



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Profiling Framework for LLM-Based Data Agents: Quality, Latency, and Energy under Parameter Sensitivity

TESI DI LAUREA MAGISTRALE IN

COMPUTER SCIENCE AND ENGINEERING - INGEGNERIA INFORMATICA

Davide Valenzisi, 245001

Ossama El Oukili, 996171

Advisor:
Prof. Danilo Ardagna

Academic year:
2025-2026

Abstract: Large Language Models (LLMs) increasingly power natural language interfaces to structured data, yet deploying them in production raises concerns beyond accuracy, namely inference latency and energy cost. To study these, this thesis builds a *Sales Agent*: an LLM-powered system, orchestrated with LangGraph, that translates a business question into a SQL query, interprets the results in text, and optionally renders a chart. A hierarchical YAML configuration exposes the studied inference parameters (sampling parameters, best-of- N , token budgets, and Chain-of-Thought refinement depth). The core contribution is a systematic profiling that jointly measures output quality, execution time, and per-step energy consumption (via CodeCarbon). Quality is assessed over 20 analytical questions, stratified across four SQL difficulty levels, along three dimensions: a graded ground-truth table-similarity score for retrieval and two LLM-as-judge scores for analysis and visualization. Five step-isolated experiments establish a baseline for three open-weight models (Gemma 4 E4B, Gemma 4 26B, Mistral Small 3.2 24B) and then vary one agent step at a time, closing with a Pareto confirmation run. They are also designed to test a common assumption in local LLM deployment, that larger thinking models are necessarily the best solution. The results show instead that model choice alone does not explain performance: the dominant effects arise from the interaction between model family, agent step, prompt difficulty, and inference-time settings. Mistral is the most efficient untuned model but collapses on the hardest aggregation prompts; once its lookup step is repaired through Chain-of-Thought refinement, however, it attains the highest strict quality and defines the final Pareto frontier. Gemma 26B is highly tunable yet energetically costly, and Gemma E4B, repaired through lookup Chain-of-Thought refinement at a lower temperature, remains a very close high-quality alternative but is dominated once energy is counted. The sensitivity analysis further shows that hyperparameter effects are strongly step-specific, that larger token budgets help only genuinely truncated model-step pairs, and that Chain-of-Thought refinement pays off only when its feedback matches the failure mode: it is the strongest intervention for verifiable SQL lookup, but harmful for visualization, where errors are semantic rather than executable. The thesis distills three recommendations for energy-aware LLM data agents: tune hyperparameters per agent step rather than globally, allocate inference-time compute by failure mode, and report accuracy and energy by prompt difficulty, since the best model for simple tasks is not the best for hard ones.

Key-words: LLM data agents, Energy-aware profiling, Quality-energy trade-off, Local open-weight LLMs, LLM-as-judge evaluation

1. Introduction

The growing adoption of Large Language Models (LLMs) in enterprise software has opened a concrete path toward natural language interfaces for structured data. Systems capable of translating a business question into a SQL query, executing it, and returning both a textual analysis and a chart represent a significant step forward in making data-driven decision making accessible to non-technical users. At the same time, deploying such systems in production raises practical questions that go beyond accuracy: how much time does each inference take, how does it scale with configuration changes, and what is the energy cost of running LLM-backed pipelines on real hardware?

This thesis addresses both aspects. On the application side, it presents the design and implementation of a *Sales Agent* — an LLM-powered conversational system that enables business analysts to query a synthetic retail dataset using plain English. Given a natural language question, the agent autonomously selects the appropriate processing steps, generates and executes a SQL query against Parquet files via DuckDB, produces a textual interpretation of the results, and optionally renders a Matplotlib chart. The agent is orchestrated through LangGraph [1], a framework for stateful graph-based LLM pipelines, and supports two interchangeable backends: locally-hosted open-weight models via Ollama and cloud-based models via the OpenAI API. A hierarchical YAML configuration system exposes the inference parameters studied in the experiments — sampling temperature, top- p , top- k , best-of- N candidates, and Chain-of-Thought refinement iterations — making every run fully reproducible and configurable without code changes.

On the experimental side, the primary focus of the thesis is the systematic profiling of the agent under varying configurations. A benchmark of 20 analytical questions, stratified across four SQL difficulty levels, is used to evaluate output quality across three complementary dimensions: a graded ground-truth table-similarity score for data retrieval accuracy, an LLM-as-judge score for the natural language analysis, and a further LLM-based score for visualization correctness. A central motivation is to test whether larger thinking models are always the best choice, or whether smaller thinking models and larger non-thinking models can be more efficient once the agent steps are tuned. The evaluation is structured as a sequence of five step-isolated experiments. The first establishes a quality and energy baseline for three locally-hosted open-weight models — Gemma 4 E4B, Gemma 4 26B, and Mistral Small 3.2 24B. The subsequent tests vary one agent step at a time: a one-factor-at-a-time sensitivity analysis across sampling hyperparameters, a token-budget ladder, and a compute-expansion study combining best-of- N sampling with Chain-of-Thought refinement. A final Pareto confirmation run then identifies the configurations that best balance output quality against inference cost. For every run, execution time and energy consumption — tracked per step using CodeCarbon [2] — are recorded alongside quality scores, providing actionable guidance for deploying LLM agents under latency or sustainability constraints.

