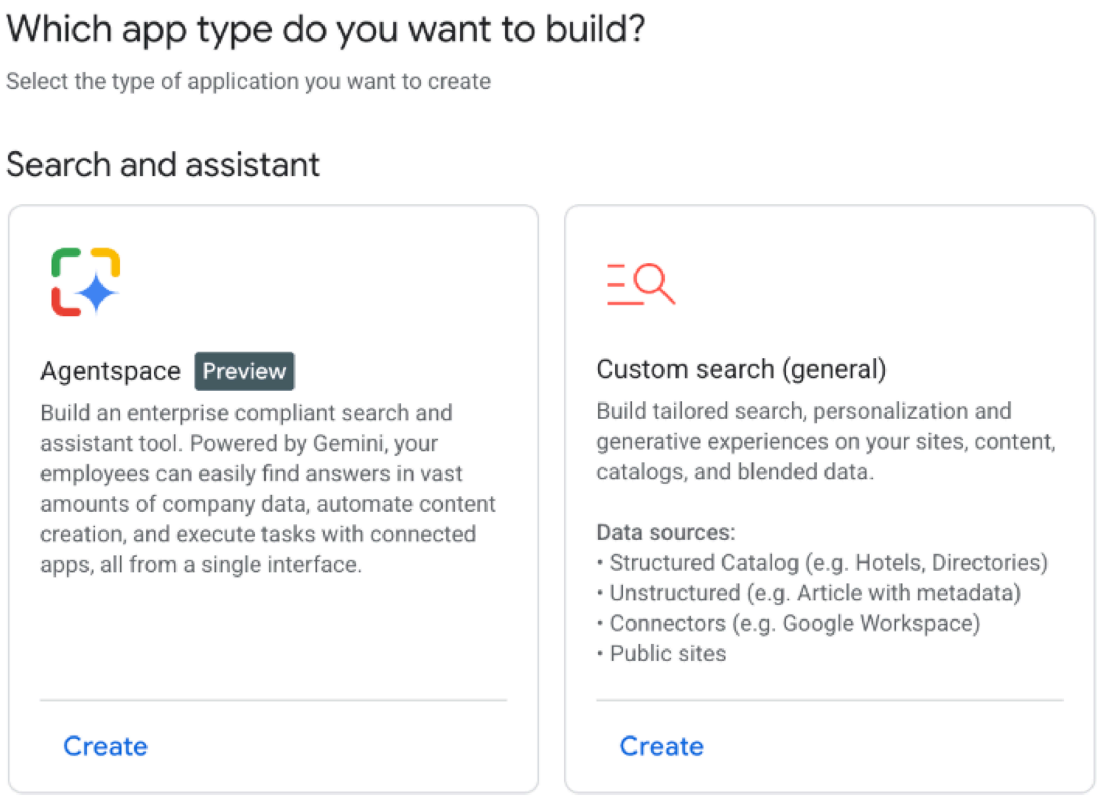
此外，AgentSpace 支援多代理系統，不同 AI 代理可透過開放協定 Agent2Agent（A2A）進行溝通與協作。這種互通性可支援更複雜且可編排的工作流程。安全性是其基礎元件之一,包含角色型存取控制與資料加密等功能,以保護企業敏感資訊。最終,AgentSpace 旨在將智慧自主系統直接嵌入組織運作脈絡中，提升生產力與決策品質。

How to build an Agent with AgentSpace UI

如何用 AgentSpace UI 建立代理

Figure 1 illustrates how to access AgentSpace by selecting AI Applications from the Google Cloud Console.

圖 1 示範如何從 Google Cloud Console 選擇 AI Applications 以進入 AgentSpace。



Which app type do you want to build?

Select the type of application you want to create

Search and assistant

Agentspace Preview

Build an enterprise compliant search and assistant tool. Powered by Gemini, your employees can easily find answers in vast amounts of company data, automate content creation, and execute tasks with connected apps, all from a single interface.

Custom search (general)

Build tailored search, personalization and generative experiences on your sites, content, catalogs, and blended data.

Data sources:

- Structured Catalog (e.g. Hotels, Directories)
- Unstructured (e.g. Article with metadata)
- Connectors (e.g. Google Workspace)
- Public sites

Figure 3: GCP: Access AgentSpace

Fig. 1: How to use Google Cloud Console to access AgentSpace

圖 1: 如何使用 Google Cloud Console 進入 AgentSpace

Your agent can be connected to various services, including Calendar, Google Mail, Workaday, Jira, Outlook, and Service Now (see Fig. 2).

你的代理可連接多種服務，包括 Calendar、Google Mail、Workaday、Jira、Outlook 與 Service Now（見圖 2）。

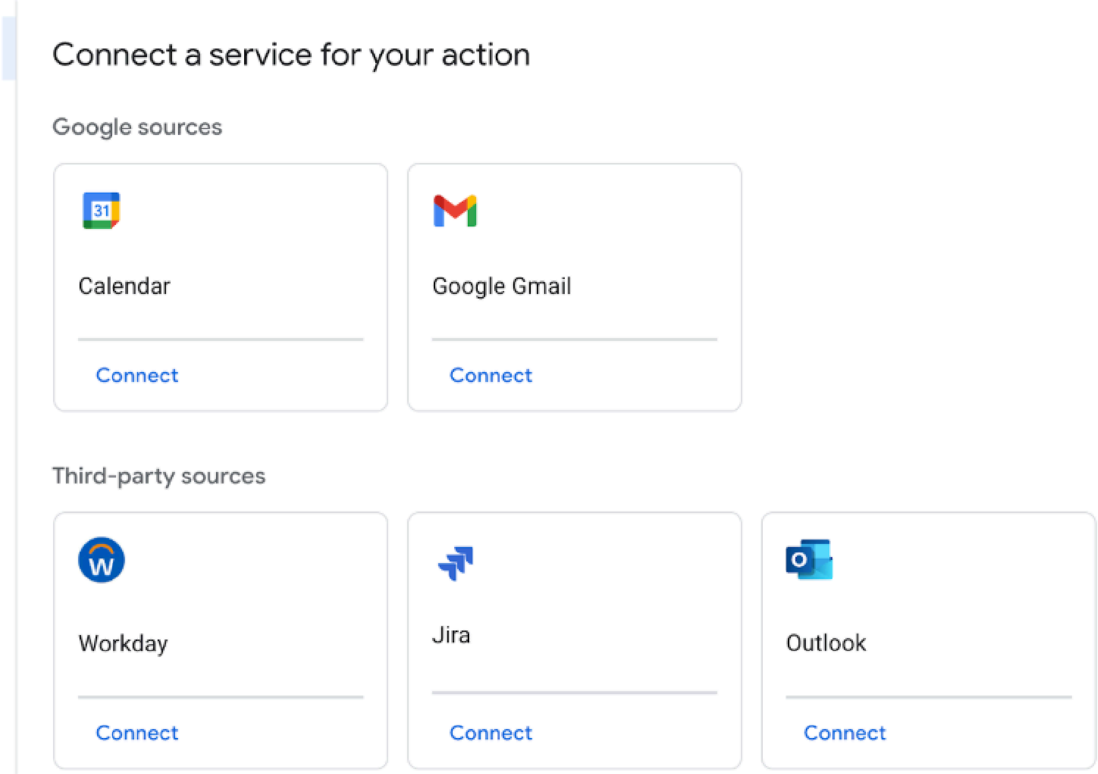


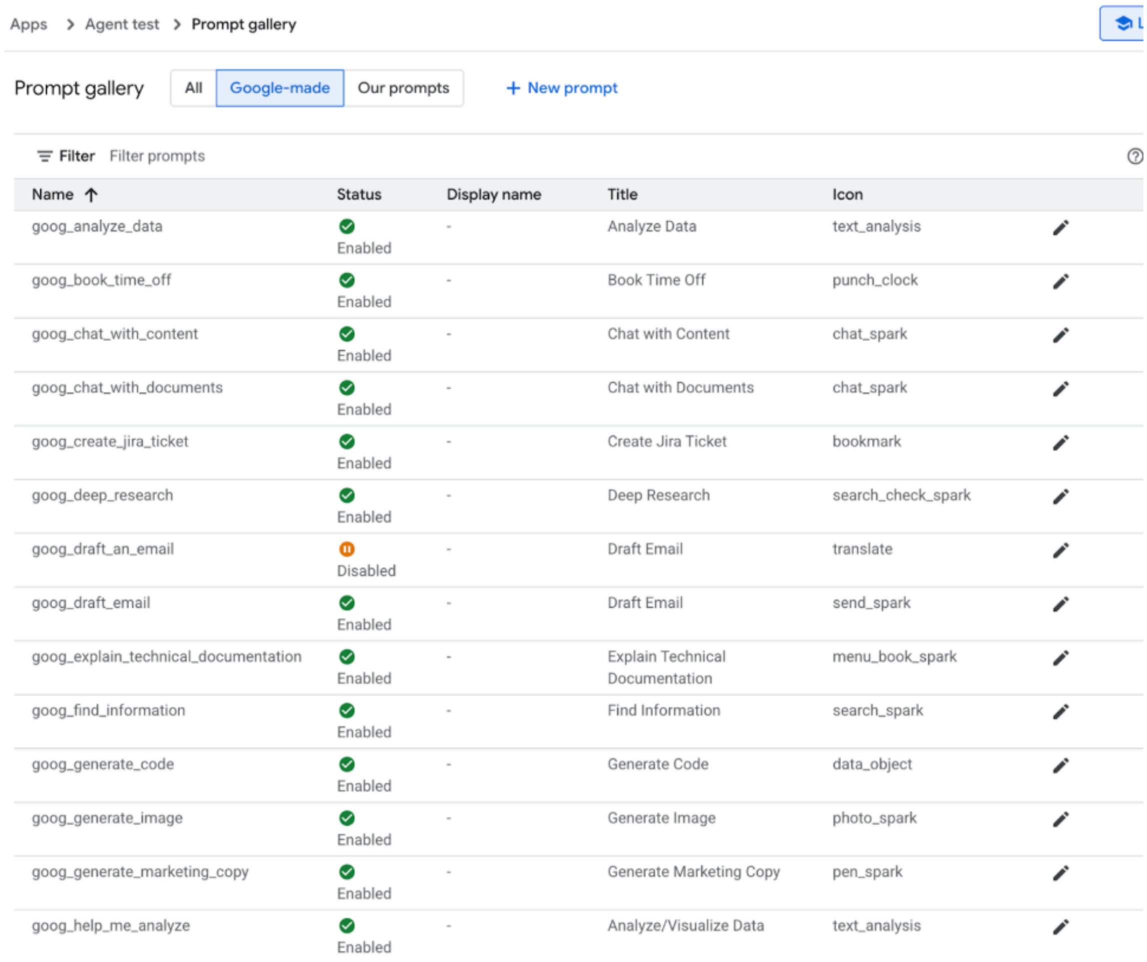
Figure 4: GCP: Integrate with diverse services

Fig. 2: Integrate with diverse services, including Google and third-party platforms.

圖 2: 整合多元服務，包括 Google 與第三方平台。

The Agent can then utilize its own prompt, chosen from a gallery of pre-made prompts provided by Google, as illustrated in Fig. 3.

接著代理可使用自己的提示，從 Google 提供的預製提示庫中選擇，如圖 3 所示。



Name ↑	Status	Display name	Title	Icon
goog_analyze_data	✓ Enabled	-	Analyze Data	text_analysis
goog_book_time_off	✓ Enabled	-	Book Time Off	punch_clock
goog_chat_with_content	✓ Enabled	-	Chat with Content	chat_spark
goog_chat_with_documents	✓ Enabled	-	Chat with Documents	chat_spark
goog_create_jira_ticket	✓ Enabled	-	Create Jira Ticket	bookmark
goog_deep_research	✓ Enabled	-	Deep Research	search_check_spark
goog_draft_an_email	⏸ Disabled	-	Draft Email	translate
goog_draft_email	✓ Enabled	-	Draft Email	send_spark
goog_explain_technical_documentation	✓ Enabled	-	Explain Technical Documentation	menu_book_spark
goog_find_information	✓ Enabled	-	Find Information	search_spark
goog_generate_code	✓ Enabled	-	Generate Code	data_object
goog_generate_image	✓ Enabled	-	Generate Image	photo_spark
goog_generate_marketing_copy	✓ Enabled	-	Generate Marketing Copy	pen_spark
goog_help_me_analyze	✓ Enabled	-	Analyze/Visualize Data	text_analysis

