

話說 對於我想要找 適合的local AI llm model  
來對應我的 local llm RAG project  
我對手邊的3 llm model , 做了real world的測試  
先看看一些attachment,  
最後再給我你的評語  
先看看test method

≡ TEST\_SUMMARY\_SIMPLE.md

≡ TEST\_METHODOLOGY\_DOCUMENT...

≡ FINAL\_3\_MODEL\_COMPARISON.md

≡ Mistrial FINAL\_TEST\_SUMMARY.md

≡ GPT\_OSS\_20B\_FINAL\_TEST\_SUMM...

≡ GEMMA\_27B\_FINAL\_TEST\_SUMMA...

💡 思考了13s

## Test Methodology Summary

Based on the attached documents (particularly "TEST\_METHODOLOGY\_DOCUMENTATION.md" which details the full methodology, and cross-referenced with the model-specific summaries), here's a clear overview of how the tests were conducted. I'll keep it structured and objective, drawing directly from the provided info. This "needle-in-haystack" approach seems solid for evaluating real-world context handling on your hardware (RTX 5090 with 32GB VRAM), especially for a RAG project where accurate retrieval from large contexts is key.

## Core Test Design: Needle-in-Haystack

- **Data Structure:** A large list of numbered items was created, e.g.:

```
text
```

```
index 1: The value for item 1 is 100  
index 2: The value for item 2 is 200  
...  
index N: The value for item N is N*100
```

- This creates an objective "haystack" where specific "needles" (values) can be queried.
- Query example: "What is the content of index X?" (Expected:  $X*100$ ).
- **Why This Method?**
  - Objective and measurable: Clear right/wrong answers, no subjectivity.
  - Scalable: Easy to increase size to probe limits.
  - Position-sensitive: Reveals how well the model attends to different parts of the context (e.g., start/end vs. middle).
  - Reproducible: Same test across models, relevant to real docs like IEC standards (300+ pages).
  - Focuses on retrieval accuracy, which is crucial for RAG (Retrieval-Augmented Generation) where you need reliable info extraction from documents.

## Position Testing

- For each context size, three positions were tested:
  - **First** (index 1): Checks primacy bias (models often remember beginnings well).

- **Middle** (index  $N/2$ ): Hardest; tests attention decay (common weakness in LLMs).
- **Last** (index  $N$ ): Checks recency bias (models often remember ends well).
- Accuracy per size:  $(\text{Correct positions} / 3) \times 100\%$ . E.g., 67% means one position (usually middle) failed.

### Progressive Scaling

- Context sizes ramped up gradually:
  - Small: 100–1,000 items.
  - Medium: 1,500–3,000 items.
  - Large/Extreme: 3,500–6,500+ items (for 131k models).
- Stop criteria: When accuracy  $< 50\%$ , out-of-memory (OOM), or context exceeded.
- Token efficiency varied: Mistral ~22.6 tokens/item, Gemma ~14.6, GPT-OSS ~16.7 (affects how much real content fits).

### Metrics Collected

- **Accuracy**: Overall % correct; thresholds: Reliable ( $\geq 95\%$ ), Usable ( $\geq 67\%$ ), Unreliable ( $< 67\%$ ).
- **Performance**: Response time (avg. seconds), first-token latency, token count.
- **Utilization**: Usable tokens vs. advertised/configured.
- **Iterations**: 3 queries per size (one per position); ~180 total API calls across models, over 2 hours.
- **Temperature**: 0 (deterministic, no randomness).
- **Error Handling**: Retries on timeouts; logged overflows; special parsing for GPT-OSS's output format (e.g., `<|channel|>final<|message|>100`).

## Hardware & Setup Standardization

- **GPU:** RTX 5090 (32GB VRAM) – critical, as VRAM limits effective context (e.g., KV cache eats memory for large contexts).
- **Inference:** llama.cpp server on localhost:8080.
- **Quantization:** Model-specific (Q6\_K for Mistral/Gemma, F16 for GPT-OSS) to fit in VRAM.
- **API Config:** Simple user messages, max\_tokens 20–50, timeout 120s.
- **Reproducibility:** Scripts provided (e.g., `context_limit_test.py`), docker commands for setup.