The thesis is guided by the following research questions:

- **RQ1:** How do the evaluated locally hosted LLMs compare in terms of end-to-end output quality, execution latency, and energy consumption when used as the backbone of a multi-step data-analysis agent?
- **RQ2:** How sensitive are the individual agent steps — data lookup, textual analysis, and visualization — to changes in sampling parameters and output-token budgets?
- **RQ3:** Under which conditions does additional inference-time computation, through iterative refinement or best-of- N sampling, improve output quality sufficiently to justify its additional latency and energy cost?
- **RQ4:** How does SQL-query difficulty influence the relative performance of the evaluated models and the resulting quality–energy trade-off?
- **RQ5:** Which evaluated model and inference-setting combinations provide the most favourable empirical trade-offs between strict end-to-end quality, completion, latency, and local energy consumption?

Accordingly, the main contributions of this work are:

- a configurable multi-step data-analysis agent for querying structured retail data through natural language;
- a profiling pipeline that records per-step quality, latency, and energy measurements for local LLM executions;
- a controlled sensitivity analysis across model–step pairs, separating the effect of lookup, analysis, and visualization parameters;
- empirical deployment recommendations derived from strict end-to-end quality, completion, latency, and energy trade-offs.

This thesis is organised as follows:

- Section 2 reviews related work on LLM agents for structured data analysis, text-to-SQL generation, local model families and inference-time reasoning strategies, evaluation of data-agent outputs, and efficiency-aware benchmarking of local LLM agents;
- Section 3 describes the architecture of the Sales Agent, covering the agent graph, each processing step, the configuration system, and the benchmark dataset;

- Section 4 presents the experimental design, including the server hardware environment, execution protocol, and the specification of the five test campaigns;
- Section 5 presents the experimental results, analysing output quality, energy consumption, and the Pareto trade-offs across models and configurations;
- Section 6 summarises the findings and outlines directions for future work.

2. State of the Art

This section reviews the research areas that frame the thesis. Section 2.1 introduces LLM agents for structured data analysis, while Section 2.2 discusses Text-to-SQL and database-grounded generation as the core mechanism behind the lookup step. Sections 2.3-2.5 review the model families used for local agent inference, the orchestration and inference-time reasoning strategies that can improve multi-step behaviour, and the evaluation methods required to score data, textual, and visual outputs. Finally, Sections 2.6 and 2.7 consider efficiency-aware benchmarking of locally hosted LLMs and summarise the research gap addressed by this work.

2.1. LLM Agents for Structured Data Analysis

Large Language Models have evolved from single-turn text generators into the reasoning core of multi-step, tool-using systems. An *LLM agent* extends a base model with the ability to invoke external tools — such as code executors, search engines, or database interfaces — and to maintain state across multiple interaction steps [3]. This architectural shift enables a qualitatively different class of applications: rather than producing a single response to a question, the agent orchestrates a sequence of operations whose intermediate outputs inform subsequent decisions.

In the context of data analysis, this capability is especially significant. A *data-analysis agent* receives a natural language business question and must translate it into a sequence of formal operations: identifying the relevant data sources, generating and executing a database query, interpreting the numerical results, and optionally producing a visualization. Each step depends on the output of the previous one, and the final answer is not a single text response but a composite artifact comprising retrieved data, a written interpretation, and possibly chart code. This makes the task structurally closer to an analytical workflow than to a conversational exchange. The central challenge for such agents is reliability across all steps. The model must correctly ground the question in the available schema, produce syntactically and semantically valid queries, avoid unsupported numerical claims in the textual interpretation, and generate executable visualization code. Failure at any step propagates downstream: an incorrect query yields a wrong result set, which in turn can produce a plausible-sounding but factually incorrect analysis.

Connection to this work. This thesis instantiates this class of systems as a *Sales Agent*: an LLM-powered analytical workflow over a retail dataset that follows the same three-phase structure — data retrieval, textual interpretation, and visualization — that characterises the broader class of data-analysis agents described above.