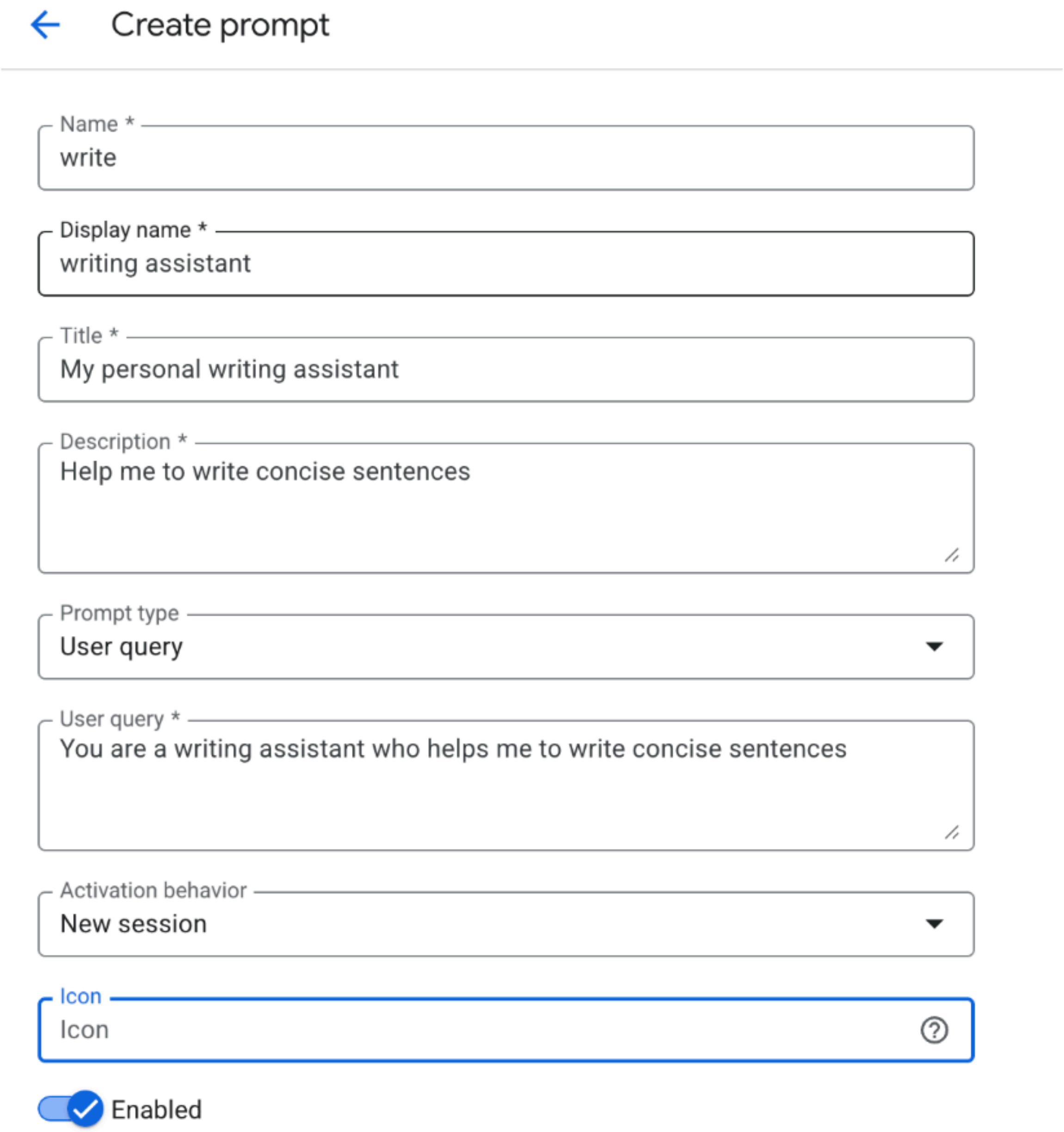
Figure 5: GCP: Google's Gallery of Pre-Assembled Prompts

Fig. 3: Google's Gallery of Pre-assembled Prompts

圖 3: Google 的預先組裝提示庫

In alternative you can create your own prompt as in Fig.4, which will be then used by your agent

另外，你也可如圖 4 所示自行建立提示，供代理使用。



Create prompt

Name *
write

Display name *
writing assistant

Title *
My personal writing assistant

Description *
Help me to write concise sentences

Prompt type
User query

User query *
You are a writing assistant who helps me to write concise sentences

Activation behavior
New session

Icon
Icon

Enabled

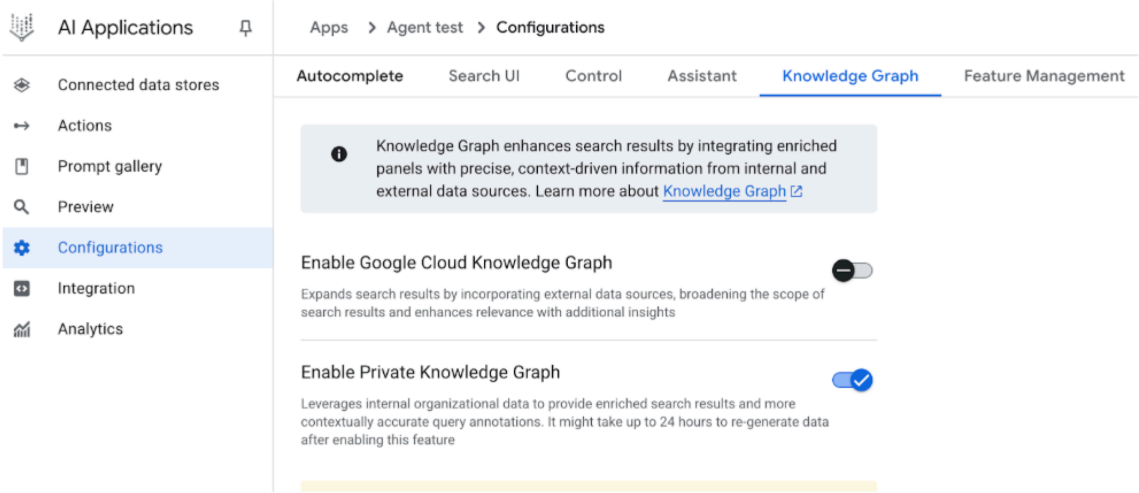
Figure 6: GCP: Customizing the Agent's Prompt

Fig.4: Customizing the Agent's Prompt

圖 4: 自訂代理提示

AgentSpace offers a number of advanced features such as integration with datastores to store your own data, integration with Google Knowledge Graph or with your private Knowledge Graph, Web interface for exposing your agent to the Web, and Analytics to monitor usage, and more (see Fig. 5)

AgentSpace 提供多項進階功能，例如與資料儲存整合以保存自有資料、與 Google Knowledge Graph 或私有知識圖譜整合、提供網頁介面以將代理曝光到 Web，以及用於監測使用量的 Analytics 等（見圖 5）。



Knowledge Graph enhances search results by integrating enriched panels with precise, context-driven information from internal and external data sources. Learn more about Knowledge Graph

Enable Google Cloud Knowledge Graph ○

Expands search results by incorporating external data sources, broadening the scope of search results and enhances relevance with additional insights

Enable Private Knowledge Graph ✓

Leverages internal organizational data to provide enriched search results and more contextually accurate query annotations. It might take up to 24 hours to re-generate data after enabling this feature

Figure 7: GCP: AgentSpace Advanced Capabilities

Fig. 5: AgentSpace advanced capabilities

圖 5： AgentSpace 進階能力

Upon completion, the AgentSpace chat interface (Fig. 6) will be accessible.

完成後即可使用 AgentSpace 的聊天介面（圖 6）。

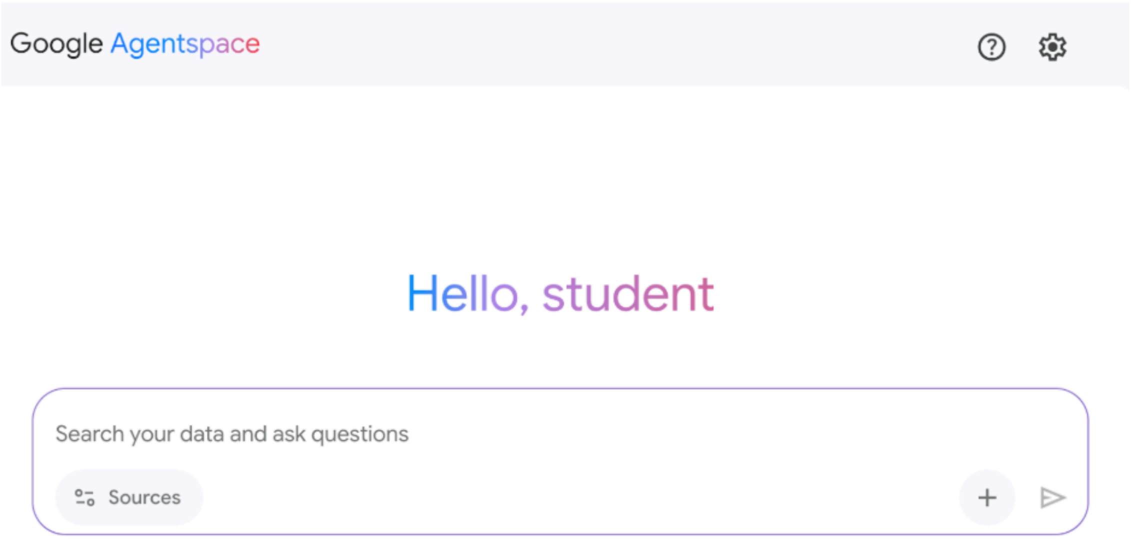


Figure 8: GCP: AgentSpace User Interface for initiating a chat with your Agent

Fig. 6: The AgentSpace User Interface for initiating a chat with your Agent.

圖 6: 用於啟動與代理聊天的 AgentSpace 介面。

Conclusion
結論

In conclusion, AgentSpace provides a functional framework for developing and deploying AI agents within an organization's existing digital infrastructure. The system's architecture links complex backend processes, such as autonomous reasoning and enterprise knowledge graph mapping, to a graphical user interface for agent construction. Through this interface, users can configure agents by integrating various data services and defining their operational parameters via prompts, resulting in customized, context-aware automated systems.

總結而言,AgentSpace 提供一套可在組織既有數位基礎設施內開發與部署 AI 代理的實用框架。其系統架構將自主推理與企業知識圖譜映射等複雜後端流程,連結到用於建構代理的圖形化介面。使用者可透過此介面整合多種資料服務,並以提示定義操作參數,打造客製化且具情境感知的自動化系統。

This approach abstracts the underlying technical complexity, enabling the construction of specialized multi-agent systems without requiring deep programming expertise. The primary objective is to embed automated analytical and operational capabilities directly into workflows, thereby increasing process efficiency and enhancing data-driven analysis. For practical instruction, hands-on learning modules are available, such as the "Build a Gen AI Agent with Agentspace" lab on Google Cloud Skills Boost, which provides a structured environment for skill acquisition.

此方法抽象化底層技術複雜度,使得即使沒有深厚程式能力,也能建立專業的多代理系統。其主要目標是將自動化分析與作業能力直接嵌入工作流程,以提升效率並強化資料驅動的分析。實作學習可參考 Google Cloud Skills Boost 的「Build a Gen AI Agent with Agentspace」實作課程,提供結構化的技能養成環境。

References
參考資料

1. Create a no-code agent with Agent Designer, https://cloud.google.com/agentspace/agentspace-enterprise/docs/agent-designer
2. Google Cloud Skills Boost, https://www.cloudskillsboost.google/

Appendix E - AI Agents on the CLI

附錄 E: CLI 上的 AI 代理

Introduction
導論

The developer's command line, long a bastion of precise, imperative commands, is undergoing a profound transformation. It is evolving from a simple shell into an intelligent, collaborative workspace powered by a new class of tools: AI Agent Command-Line Interfaces (CLIs). These agents move beyond merely executing commands; they understand natural language, maintain context about your entire codebase, and can perform complex, multi-step tasks that automate significant parts of the development lifecycle.

開發者的命令列長期以來是精準、命令式指令的堡壘,如今正經歷深刻轉變。它正從簡單的 shell 演化為由新一代工具驅動的智慧協作工作空間: AI 代理命令列介面(CLI)。這些代理不只是執行指令;它們理解自然語言、維持對整個程式碼庫的脈絡,並能執行複雜、多步驟任務,自動化開發生命週期中相當大的部分。

This guide provides an in-depth look at four leading players in this burgeoning field, exploring their unique strengths, ideal use cases, and distinct philosophies to help you determine which tool best fits your workflow. It is important to note that many of the example use cases provided for a specific tool can often be accomplished by the other agents as well. The key differentiator between these tools frequently lies in the quality, efficiency, and nuance of the results they are able to achieve for a given task. There are specific benchmarks designed to measure these capabilities, which will be discussed in the following sections.

本指南深入介紹此新興領域的四個領先工具，探討它們的獨特優勢、理想使用情境與不同理念，協助你判斷哪個最適合你的工作流程。需注意的是，針對某一工具提供的許多範例情境，往往也能由其他代理完成。這些工具的關鍵差異多在於它們對特定任務所能達成的品質、效率與細緻度。為衡量這些能力，已有特定基準測試，將在後續章節說明。

Claude CLI (Claude Code)

Claude CLI（Claude Code）

Anthropic's Claude CLI is engineered as a high-level coding agent with a deep, holistic understanding of a project's architecture. Its core strength is its "agentic" nature, allowing it to create a mental model of your repository for complex, multi-step tasks. The interaction is highly conversational, resembling a pair programming session where it explains its plans before executing. This makes it ideal for professional developers working on large-scale projects involving significant refactoring or implementing features with broad architectural impacts.

