

ChatGPT 5.1 ▾

Share

Add people

...

TEST_TECHNICAL_REPORT.md

File

TEST_SUMMARY_SIMPLE.md

File

TEST_METHODOLOGY_DOCUMENTATI...

File

GPT_OSS_20B_FINAL_TEST_SUMMARY....

File

Mistrial FINAL_TEST_SUMMARY.md

File

GEMMA_27B_FINAL_TEST_SUMMARY....

File

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

Thought for 12s >

先說結論：你的 test method 本身是蠻紮實、而且很適合你現在要比較「context 能力」的目標的，只是如果要直接拿來決定「哪顆 model 最適合整個 RAG 專案」，還可以再加幾層現實世界的測試。

我先幫你把你自己的 methodology 梳理一下，再講優點 / 盲點 / 建議。

你現在的 Test Method，我看到的是什麼

1. 目標定義得很清楚

你是要測：

- 每顆 model 在 RTX 5090 (32GB) 上實際可用的 context 上限
- 在不同 context size 下，正確率從哪裡開始跌破 95%
- 起點 / 中間 / 結尾這三種位置的 retrieval 能力差異
- 延遲 (latency) 與 token 數、VRAM 的關係

TEST METHODOLOGY DOCUMENTATION

這個目標對「我要選一顆 local LLM 來當 RAG backend」是非常對焦的。

2. Needle-in-Haystack + 三點位測試

你用的是經典的 needle-in-haystack：

text

Copy code

```
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
```

Question: What is the content of index X?

Expected: X*100

然後一個 size 測三個位置：

- 第一個 (primacy)
- 正中間 (最困難)
- 最後一個 (recency) □ TEST_METHODOLOGY_DOCUMENTATION

這個設計很好，因為：

- 答案 **客觀可判斷** (要嘛對要嘛錯)
 - 可以直接觀察「中間段掉漆程度」
 - 容易 scale 到幾千條
-

3. Progressive scaling + 清楚的成功門檻

你從小到大一直加：

- 100, 250, 500, 750, 1000, ..., 一直到 6000+ items □ TEST_METHODOLOGY_DOCUMENTATION
- 每個 size 三次測試 (頭 / 中 / 尾)，以 33/67/100% 表示
- 「可靠」定義為 $\geq 95\%$ accuracy，「可用」定義為 $\geq 67\%$ □ TEST_METHODOLOGY_DOCUMENTATION

這個門檻設計是合理的：

- 95% 當 production 級「放心用」
 - 67% 當「知道有洞，但有機會用 workaround 改一下」
-

4. 實作細節有顧到 reproducibility

- 統一硬體：RTX 5090 32GB + llama.cpp server + localhost:8080 □ TEST_METHODOLOGY_DOCUMENTATION
- temperature = 0 、max_tokens 20–50，只取數字答案 □ TEST_METHODOLOGY_DOCUMENTATION
- 有處理錯誤：timeout 重試、context overflow、GPT-OSS 特殊 output format parsing 等 □ TEST_METHODOLOGY_DOCUMENTATION

這部分我覺得做得比很多 paper 還「工程實務」。

優點：為什麼我說這個 method 很對你現在的需求

✓ 1. 直接打臉「官方 context 宣稱」

你用這個方法，成功量化出：

- Mistral 24B：實際可用 31,710 tokens (\approx 97% 的 32k) □ TEST_TECHNICAL_REPORT
- Gemma 27B：雖然宣稱 131k，實際只有 24k 設定、約 21,928 usable (18.7% of 131k) □ TEST_TECHNICAL_REPORT
- GPT-OSS 20B：實際用到 100k tokens，context 利用率 ~76.4% (但準確度很 chaotic) □ GPT_OSS_20B_FINAL_TEST_SUMMARY

這些數字直接回答你最在意的：哪顆 model 在 5090 上真的吃得了多大文件。

✓ 2. 能看出 model 的「性格」

從你 summary 裡可以看出三顆 model 的人格：

- Mistral 24B：
 - context 32k 幾乎吃滿
 - latency 最快 (0.6s) 、token/sec 最高 □ TEST_TECHNICAL_REPORT

