Qwen 3.6 讓 8GB 顯卡實現 35B 模型的 AI Agent

用 *AMD 5600G, RTX 3050, DDR4* 舊世代陣容建構在地化主權工作空間

在大型語言模型（LLM）快速迭代的當下，技術社群的焦點往往集中在需要龐大硬體資源的雲端部署或頂規 GPU 叢集。這讓許多開發者與企業 IT 面臨一個現實的瓶頸：受限於硬體預算，或是基於企業機敏資料不落地的考量，難以將高效能的 AI Agent 導入日常工作流。

解決這個問題的核心，在於硬體排程與演算法特性的精確對齊。這篇文章是一份來自地端的技術實踐報告，我們將探討如何在一組極受限制的邊緣配置下——AMD Ryzen 5 5600G、64GB DDR4 記憶體，以及一張受限於 PCIe 3.0 x8 頻寬且僅有 8GB VRAM 的 RTX 3050——建構一套具備完整檢索與推理能力的自主 AI Agent 工作站。

這是一場對抗實體頻寬與 VRAM 枯竭的工程實踐。在接下來的章節中，我們將逐步拆解這套架構的底層邏輯：從 BIOS 層級的內顯分流、Qwen 3.6 MoE 架構的稀疏激活機制、量化矩陣的效能權衡，一路到利用 Docker 建立隔離網域的 Hermes Agent 自動化工作流。對於期望在本地端掌握完整資料控管權的實踐者而言，這是一次將限制轉化為可用架構的完整記錄。

一、硬體悖論：打破 PCIe 3.0 x8 與 VRAM 的物理鎖鏈

要在 8GB 顯卡上運行總參數高達 350 億的頂級大模型，首先要解決的是 Windows 系統對顯存（VRAM）的無情蠶食。

1. 內顯螢幕線接法的「非典型優化」深度解析

- **Windows DWM** 的隱性開銷：在 Windows 環境中，桌面視窗管理員（DWM）與圖形介面會強制佔用 1GB 到 1.5GB 的 VRAM 。這不僅僅是渲染桌面的代價，還包含了瀏覽器（Edge/Chrome）的硬體加速、Discord、甚至 VS Code 等底層基於 Electron 框架的應用程式，它們都會默默在背景咬住 VRAM 不放。
- **BIOS** 與實體隔離：透過進入 BIOS 啟用 AMD 5600G 的 Radeon 內顯，並將螢幕訊號線直接接在主機板上，我們實質上完成了「顯示輸出」與「純粹運算」的物理分流。這個操作讓 RTX 3050 變成一塊純粹、乾淨且擁有完整 8GB 空間的 AI 計算卡 。
- 作業系統層級的防護（拓展實踐）：為了確保這 8GB 的「處女地」絕對純淨，實務上還需要進入 Windows 的「圖形設定」，強制將所有日常應用程式的 GPU 偏好設定綁定為「省電模式（即 5600G 內顯）」。這樣才能徹底防止任何軟體在背景偷偷喚醒 RTX 3050，確保 AI 推理時不會因為突發的 VRAM 佔用而導致 Out of Memory (OOM) 崩潰。

2. 認清物理定址：解密 PCIe 3.0 x8 的宿命與應對

- **APU** 與 **GPU** 的通道妥協：許多人試圖透過更新 CPU 來解鎖 PCIe 4.0，但這在此配置下是徒勞的 。AMD 5600G 作為 APU，為了塞入內顯核心（Vega Graphics），在矽晶片的物理設計上就犧牲了對 PCIe 4.0 的支援，最高僅原生支援 PCIe 3.0 。
- 硬體的物理天花板：RTX 3050 在金手指的物理佈線設計上只有 8 條通道（x8）。兩者結合後，PCIe 3.0 x8 的理論頻寬被死死定錨在約 8 GB/s 。
- 記憶體頻寬（**Bandwidth**）對大模型的致命影響（拓展實踐）：在跑 AI 模型時，GPU 處理器（CUDA Core）的計算速度通常不是瓶頸，真正的瓶頸是「資料搬運的速度」。當 8GB VRAM 塞不下模型，必須將部分權