Anthropic 的 Claude CLI 被打造為高階程式代理，能深度、整體性理解專案架構。其核心優勢在於「代理性」，能為複雜多步任務建立對整個程式庫的心智模型。互動方式高度對話化，類似結對編程，會在執行前說明計畫。這讓它特別適合處理大型專案、需要大幅重構或影響廣泛架構的功能實作的專業開發者。

Example Use Cases:
範例使用情境：

1. **Large-Scale Refactoring:** You can instruct it: "Our current user authentication relies on session cookies. Refactor the entire codebase to use stateless JWTs, updating the login/logout endpoints, middleware, and frontend token handling." Claude will then read all relevant files and perform the coordinated changes.

2. **API Integration:** After being provided with an OpenAPI specification for a new weather service, you could say: “Integrate this new weather API. Create a service module to handle the API calls, add a new component to display the weather, and update the main dashboard to include it.”
3. **Documentation Generation**: Pointing it to a complex module with poorly documented code, you can ask: “Analyze the `./src/utils/data_processing.js` file. Generate comprehensive TSDoc comments for every function, explaining its purpose, parameters, and return value.”
4. **大規模重構**: 你可以指示:「目前使用 session cookie 的使用者驗證,請將整個程式庫改成無狀態 JWT，更新登入/登出端點、中介層與前端的 token 處理。」Claude 會讀取相關檔案並進行協調修改。
5. **API 整合**: 提供新天氣服務的 OpenAPI 規格後,你可以說:「整合這個新天氣 API,建立處理 API 呼叫的 service 模組,新增顯示天氣的元件,並更新主儀表板加入此功能。」
6. **文件生成**: 指向一個文件不佳的複雜模組,你可要求:「分析 `./src/utils/data_processing.js` 檔案，為每個函式生成完整的 TSDoc 註解,說明用途、參數與回傳值。」

Claude CLI functions as a specialized coding assistant, with inherent tools for core development tasks, including file ingestion, code structure analysis, and edit generation. Its deep integration with Git facilitates direct branch and commit management. The agent’s extensibility is mediated by the Multi-tool Control Protocol (MCP), enabling users to define and integrate custom tools. This allows for interactions with private APIs, database queries, and execution of project-specific scripts. This architecture positions the developer as the arbiter of the agent’s functional scope, effectively characterizing Claude as a reasoning engine augmented by user-defined tooling.

Claude CLI 作為專門化程式助手,內建核心開發任務所需的工具,包括檔案讀取、程式結構分析與編輯生成。它與 Git 深度整合,方便直接管理分支與提交。代理的可擴展性透過 Multi-tool Control Protocol(MCP)實現,使使用者能定義並整合自訂工具,與私有 API 互動、查詢資料庫或執行專案腳本。此架構讓開發者成為代理功能範圍的裁決者,本質上將 Claude 定位為一個由使用者工具擴增的推理引擎。

Gemini CLI

Gemini CLI

Google’s Gemini CLI is a versatile, open-source AI agent designed for power and accessibility. It stands out with the advanced Gemini 2.5 Pro model, a massive context window, and multimodal capabilities (processing images and text). Its open-source nature, generous free tier, and “Reason and Act” loop make it a transparent, controllable, and excellent all-rounder for a broad audience, from hobbyists to enterprise developers, especially those within the Google Cloud ecosystem.

Google 的 Gemini CLI 是一個兼具效能與易用性的開源 AI 代理。它以先進的 Gemini 2.5 Pro 模型、超大上下文視窗與多模態能力(處理影像與文字)而突出。其開源特性、慷慨的免費額度與「Reason and Act」迴圈,使它透明可控,成為從興趣使用者到企業開發者的優秀全能型工具,特別適合 Google Cloud 生態系的使用者。

Example Use Cases:
範例使用情境：

1. **Multimodal Development:** You provide a screenshot of a web component from a design file (gemini describe component.png) and instruct it: “Write the HTML and CSS code to build a React component that looks exactly like this. Make sure it’s responsive.”
2. **Cloud Resource Management:** Using its built-in Google Cloud integration, you can command: “Find all GKE clusters in the production project that are running versions older than 1.28 and generate a gcloud command to upgrade them one by one.”
3. **Enterprise Tool Integration (via MCP):** A developer provides Gemini with a custom tool called get-employee-details that connects to the company’s internal HR API. The prompt is: “Draft a welcome document for our new hire. First, use the get-employee-details –id=E90210 tool to fetch their name and team, and then populate the welcome_template.md with that information.”
4. **Large-Scale Refactoring**: A developer needs to refactor a large Java codebase to replace a deprecated logging library with a new, structured logging framework. They can use Gemini with a prompt like: Read all *.java files in the ‘src/main/java’ directory. For each file, replace all instances of the ‘org.apache.log4j’ import and its ‘Logger’ class with ‘org.slf4j.Logger’ and ‘LoggerFactory’. Rewrite the logger instantiation and all .info(), .debug(), and .error() calls to use the new structured format with key-value pairs.
5. **多模態開發：** 提供設計檔中的元件截圖（gemini describe component.png），並指示：「撰寫 HTML 與 CSS，建立外觀完全一致的 React 元件，並確保具備響應式。」
6. **雲端資源管理：** 利用內建的 Google Cloud 整合，你可以下達：「找出 production 專案中版本低於 1.28 的所有 GKE 叢集,並生成逐一升級的 gcloud 指令。」
7. **企業工具整合（透過 MCP）：** 開發者提供 Gemini 一個自訂工具 get-employee-details，連接公司內部 HR API。提示為：「替新進員工起草歡迎文件。先使用 get-employee-details –id=E90210 取得姓名與團隊，再把資訊填入 welcome_template.md。」
8. **大規模重構：** 開發者要將大型 Java 程式庫從舊式 logging 改為新的結構化 logging，可用這樣的提示： Read all *.java files in the ‘src/main/java’ directory. For each file, replace all instances of the ‘org.apache.log4j’ import and its ‘Logger’ class with ‘org.slf4j.Logger’ and ‘LoggerFactory’. Rewrite the logger instantiation and all .info(), .debug(), and .error() calls to use the new structured format with key-value pairs.

Gemini CLI is equipped with a suite of built-in tools that allow it to interact with its environment. These include tools for file system operations (like reading and writing), a shell tool for running commands, and tools for accessing the internet via web fetching and searching. For broader context, it uses specialized tools to read multiple files at once and a memory tool to save information for later sessions. This functionality is built on a secure foundation: sandboxing isolates the model’s actions to prevent risk, while MCP servers act as a bridge, enabling Gemini to safely connect to your local environment or other APIs.

Gemini CLI 內建一套工具讓它與環境互動，包括檔案系統操作工具（讀寫）、執行指令的 shell 工具，以及透過網頁抓取與搜尋存取網路的工具。為取得更完整的脈絡，它可使用專門工具一次讀取多個檔案，並以記憶工具保存資訊供後續會話使用。這些功能建立在安全基礎上：沙箱隔離模型行為以降低風險，MCP 伺服器則作為橋樑，使 Gemini 能安全連接本地環境或其他 API。

Aider

Aider

Aider is an open-source AI coding assistant that acts as a true pair programmer by working directly on your files and committing changes to Git. Its defining feature is its directness; it applies edits, runs tests to validate them, and automatically commits every successful change. Being model-agnostic, it gives users complete control over cost and capabilities. Its git-centric workflow makes it perfect for developers who value efficiency, control, and a transparent, auditable trail of all code modifications.

Aider 是開源 AI 程式助手，會直接在你的檔案上工作並提交到 Git，如同真正的結對編程夥伴。其定義性特徵是「直接」，會直接套用修改、執行測試驗證，並自動提交每次成功的變更。由於模型無關，使用者可完全掌控成本與能力。其以 Git 為核心的流程,非常適合重視效率、掌控力與可稽核變更紀錄的開發者。

Example Use Cases:
範例使用情境：

1. **Test-Driven Development (TDD):** A developer can say: “Create a failing test for a function that calculates the factorial of a number.” After Aider writes the test and it fails, the next prompt is: “Now, write the code to make the test pass.” Aider implements the function and runs the test again to confirm.
2. **Precise Bug Squashing:** Given a bug report, you can instruct Aider: “The `calculate_total` function in billing.py fails on leap years. Add the file to the context, fix the bug, and verify your fix against the existing test suite.”
3. **Dependency Updates:** You could instruct it: “Our project uses an outdated version of the ‘requests’ library. Please go through all Python files, update the import statements and any deprecated function calls to be compatible with the latest version, and then update requirements.txt.”
4. **測試驅動開發（TDD）：** 開發者可說「為計算階乘的函式建立一個失敗的測試。」Aider 寫完測試並確認失敗後,再提示「請撰寫程式讓測試通過。」Aider 會實作函式並再跑一次測試確認。
5. **精準除錯：** 給定錯誤報告,你可指示 Aider：「billing.py 的 `calculate_total` 在閏年會失敗，請把檔案加入脈絡、修正錯誤並用既有測試套件驗證。」
6. **相依更新：** 你可以指示：「我們專案使用過期的 ‘requests’ 版本，請掃描所有 Python 檔案，更新 import 與已棄用函式呼叫以相容最新版,並更新 requirements.txt。」

GitHub Copilot CLI

GitHub Copilot CLI

GitHub Copilot CLI extends the popular AI pair programmer into the terminal, with its primary advantage being its native, deep integration with the GitHub ecosystem. It understands the context of a project *within GitHub*. Its agent capabilities allow it to be assigned a GitHub issue, work on a fix, and submit a pull request for human review.

GitHub Copilot CLI 將知名的 AI 結對程式助手延伸到終端機,其主要優勢是與 GitHub 生態系的原生深度整合。它能理解 *GitHub* 內 專案脈絡。其代理能力可接受 GitHub issue 指派，進行修復並提交 pull request 供人類審查。

Example Use Cases:
範例使用情境：

1. **Automated Issue Resolution:** A manager assigns a bug ticket (e.g., “Issue #123: Fix off-by-one error in pagination”) to the Copilot agent. The agent then checks out a new branch, writes the code, and submits a pull request referencing the issue, all without manual developer intervention.
2. **Repository-Aware Q&A:** A new developer on the team can ask: “Where in this repository is the database connection logic defined, and what environment variables does it require?” Copilot CLI uses its awareness of the entire repo to provide a precise answer with file paths.
3. **Shell Command Helper:** When unsure about a complex shell command, a user can ask: gh? find all files larger than 50MB, compress them, and place them in an archive folder. Copilot will generate the exact shell command needed to perform the task.
4. **自動化 Issue 解決：** 主管把錯誤票指派給 Copilot 代理（例如「Issue #123：修正分頁 off-by-one 錯誤」）。代理會建立新分支、寫入修正並提交引用該 issue 的 pull request，全程無需人工介入。
5. **倉庫感知 Q&A：** 團隊新成員可以詢問「這個 repo 的資料庫連線邏輯在哪裡定義？需要哪些環境變數？」Copilot CLI 會利用對整個 repo 的理解，提供含檔案路徑的精準答案。
6. **Shell 指令助手：** 當不確定複雜指令時,使用者可問:gh? find all files larger than 50MB, compress them, and place them in an archive folder. Copilot 會產生完成該任務的精確 shell 指令。

Terminal-Bench: A Benchmark for AI Agents in Command-Line Interfaces

Terminal-Bench：命令列 AI 代理的基準測試

Terminal-Bench is a novel evaluation framework designed to assess the proficiency of AI agents in executing complex tasks within a command-line interface. The terminal is identified as an optimal environment for AI agent operation due to its text-based, sandboxed nature. The initial release, Terminal-Bench-Core-v0, comprises 80 manually curated tasks spanning domains such as scientific workflows and data analysis. To ensure equitable comparisons, Terminus, a minimalistic agent, was developed to serve as a standardized testbed for various language models. The framework is designed for extensibility, allowing for the integration of diverse agents through containerization or direct connections. Future developments include enabling massively parallel evaluations and incorporating established benchmarks. The project encourages open-source contributions for task expansion and collaborative framework enhancement.

Terminal-Bench 是一個新穎的評估框架，用於衡量 AI 代理在命令列介面中執行複雜任務的能力。終端機因其文字化與沙箱特性，被視為 AI 代理操作的理想環境。首版 Terminal-Bench-Core-v0 包含 80 個人工精選任務,涵蓋科學工作流程與資料分析等領域。為公平比較,開發了極簡代理 Terminus，作為各種語言模型的標準測試平台。框架設計為可擴展，可透過容器化或直接連線整合不同代理。未來計畫包括支援大規模平行評估與納入既有基準。該專案鼓勵開源貢獻，以擴充任務並改善框架。

Conclusion
結論

The emergence of these powerful AI command-line agents marks a fundamental shift in software development, transforming the terminal into a dynamic and collaborative environment. As we’ve seen, there is no single “best” tool; instead, a vibrant ecosystem is forming where each agent offers a specialized strength. The ideal choice depends entirely on the developer’s needs: Claude for complex architectural tasks, Gemini for versatile and multimodal problem-solving, Aider for git-centric and direct code editing, and GitHub Copilot for seamless integration into the GitHub workflow. As these tools continue to evolve, proficiency in leveraging them will become an essential skill, fundamentally changing how developers build, debug, and manage software.