2.2. Text-to-SQL and Database-Grounded Generation

Text-to-SQL is the task of mapping a natural language question to an executable database query. It has been studied extensively through shared benchmarks such as Spider [4], which introduced cross-domain multi-table schemas as a standard evaluation setting, and BIRD [5], which extended the problem to real-world databases with noisy values and ambiguous content, making execution accuracy — whether the query returns the correct result set — a more meaningful correctness target than exact SQL string matching. While modern models now achieve over 85% execution accuracy on Spider, the same generation of models reached only approximately 55% on BIRD at the time of its release, highlighting that real-world database complexity remains a substantially harder challenge than cross-domain schema generalisation alone. These benchmarks established the core challenges of the field: schema understanding, column grounding, join selection, aggregation logic, filtering, grouping, and date handling. Multi-table schemas are particularly demanding because the model must identify which tables are relevant, how they relate, and which join keys to use, without being able to rely on column names alone.

A structural difficulty in evaluation is that SQL equivalence cannot be measured by string comparison: semantically different queries can produce identical result sets, and vice versa. Execution accuracy is therefore a more robust correctness signal, though it too has limits when result sets are large or partially ordered. In data-agent settings, where the query output is consumed directly by downstream steps, the meaningful correctness target is the returned table rather than the SQL string itself.

Prompt engineering has become a primary lever for improving text-to-SQL quality with instruction-tuned LLMs. Static few-shot examples embedded in the system prompt — fixed question-to-query pairs that illustrate expected output format, join patterns, and aggregation conventions — provide in-context guidance without requiring an additional retrieval step. More sophisticated approaches such as DIN-SQL [6] decompose the generation into sub-tasks — schema linking, query classification, and self-correction — each guided by static few-shot examples, achieving higher accuracy on complex queries at the cost of additional LLM calls. DIN-SQL with GPT-4 reached 85.3% execution accuracy on Spider, compared to 67.4% achieved by GPT-4 with direct few-shot prompting, demonstrating the benefit of task decomposition over end-to-end generation.

Connection to this work. This thesis treats SQL generation as a dedicated agent step that uses static few-shot examples and schema descriptions to ground the model in the available data. The benchmark used for evaluation spans four difficulty levels — flat aggregation, time bucketing, single join, nested aggregation, conditional metrics, ratios, and year-over-year comparisons — which map directly onto the failure modes documented in the text-to-SQL literature: incorrect aggregation level, wrong date extraction, missing filters, and incorrectly structured output tables.

2.3. Model Families for Local Agent Inference

Model choice is a first-order variable in LLM agent performance, but its effect is not uniform: size, architecture, and reasoning orientation interact with the task structure in ways that matter for both quality and efficiency.

Model size. Larger models generally achieve higher task accuracy, but the relationship is not proportional across all tasks. For straightforward formatting or routing decisions, small models can match the output of much larger ones; for complex multi-step reasoning such as nested SQL generation, the gap widens substantially. At the same time, larger models increase inference latency, memory pressure, and energy consumption, making size a genuine trade-off rather than a free variable.

Reasoning-oriented vs. instruction-following models. A growing number of open-weight models are trained or fine-tuned to expose explicit intermediate reasoning — a chain of thought produced before the final answer — at inference time. These *thinking* models can improve accuracy on tasks that benefit from deliberate step-by-step analysis, such as complex SQL generation or numerical interpretation, but they produce longer outputs and are therefore slower and more energy-intensive than equivalently sized instruction-following models. Instruction-following models, by contrast, are optimised for direct, concise responses and tend to be more efficient on tasks where structured output format matters more than reasoning depth — such as tool routing or chart configuration.

Dense vs. Mixture-of-Experts architecture. In a standard dense model, all parameters participate in every forward pass; inference cost scales roughly with the total parameter count. Mixture-of-Experts (MoE) models partition the feed-forward layers into a set of specialist sub-networks and activate only a learned subset of them per token [7]. This allows MoE models to maintain large representational capacity while keeping active compute lower than an equivalently-sized dense model, producing a quality-per-FLOP profile that differs from both small and large dense models.

Local inference. When models are deployed locally on a single GPU rather than called through a cloud API, these architectural differences become directly observable: GPU energy, memory pressure, tokens-per-second throughput, and wall-clock latency all vary with model family and can be measured per step under controlled conditions. The same configuration that is cost-effective for SQL generation may be unnecessarily expensive for the simpler analysis or visualization steps.

Connection to this work. The experiments compare three models serving distinct roles in the quality-efficiency space: *Gemma 4 E4B* [8] is a small thinking model that represents the low-cost end of the reasoning spectrum; *Gemma 4 26B* [8], a larger MoE thinking model, offers higher representational capacity with partially reduced active compute; and *Mistral Small 3.2 24B* [9] is a large dense instruction-following model that serves as a non-thinking reference. All models are served locally via Ollama on a single GPU, making per-step latency and energy directly comparable across configurations.