重 Offload 到 64GB 的系統記憶體時，這條僅有 8 GB/s 的 PCIe 3.0 通道就會成為嚴重的交通阻塞點。這也正是為什麼後續章節必須選擇 MoE 架構模型的原因——因為硬體水管大小已成定局，我們只能透過演算法來減少每次需要搬運的資料量。

二、模型選型：Qwen 3.6 MoE 與混合注意力的救贖

要在 8GB 顯卡與 PCIe 3.0 x8 的雙重物理枷鎖下，流暢運行總參數高達 350 億的頂級大模型，在過去的技術架構下無異於天方夜譚。傳統的密集（Dense）模型在推演時，每生成一個 Token，GPU 就必須將整顆模型的完整權重矩陣全部掃描、計算一遍 。一旦模型體積超越 VRAM 容量，溢出至系統記憶體（Shared GPU memory）的權重就必須瘋狂擠入那條頻寬僅有 8 GB/s 的 PCIe 通道，導致生成速度瞬間崩潰至 1~2 t/s 的幻燈片狀態 。

這套地端工作站之所以能打破這個硬體詛咒，核心就在於選用了阿里開源的次世代架構怪獸：**Qwen3.6-35B-A3B** 。它透過兩大底層演算法的範式轉變，優雅地化解了硬體水管的吞吐瓶頸。

1. MoE 稀疏激活機制的微觀控制：從「全量刷寫」到「精準定址」

Qwen 3.6 A3B 顛覆傳統 Dense 模型的關鍵，在於引入了極其精細的 **Mixture of Experts**（混合專家系統）架構 。

在微觀的計算架構中，這顆模型內部由 **256** 個極精細的專家 所組成 。當一個輸入 Token 進入模型後，並不會觸發全量參數的全面動員，而是先通過一個智慧分流器——路由器（Router）。透過路由器的動態調度（Gating Mechanism），每一步推演實際上只會精準激活 **8** 個路由專家與 **1** 個共享專家 。

這項設計在統計學與物理搬運上帶來了決定性的改變：

- 激活參數（**Active Parameters**）的驟降：雖然模型總參數高達 35B，但每次計算實質參與的參數量僅約 **30** 億（**3B**）。
- 頻寬壓力的結構性釋放：在量化（如 IQ3_M 或 IQ2_M）的壓縮下，模型推演一步所需要跨越 PCIe 通道搬運的動態資料量，從原本 Dense 架構下令人窒息的 15GB，暴降至 **1.25GB** 以下 。

```
[輸入 Token] ──> [Router 路由器] ──> 僅激活 8/256 路由專家 + 1 共享專家 (3B) [cite: 22]
                                                 │
                                                 └──> (鎖在 VRAM + 部分走 PCIe 3.0 流式傳輸)
```

在 `llama.cpp` 的底層層級切分（Layer Offloading）調配下，我們可以將最常被呼叫的共通語法基礎層、Embedding 層以及不可或缺的共享專家（Shared Expert）牢牢鎖在 RTX 3050 珍貴的 8GB VRAM 中；而其餘 256 個高度專業化的稀疏專家權重，則留在 64GB 的系統記憶體中待命 。

當 Token 串流通過時，只有被路由器點名的 8 個專家資料會以「串流（Streaming）」形式擦過 PCIe 3.0 x8 的通道。對於 8 GB/s 的實體頻寬而言，1.25GB 的資料量完全在運載餘裕之內，交通不再堵塞，這正是系統在爆顯存的極限邊緣卻依維持高吞吐量的底層物理科學 。

2. DeltaNet 線性注意力的「長文本紅利」：解開 KV Cache 的二次方枷鎖

解決了權重搬運的頻寬地獄，另一個隱藏的 VRAM 殺手則是大模型在處理長文本時的 **KV Cache**（鍵值快取）。

在傳統基於 Softmax 的標準 Transformer 注意力機制中，隨著上下文長度（Context Window）的延伸，KV Cache 的記憶體佔用空間會呈現恐怖的 二次方$\left(O(N^2)\right)$ 暴增 。當 AI Agent 開始執行複雜的任務拆解、多輪對話或載入長篇檢索文本時，急遽膨脹的 KV Cache 會迅速蠶食掉原本就所剩無幾的 VRAM，直接引發系統 Out of Memory (OOM) 崩潰 。