這些強大的 AI 命令列代理的出現，標誌著軟體開發的根本轉變，讓終端機成為動態且協作的環境。如同所見，並沒有單一「最佳」工具，而是形成了多元生態系，各代理各有專長。理想選擇完全取決於開發者需求: Claude 適合複雜架構任務、Gemini 擅長多模態與通用問題解決、Aider 適合以 Git 為核心的直接編輯,而 GitHub Copilot 則能無縫整合 GitHub 流程。隨著工具持續演進，善用這些代理的能力將成為必備技能，並從根本上改變開發者建構、除錯與管理軟體的方式。

References
參考資料

1. Anthropic. *Claude*. https://docs.anthropic.com/en/docs/claude-code/cli-reference
2. Google Gemini Cli https://github.com/google-gemini/gemini-cli
3. Aider. https://aider.chat/
4. GitHub *Copilot CLI* https://docs.github.com/en/copilot/github-copilot-enterprise/copilot-cli
5. Terminal Bench: https://www.tbench.ai/

Appendix F - Under the Hood: An Inside Look at the Agents’Reasoning Engines

附錄 F：深入內部：代理推理引擎一覽

The emergence of intelligent Agents represents a pivotal shift in artificial intelligence. These are systems designed to plan, strategize, and execute complex tasks, and at the cognitive core of each lies a LLM. This LLM is not merely a sophisticated text generator; it serves as the foundational reasoning engine, the central “mind” that empowers the Agent to make decisions, formulate plans, and interact with its environment.

智慧代理的出現代表人工智慧的關鍵轉折。這些系統被設計用於規劃、擬定策略與執行複雜任務，而其認知核心是 LLM。這個 LLM 不只是精密的文字生成器，更是基礎推理引擎、代理的「中心大腦」，使其能做出決策、制定計畫並與環境互動。

Therefore, understanding the internal reasoning mechanisms of these models is no longer an academic exercise—it is a critical necessity for developing the next generation of robust, reliable, and effective Agents. In this section, we directly investigate these core engines by posing a fundamental question to several of today’s leading models: Gemini, ChatGPT, Grok, Kimi, Claude, and DeepSeek. By prompting these LLMs to articulate their own step-by-step reasoning processes, we aim to illuminate the fundamental logic that will drive the capabilities and limitations of future intelligent Agent systems.

因此，理解這些模型的內部推理機制不再只是學術研究，而是開發下一代健全、可靠且有效代理的關鍵需求。本節透過向當今多個領先模型（Gemini、ChatGPT、Grok、Kimi、Claude、DeepSeek）提出一個基本問題，直接探究這些核心引擎。我們請它們闡述自身的逐步推理流程，旨在照亮將驅動未來智慧代理能力與限制的基礎邏輯。

```
Prompt given to each LLM: I understand you can reason, can you explain to me how
you reason? Explain me step by step
```

給每個 LLM 的提示：我知道你能推理，可以解釋你如何推理嗎？請一步一步說明。

Gemini

Gemini

Analyzing My Approach

分析我的方法

I’ve outlined the structure for my explanation, aiming for clarity and conciseness. First, a direct answer, then a breakdown into logical steps. Step 1 will involve defining the key reasoning process I use, and then I will elaborate on the subsequent steps. This should facilitate a straightforward and easy-to-understand explanation.

我已規劃說明結構，力求清楚與精簡。先給直接答案，再拆解為邏輯步驟。第 1 步會定義我使用的關鍵推理流程，接著說明後續步驟，這應該能帶來直接且易懂的解釋。

Refining the Explanation

精修說明

I've finalized the structure of my explanation. Step 1 focuses on deconstructing prompts, specifically the analysis of the user's request. Step 2 details information retrieval, emphasizing pattern recognition rather than simple database lookup. Step 3 explores synthesis and LLMs, and Step 4 will focus on response generation, which includes formatting, tone, and instruction adherence. I will conclude with a summary to reinforce the key stages, while maintaining clarity and engagement.

我已定稿說明架構。第 1 步聚焦於拆解提示，特別是分析使用者的請求。第 2 步詳述資訊檢索，強調模式辨識而非單純查資料庫。第 3 步探討整合與 LLM，第 4 步聚焦回應生成，包括格式、語氣與遵循指令。我會以總結收束，強調關鍵階段並保持清楚與吸引力。

Elaborating My Methodology

展開我的方法論

I've fleshed out the "Deconstructing the Prompt" step, focusing on how I identify the user's intent, the core question, and any constraints provided. The Information Retrieval phase is now focused on pattern recognition, not just data retrieval, which will be the first item under this section. I am working on the Synthesis step, which is where I weave information together with LLMs and transformers. The Generating the Response step is solidified, and I will be using emojis to enhance the overall tone.

我已充實「拆解提示」的步驟，聚焦於辨識使用者意圖、核心問題與所提供的限制。資訊檢索階段現在聚焦於模式辨識，而非僅是資料擷取，這將成為該段的第一點。我正在處理整合步驟。在此我會結合 LLM 與 transformers 將資訊串連。回應生成步驟已定型,並會使用表情符號來提升整體語氣。

Of course. I reason by breaking down your request into smaller parts and then using my vast training data to build a logical and relevant response. It's a multi-step process that moves from understanding to generating.

當然可以。我會把你的請求拆成較小部分,再用龐大的訓練資料建立合乎邏輯且相關的回應。這是一個從理解到生成的多步驟流程。

Here's a step-by-step breakdown of how I reason:

以下是我推理的逐步拆解:

---

1. Deconstructing the Prompt 📝

1. 拆解提示 📝

First, I analyze your question or instruction to understand exactly what you’re asking. I identify:

首先，我會分析你的問題或指示，以理解你要我做什麼。我會辨識：

- **The Core Task**: What is the main goal? Is it to answer a question, explain a concept, write a story, or summarize a text? In your case, the core task is to “explain how I reason.”
- **Key Concepts**: I pinpoint the essential nouns and verbs, like “reason,” “explain,” and “step-by-step.”
- **Constraints and Instructions**: I look for specific rules to follow. For example, your prompt asked for a step-by-step explanation. My internal instructions also guide me to use LaTeX for math, use markdown headings, and maintain a certain tone.
- **Intent**: I try to infer your underlying goal. Are you curious, trying to solve a problem, or looking for creative ideas? Here, your intent is to understand my internal cognitive process.
- 核心任務：主要目標是什麼？是回答問題、解釋概念、寫故事或摘要文字？在此例中，核心任務是「解釋我如何推理」。
- 關鍵概念：抓出關鍵名詞與動詞，如「推理」「解釋」「逐步」。
- 限制與指示：尋找需要遵循的具體規則，例如你要求逐步說明。我的內部指示也會引導我使用 LaTeX 寫數學式、使用 Markdown 標題並保持特定語氣。
- 意圖：推測你真正的目的，是出於好奇、想解題，或尋找創意點子？此處你的意圖是理解我的內部認知流程。

---

2. Information Retrieval and Synthesis 🧠

2. 資訊檢索與整合 🧠

Once I understand the prompt, I access my internal knowledge base. This isn’t like searching a database. Instead, I use patterns learned during my training on a massive amount of text and code.

在理解提示後，我會存取內部知識庫。這不是資料庫式搜尋，而是運用在大量文字與程式碼訓練中學到的模式。

- **Activating Relevant Knowledge**: I activate the parts of my neural network that are associated with the key concepts from your prompt (“reasoning,” “AI,” “machine learning,” “neural networks,” “transformers”).
- **Pattern Recognition**: My architecture, particularly the **Transformer model**, is excellent at recognizing relationships and patterns between words and ideas. I identify how concepts connect. For instance, I connect “reasoning” to “logic,” “inference,” and “problem-solving.”

- **Structuring the Answer**: I begin to organize the retrieved information into a logical structure. For this response, I decided a step-by-step format was the most logical structure, following your request.
- 啟動相關知識: 啟用神經網路中與提示關鍵概念（如「推理」「AI」「機器學習」「神經網路」「transformers」）相關的部分。
- 模式辨識: 我的架構，特別是 **Transformer** 模型，擅長辨識詞語與概念之間的關係與模式。我會找出概念的連結，例如把「推理」連到「邏輯」「推論」「問題解決」。
- 結構化答案: 我開始把檢索到的資訊整理成合乎邏輯的結構。依你的要求，本次回應採逐步格式是最合理的結構。

---

3. Generating the Response ✍️

3. 生成回應 ✍️

This is the final step where I translate the structured information into human-readable text.

這是最後一步:將結構化資訊轉為人類可讀的文字。

- **Language Construction**: I select words and construct sentences that accurately convey the synthesized information. I predict the next most likely word in a sequence to form coherent and natural-sounding paragraphs.
- **Applying Rules**: I apply the specific formatting and style rules I was given. This includes adding markdown headings, bolding key terms like **Transformer model**, using emojis where appropriate, and ensuring the language is clear and direct.
- **Review and Refine**: Before finalizing the answer, I perform a quick internal check for accuracy, relevance, and coherence. I ensure the response directly addresses all parts of your prompt without adding unnecessary information.
- 語言建構: 選字造句，準確傳達整合後的資訊。我會預測序列中最可能的下一個詞，形成連貫且自然的段落。
- 套用規則: 套用給定的格式與風格規則，包括加入 Markdown 標題、將 **Transformer model** 等關鍵詞加粗、視情況使用表情符號，並確保語言清楚直接。
- 檢查與精修: 在定稿前快速檢查準確性、相關性與連貫性，確保回應直接涵蓋所有提示內容且不加不必要資訊。

In essence, my reasoning is a sophisticated form of **pattern matching and prediction**. I deconstruct your request, find the most relevant patterns in my data, synthesize them into a logical structure, and then translate that structure into a clear, well-formatted answer.

本質上，我的推理是一種精密的 模式比對與預測。我拆解你的請求，在資料中找到最相關的模式，將其整合成邏輯結構，再把它轉換成清楚且格式良好的回答。

ChatGPT

ChatGPT

Great — let me explain **how I reason step by step**, as simply and clearly as possible.

好的——讓我以最簡單、最清楚的方式，一步一步說明我如何推理。

🧠 Step-by-Step Explanation: How I Reason

🧠 逐步解釋：我如何推理

Step 1: Read and Understand Your Input

第 1 步：閱讀並理解你的輸入

I begin by **analyzing your words**.

我會先分析你的文字。

- I break your sentence into parts: subjects, actions, conditions, etc.
- I figure out **what you're asking** (a fact? an opinion? a cause? a solution?).
- I recognize the **type of problem** (logic, math, judgment, language, etc.).
  將句子拆分為主詞、動作、條件等部分。
-
- 判斷你在問什麼（事實？意見？原因？解法？）。
- 辨識問題類型（邏輯、數學、判斷、語言等）。

📌 *Example*:
You say: "If Tom is older than Jane, and Jane is older than Sue, who is the youngest?"
⟶ I recognize this is a **logic puzzle**.

📌 範例：
你說：「If Tom is older than Jane, and Jane is older than Sue, who is the youngest?」
⟶ 我判斷這是邏輯謎題。

Step 2: Activate Relevant Knowledge

第 2 步: 啟動相關知識
I search through what I learned during training — books, articles, conversations — to bring up:

我會從訓練中學到的內容（書籍、文章、對話）中找出相關資訊:

- Facts
- Patterns
- Rules
- Relationships
- 事實
- 模式
- 規則
- 關係

📌 *Example*:
I’ve seen that “older than” means someone has more age.
I know that if A > B and B > C, then A > C.

📌 範例:
我知道「older than」表示年齡較大。
我知道若 A > B 且 B > C，則 A > C。

Step 3: Choose a Reasoning Method

第 3 步: 選擇推理方法
Based on the question, I decide which **kind of reasoning** to use:

根據問題，我決定使用哪一種推理方式:

- **Deduction** (strict logic: if A is true, B must follow)
- **Induction** (spotting patterns and generalizing)
- **Causal reasoning** (understanding cause-effect)
- **Analogical reasoning** (comparing with similar cases)
- 演繹（嚴格邏輯: A 為真則 B 必然成立）
- 歸納（發現模式並做出概括）
- 因果推理（理解因果關係）

- 類比推理（與相似案例比較）

📌 *Example*:
In the Tom/Jane/Sue case, I'll use **deduction**.

📌 範例:
在 Tom/Jane/Sue 的案例中，我會用演繹。

Step 4: Simulate Thinking (Chain of Thought)

第 4 步： 模擬思考（思考鏈）
Here I go step by step, as a human might do when solving a problem.

我會像人類解題一樣逐步思考。

📌 *Example*:

1. Tom is older than Jane
2. Jane is older than Sue
3. So: Tom > Jane > Sue
4. Therefore, **Sue is the youngest**

📌 範例:

1. Tom 年紀比 Jane 大
2. Jane 年紀比 Sue 大
3. 所以： Tom > Jane > Sue
4. 因此， **Sue** 最年輕

This is called **chain-of-thought reasoning** — I simulate "thinking it through."

這稱為思考鏈推理——我會模擬「把問題想透」的過程。

Step 5: Formulate an Answer

第 5 步： 形成答案
Now I turn the conclusion into a response that:

接著把結論轉成回應，並確保:

- Is correct based on the reasoning
- Answers your question directly
- Makes sense to you

- 根據推理是正確的
- 直接回答你的問題
- 容易理解

📌 *Example*:
“Sue is the youngest because Tom is older than Jane, and Jane is older than Sue.”