- pattern 是「U型」：中間位置偶爾失誤，但可預期修正 □ TEST_TECHNICAL_REPORT
- Gemma 27B：
 - 被 VRAM 卡死，只能用 24k context
 - 準確度一開始超穩，後面突然崩（2000 items 直接 33%）□ GEMMA_27B_FINAL_TEST_SUMMARY
 - token 計數還有 anomaly，一看就知道 sliding window / internal truncation 在偷偷動作
- GPT-OSS 20B：
 - context 最大（100k usable），但準確度完全隨機：33 ↔ 67 ↔ 100% 來回跳□ GPT_OSS_20B_FINAL_TEST_SUMMARY
 - 還有奇怪 channel 格式，需要額外 parsing□ GPT_OSS_20B_FINAL_TEST_SUMMARY

這些結果都來自同一套 method，代表方法本身有區分度，很有 diagnostic value。

3. 指標設計跟實務很貼近

你不只看「能不能跑」，還看：

- latency / tokens/sec
- VRAM 用量（weights / KV cache / activations 分拆）□ TEST_TECHNICAL_REPORT
- position-wise accuracy（頭 / 中 / 尾）□ TEST_TECHNICAL_REPORT

這些資訊，對你要設計 RAG pipeline（chunk 大小、一次塞幾個 chunk、能不能讓 user 等一秒以內）都很有用。

盲點 / 限制：哪裡跟「真實 RAG 場景」還有落差

不是批評，是幫你標出「下一階段可以加的測試」。

⚠ 1. 資料是人工合成、結構超乾淨

現在的 corpus 是：

- 每行格式完全一致 (index N: value ...)
- 沒有自然語言噪音、沒有 table、沒有跨段 reference
- 問題也是單一 pattern：「index X 的值是？」 TEST METHODOLOGY DOCUMENTATION

→ 很適合測 attention + position，但：

- 不會暴露「model 在 technical 文本裡的語義理解力」
- 不會反映它在 **多段文字 + 雜訊 + 標點** 情況的 robustness

建議下一步：

加一組「真實標準文件（例如 IEC 62443 的節錄）」的 QA 測試，問：

- 指定 clause 的內容
- 比較兩個條文差異
- 問「某條文是否包含特定要求」

這樣可以同時看到：

「context 能力 + 語義理解 / 推理」合在一起的實際表現。

⚠ 2. 還沒有把「RAG 的檢索層」一起納入

現在測的是：

「整包 context 都塞給 model 之後，能不能準確指出其中一個 item 的值？」

這是 **純 LLM context/attention test**，還沒考慮：

- chunking 策略 (400 / 1000 / 2000 tokens)

- retriever (BM25 / SPLADE / ColBERT) 會不會選錯段落
- 多 chunk merge 後的順序與位置

也就是說，你的 test 很適合作為：

「LLM 在 已經拿到正確 chunk 的前提下，能否不掉針？」

但如果你要決定「整個 RAG stack 哪個 model 較適合」，未來可以加一個：

- 真實 pipeline : documents → chunk → index → retrieve top-k → concatenate → LLM answer
 - 然後評估 end-to-end 的正確率 / 漏答率
-

⚠ 3. 只看「單點查值」，還沒考 reasoning / summarization

現在的 needle 問題，是「查號碼 + echo 回數字」，這對：

- embedding / attention pattern 很敏感 ✅
- 但對「推理深度、總結品質、多步 reasoning」沒什麼壓力 ❌

未來你可能會：

- 叫 model 整合多段 standard 條文、給出一份合規建議
- 要他比較不同版本的標準差異
- 讓他針對 RFC + IEC + local law 做 cross-reference

這種情境就需要第二類測試：

Real-world task benchmark (例如 5-10 個你實際會丟給 RAG 的合規問題)