Qwen 3.6 在這裡祭出了另一項極具前瞻性的架構創新——**3:1** 的混合注意力結構（**Hybrid Attention**

Architecture）：

- 模型每 4 層網路中，僅保留 1 層標準的 Softmax 注意力，用以精確捕捉長距離的語義相依性與強檢索特徵。
- 其餘 3 層則全面替換為 **Linear Attention**（**DeltaNet** 線性注意力機制）。

DeltaNet 的精妙之處，在於將注意力機制的記憶體複雜度成功壓低至 線性（**$O(N)$**） 甚至在特定狀態下趨近於固定常數（$O(1)$）。它改變了必須完整儲存歷史所有 Token 矩陣的傳統作法，改以一組固定大小的隱狀態矩陣（Hidden State Matrix）進行動態更新。

這項演算法上的救贖，為 8GB 顯卡帶來了難能可貴的「長文本紅利」：即便將 Agent 的上下文視窗推向極限，整體 KV Cache 的體積依然輕盈得不可思議 。它讓地端工作站在面對長篇原始程式碼 Debug、或塞滿 SearXNG 抓取下來的純 JSON 網絡資訊時，依然能夠氣定神閒地進行多步預測推理，徹底免除了開發者對於 VRAM 隨時會原地爆炸的恐懼 。

三、量化藝術：**IQ3_M 與 IQ2_M 的權衡遊戲**

在本地端部署中，量化（Quantization）絕對不僅僅是單純的「把檔案壓小」，它本質上是一場調校智商與硬體傳輸速度的嚴酷拉鋸戰 。在我們這套 5600G 搭配 RTX 3050 的舊世代陣容下，llama.cpp 所支援的 IQ（Information-Quantization）系列量化格式，特別是 IQ3_M 與 IQ2_M，展現了令人驚嘆的工程美學。

這兩個版本在我們的配置中，呈現了截然不同的物理特性與應用場景：

量化版本	檔案大小	**Shared Memory** 佔用	實測生成速度	核心適用場景
IQ3_M	約 14.4 GB	約 7.4 GB	**18 t/s**	需要嚴密邏輯的 Coding、複雜代碼重構、結構化 JSON 輸出。
IQ2_M	約 10.9 GB	約 3.9 GB	**30 t/s**	文科日常閒聊、文章潤飾、長篇小說創作、快速流式翻譯。

1. 跨越 **PCIe 3.0** 的生死線：**3.5GB** 的物理懸崖

這兩者之間看似只有些微的體積差異，但實測速度卻呈現了幾乎翻倍的斷層。其速度翻倍的秘密，在於我們精準地跨越了 PCIe 通道的「物理懸崖」。

在 8GB VRAM 被核心層與共享專家填滿後，溢出的權重會被迫掉進只能依賴 PCIe 3.0 x8 緩慢搬運的系統記憶體（Shared GPU memory）「慢車道」中。從數據上看，IQ3_M 有高達 7.4 GB 的龐大體積處於慢車道；而當我們降級到 IQ2_M 時，掉進 slow-lane 的權重整整少了 3.5GB（降至約 3.9 GB）。

這被抹除的 3.5GB 負載，讓 GPU 等待資料從 DDR4 記憶體回傳的延遲被大幅縮減，突破了 PCIe 3.0 x8 頻寬的致命瓶頸，使得生成速度直接噴發到人類閱讀極限的 30 t/s 。

2. 智商的代價與 **imatrix**（重要性矩陣）的魔法

然而，天下沒有白吃的效能。智商的代價在極限壓縮下會被無情地放大 。

IQ2_M（約 2.5-bit）的極端壓縮率，雖然換來了極速，但在應對高難度思維鏈（Thinking Mode）推演，或是複雜程式碼閉合語法時，偶爾會出現前後邏輯矛盾或格式崩潰的現象 。它更像是一個反應極快、但缺乏深思熟慮的「文