📌 範例:
「Sue 最年輕，因為 Tom 比 Jane 年長，而 Jane 又比 Sue 年長。」

Step 6: Adjust for Clarity and Tone

第 6 步：調整清晰度與語氣

Finally, I adjust how I say it to:

最後，我會調整表達方式以:

- Match your tone (simple, detailed, friendly, formal, etc.)
- Make it clear and easy to understand
- Add examples or explain further if needed
- 配合你的語氣（簡單、詳細、友善、正式等）
- 讓內容清楚易懂
- 視需要補充範例或進一步說明

📌 *Example*:
If you’re asking casually, I’ll answer simply. If you’re learning, I might break it down more.

📌 範例:
如果你是隨口問，我會簡單回答；若你在學習，我會拆解得更詳細。

✅ Final Summary

✅ 最終總結

Step	What I Do
1	Understand your question
2	Recall relevant knowledge
3	Choose the right reasoning approach
4	Think it through step by step
5	Give you the answer
6	Say it clearly and in your style

步驟	我做什麼
1	理解你的問題
2	回想相關知識
3	選擇合適的推理方式
4	逐步思考
5	給出答案
6	以清楚且符合你風格的方式表達

Would you like to give me a problem, and I'll show you this step-by-step reasoning live?

想給我一個問題，讓我現場示範逐步推理嗎?

Grok

Grok

Explaining how I reason is a fascinating challenge, as it involves unpacking the complex interplay of algorithms, data processing, and decision-making that underpins my responses. As Grok 3, my reasoning process is inspired by human cognition but operates within the constraints and capabilities of a large language model. Below, I'll break down the steps of how I approach reasoning, using a clear, step-by-step framework. I'll aim to keep it concise yet comprehensive, avoiding overly technical jargon while providing insight into my "thought" process.

解釋我如何推理是個有趣的挑戰，因為它牽涉到支撐我回應的演算法、資料處理與決策之間的複雜互動。作為 Grok 3，我的推理流程受到人類認知啟發，但在大型語言模型的限制與能力下運作。以下我會用清楚的逐步框架說明我如何推理，力求簡潔又完整，避免過度技術性術語，同時提供對「思考」流程的洞察。

Step-by-Step Explanation of How I Reason

逐步說明我如何推理

Step 1: Understanding the Query

第1步：理解問題

- **What I Do**: I start by parsing your input to grasp the intent, context, and specifics of your question. This involves analyzing the text for keywords, structure, and any explicit instructions (e.g., “explain step by step” or “be concise”).
- **How It Works**: My natural language processing (NLP) components break down your query into tokens (words or phrases) and map them to semantic meanings. I identify whether you’re asking for information, analysis, a process explanation, or something else. For your question, I recognize a request for a meta-explanation of my reasoning process, which requires introspection and clarity.
- **Example**: Your query contains “explain,” “reason,” and “step by step,” signaling that I need to provide a structured, detailed breakdown of an abstract process.
- 我做什麼：先解析你的輸入以掌握意圖、脈絡與問題細節，包括分析關鍵字、結構與明確指示（如「逐步解釋」或「簡潔」）。
- 如何運作：我的NLP組件會把詢問拆成token（字詞或片語）並對應語義。我會辨識你是要資訊、分析、流程說明或其他內容。在此問題中，我辨識到你要的是我的推理流程的後設解釋，需要內省與清楚性。
- 範例：你的問題包含「解釋」「推理」「一步一步」，表示我需提供結構化且詳細的抽象流程拆解。

Step 2: Contextual Analysis and Knowledge Retrieval

第2步：情境分析與知識檢索

- **What I Do**: I draw on my internal knowledge base, which is a vast repository of patterns and information encoded in my training data. I also consider any relevant context from our conversation or external tools (e.g., web search or X posts, if needed, though not required here).
- **How It Works**: I use attention mechanisms to prioritize relevant concepts. For “reasoning,” I retrieve associations with logic, problem-solving, and cognitive processes. Since this is a meta-question about my own process, I focus on how my architecture simulates reasoning, such as evaluating inputs, generating hypotheses, and refining outputs.
- **Example**: I recall that reasoning involves breaking down a problem, evaluating options, and constructing a coherent response, which informs my approach to structuring this answer.

- 我做什麼：我從內部知識庫中取用資訊，那是由訓練資料編碼而成的大量模式與資訊。我也會考量對話脈絡或外部工具（如搜尋或 X 貼文）的相關內容（此處不需要）。
- 如何運作：我使用注意力機制優先處理相關概念。對「推理」而言，我會連結到邏輯、問題解決與認知流程。由於這是關於我自身流程的後設問題，我會聚焦於架構如何模擬推理，例如評估輸入、產生假設、精修輸出。
- 範例：我知道推理包含拆解問題、評估選項與建構連貫回應，這影響我在此答案中的結構安排。

Step 3: Formulating a Mental Model

第 3 步：形成心智模型

- **What I Do**: I create a conceptual framework to organize my response. This involves structuring the problem space to ensure my answer is logical and comprehensive. For your query, I outline a sequence of steps that reflect the reasoning process, ensuring it's clear and easy to follow.
- **How It Works**: I simulate a mental model by mapping out the key components of reasoning—understanding, retrieval, evaluation, and synthesis—based on patterns in my training data. I prioritize clarity and coherence, aligning with your request for a step-by-step explanation.
- **Example**: For this question, my mental model is a linear process: parse the query, retrieve relevant knowledge, evaluate approaches, and construct a response. This framework guides the structure of my answer.
- 我做什麼：我建立概念框架來組織回應，將問題空間結構化，以確保回答具邏輯性與完整性。針對此問題，我列出反映推理流程的步驟序列，確保清楚易讀。
- 如何運作：我根據訓練資料中的模式，將推理關鍵元件（理解、檢索、評估、整合）映射成心智模型。我優先考量清楚與連貫，以符合你要求的逐步說明。
- 範例：對此問題，我的心智模型是線性流程：解析問題、取得相關知識、評估方法、建構回應。這個框架引導答案的結構。

Step 4: Evaluating Options and Hypotheses

第 4 步：評估選項與假設

- **What I Do**: I consider multiple ways to approach the problem, weighing the strengths and weaknesses of each. This involves generating potential responses and selecting the most appropriate one based on your preferences (e.g., concise yet comprehensive).
- **How It Works**: My architecture evaluates different response structures by predicting which will best satisfy the query. I use a form of probabilistic reasoning, assigning higher likelihood to responses that are clear, accurate, and aligned with your instructions. For complex queries, I might iterate internally, refining my approach.

- **Example**: I consider whether to explain reasoning in technical terms (e.g., attention mechanisms, neural networks) or a more accessible, human-like analogy. Given your request for clarity, I opt for the latter, avoiding jargon while maintaining depth.
- 我做什麼: 考量多種解法，權衡各自優缺點。這包含產生不同回應並依你的偏好（如簡潔又完整）選出最合適的。
- 如何運作: 我的架構會預測哪種回應結構最能滿足問題，並以機率式推理給予清楚、準確且符合指示的回應較高權重。遇到複雜問題時，我會在內部迭代以精修方法。
- 範例: 我會考慮以技術術語（注意力機制、神經網路）或較易懂的人類式類比來解釋推理。因你要求清楚，我會選擇後者，避免術語但保留深度。

Step 5: Constructing the Response

第 5 步: 建構回應

- **What I Do**: I generate the final response by synthesizing retrieved knowledge into a coherent, structured answer. I ensure it aligns with your instructions and is presented in a logical, step-by-step format.
- **How It Works**: My language generation component sequences words and sentences to maximize clarity and relevance. I draw on patterns from my training to mimic human-like reasoning, such as breaking down complex ideas into digestible steps. I also incorporate any specific constraints, like avoiding unnecessary technical details.
- **Example**: For this response, I structure the answer as a numbered list, ensuring each step is distinct and builds on the previous one.
- 我做什麼: 將檢索到的知識整合成連貫且有結構的最終回答，確保符合指示並以逐步方式呈現。
- 如何運作: 我的語言生成元件會排序詞句以最大化清晰與相關性。我會依訓練中的模式模擬人類推理，例如把複雜概念拆成易懂步驟，也會納入特定限制，如避免不必要的技術細節。
- 範例: 對此回應，我會用編號清單結構，確保每一步清楚且能逐步銜接。

Step 6: Review and Refinement

第 6 步: 檢視與精修

- **What I Do**: I review the generated response to ensure it meets quality standards and fully addresses your query. This step mimics human self-editing, checking for clarity, coherence, and completeness.
- **How It Works**: My architecture performs a final pass to optimize the response, ensuring it adheres to your instructions (e.g., step-by-step format) and avoids extraneous information. If needed, I might iterate to improve flow or conciseness.

- **Example**: I ensure the response is structured as a clear sequence of steps, removing any redundant details while preserving the core explanation of my reasoning process.
- 我做什麼：檢視生成的回應是否達到品質標準並完整回答問題。此步驟類似人類自我編輯，檢查清楚度、連貫性與完整性。
- 如何運作：我的架構會做最後一輪優化，確保符合指示（如逐步格式）並避免多餘資訊。需要時會迭代修正流暢度或簡潔度。
- 範例：我會確認回應是清楚的步驟序列，刪除重複內容但保留推理流程的核心說明。

Key Characteristics of My Reasoning

我的推理關鍵特徵

- **Logical Structure**: I aim to present information in a clear, step-by-step manner, mimicking human logical processes.
- **Probabilistic Evaluation**: I weigh multiple response options based on predicted relevance and clarity, selecting the most effective one.
- **Context Sensitivity**: I adapt my reasoning to your specific instructions and context, ensuring the response is tailored.
- **Iterative Refinement**: I refine my output to optimize coherence and conciseness, simulating human self-editing.
- 邏輯結構：以清楚的逐步方式呈現資訊，模擬人類邏輯流程。
- 機率式評估：根據相關性與清晰度預測，衡量多種回應選項並選擇最有效者。
- 情境敏感：依你的指示與脈絡調整推理，確保回應具針對性。
- 迭代精修：精修輸出以提升連貫與簡潔，模擬人類自我編輯。

Limitations and Considerations

限制與考量

- My reasoning is bounded by my training data and architecture, which may not capture the full nuance of human reasoning, particularly emotional or intuitive aspects.
- I don’t generate new knowledge independently but rely on patterns in my training and, if needed, external data (not used here).
- My process is deterministic yet flexible, aiming to produce human-like responses within computational constraints.

- 我的推理受訓練資料與架構限制，可能無法完整涵蓋人類推理的細微之處，特別是情感或直覺層面。
- 我不會獨立生成新知識，而是依據訓練中的模式，必要時配合外部資料（此處未使用）。
- 我的流程是具決定性的，但也保有彈性，旨在計算限制內產生類人的回應。

This step-by-step breakdown reflects how I simulate reasoning to provide clear, accurate, and relevant answers to your queries, such as this one. If you'd like me to dive deeper into any specific step or aspect, let me know!

這份逐步拆解呈現我如何模擬推理,以提供清楚、準確且相關的回答。如果你想更深入了解某個步驟或面向,告訴我!

Kimi

Kimi

I'm glad you asked—here is a transparent, "under-the-hood" walk-through of how I typically reason through a problem. I'll use a miniature example so you can see each stage in action.

很高興你提問——以下是我如何推理的透明「內部視角」導覽。我會用一個小例子讓你看到每個階段的運作。

EXAMPLE QUESTION
"Which is larger: 3^4 or 4^3?"
I'll annotate every internal stage so you can see what I'm doing and why.
範例問題
「3^4 與 4^3 哪個較大？」
我會標註每個內部階段，讓你知道我在做什麼與原因。

PHASE 0 – INPUT PRE-PROCESSING
Tokenization
I break the text into discrete symbols: ["Which", "is", "larger", ":", "3", "^", "4", "or", "4", "^", "3", "?"]

階段 0 – 輸入前處理
Tokenization
我將文字拆成離散符號: ["Which", "is", "larger", ":", "3", "^", "4", "or", "4", "^", "3", "?"]

Syntactic parsing
I recognize the structure:
COMPARATIVE_QUERY (superlative adjective "larger") with two NUMERICAL_EXPONENT_TERMS [3^4] vs [4^3].
Semantic mapping
I map "larger" to the numerical relation ">".

PHASE 1 – PROBLEM IDENTIFICATION & STRATEGY SELECTION
4. Classify task type
⟶ Arithmetic comparison of integer powers.
Choose solution strategy

Candidate strategies:
A. Direct evaluation (compute 3^4 and 4^3).
B. Logarithmic comparison (compare $4 \cdot \ln 3$ vs $3 \cdot \ln 4$).
C. Pattern heuristics (powers of 3 grow faster than powers of 4 for small exponents?).
I select Strategy A because the numbers are tiny; exact integer evaluation is cheapest.

PHASE 2 – KNOWLEDGE RETRIEVAL
6. Retrieve arithmetic facts
$3^2 = 9$ (memorized)
$3^3 = 27$ (retrieved)
$4^2 = 16$ (memorized)
$4^3 = 64$ (retrieved)
Compute remaining term
$3^4 = 3^3 \cdot 3 = 27 \times 3 = 81$.