用人工評估「答得有沒有到你當顧問會交出去的水準」。

⚠ 4. 單硬體、單 quantization、單 framework

你現在的 setup 很一致（這是優點），但也有：

- 只有 RTX 5090 + llama.cpp + Q6_K / F16 這一種組合 TEST_TECHNICAL_REPORT
- 沒有看：
 - vLLM / TensorRT-LLM 下 context / latency 會不會更好
 - Q4_K / Q5_K / AWQ 對 context / accuracy 的影響
 - 4070 + 5090 分工（雙 GPU）時，context 上限是否能再拉高

這個我知道你已經在想了，只是就 methodology 來說，這是「未來可以拓展的維度」。

我會怎麼幫你「下一步強化」這套測試

你現在這套是：

Stage 1：純 LLM context / attention stress test

我會建議再疊兩層：

Stage 2：真實文件 + QA benchmark

- 選一兩個你最常用的標準（例如 EN 18031、IEC 62443-4-2）
- 手工設計 20–30 題「真實你會問的問題」，每題有標準答案
- 流程：
 1. 用 retriever 找 top-k chunk
 2. 把 chunk (or sliding window) 丟進 model
 3. 人工或用 rule 檢查答案是否符合 ground truth

這樣可以比較：

同一套 retriever 下，不同 LLM 的 end-to-end 合規 QA 表現。

💬 Stage 3 : Reasoning / summarization 小測試

例如：

- 紿 model 3–5 段不同標準 / regulation 節錄
- 問它：「針對某產品，列出需要滿足的安全要求與測試重點」
- 用 1–5 分評分（正確性 / 完整度 / 清晰度）

這一層是補：

「哪顆 model 的文字輸出比較像你會喜歡的顧問報告？」

總結幫你一句話收斂（只針對 test method 本身）

- 你現在的 context window real-world test method，非常適合拿來回答：
“在 RTX 5090 上，哪顆 model 的實際 context 能力最好、最穩、最快？”
- 方法在「設計、指標、實作、debug」層面我覺得都很紮實，有足夠的 engineering 品質，可以當你未來所有 model 的固定 benchmark 之一。 ⓘ TEST_TECHNICAL_REPORT
- 若要進一步用它來 決定 “哪顆 model 最適合整個 local RAG 專案”，我會建議你再加上：
 - 真實標準文件 QA 測試 (RAG pipeline)
 - reasoning / summarization 測試
 - 之後有空再擴到 vLLM / 其他 quantization / 雙 GPU

你如果接下來要，我可以幫你把「Stage 2 / Stage 3」的測試腳本 & 評分表直接設計出來，讓你用同一套風格繼續 benchmark。

Sources



評價一下 你對3 llm model的看法

Thought for 47s >

直接講白話結論先：

在你現在這台 5090 上，這三顆的定位很清楚：

- Mistral 24B = **主力生產線** (RAG 前線主力)
- GPT-OSS 20B = **研究用長文怪獸** (離線 / 非關鍵任務)
- Gemma 27B = **硬體不配合的天才** (先收起來，等 48–80GB VRAM 再說)

下面我一顆一顆講我對它們的「性格評價」跟在你專案裡最適合扮演的角色。

1 Mistral-Small-24B — 你現在應該的主力工兵

我對它的看法：這顆就是「務實、好養、好預測的主力模組」。

實測特性（你自己的數據）

- 實際可用 context : 31,710 tokens ($\approx 96.8\%$ 的 32k) □ TEST_TECHNICAL_REPORT
- Max reliable items : 1,400，還能維持 95% 以上準確率 □ TEST_TECHNICAL_REPORT
- 平均延遲 : 0.6s，tokens/sec ≈ 52.8 ，first token 0.3s，很適合互動式問答 □ TEST_TECHNICAL_REPORT
- VRAM : 整體只吃大約 23GB，還留 ~9GB 紿其他東西 (RAG index / 第二顆 model)