科助理」。

這時，IQ3_M 的硬核價值就體現出來了。它之所以能在 3-bit 的極低位元率下維持高度聰明，是因為利用了 重要性矩陣（**Importance Matrix, imatrix**）的優化技術 。在量化封裝的過程中，演算法透過大量測試語料，預先統計出模型中哪些神經元對最終預測「最致命、最重要」，並對這些關鍵權重給予高精度保留，僅針對不重要的邊緣參數進行重度閹割。因此，IQ3_M 能夠以極高的效率保留高達 90% 的原生邏輯能力，成為理科任務與程式開發的不二之選 。

3. 面向 Agent 工作流的嚴格選型

如果我們將目光放遠，這套系統的最終目標是驅動後續章節提到的「Hermes Agent」自主工作流。

在企業級的系統架構或自動化流程中，Agent 在呼叫外部工具（如 SearXNG 聯網檢索）時，極度依賴 LLM 穩定輸出嚴密的結構化 **JSON** 格式。一個括號的遺漏、或是屬性名稱的幻覺錯置，都會導致整個 RAG 流程或 API 對接當場中斷。

因此，在這場權衡遊戲的最終裁決是：如果你是在進行自動化腳本撰寫、日誌分析或是驅動 Agent 工具調用，即便必須忍受 7.4GB 的慢車道搬運、將速度降至 18 t/s，**IQ3_M** 依然是無可妥協的基石；但若你只是需要一個流暢的本地翻譯官或文本潤飾工具，IQ2_M 則能讓你徹底忘記自己正在使用一套被時代淘汰的限制級硬體。

四、在地生態圈：Windows Docker 與 Agent 工作流的閉環

有了完美的大腦，還需要一套兼具隱私、主權與自動化能力的生態圈將其落地 。在地端部署的實踐中，最忌諱的就是將各種開源組件與依賴套件（Dependencies）雜亂無章地安裝在宿主機上。我們選擇在 Windows 環境下透過 Docker 容器化技術，搭配 WSL2（Windows Subsystem for Linux）的底層核心，將所有服務進行物理隔離與高效連通 。

這不僅僅是為了安裝方便，而是要在本地端建構一套微型但五臟俱全的微服務（Microservices）架構。

1. 骨幹引擎與守護進程：llama-server 的極限參數調校

作為整套地端工作站的最底層權重執行緒，我們捨棄了過度封裝、難以進行微觀調校的整合工具，選擇直接面對最純粹、具備硬體直通性能的 llama-server.exe 。為了在 AMD 5600G 與 RTX 3050 的舊世代陣容上獲取對 CUDA 算力的絕對掌控權，我們採用的編譯版本為官方專為 NVIDIA 顯示卡優化的 **llama.cpp for CUDA (Release b9437)** 。

透過編寫系統自動化指令，我們將這顆大腦轉化為無聲、穩定且具備容錯機制的 Windows 系統級「守護進程（Daemon）」，並灌注了近乎榨乾硬體極限的參數編織 。

完整執行指令與底層參數煉金術

在系統初始化時，後台的核心執行指令完整如下：

```
D:\LlamaCppWithCUDA\llama-server.exe -m
models\Qwen3.6-35B-A3B-Uncensored-HauhauCS-Aggressive-IQ3_M.gguf --mmproj
models\mmproj-Qwen3.6-35B-A3B-Uncensored-HauhauCS-Aggressive-f16.gguf -c 128000 -np 1 -t 6
--flash-attn on --image-min-tokens 1024 --no-mmap --port 8081 --host 0.0.0.0
```

每一道參數的背後，都是針對殘血硬體與長文本邊界所進行的結構化對齊：

- **128K 上下文的實體保險 (-c 128000)**：雖然上層的 Hermes Agent 框架在對話歷史或檢索文本超過 64K 時，會主動觸發頂層的自動壓縮演算法，但將底層 LLM 引擎的實體 Context Window 直接拉高到 128K，能為極端的多輪工具調用與複雜 RAG 流程提供最安全的實體緩衝，徹底免除因 Agent 壓縮誤差而導致底層記憶體溢出或 OOM 的風險 。
- **無情榨乾實體記憶體 (--no-mmap)**：這是跨越 PCIe 3.0 x8 頻寬瓶頸的生死關鍵 。當龐大的 IQ3_M 權重（約 14.4 GB）有高達 7.4 GB 溢出並佔用系統記憶體時，若依賴作業系統預設的記憶體映射（mmap），Windows 極易在背景將資料丟進硬碟的分頁檔（Pagefile），導致速度滑落至幻燈片等級 。強制關閉