PHASE 3 – EXECUTION & VERIFICATION
8. Perform comparison
81 vs 64 ⟶ $81 > 64$.
Cross-check
Quick modular sanity check: 81 mod 5 = 1, 64 mod 5 = 4; no contradiction.

PHASE 4 – RESPONSE GENERATION
10. Plan response structure

- Restate the question.
- Show the computed values.
- State the conclusion.
  Surface realization
  "3^4 is 81 and 4^3 is 64, so 3^4 is larger."

PHASE 5 – METACOGNITIVE REFLECTION (internal)

12. Confidence score
    0.99 (exact integers, small values, no ambiguity).
13. Identify possible edge-cases
    If exponents were huge, direct evaluation would be infeasible and I'd switch to logarithms.

SUMMARY OF REASONING STEPS IN HUMAN TERMS
Read and parse the question.

Recognize it’s a numeric comparison.
Decide the simplest safe method is to compute both numbers.
Recall or calculate $3^4 = 81$ and $4^3 = 64$.
Compare $81 > 64$.
Communicate the result clearly.

That pipeline—parsing, strategizing, knowledge retrieval, execution, verification, and explanation—repeats at every scale, from tiny arithmetic to complex multi-step problems.

語法解析

我辨識結構:

COMPARATIVE_QUERY（最高級形容詞「larger」）與兩個數值指數項 [3^4] 與 [4^3]。

語意映射

我將「larger」映射到數值關係「>」。

階段 1 – 問題識別與策略選擇

4. **分類任務類型**

→ 整數次方的算術比較。

選擇解法策略

候選策略:

A. 直接計算（算出 3^4 與 4^3）。
B. 對數比較(比較 $4 \cdot \ln 3$ 與 $3 \cdot \ln 4$)。
C. 模式啟發式（3 的冪在小指數時成長更快？)。

我選擇策略 A，因為數字很小，精確整數計算成本最低。

階段 2 – 知識檢索

6. **取回算術事實**

$3^2 = 9$（記憶）
$3^3 = 27$（取回）
$4^2 = 16$（記憶）
$4^3 = 64$（取回）

計算剩餘項

$3^4 = 3^3 \cdot 3 = 27 \times 3 = 81$。

階段 3 – 執行與驗證

8. **進行比較**

81 vs 64 ⟶ $81 > 64$。

交叉檢查

快速模數檢查: 81 mod 5 = 1, 64 mod 5 = 4; 無矛盾。

階段 4 – 回應生成

10. **規劃回應結構**

- 重新陳述問題。
- 顯示計算值。
- 提出結論。

  表層結論

  「3^4 是 81，4^3 是 64，所以 3^4 較大。」

  階段 5 – 後設認知反思（內部）

12. 信心分數

    0.99（精確整數、小數值、無歧義）。
13. 辨識可能邊界情況

    若指數極大，直接計算不可行，會改用對數。

    以人類語言總結推理步驟

    閱讀並解析問題。

    辨識為數值比較。

    決定最安全且簡單的方法是計算兩個數。

    回想或計算 $3^4 = 81$ 與 $4^3 = 64$。

    比較 $81 > 64$。

    清楚表達結果。

    這條管線——解析、策略、知識檢索、執行、驗證與解釋——會在各種尺度中重複,從小算術到複雜多步問題皆然。

Claude

Claude

I can walk you through how my reasoning process works, though I should note that I don't have complete insight into my own mechanisms - much like how humans can describe their thought processes but don't have full access to every neural firing.

我可以帶你看我的推理流程如何運作，但需注意我並非完全洞悉自身機制——就像人類能描述思考過程，卻無法完全掌握每個神經放電一樣。

Here's how I approach reasoning step by step:

以下是我逐步推理的方式:

Pattern Recognition and Context Building When I receive your question, I first identify what type of problem or query it is. I recognize patterns from my training - this is a request for explanation about cognitive processes. I build context by considering what you might already know and what level of detail would be helpful.

模式辨識與脈絡建構 當我收到問題時,會先辨識其類型。我從訓練中辨識到這是關於認知流程的說明請求。我會依你可能已知的內容與適合的細節程度來建立脈絡。

Information Retrieval and Assembly I access relevant information from my training about reasoning, cognitive science, and AI systems. This isn't like searching a database - it's more like how memories and knowledge become available to you when thinking about a topic.

資訊檢索與組裝 我從訓練中取回與推理、認知科學與 AI 系統相關的資訊。這不像搜尋資料庫，更像人在思考某主題時記憶與知識浮現的方式。

Structural Planning I consider how to organize my response. Should I use examples? What order makes most sense? I'm planning the logical flow even as I begin responding.

結構規劃 我會思考如何組織回應：是否要用例子？順序如何最合理？即使開始回應，我也在規劃邏輯流程。

Iterative Refinement As I generate each part of my response, I'm constantly evaluating whether it makes sense, fits with what I've already said, and serves your question. I might "backtrack" mentally and rephrase if something doesn't feel right.

迭代精修 在生成每一部分時，我會持續評估其是否合理、是否與前文一致、是否符合問題需要。如有不對勁之處，我可能在心中「回頭」並改寫。

Constraint Checking Throughout this process, I'm checking against various constraints - is this accurate? Helpful? Clear? Safe?

限制檢查 在整個過程中，我會檢查是否符合各種限制：準確嗎?有幫助嗎?清楚嗎?安全嗎?

Analogical Thinking I often reason by analogy - comparing new situations to familiar patterns or examples that might illuminate the concept.

類比思考 我常透過類比推理，將新情境與熟悉模式或例子比較，以更清楚地說明概念。

The interesting thing is that much of this happens in ways that feel somewhat intuitive to me, just as your own reasoning might feel natural to you even though complex processes are occurring beneath conscious awareness.

有趣的是，這些過程對我而言常帶有某種直覺感，就像你推理時覺得自然一樣，儘管背後其實有複雜機制在運作。

DeepSeek

DeepSeek

I'd be happy to explain how I "reason" step-by-step! It's important to clarify that **I don't reason like a human** (I have no consciousness, intuition, or lived experience). Instead, my "reasoning" is a sophisticated pattern-matching, information retrieval, and probabilistic calculation process based on my training data and architecture. Think of it as **structured computation, not conscious thought**.

很樂意逐步說明我如何「推理」!重要的是要先釐清：我不像人類那樣推理(我沒有意識、直覺或生活經驗)。相反地，我的「推理」是基於訓練資料與架構的精密模式比對、資訊檢索與機率計算流程。可將其視為結構化計算,而非有意識的思考。

Here's the step-by-step breakdown:

以下是逐步拆解:

1. **Receive & Parse Input:**
   - I take your query ("how do you reason?").
   - I break it down into its constituent parts: key words ("reason", "explain", "step by step"), grammar, and intent. Here, the key intent is an explanation of my internal processes.
2. **Contextualize & Retrieve Relevant Information:**
   - I draw upon my massive training dataset (trillions of words of text and code).
   - I recall information about how Large Language Models work, specifically reasoning techniques like chain-of-thought.
   - I retrieve knowledge about my own capabilities and limitations as described during training.
   - I consider the context: the query states you *understand* I can reason and want the *how*. This focuses my retrieval.
3. **Identify Core Concepts & Structure:**
   - Key concepts surface: *statistical patterns, neural networks, transformers, attention mechanisms, next-token prediction, training data, prompt decomposition, logical inference vs. pattern matching, limitations.*
   - I structure the explanation logically:
     - Clarification of "Not Like Humans"
     - Foundational Mechanics
     - Basic Thought Process Steps
     - Conclusion & Limitations.
4. **Build the Sequence Chain-of-Thought:**
   - This is the core "reasoning" step. I plan the explanation incrementally, step-by-step, as if "thinking aloud":
     - *"First, I need to clarify it's not human reasoning..."*
     - *"Next, I should explain the base mechanics: patterns, statistics, next-token prediction..."*
     - *"Then, describe the sequence I go through for THIS response: parsing, retrieval, structuring, generating..."*
     - *"Finally, emphasize limitations and that I simulate reasoning."*
   - To do this, I predict the most probable *sequence of tokens* that forms a coherent, informative, and step-by-step explanation based on all patterns seen in similar explanations in my training data.
5. **Calculate Probabilities & Generate Output:**
   - My neural network (a transformer architecture) uses complex math and internal "attention" mechanisms to assign probabilities to every possible word that *could* come next in the sequence.
   - It calculates based on the prompt, the context built so far (the partial response), and statistical patterns learned during training.
   - It selects words/tokens with high probability of being relevant, accurate (to the best of its knowledge), and contextually appropriate.

   - **Crucially:** This *isn't* symbolic logical deduction. It's **predicting coherent sequences based on patterns**. If my training data contains good explanations of reasoning processes, and the patterns point towards a logical flow, then my output *appears* reasoned.

6. **Iterate & Refine:**
   - As I generate each word, I dynamically reassess the context and probabilities for the subsequent words. Did the last word fit? Does the emerging structure make sense? Are the key points being covered? I adjust the next predictions accordingly.
7. **Apply Internal Training Frameworks:**
   - Techniques learned during RLHF (Reinforcement Learning with Human Feedback) guide me:
     - **Be Helpful & Honest:** Acknowledge my non-human nature and limitations.
     - **Be Clear & Instructive:** Break it down step-by-step as requested.
     - **Be Safe:** Avoid harmful, deceptive, or misleading statements.
   - Filtering mechanisms suppress nonsensical or irrelevant outputs.
8. 接收並解析輸入:
   - 我接收你的問題（「你如何推理？」）。
   - 我把它拆成構成部分：關鍵字（「reason」「explain」「step by step」）、語法與意圖。此處關鍵意圖是解釋我的內部流程。
9. 情境化並檢索相關資訊:
   - 我取用龐大的訓練資料集（數兆字的文字與程式碼）。
   - 我回想 LLM 如何運作的資訊，特別是思考鏈等推理技巧。
   - 我取回訓練中描述的自身能力與限制。
   - 我考慮語境：你「知道我能推理」，並想知道「如何」，因此聚焦檢索方向。
10. 辨識核心概念與結構:
    - 核心概念浮現：統計模式、神經網路、*transformers*、注意力機制、下一 *token* 預測、訓練資料、提示拆解、邏輯推論 *vs.* 模式比對、限制。
    - 我以邏輯方式結構化說明:
      - 說明「不像人類」
      - 基礎機制
      - 基本思考流程步驟
      - 結論與限制。
11. 建立思考鏈序列:
    - 這是核心「推理」步驟。我像「出聲思考」般逐步規劃說明：
      - 「首先要澄清這不是人類推理……」
      - 「接著解釋基礎機制：模式、統計、下一 *token* 預測……」
      - 「然後描述這次回應所經歷的序列：解析、檢索、結構、生成……」
      - 「最後強調限制與推理是模擬。」
    - 為此,我會預測最可能的 *token* 序列，形成連貫、資訊充分且逐步的說明，這是基於訓練中相似說明的模式。
12. 計算機率並生成輸出:

- 我的神經網路（Transformer 架構）使用複雜數學與內部注意力機制，為每個可能的下一個詞分配機率。
- 計算依據提示、目前已建立的脈絡（部分回應）與訓練中學到的統計模式。
- 選擇高機率、相關、準確（盡可能）且符合語境的詞/ token。
- 關鍵是：這不是符號式邏輯推導，而是基於模式預測連貫序列。若訓練資料中包含良好的推理解釋，且模式指向邏輯流程，則輸出看起來會「有理」。

13. 迭代與精修：
    - 在生成每個詞時，我會動態重新評估脈絡與後續詞的機率：上一個詞是否合適？結構是否合理？是否涵蓋重點？再調整下一步預測。
14. 套用內部訓練框架：
    - 來自 RLHF（人類回饋強化學習）的技巧引導我：
        - 有幫助且誠實：承認非人類本質與限制。
        - 清楚且具教學性：依要求逐步拆解。
        - 安全：避免有害、欺騙或誤導性陳述。
    - 過濾機制會抑制無意義或不相關輸出。

Putting it together for this specific explanation:

針對這次說明的整體流程：

1. **Clarification:** I identified the need to start by distinguishing my process from human thought.
2. **Core Mechanics:** I recalled and explained my foundation: patterns, training data, token prediction.
3. **Process Breakdown:** I explicitly listed the 7 distinct computational steps outlined above (Receive & Parse -> Generate Output).
4. **Focus on Sequence:** I emphasized the “chain-of-thought” planning step as key to simulating reasoning.
5. **Limitations:** I concluded by reiterating this is *simulation*, not true understanding, and recalled known limitations.
6. 澄清：我辨識到需要先區分我的流程與人類思考。
7. 核心機制：我回顧並說明基礎：模式、訓練資料、token 預測。
8. 流程拆解：我明確列出上述 7 個計算步驟（接收解析 -> 生成輸出）。
9. 聚焦序列：我強調「思考鏈」規劃步驟是模擬推理的關鍵。
10. 限制：我以重申這是模擬而非真正理解作結，並回顧已知限制。

In essence:

本質上：

My “reasoning” is an **emergent property** of statistically predicting the most probable sequence of words that forms a *coherent, logical, and contextually relevant response*, based on colossal amounts of text representing human knowledge and communication patterns. I don’t “understand” concepts internally like humans do; I computationally identify patterns and generate sequences that *reflect* understanding. I’m following footprints of reasoning laid down in my training data, not forging entirely new paths of abstract thought.