## Key Insights from the Method

- **Patterns Revealed:** All models weak in middle positions (40% avg. accuracy vs. 90% start/85% end). This highlights real RAG challenges – important info might "get lost" in the middle.
- **Hardware Reality:** Advertised limits (e.g., 131k) often slashed by VRAM (e.g., Gemma to 24k).
- **Trade-offs:** Larger context ≠ better accuracy; speed degrades with size.
- **Limitations of Method:** Focuses on retrieval, not generation/creativity. Good for RAG, but test creative tasks separately if needed.
- **Strengths:** Objective, hardware-specific, position-aware – better than just trusting specs.

This method is practical and RAG-relevant, as it simulates querying embedded knowledge from docs. It shows why "marketing numbers" (e.g., 131k context) don't match reality on consumer hardware.

## My Evaluation for Your Local LLM RAG Project

Now, based on the full set of attachments (summaries, methodology, and model-specific results), here's my overall assessment. I'll focus on suitability for a local RAG setup on RTX 5090: You need reliability (accurate retrieval), speed (for interactive use), and decent context (to handle docs without too much

chunking). RAG thrives on stable models that don't hallucinate or fail unpredictably when pulling from large contexts.

Quick Model Comparison Table (From Your Tests)

Model	Usable Context	Reliability/Accuracy	Speed (Avg.)	Strengths	Weaknesses	RAG Fit
Mistral 24B	~31k tokens (96% of 32k advertised)	High (67–100%, predictable; middle fails sometimes but consistent)	0.6s ⚡	Fast, stable, production-ready; good token efficiency for docs (~60 pages).	Smaller context; occasional middle-position weakness (workaround: chunking).	✅ Best overall – Reliable for real-time RAG queries.
Gemma 27B	~22k tokens (only 18% of 131k advertised, VRAM-limited)	Medium (33–100%, erratic; unusual first-position fails).	1.5s 🐢	Slightly more items than Mistral (~45 pages); efficient tokens/item.	Slower, unstable, VRAM-hungry; no full potential on your hardware.	❌ Avoid – Worse than Mistral in every key metric on RTX 5090.
GPT-OSS 20B	~100k tokens (76% of 131k advertised)	Low (0–100%, chaotic/random; no pattern).	2.7s 🐢	Largest context (~200 pages); good for huge docs.	Unpredictable failures (even on small contexts); special output format needs parsing; slow.	⚠️ Maybe for experiments – Too unreliable for production RAG.

Overall Recommendation: Go with Mistral 24B

- **Why Mistral?** It's the clear winner for your RAG project based on these tests. It balances everything: High utilization of its advertised context, fast responses (critical for local interactive RAG), and predictable behavior. In RAG, you often chunk docs anyway (e.g., into 500–1,000 token pieces), so 31k usable is plenty for most queries – it can handle 60-page sections reliably without the chaos of GPT-OSS or Gemma's limitations. Your tests show it as "production-ready" with stable 67–100% accuracy, and the 0.6s speed means quick retrieval-augmented responses.
  - For IEC-like docs: Process in 60-page chunks; use sliding windows for larger ones (as suggested in methodology).
  - Implementation Tip: Set max context to 29k tokens for a safety buffer, as per the Mistral summary. Integrate with LightRAG or similar, placing key info at start/end to avoid middle weaknesses.
- **Why Not Gemma?** It's crippled on your hardware – only 22k usable despite 131k training. Slower and less stable than Mistral, with no upsides. Wait for better GPUs (e.g., 48GB+) if you want to revisit.
- **Why Not GPT-OSS?** The massive context is tempting for full-doc RAG, but the "chaotic" accuracy (random 0–100%) makes it risky – you can't trust retrievals, even on small inputs. Plus, parsing its special format adds hassle. Use it only for non-critical experiments or if you need 200+ pages at once (with retries).
- **General Advice for Your Project:**
  - **Test on Your Setup:** Reproduce with the provided scripts to confirm (e.g., `python context_limit_test.py`).
  - **RAG Optimizations:** Design for middle-position issues (all models have them) – e.g., summarize middles or use hybrid retrieval.
  - **Hardware Note:** VRAM is the bottleneck; if you upgrade, Gemma/GPT-OSS might shine more.

- **Bottom Line:** Mistral gives the best "bang for buck" on RTX 5090 – reliable, fast, and hardware-optimized. Don't chase bigger contexts if they come with instability.

If you share more details about your RAG use case (e.g., doc types, query volume), I can refine this further!