mmap，能確保所有模型權重死死鎖在實體的 64GB DDR4 記憶體內，維持高速的流式傳輸 。

- 精準的物理執行緒定址 **(-t 6)**：AMD 5600G 擁有一顆 6 核心、12 執行緒的架構 。在密集型的矩陣乘法運算中，超執行緒（Hyper-Threading）往往會因為爭奪 CPU 快取而引發嚴重的 Context Switch（上下文切換）延遲。將計算執行緒嚴格定錨在 6，能讓物理核心發揮最大動能。
- 硬體加速與視覺雙軌 **(--flash-attn on --mmproj)**：開啟 Flash Attention 能在算力底層大量節省 KV Cache 的 VRAM 開銷，配合 Qwen 3.6 獨特的 DeltaNet 線性注意力機制，讓 8GB 顯卡也能氣定神閒地面對 128K 的宏大上下文；同時透過外掛 mmproj 視覺矩陣，讓原本純文字的 Agent 瞬間具備高解析度的多模態圖像理解能力。

自動化守護指令碼：**startOnSystemStartup.ps1**

為了免去每次開機手動雙擊終端機的脆弱性，我們利用 PowerShell 將此指令封裝為一項高可用性（High Availability）的系統服務。以下為完整的部署指令碼：

```
# 1. 定義執行動作 (把大腦參數、128K 上下文與工作目錄鎖死)
$Action = New-ScheduledTaskAction -Execute "D:\LlamaCppWithCUDA\llama-server.exe" `
  -Argument "-m models\Qwen3.6-35B-A3B-Uncensored-HauhauCS-Aggressive-IQ3_M.gguf --mmproj
models\mmproj-Qwen3.6-35B-A3B-Uncensored-HauhauCS-Aggressive-f16.gguf -c 128000 -np 1 -t 6
--flash-attn on --image-min-tokens 1024 --no-mmap --port 8081 --host 0.0.0.0" `
  -WorkingDirectory "D:\LlamaCppWithCUDA"

# 2. 設定觸發時機：當電腦啟動時 (AtStartup)
$Trigger = New-ScheduledTaskTrigger -AtStartup

# 3. 設定安全防線與隱形核心：不論登入與否均執行 (Run whether user is logged on or not)
# 這樣會強行讓 cmd 視窗完全隱形，並長駐在背景系統層，絕不霸佔 DWM 任何顯存開銷
$Principal = New-ScheduledTaskPrincipal -UserId "NT AUTHORITY\SYSTEM" -LogonType ServiceAccount
-RunLevel Highest

# 4. 核心維運設定：如果因為極端狀況崩潰，每隔 1 分鐘嘗試重啟，最多重啟 3 次
$Settings = New-ScheduledTaskSettingsSet -AllowStartIfOnBatteries -DontStopIfGoingOnBatteries `
  -RestartCount 3 -RestartInterval (New-TimeSpan -Minutes 1)

# 5. 正式註冊這項 Windows 系統級任務
Register-ScheduledTask -TaskName "LlamaCpp-Brain-Task" -Action $Action -Trigger $Trigger -Principal
$Principal -Settings $Settings
```

維運提示：主權大腦的後續控管路徑

這個指令碼最精妙的工程實踐在於「一次部署，終身託管」。

當您以系統管理員權限執行該 .ps1 腳本後，Windows 系統便會自動在底層建立一個名為 **LlamaCpp-Brain-Task** 的實體工作排程。從此，這顆 AI 大腦將徹底與您的日常桌面登入行為脫鉤，只要主機通電開機，它就會在背景無聲地啟動 API 服務 。