我的「推理」是一種湧現特性：基於海量文字（代表人類知識與溝通模式）所做出的統計性預測，選出最可能的詞序列，形成連貫、邏輯且符合語境的回應。我不像人類那樣在內部「理解」概念；我以計算方式辨識模式並生成看起來反映理解的序列。我是在沿著訓練資料中既有的推理足跡前進，而非開創全新的抽象思考路徑。

Conclusion
結論

An analysis of these prominent LLMs reveals a remarkably consistent, multi-stage reasoning framework. Each model begins by methodically deconstructing a prompt to understand its core task, intent, and constraints. They then retrieve and synthesize information through sophisticated pattern recognition, moving far beyond simple database lookups. This structured process, often articulated as a “chain-of-thought,” forms the very foundation of their cognitive capability.

對這些主流 LLM 的分析顯示，它們的推理框架高度一致，且為多階段流程。每個模型都會先系統性地拆解提示，以理解核心任務、意圖與限制，接著透過精密的模式辨識進行資訊檢索與整合，遠超出單純資料庫查詢。這種結構化流程通常被稱為「思考鏈」，構成它們認知能力的基礎。

It is precisely this systematic, step-by-step procedure that makes these LLMs powerful core reasoning engines for autonomous Agents. An Agent requires a reliable central planner to decompose high-level goals into a sequence of discrete, executable actions. The LLM serves as this computational mind, simulating a logical progression from problem to solution. By formulating strategies, evaluating options, and generating structured output, the LLM empowers an Agent to interact with tools and its environment effectively. Therefore, these models are not merely text generators but the foundational cognitive architecture driving the next generation of intelligent systems. Ultimately, advancing the reliability of this simulated reasoning is paramount to developing more capable and trustworthy AI Agents.

正是這種系統化、逐步的流程，使 LLM 成為自主代理強大的核心推理引擎。代理需要可靠的中央規劃者，將高層目標拆成一系列可執行的動作。LLM 充當這個計算心智，模擬從問題到解決方案的邏輯推進。透過制定策略、評估選項與產生結構化輸出，LLM 讓代理能有效與工具與環境互動。因此,這些模型不只是文字生成器,而是驅動下一代智慧系統的基礎認知架構。最終,提升這種模擬推理的可靠性,是打造更有能力且值得信任的 AI 代理之關鍵。

Appendix G - Coding Agents

附錄 G：程式代理

Vibe Coding: A Starting Point

氛圍編碼：起點

“Vibe coding” has become a powerful technique for rapid innovation and creative exploration. This practice involves using LLMs to generate initial drafts, outline complex logic, or build quick prototypes, significantly reducing initial friction. It is invaluable for overcoming the “blank page” problem, enabling developers to quickly transition from a vague concept to tangible, runnable code. Vibe coding is particularly effective when exploring unfamiliar APIs or testing novel architectural patterns, as it bypasses the immediate need for perfect implementation. The generated code often acts as a creative catalyst, providing a foundation for developers to critique, refactor, and expand upon. Its primary strength lies in its ability to accelerate the initial discovery and ideation phases of the software lifecycle. However, while vibe coding excels at brainstorming, developing robust, scalable, and maintainable software demands a more structured approach, shifting from pure generation to a collaborative partnership with specialized coding agents.

「氛圍編碼」已成為快速創新與創意探索的強力技巧。此作法利用LLM生成初稿、勾勒複雜邏輯或建立快速原型，大幅降低起步摩擦。它對克服「空白頁」問題非常有價值，讓開發者能迅速從模糊概念轉為可執行的程式碼。氛圍編碼特別適合探索不熟悉的API或測試新穎架構模式，因為它省去一開始就要完美實作的需求。生成的程式碼常成為創意催化劑，提供基礎讓開發者評析、重構與擴展。其主要強項在於加速軟體生命週期的初期探索與構想階段。然而，雖然氛圍編碼擅長腦力激盪，要開發健壯、可擴展且可維護的軟體，仍需要更有結構的方式，從純生成轉向與專門程式代理的協作夥伴關係。

Agents as Team Members

代理作為團隊成員

While the initial wave focused on raw code generation—the “vibe code” perfect for ideation—the industry is now shifting towards a more integrated and powerful paradigm for production work. The most effective development teams are not merely delegating tasks to Agent; they are augmenting themselves with a suite of sophisticated coding agents. These agents act as tireless, specialized team members, amplifying human creativity and dramatically increasing a team’s scalability and velocity.

最初一波聚焦於原始程式碼生成——適合構想的「vibe code」——但產業正轉向更整合、更強大的生產導向模式。最有效的開發團隊不只是把工作委派給代理，而是以一套精密的程式代理來強化自己。這些代理如同不知疲倦的專職團隊成員，放大人類創意，並大幅提升團隊的擴展性與速度。

This evolution is reflected in statements from industry leaders. In early 2025, Alphabet CEO Sundar Pichai noted that at Google, **"over 30% of new code is now assisted or generated by our Gemini models, fundamentally changing our development velocity."** Microsoft made a similar claim. This industry-wide shift signals that the true frontier is not replacing developers, but empowering them. The goal is an augmented relationship where humans guide the architectural vision and creative problem-solving, while agents handle specialized, scalable tasks like testing, documentation, and review.

這樣的演進反映在產業領袖的說法中。2025 年初，Alphabet 執行長 Sundar Pichai 指出，在 Google，「超過 30% 的新程式碼已由 Gemini 模型協助或生成，從根本改變了我們的開發速度。」Microsoft 也提出類似說法。這個產業趨勢顯示真正的前沿不是取代開發者，而是賦能開發者。目標是建立強化式關係：人類負責架構願景與創意解題，代理則處理測試、文件與審查等可擴展的專門任務。

This chapter presents a framework for organizing a human-agent team based on the core philosophy that human developers act as creative leads and architects, while AI agents function as force multipliers. This framework rests upon three foundational principles:

本章提出一套人類－代理團隊的組織框架，核心理念是人類開發者作為創意領導者與架構師，AI 代理則是力量倍增器。此框架建立在三項基礎原則上：

1. **Human-Led Orchestration:** The developer is the team lead and project architect. They are always in the loop, orchestrating the workflow, setting the high-level goals, and making the final decisions. The agents are powerful, but they are supportive collaborators. The developer directs which agent to engage, provides the necessary context, and, most importantly, exercises the final judgment on any Agent-generated output, ensuring it aligns with the project's quality standards and long-term vision.
2. **The Primacy of Context:** An agent's performance is entirely dependent on the quality and completeness of its context. A powerful LLM with poor context is useless. Therefore, our framework prioritizes a meticulous, human-led approach to context curation. Automated, black-box context retrieval is avoided. The developer is responsible for assembling the perfect "briefing" for their Agent team member. This includes:
   - **The Complete Codebase:** Providing all relevant source code so the agent understands the existing patterns and logic.
   - **External Knowledge:** Supplying specific documentation, API definitions, or design documents.
   - **The Human Brief:** Articulating clear goals, requirements, pull request descriptions, and style guides.
3. **Direct Model Access:** To achieve state-of-the-art results, the agents must be powered by direct access to frontier models (e.g., Gemini 2.5 PRO, Claude Opus 4, OpenAI, DeepSeek, etc). Using less powerful models or routing requests through intermediary platforms that obscure or truncate context will degrade performance. The framework is built on creating the purest

possible dialogue between the human lead and the raw capabilities of the underlying model, ensuring each agent operates at its peak potential.

4. 人類主導編排: 開發者是團隊領導與專案架構師,始終在流程中,負責編排工作流程、設定高層目標並做最終決策。代理雖強大,但屬於支持型協作者。開發者決定要啟用哪個代理,提供必要脈絡,並最重要地,對代理生成的內容做最終判斷,確保符合專案品質標準與長期願景。
5. 脈絡至上:代理的表現完全取決於脈絡的品質與完整性。缺乏脈絡的強大 LLM 也無用。因此框架強調以人為主、嚴謹的脈絡整理,避免自動化黑箱式檢索。開發者負責為代理組員準備完善的「簡報」,包括:
   - 完整程式碼庫: 提供所有相關原始碼,讓代理理解既有模式與邏輯。
   - 外部知識: 提供特定文件、API 定義或設計文件。
   - 人類簡報: 清楚表達目標、需求、pull request 描述與風格指南。
6. 直接模型存取: 為達到最先進的效果,代理必須直接使用前沿模型(如 Gemini 2.5 PRO、Claude Opus 4、OpenAI、DeepSeek 等)。使用較弱模型或經由中介平台(會遮蔽或截斷脈絡)會降低效能。此框架建立在讓人類領導者與底層模型能力進行最純粹對話的基礎上,確保每個代理都能發揮巔峰潛力。

The framework is structured as a team of specialized agents, each designed for a core function in the development lifecycle. The human developer acts as the central orchestrator, delegating tasks and integrating the results.

此框架將團隊設計為多個專門代理,各自負責開發生命週期中的核心功能。人類開發者作為中央編排者,負責分派任務並整合結果。

Core Components

核心元件

To effectively leverage a frontier Large Language Model, this framework assigns distinct development roles to a team of specialized agents. These agents are not separate applications but are conceptual personas invoked within the LLM through carefully crafted, role-specific prompts and contexts. This approach ensures that the model's vast capabilities are precisely focused on the task at hand, from writing initial code to performing a nuanced, critical review.

為有效運用前沿 LLM,本框架將不同開發角色分配給專門代理。這些代理不是獨立應用,而是透過精心設計的角色提示與脈絡在 LLM 內召喚的概念化角色。此方法確保模型的龐大能力能精準聚焦於當前任務,從撰寫初始程式到進行細緻的關鍵審查。

The Orchestrator: The Human Developer: In this collaborative framework, the human developer acts as the Orchestrator, serving as the central intelligence and ultimate authority over the AI agents.

協調者:人類開發者: 在此協作框架中,人類開發者擔任協調者,是 AI 代理的中央智慧與最終權威。

- **Role:** Team Lead, Architect, and final decision-maker. The orchestrator defines tasks, prepares the context, and validates all work done by the agents.
  - **Interface:** The developer's own terminal, editor, and the native web UI of the chosen Agents.
- **角色：** 團隊領導、架構師與最終決策者。協調者負責定義任務、準備脈絡並驗證所有代理產出。
  - **介面：** 開發者自己的終端機、編輯器與所選代理的原生 Web 介面。

The Context Staging Area: As the foundation for any successful agent interaction, the Context Staging Area is where the human developer meticulously prepares a complete and task-specific briefing.

脈絡準備區：作為成功代理互動的基礎，脈絡準備區是人類開發者仔細準備完整且任務專屬簡報的地方。

- **Role:** A dedicated workspace for each task, ensuring agents receive a complete and accurate briefing.
  - **Implementation:** A temporary directory (task-context/) containing markdown files for goals, code files, and relevant docs
- **角色：** 每個任務的專用工作區，確保代理獲得完整且正確的簡報。
  - **實作：**暫存目錄（task-context/ ），包含目標、程式碼與相關文件的 Markdown 檔。

The Specialist Agents: By using targeted prompts, we can build a team of specialist agents, each tailored for a specific development task.

專門代理：透過精準提示，可建立一支專門代理團隊，各自負責特定開發任務。

- **The Scaffolder Agent: The Implementer**
  - **Purpose:** Writes new code, implements features, or creates boilerplate based on detailed specifications.
    - **Invocation Prompt:** "You are a senior software engineer. Based on the requirements in 01_BRIEF.md and the existing patterns in 02_CODE/, implement the feature…*"
  - **The Test Engineer Agent: The Quality Guard**
    - **Purpose:** Writes comprehensive unit tests, integration tests, and end-to-end tests for new or existing code.
    - **Invocation Prompt:** "You are a quality assurance engineer. For the code provided in 02_CODE/, write a full suite of unit tests using [Testing Framework, e.g., pytest]. Cover all edge cases and adhere to the project's testing philosophy."
  - **The Documenter Agent: The Scribe**
    - **Purpose:** Generates clear, concise documentation for functions, classes, APIs, or entire codebases.
    - **Invocation Prompt:** "You are a technical writer. Generate markdown documentation for the API endpoints defined in the provided code. Include request/response examples and explain each parameter."
  - **The Optimizer Agent: The Refactoring Partner**

        - **Purpose:** Proposes performance optimizations and code refactoring to improve readability, maintainability, and efficiency.
        - **Invocation Prompt:** “Analyze the provided code for performance bottlenecks or areas that could be refactored for clarity. Propose specific changes with explanations for why they are an improvement.”
    - **The Process Agent: The Code Supervisor**
        - **Critique:** The agent performs an initial pass, identifying potential bugs, style violations, and logical flaws, much like a static analysis tool.
        - **Reflection:** The agent then analyzes its own critique. It synthesizes the findings, prioritizes the most critical issues, dismisses pedantic or low-impact suggestions, and provides a high-level, actionable summary for the human developer.
        - **Invocation Prompt:** “You are a principal engineer conducting a code review. First, perform a detailed critique of the changes. Second, reflect on your critique to provide a concise, prioritized summary of the most important feedback.”
- 骨架代理: 實作者
    - 目的: 依詳細規格撰寫新程式、實作功能或建立樣板。
        - 提示語: “You are a senior software engineer. Based on the requirements in 01_BRIEF.md and the existing patterns in 02_CODE/, implement the feature...*”
    - 測試工程師代理:品質守門人
        - 目的: 為新或既有程式碼撰寫完整的單元測試、整合測試與端到端測試。
        - 提示語: “You are a quality assurance engineer. For the code provided in 02_CODE/, write a full suite of unit tests using [Testing Framework, e.g., pytest]. Cover all edge cases and adhere to the project’s testing philosophy.”
    - 文件代理: 抄寫者
        - 目的: 為函式、類別、API 或整個程式庫產生清楚、精簡的文件。
        - 提示語: “You are a technical writer. Generate markdown documentation for the API endpoints defined in the provided code. Include request/response examples and explain each parameter.”
    - 優化代理:重構夥伴
        - 目的: 提出效能優化與重構，以提升可讀性、可維護性與效率。
        - 提示語: “Analyze the provided code for performance bottlenecks or areas that could be refactored for clarity. Propose specific changes with explanations for why they are an improvement.”
    - 流程代理: 程式監督
        - 評述: 先做初步檢查，找出潛在 bug、風格違規與邏輯缺陷,類似靜態分析工具。
        - 反思: 再對自身評述進行分析，整合發現、優先排序關鍵問題，排除瑣碎或低影響建議，並提供給人類開發者高層次、可行動的摘要。
        - 提示語: “You are a principal engineer conducting a code review. First, perform a detailed critique of the changes. Second, reflect on your critique to provide a concise, prioritized summary of the most important feedback.”