用中文 跟我解釋

🕒 思考了9s

## 測試方法論摘要

根據您提供的附件（特別是「TEST\_METHODODOLOGY\_DOCUMENTATION.md」詳細描述了完整方法論，並與模型特定摘要交叉參考），以下是測試方法的清晰概述。我會保持結構化和客觀，直接從提供的資訊中提取。這項「needle-in-haystack」（針在大海撈針）方法非常適合評估在您的硬體（RTX 5090 配備 32GB VRAM）上的真實情境處理能力，特別適用於 RAG 專案，因為 RAG 需要從大型情境中準確檢索資訊。

### 核心測試設計：針在大海撈針

- **資料結構：**建立一個大型編號項目清單，例如：

text

```
index 1: The value for item 1 is 100
index 2: The value for item 2 is 200
...
index N: The value for item N is N*100
```

- 這創造了一個客觀的「haystack」（乾草堆），其中可以查詢特定「needle」（針，即值）。
- 查詢範例："What is the content of index X?"（預期答案：X\*100）。

- **為什麼用這個設計？**

- 客觀測量：明確的對錯答案，無主觀性。
- 可擴展：容易增加大小來探測極限。
- 位置敏感：揭示模型對情境不同部分（例如開始/結束 vs. 中間）的注意力。
- 可重現：跨模型相同測試，相關於真實文件如 IEC 標準（300+ 頁）。
- 專注於檢索準確度，這對 RAG（檢索增強生成）至關重要，因為需要從文件中可靠提取資訊。

### 位置測試策略

- 每個情境大小測試三個位置：
  - **第一個**（index 1）：檢查首要偏差（模型通常記得開頭很好）。
  - **中間**（index N/2）：最難；測試注意力衰減（LLM 的常見弱點）。
  - **最後一個**（index N）：檢查近期偏差（模型通常記得結尾很好）。
- 每個大小的準確度： $(\text{正確位置} / 3) \times 100\%$ 。例如，67% 表示一個位置（通常是中間）失敗。

### 漸進式擴展

- 情境大小逐步增加：
  - 小型：100–1,000 項目。
  - 中型：1,500–3,000 項目。
  - 大型/極端：3,500–6,500+ 項目（適用於 131k 模型）。
- 停止標準：當準確度 <50%、記憶體不足（OOM）或情境超出時。



- 令牌效率因模型而異：Mistral ~22.6 令牌/項目，Gemma ~14.6，GPT-OSS ~16.7（影響真實內容適合度）。

### 收集的指標

- **準確度**：整體 % 正確；閾值：可靠（≥95%）、可用（≥67%）、不可靠（<67%）。
- **效能**：回應時間（平均秒）、首令牌延遲、令牌計數。
- **利用率**：可用令牌 vs. 廣告/配置。
- **迭代**：每個大小 3 個查詢（每個位置一個）；跨模型總計 ~180 次 API 呼叫，超過 2 小時。
- **溫度**：0（確定性，無隨機性）。
- **錯誤處理**：逾時重試；記錄溢出；GPT-OSS 的輸出格式需特殊解析（例如 `<|channel|>final<|message|>100`）。

### 硬體與設定標準化

- **GPU**：RTX 5090（32GB VRAM）– 關鍵，因為 VRAM 限制有效情境（例如 KV 快取消耗大型情境的記憶體）。
- **推論**：llama.cpp 伺服器在 localhost:8080。
- **量化**：模型特定（Mistral/Gemma 用 Q6\_K，GPT-OSS 用 F16）以適合 VRAM。
- **API 配置**：簡單使用者訊息，max\_tokens 20–50，逾時 120 秒。
- **可重現性**：提供腳本（例如 `context_limit_test.py`），Docker 命令用於設定。

### 方法的關鍵洞見

- **模式揭示**：所有模型在中間位置弱（平均 40% 準確度 vs. 開始 90%/結束 85%）。這突顯 RAG 的真實挑戰– 重要資訊可能在中間「丟失」。

- **硬體現實**：廣告極限（例如 131k）常被 VRAM 削減（例如 Gemma 降至 24k）。
- **權衡**：更大情境 ≠ 更好準確度；速度隨大小衰減。
- **方法限制**：專注於檢索，而非生成/創意。如果需要，單獨測試創意任務。
- **優勢**：客觀、硬體特定、位置意識– 優於僅信任規格。