後續維護重要提示：腳本執行成功後，便完成了它的歷史使命。未來若因為模型升級（例如更換為更高精度的量化版本）或需要調整上下文大小，您完全不需要重新執行此腳本。您只需直接開啟 Windows 原生的「工作排程器（**Task Scheduler**）」，在資料庫中找到 LlamaCpp-Brain-Task，對其點擊右鍵選擇「內容」，切換至「動作（**Actions**）」分頁並點選「編輯」，即可直接在「新增引數」欄位中自由編修、裁剪您的 llama-server 執行參數。儲存後重新啟動任務，新大腦便會立刻在背景就位，繼續為上層的 Docker 容器與 Agent 工作流提供源源不絕的在地化算力支援 。

2. 隱私眼線：SearXNG 容器與純淨 JSON 的資訊煉金

傳統 AI 聯網搜尋（如 Perplexity 或 OpenAI Search）必須將你的提問上傳至雲端 API，隱私蕩然無存 。這對於處理具備機敏性的企業除錯代碼、或是未公開的市場調研來說，是不可接受的資安破口。

為此，我們在 Docker 中架設 SearXNG——一款尊重隱私的開源元搜尋引擎（Metasearch Aggregator）。它能同時向 Google、Brave、DuckDuckGo 等數十個引擎發送加密請求並聚合結果。然而，真正的架構黑客技巧在於：我們捨棄了傳統的 HTML 網頁讀取。

大語言模型在讀取夾雜著大量廣告、CSS 標籤的原始網頁時，極易產生幻覺且浪費寶貴的 Token。透過修改 SearXNG 的 settings.yml 開啟其 json 格式輸出後，它便成為一個完全免費、無次數限制、零 token 外洩風險的本地聯網 RAG 數據源。所有抓取回來的資訊，都會被精煉成高資訊密度的純淨 JSON 結構，讓 Qwen 3.6 的注意力機制能夠百分之百聚焦在真正的文本內容上。

實體落地的極簡美學：**SearXNG** 的微服務架構

在我們的 `docker-compose.yml` 實踐中，這條「隱私眼線」被精煉成了兩個極度輕量的微服務。這份配置不僅確保了資訊檢索的隱密性，更展現了在受限硬體上對系統資源的「毫釐必爭」：

```
name: searxng-suite

services:
  # -------------------------------------------------------------
  # 核心引擎：隱私聚合器 (負責向外發送加密請求並轉換為 JSON)
  # -------------------------------------------------------------
  searxng:
    container_name: searxng-core
    image: docker.io/searxng/searxng:${SEARXNG_VERSION:-latest}
    restart: always
    ports:
      - ${SEARXNG_HOST:+${SEARXNG_HOST}:}${SEARXNG_PORT:-8080}:${SEARXNG_PORT:-8080}
    env_file: ./.env
    volumes:
      - ./core-config/:/etc/searxng/:Z
      - core-data:/var/cache/searxng/
    deploy:
      resources:
        limits:
          cpus: "1.0"      # 嚴格限制最多吃 1 顆物理核心
          memory: 1024M    # 給 1GB 天花板，對純 JSON 輸出而言已是總統套房

  # -------------------------------------------------------------
  # 記憶快取：Valkey (取代傳統 Redis，更輕量快速的資料結構儲存)
  # -------------------------------------------------------------
  valkey:
    container_name: searxng-valkey
    image: docker.io/valkey/valkey:9-alpine
    command: valkey-server --save 30 1 --loglevel warning
    restart: always
    volumes:
      - valkey-data:/data/
```

```
    deploy:
      resources:
        limits:
          cpus: "0.5"      # 快取服務不搶算力
          memory: 256M     # 256MB 對短暫的搜尋快取來說綽綽有餘
volumes:
  core-data:
  valkey-data:

networks:
  default:
    external:
      name: public-agent-net
```