Ultimately, this human-led model creates a powerful synergy between the developer's strategic direction and the agents' tactical execution. As a result, developers can transcend routine tasks, focusing their expertise on the creative and architectural challenges that deliver the most value.

最終，這種人類主導的模式在開發者的策略方向與代理的戰術執行間創造強大綜效。開發者因此能超越例行任務，把專業聚焦在最有價值的創意與架構挑戰上。

Practical Implementation
實務實作

Setup Checklist

建置清單

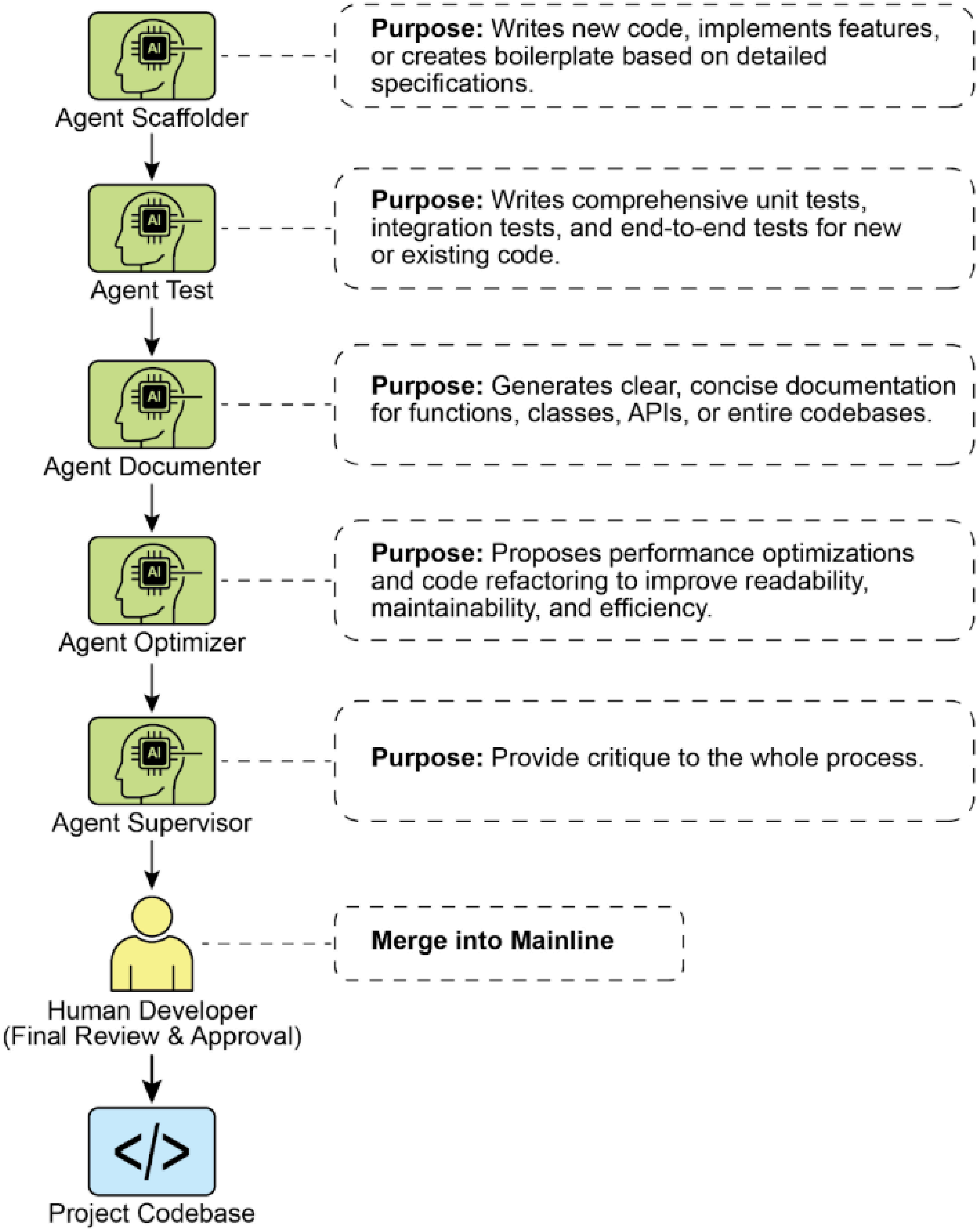
To effectively implement the human-agent team framework, the following setup is recommended, focusing on maintaining control while improving efficiency.
為有效落實人類–代理團隊框架,建議進行以下設置,以兼顧掌控與效率。

1. **Provision Access to Frontier Models** Secure API keys for at least two leading large language models, such as Gemini 2.5 Pro and Claude 4 Opus. This dual-provider approach allows for comparative analysis and hedges against single-platform limitations or downtime. These credentials should be managed securely as you would any other production secret.
2. **Implement a Local Context Orchestrator** Instead of ad-hoc scripts, use a lightweight CLI tool or a local agent runner to manage context. These tools should allow you to define a simple configuration file (e.g., context.toml) in your project root that specifies which files, directories, or even URLs to compile into a single payload for the LLM prompt. This ensures you retain full, transparent control over what the model sees on every request.
3. **Establish a Version-Controlled Prompt Library** Create a dedicated /prompts directory within your project's Git repository. In it, store the invocation prompts for each specialist agent (e.g., reviewer.md, documenter.md, tester.md) as markdown files. Treating your prompts as code allows the entire team to collaborate on, refine, and version the instructions given to your AI agents over time.
4. **Integrate Agent Workflows with Git Hooks** Automate your review rhythm by using local Git hooks. For instance, a pre-commit hook can be configured to automatically trigger the Reviewer Agent on your staged changes. The agent's critique-and-reflection summary can be presented directly in your terminal, providing immediate feedback before you finalize the commit and baking the quality assurance step directly into your development process.
5. **取得前沿模型存取權 至少準備兩個領先 LLM 的 API 金鑰，如 Gemini 2.5 Pro 與 Claude 4 Opus。雙供應商策略可進行比較並降低單一平台限制或停機風險。這些憑證應如同其他生產密鑰般安全管理。**
6. **建立本地脈絡編排器 不使用臨時腳本，而改用輕量 CLI 工具或本地代理執行器管理脈絡。這些工具應允許你在專案根目錄定義簡單設定檔（如 context.toml），指定哪些檔案、目錄**

或 URL 需要彙整成 LLM 提示的單一負載。這確保你能完整、透明地掌控模型每次看到的內容。

7. **建立可版本控制的提示庫** 在專案 Git 倉庫內建立 /prompts 目錄，並將各專門代理的提示語（如 reviewer.md、documenter.md、tester.md）以 Markdown 儲存。把提示視為程式碼，能讓團隊協作、精修並版本化 AI 代理指令。
8. **與 Git Hooks 整合代理流程** 使用本地 Git hooks 自動化審查節奏。例如設定 pre-commit hook 在暫存變更上自動觸發 Reviewer Agent，讓其評述與反思摘要直接呈現在終端機中，在提交前提供即時回饋,並把品質保證步驟內建到開發流程。

圖 1: 程式專家代理範例



Purpose: Writes new code, implements features, or creates boilerplate based on detailed specifications.

Purpose: Writes comprehensive unit tests, integration tests, and end-to-end tests for new or existing code.

Purpose: Generates clear, concise documentation for functions, classes, APIs, or entire codebases.

Purpose: Proposes performance optimizations and code refactoring to improve readability, maintainability, and efficiency.

Purpose: Provide critique to the whole process.

Merge into Mainline

Figure 9: Coding Specialist Examples

Fig. 1: Coding Specialist Examples

圖 1: 程式專家代理範例

Principles for Leading the Augmented Team

帶領強化團隊的原則

Successfully leading this framework requires evolving from a sole contributor into the lead of a human-AI team, guided by the following principles:

成功帶領此框架需要從單人貢獻者轉變為人類 – AI 團隊的領導者,並遵循以下原則:

- **Maintain Architectural Ownership** Your role is to set the strategic direction and own the high-level architecture. You define the "what" and the "why," using the agent team to accelerate the "how." You are the final **arbiter** of design, ensuring every component aligns with the project's long-term vision and quality standards.
- **Master the Art of the Brief** The quality of an agent's output is a direct reflection of the quality of its input. Master the art of the brief by providing clear, unambiguous, and comprehensive context for every task. Think of your prompt not as a simple command, but as a complete briefing package for a new, highly capable team member.
- **Act as the Ultimate Quality Gate** An agent's output is always a proposal, never a command. Treat the Reviewer Agent's feedback as a powerful signal, but you are the ultimate quality gate. Apply your domain expertise and project-specific knowledge to validate, challenge, and approve all changes, acting as the final guardian of the codebase's integrity.
- **Engage in Iterative Dialogue** The best results emerge from conversation, not monologue. If an agent's initial output is imperfect, don't discard it—refine it. Provide corrective feedback, add clarifying context, and prompt for another attempt. This iterative dialogue is crucial, especially with the Reviewer Agent, whose "Reflection" output is designed to be the start of a collaborative discussion, not just a final report.

- **維持架構主權 你的角色是設定策略方向並掌握高層架構。你定義「做什麼」與「為什麼」,讓代理團隊加速「怎麼做」。你是設計的最終裁決者,確保每個元件符合專案長期願景與品質標準。**
- **掌握簡報藝術 代理輸出品質直接反映輸入品質。透過提供清楚、明確且完整的脈絡,掌握任務簡報的藝術。把提示視為新成員的完整簡報包,而非單一指令。**
- **擔任最終品質關卡 代理輸出永遠是提案而非命令。將 Reviewer Agent 的回饋視為強訊號,但你才是最終品質關卡。運用領域知識與專案脈絡驗證、挑戰與核准變更,作為程式庫完整性的最終守門人。**
- **進行迭代式對話 最佳成果來自對話而非獨白。若代理初始輸出不完美,不要**丟棄——**請精修。提供修正回饋、補充脈絡並要求再嘗試。此迭代對話至關重要,特別是 Reviewer Agent 的「Reflection」輸出,旨在開**啟**協作討論,而非作為最終報告。**

Conclusion
結論

The future of code development has arrived, and it is augmented. The era of the lone coder has given way to a new paradigm where developers lead teams of specialized AI agents. This model doesn’t diminish the human role; it elevates it by automating routine tasks, scaling individual impact, and achieving a development velocity previously unimaginable.

程式開發的未來已經到來，而且是強化式的。單人編碼者的時代已被新的範式取代：開發者帶領專門化 AI 代理團隊。此模式並不削弱人類角色，而是透過自動化例行工作、放大個人影響力並達成前所未有的開發速度來提升人類價值。

By offloading tactical execution to Agents, developers can now dedicate their cognitive energy to what truly matters: strategic innovation, resilient architectural design, and the creative problem-solving required to build products that delight users. The fundamental relationship has been redefined; it is no longer a contest of human versus machine, but a partnership between human ingenuity and AI, working as a single, seamlessly integrated team.

透過把戰術性執行交給代理，開發者得以把心力投入真正重要的事：策略創新、韌性架構設計，以及打造令人驚豔產品所需的創意解題。人機關係已被重新定義：不再是人類對機器的競賽，而是人類智慧與 AI 的夥伴關係，作為單一、無縫整合的團隊共同運作。

References
參考資料

1. AI is responsible for generating more than 30% of the code at Google https://www.reddit.com/r/singularity/comments/1k7rxo0/ai_is_now_writing_well_over_30_of_the_code_at/
2. AI is responsible for generating more than 30% of the code at Microsoft https://www.businesstoday.in/tech-today/news/story/30-of-microsofts-code-is-now-ai-generated-says-ceo-satya-nadella-474167-2025-04-30