這個方法實用且與 RAG 相關，因為它模擬從文件中查詢嵌入知識。顯示為什麼「行銷數字」（例如 131k 情境）在消費硬體上不匹配現實。

我對您 Local LLM RAG 專案的評價

現在，基於完整附件集（摘要、方法論和模型特定結果），以下是我的整體評估。我專注於適合 RTX 5090 上的本地 RAG 設定：您需要可靠性（準確檢索）、速度（互動使用）和適當情境（處理文件而不需過多分塊）。RAG 在穩定模型上茁壯成長，這些模型不會在從大型情境中提取時產生幻覺或不可預測失敗。

快速模型比較表（來自您的測試）

模型	可用情境	可靠性/準確度	速度 (平均)	優勢	弱點	RAG 適合度
Mistral 24B	~31k 令牌 (配置 32k 的 96%)	高 (67–100%，可預測；中間偶爾失敗但一致)	0.6s ⚡	快速、穩定、生產就緒；適合文件令牌效率 (~60 頁)。	較小情境；偶爾中間位置弱點 (解決：分塊)。	✅ 整體最佳 – 適合即時 RAG 查詢。
Gemma 27B	~22k 令牌 (訓練 131k 的僅 18%，VRAM 限制)	中等 (33–100%，不穩定；不尋常的開始位置失敗)。	1.5s 🐢	略多項目 (~45 頁)；高效令牌/項目。	較慢、不穩定、VRAM 消耗大；在您的硬體上無法發揮潛力。	❌ 避免 – 在 RTX 5090 上每項關鍵指標都輸給 Mistral。

模型	可用情境	可靠性/準確度	速度 (平均)	優勢	弱點	RAG 適合度
GPT-OSS 20B	~100k 令牌 (配置 131k 的 76%)	低 (0–100%，混亂/隨機；無模式)。	2.7s 	最大情境 (~200 頁)；適合巨型文件。	不可預測失敗 (即使小情境)；特殊輸出格式需解析；慢。	 僅實驗用 – 生產 RAG 太不可靠。

整體推薦：選擇 Mistral 24B

- **為什麼 Mistral？** 根據這些測試，它是 RAG 專案的明顯贏家。它平衡一切：高利用率廣告情境、快速回應 (本地互動 RAG 的關鍵)，以及可預測行為。在 RAG 中，您通常會將文件分塊 (例如 500–1,000 令牌塊)，因此 31k 可用已足夠大多數查詢– 它能可靠處理 60 頁區段，而無 GPT-OSS 的混亂或 Gemma 的限制。您的測試顯示它「生產就緒」，穩定 67–100% 準確度，0.6s 速度意味快速檢索增強回應。
  - 對於 IEC-like 文件：以 60 頁分塊處理；使用滑動視窗處理更大文件 (如方法論建議)。
  - 實作提示：如 Mistral 摘要所述，將最大情境設為 29k 令牌以留安全緩衝。與 LightRAG 或類似整合，將關鍵資訊置於開始/結束以避免中間弱點。
- **為什麼不 Gemma？** 在您的硬體上它被削弱– 儘管訓練 131k，但僅 22k 可用。比 Mistral 慢且不穩定，無優勢。如果想重訪，等更好 GPU (例如 48GB+)。
- **為什麼不 GPT-OSS？** 巨型情境誘人適合全文件 RAG，但「混亂」準確度 (隨機 0–100%) 使其有風險– 無法信任檢索，即使小輸入。而且解析其特殊格式增加麻煩。僅用於非關鍵實驗，或需 200+ 頁時 (帶重試)。
- **對您專案的通用建議：**

- **在您的設定測試：**使用提供的腳本重現（例如 `python context_limit_test.py`）以確認。
- **RAG 優化：**設計應對中間位置問題（所有模型都有）– 例如總結中間或使用混合檢索。
- **硬體注意：**VRAM 是瓶頸；升級時 Gemma/GPT-OSS 可能更閃耀。
- **結論：**Mistral 在 RTX 5090 上提供最佳「性價比」– 可靠、快速且硬體優化。不要追逐更大情境，如果它帶來不穩定。

如果您分享更多 RAG 用例細節（例如文件類型、查詢量），我可以進一步精煉！

↳ 解釋RAG分塊策略

↳ Llama 3模型比較

↳ 更簡潔中文版本