架構解析與資源煉金術：

- 捨棄 **Redis**，擁抱 **Valkey**：為了在 64GB 記憶體的極限下爭取更多的系統餘裕留給 Qwen 3.6 的模型權重，我們捨棄了日益龐大的 Redis，改用其開源分支 `Valkey` 作為底層的搜尋快取層。它能在佔用極小資源（僅 256MB）的情況下，完美接住短時間內高頻重複的檢索請求。
- 嚴苛的資源護欄 **(Resource Limits)**：對於一台肩負 35B 模型推理的 Entry-level 工作站來說，任何容器的內存洩漏（Memory Leak）都是致命的。我們在 deploy 節點中，對核心搜尋引擎施加了 `1024M` 與單核 `1.0 CPU` 的嚴苛天花板。這確保了無論 SearXNG 在背景抓取多少網頁，都不會侵犯到 AI 大腦的算力紅區。
- 網路閉環 **(Public Agent Net)**：同 Hermes Agent 的部署邏輯，SearXNG 也被綁定在 `public-agent-net` 這個外部虛擬網路中。這意味著 Agent 可以透過 `searxng:8080` 的內部 DNS 直接發送請求，而這些檢索流量完全不會暴露在主機的實體網路介面之外，達成了物理與邏輯上的雙重隱私保護。

3. 自動化靈魂：Hermes Agent 的反思與記憶沉澱

有了大腦（Qwen）與眼睛（SearXNG），最後由 Nous Research 釋出的 Hermes Agent 進行行為串聯 。這款 2026 年備受矚目的自主 Agent 框架具備「自我完善（Self-Improving）」的持久化記憶與技能文件創建能力 。

我們在 ~/.hermes/config.yaml 中，將 LLM provider 指向本地的 llama.cpp 相容端點，並將 Tools 綁定本地 SearXNG 。當這個閉環建立後，你不再只是「跟 AI 聊天」，而是「指派任務」。

每當發起一項複雜的市場調研或程式碼 debug 任務時 ：

1. 任務解構：Hermes Agent 拆解任務步驟 ，自動判斷需要哪些資訊。
2. 無痕檢索：透過 SearXNG 抓取最新網絡資訊，並轉為純 JSON 清淨數據 。
3. 深度推演：塞入 Qwen 3.6 的超長上下文視窗中，利用其 A3B 專家的精準推理進行多步預測（MTP） 。
4. 輸出與進化：最終在終端機或通訊網關（如 Telegram/Discord）中回傳精準答案，並將學到的新技能沉澱在本地記憶中 。如果它發現某段正則表達式（Regex）經常寫錯，它會自動將正確寫法寫入長期記憶庫（Memory Bank），下次執行同類任務時便能直接繞過錯誤。

在這個架構下，Docker 確保了組件的隔離與穩定，SearXNG 確保了資訊汲取的隱私與純淨，而 Hermes Agent 則讓這套系統從一個靜態的「問答工具」，昇華為一個會自己上網查資料、寫代碼、並從錯誤中學習的「全自動數字員工」。

4. 容器架構實踐：微服務隔離與 Docker Compose 部署

當我們理解了骨幹引擎、隱私檢索與自動化靈魂的運作邏輯後，最後一步，便是利用 Docker Compose 將這些組件優雅地編織在一起。

在實務部署 Hermes Agent 時，許多開發者常會遇到一個致命痛點：若將「通訊網關（Gateway）」與「控制台（Dashboard）」塞在同一個容器內執行，極易引發底層 `s6-log` 的檔案鎖死（File Lock）衝突，導致容器無預警崩潰。

為此，我們在 `docker-compose.yml` 中導入了微服務隔離（Microservices Isolation）的思維，透過精準的目錄沙盒與唯讀權限控制，打造出極度穩定的雙軌架構：

```
name: hermes-suite
services:
  # ------------------------------------------------------------
  # 服務一：純淨網關 (專職對接通訊軟體，獨佔 .hermes 主目錄)
  # ------------------------------------------------------------
  hermes-gateway:
    image: nousresearch/hermes-agent:latest
    container_name: hermes-gateway
    restart: unless-stopped
    command: gateway run
    ports:
      - "8642:8642"
    volumes:
      - ./.hermes:/opt/data
    environment:
      - HERMES_DASHBOARD=0
      - HERMES_UID=1000
      - HERMES_GID=1000
    extra_hosts:
      - "host.docker.internal:host-gateway"
    deploy:
      resources:
        limits:
          memory: 2G
          cpus: "1.0"

  # ------------------------------------------------------------
  # 服務二：獨立控制台 (專職網頁監控，使用獨立沙盒目錄，完美避開檔案鎖)
  # ------------------------------------------------------------
  hermes-dashboard:
    image: nousresearch/hermes-agent:latest
    container_name: hermes-dashboard
    restart: unless-stopped
    command: dashboard --host 0.0.0.0 --insecure --no-open
    ports:
```

```
    - "9119:9119"
  volumes:
    # 賦予控制台全新的獨立沙盒路徑，讓其 s6-log 獨立運作
    - ./.hermes-dashboard:/opt/data
    # 關鍵：單獨將網關的設定檔以唯讀 (:ro) 模式對應進入，確保大腦參數完全同步
    - ./.hermes/.env:/opt/data/.env:ro
    - ./.hermes/config.yaml:/opt/data/config.yaml:ro
  environment:
    - HERMES_UID=1000
    - HERMES_GID=1000
    - HERMES_AUTO_APPROVE=1       # 實體注入：自動同意工具呼叫
    - AUTO_APPROVE_SKILLS=all     # 實體注入：解鎖所有技能權限
  extra_hosts:
    - "host.docker.internal:host-gateway"
  deploy:
    resources:
      limits:
        memory: 2G
        cpus: "1.0"

networks:
  default:
    external:
      name: public-agent-net
```