□ TEST_TECHNICAL_REPORT

- pattern：標準 U 型，頭尾幾乎完美，中間有固定 pattern 的掉漆，但行為是「可預測」的

TEST_TECHNICAL_REPORT

我怎麼解讀

- 對你最重要的三件事：開發速度、品質、再來才是推理速度
 - 行為穩定 + pattern 可預測 → debug 成本最低、Prompt / chunk 策略比較好調
 - 反而比起一顆「很大但亂飄」的 model，更符合你「希望系統可控」的習慣。
- 32k context 在 5090 上已經被你吃到極限，而且利用率接近 97% → 這顆在這張卡上，算是「規格剛剛好，發揮完全」。

我給它的定位

- RAG 主力 QA 模型 (production front-line)
 - 所有互動式查詢、合規 Q&A、條文對比、測試計畫草案 → 先走 Mistral。
- 設計 LightRAG / MiniRAG 時的 baseline：
 - 以 ~29k tokens 保守上限去設計 chunk/merge Mistral FINAL_TEST_SUMMARY
 - 用 sliding window 補中間段的弱點 Mistral FINAL_TEST_SUMMARY

簡單講：在你現在的硬體條件下，這顆就是「標準答案」。

2 Gemma-3-27B — 被 5090 繩手繩腳的天才

我對它的看法：理論很美、實務在你這台機器上完全發揮不了。

實測特性

- 訓練 context : 131k tokens，但
- 在 5090 (32GB VRAM) 上只能設定 24,576 tokens (約 18.7% 能力) GEMMA_27B_FINAL_TEST_SUMMARY

- 實際可用 context (在這個 24k 設定下)：21,928 tokens (利用率 89.2%) □ GEMMA_27B_FINAL_TEST_SUMMARY
- Max reliable items：約 1,500，但 2,000 items 直接 33% 大爆死 □ GEMMA_27B_FINAL_TEST_SUMMARY
- TEST_TECHNICAL_REPORT
- 延遲：平均 1.5s，P95 到 3.9s，比 Mistral 慢 2.5 倍以上 □ TEST_TECHNICAL_REPORT
- VRAM 佔用：大約 29GB，只剩 3GB 餘裕，空間超緊 □ TEST_TECHNICAL_REPORT
- 行為特性：accuracy pattern 很怪、很 erratic，甚至有 first-position fail 的情況 □ GEMMA_27B_FINAL_TEST_SUMMARY
- GEMMA_27B_FINAL_TEST_SUMMARY

我怎麼解讀

- 你實際測出來的對比表裡，Mistral 在 7 項裡贏 5 項：context、usable tokens、速度、穩定度、VRAM usage，全面輾壓 □ GEMMA_27B_FINAL_TEST_SUMMARY
- Gemma 真正的優點 (131k、local+global attention、sliding window) 在 24k config 下通通被閹割 □ GEMMA_27B_FINAL_TEST_SUMMARY
- 換句話說：在你現在這張 5090 上，Gemma 只是一顆又大又貴、表現還比 Mistral 差的 27B。

我給它的定位

- 在「單張 5090、本地部署」這個世界線 → 直接 PASS：
 - RAG 主力不適合
 - 當 re-ranker / summarizer 也沒有特別優勢 (context 小、速度慢、VRAM 重)
- 可以保留當「未來雲端或 48GB+ GPU」候選：
 - 如果你之後搞 A6000 / H100 或 DGX，那時候再讓 Gemma 上場，就有機會真的用到 65k–131k context □ GEMMA_27B_FINAL_TEST_SUMMARY