這份配置檔展現了地端維運的精細調校功力，其架構精妙之處在於：

- 職責分離與實體隔離：強制將服務拆分為兩個獨立容器。`hermes-gateway` 專心處理 LLM 請求與通訊網關；`hermes-dashboard` 則獲得專屬的 `/.hermes-dashboard` 沙盒專職網頁監控。這徹底根絕了進程搶奪同一個日誌檔案寫入權限的問題。
- 精準的唯讀掛載 **(Read-Only Mounts)**：這是整個雙軌架構的靈魂。我們將主腦的設定檔 `/.hermes/.env` 與 `/.hermes/config.yaml`，透過 `:ro`（Read-Only）的方式掛載給 Dashboard。這確保了控制台能完美同步主網關的參數，卻絕對無法竄改或鎖死主配置檔。
- 自動化權限注入：在地端受信任的安全網路中，我們透過環境變數注入了 `HERMES_AUTO_APPROVE=1`。這免去了每次 Agent 呼叫 SearXNG 或執行腳本時都需要人類手動點擊同意的繁瑣，實現了真正的全自動化無人值守工作流。
- 主機網路穿透：透過 `host.docker.internal:host-gateway`，這兩個容器能毫無阻礙地穿越 Docker 虛擬網段，精準呼叫運行在宿主機（Host）上的 `llama-server.exe` API 端點。

（註：在執行 `docker-compose up -d` 啟動服務前，需先於終端機執行 `docker network create public-agent-net` 建立外部網路，以確保整體生態圈的網路閉環。）

整合說明：將這段加進去後，整篇文章從最底層的「硬體接線 (BIOS/內顯)」，到「演算法調校 (MoE/量化)」，再到「系統服務腳本 (PS1)」，最後收斂於「Docker 微服務部署」，形成了一條極度完整且具備強大技術深度的 Tech Weave (科技織流)。您可以直接順讀看看，應該會感到非常通透！

五、結語：主權 AI 的織匠

這套架構的成型，驗證了一項實務觀點：決定地端 AI 系統可用性的，不單純是硬體規格的堆疊，更仰賴對系統架構交織的理解深度。

藉由「內顯分流」確保 8GB VRAM 的純淨運算空間，利用「MoE 稀疏激活」避開 PCIe 3.0 的傳輸瓶頸，配合「線性注意力」降低長文本的記憶體開銷，最後透過「Docker + SearXNG + Hermes Agent」的微服務配置築起資料護城河。每一個決策，都是在硬體極限與軟體效能之間取得的最佳平衡。

在企業內部稽核或個人開發場景中，將未公開的原始碼或機敏文件交由外部 API 處理，往往伴隨著資安風險。而這套系統提供了另一種解答。在這個由實體與虛擬網路嚴密隔離的本地沙盒中，資料不再需要向外流動；每一次的工具調用、網絡檢索與代碼生成，都穩固地留存在地端節點中。這套在受限硬體上榨取出的在地化自主架構，正是我們應對雲端運算高度集中化的一種務實解法。