你自己的結論其實已經很準：在 5090 上，用 Gemma 是浪費，這點我完全同意。

□ GEMMA_27B_FINAL_TEST_SUMMARY

3 GPT-OSS-20B F16 — 超長 context 但「人格混亂」的怪胎

我對它的看法：是一顆「研究向 / 玩具向」的長上下文怪獸，適合做特殊任務，不適合當前線。

實測特性

- 實際可用 context : 100,093 tokens (約 76.4% 的 131k config) □ GPT_OSS_20B_FINAL_TEST_SUMMARY
□ TEST_TECHNICAL_REPORT
- Max reliable (100% 準確) 只有 500 items，雖然往上到 6,000 items 還能跑，但 accuracy 在 33 / 67 /100% 之間亂跳 □ GPT_OSS_20B_FINAL_TEST_SUMMARY
□ TEST_TECHNICAL_REPORT
- pattern：混亂振盪，100 items 可以 33%，500 又變 100%，2500 再 100%，5000 又掉到 33%，完全無法預測 □ TEST_TECHNICAL_REPORT
□ GPT_OSS_20B_FINAL_TEST_SUMMARY
- 平均延遲：2.7s，tokens/sec ≈ 14.8，比 Mistral 慢約 4.5 倍 □ TEST_TECHNICAL_REPORT
□ TEST_TECHNICAL_REPORT
- VRAM：最高到 31GB 左右，只剩 1GB margin □ TEST_TECHNICAL_REPORT
- 特殊問題：輸出是 <|channel|>analysis + <|channel|>final 兩段，要額外寫 parser 處理 □ GPT_OSS_20B_FINAL_TEST_SUMMARY
□ TEST METHODOLOGY DOCUMENTATION

我怎麼解讀

- 它最大的優點只有一個：context 超大 → 你真的可以一口氣塞一整本 IEC 標準進去。
□ GPT_OSS_20B_FINAL_TEST_SUMMARY
- 但你也測到：大 context ≠ 好準確率，甚至連小 context 都可能翻車 (100 items 33%)
□ GPT_OSS_20B_FINAL_TEST_SUMMARY
- 以你這種「要做正式合規、要寫 test plan、要給客戶報告」的使用場景，這種隨機跳動的 accuracy，完全沒辦法當主力。

我給它的定位

- 不能當 production front-line model：
你自己在 report 也寫了 "DO NOT use GPT-OSS 20B for production"，我完全同意。

GPT_OSS_20B_FINAL_TEST_SUMMARY

- 但我覺得它可以保留作為「背景專用工具」：

1. 離線全書總結 / mapping agent

- 一次丟整本 EN 18031 / IEC 62443 進去，讓它產出「粗略章節 summary / dependency map」，再由 Mistral 做細緻 QA。

2. 第二意見 / cross-check agent (低頻觸發)

- 對某些 edge case 問題，讓 Mistral 和 GPT-OSS 都回答一次，你寫個小工具做 diff，比較觀點。

3. 實驗場：測試各種長 context prompt 工程想法

- 例如超長 system prompt、整本 spec 當前文 → 觀察行為，幫你理解「大 context 下 LLM 的 failure mode」。

一句話：GPT-OSS 20B 很有趣、很好玩，但不是你要拿去跟客戶交差的那種 model。

最後幫你收斂：在你的 local RAG 專案裡，三顆怎麼分工？

- 主力（必上）：Mistral 24B Q6_K

- 所有「合規 QA、test case 設計、報告草稿、日常 RAG 查詢」→ 這顆當 default。

Mistrial FINAL_TEST_SUMMARY

- 備用工具 / 研究用：GPT-OSS 20B F16

- 只在你需要「一次吞 100k tokens」的時候出動，
- 永遠當「非關鍵、可接受錯、可人工檢查」的 background job。 GPT_OSS_20B_FINAL_TEST_SUMMARY

- 暫時下放冷凍：Gemma 27B Q6_K

- 不列入 5090 的 production 設計，
- 保留權利：未來如果你搞到 48GB/80GB VRAM，再重新評估它當超長 context 主力。

GEMMA_27B_FINAL_TEST_SUMMARY

如果你接下來想，我可以直接幫你畫一張「三顆 model 在 LightRAG / MiniRAG pipeline 裡的分工圖」，順便把 call 流程（哪些任務丟哪一顆、什麼情況 fallback、怎麼做 voting）寫成 pseudo-code，讓你直接貼進你的 orchestrator。

Sources