2.4. Agent Orchestration and Inference-Time Reasoning Strategies

Building a multi-step data-analysis agent requires more than choosing a capable model: the structure of the workflow, the allocation of inference compute across steps, and the prompting strategy applied to each node all influence the final output quality and efficiency independently of model choice.

Graph-based orchestration. Agent frameworks organise LLM calls, tool invocations, and state transitions into explicit workflows that are easier to inspect, test, and reconfigure than monolithic prompt chains. Graph-based orchestration is particularly appropriate when an agent must decide at runtime which step to execute next and when to terminate — decisions that depend on intermediate outputs rather than a fixed linear schedule. LangGraph [1] represents an LLM application as a stateful directed graph: nodes encapsulate processing steps, edges encode conditional routing logic, and a shared state object carries information between nodes across iterations. This design makes it possible to expose individual nodes as independent experimental units, varying one step’s configuration while keeping the others fixed.

Observability and distributed tracing. As agentic workflows grow in complexity, understanding what happens inside each execution becomes essential for debugging and iterative improvement. OpenTelemetry [10] is an open-source, vendor-neutral standard for distributed tracing: it defines APIs and SDKs for emitting *spans* — structured records of individual units of work, each carrying a start time, duration, input/output attributes, and a parent reference that links it to the enclosing operation. Arize Phoenix [11] is an LLM observability platform built on top of OpenTelemetry that provides span-level visibility into LLM calls, including prompt templates, model responses, token counts, and latency breakdowns. In multi-node agent workflows, per-span tracing makes it possible to attribute latency and failures to specific nodes, inspect the exact prompt sent to the model at each step, and replay individual spans for offline evaluation — none of which is visible from run-level logs alone.

Chain-of-Thought prompting. Chain-of-Thought (CoT) prompting [12] instructs the model to produce explicit intermediate reasoning steps before generating a final answer. Externalising the reasoning process reduces the likelihood of skipped steps or implicit assumptions, and improves accuracy on tasks that require multi-step inference such as arithmetic, symbolic reasoning, and structured query generation. **Iterative CoT refinement** extends this idea: the model’s initial output is fed back into the same prompt together with a directive to verify or improve it, forming a self-correction loop that can recover from minor errors without human intervention. This self-correction is not universally beneficial, however: without a reliable external feedback signal, large language models often fail to improve — and can even degrade — their own outputs through iterative self-criticism [13], so the value of refinement depends on whether the verification signal is aligned with the property being evaluated.

Best-of- N sampling and self-consistency. Best-of- N sampling generates multiple candidate outputs for the same input and selects the best one according to an evaluation criterion. Self-consistency [14] is a specific realisation of this strategy in which the selection rule is consensus: the candidate that agrees most with the others is preferred, exploiting output redundancy rather than requiring a reference answer. Both approaches treat inference-time compute as a resource that can be exchanged for quality: generating N candidates multiplies LLM calls by N and increases energy consumption proportionally, so the quality gain must be assessed against the additional cost.

Per-step compute allocation and token budget. Because different agent steps have different failure modes, optimal compute allocation is not uniform across the workflow. SQL generation is sensitive to structural errors — wrong joins or incorrect aggregation — that benefit from candidate diversity and re-ranking. Textual analysis requires numerical consistency across potentially many retrieved values. Chart code generation requires syntactic correctness that can be verified deterministically by attempting execution. Each step therefore has a different profile of how much additional inference compute converts into measurable quality improvement. Token budget is a complementary control variable. Setting the maximum output length too low can truncate reasoning chains or produce incomplete code, while a cap that is already non-binding could add no measurable cost.

Connection to this work. The thesis uses a LangGraph workflow with four nodes — tool routing, lookup, analysis, and visualization — each with an independent set of sampling and compute parameters. This per-step parameterisation enables step-isolated experiments: one node’s configuration is varied at a time while the others

are held at their baseline values, so that the effect of each form of additional inference compute can be measured separately against both quality and energy metrics.

2.5. Evaluation of Data-Agent Outputs

Evaluating a data-analysis agent is fundamentally harder than evaluating a single chat response because the output is composite: a retrieved table, a natural language interpretation, and visualization code each require a different correctness criterion, and failure at any one step is not necessarily reflected in the others.

Table and SQL evaluation. Since many syntactically different SQL queries can produce the same correct result, string comparison of generated SQL against a reference is not a reliable quality signal. Execution accuracy — comparing the result set returned by the agent’s query against a reference result set — is more meaningful, but exact row-level comparison is brittle: the agent may return correct values with different column names, different column ordering, boolean labels encoded as integers instead of strings, or results in long format where the reference is wide (or vice versa). Graded similarity metrics that tolerate these surface variations while measuring the overlap of actual data values are more appropriate for benchmarking, where the goal is to rank configurations rather than pass or fail individual queries.

A natural choice for this purpose is *Intersection over Union* (IoU), a set-overlap metric widely used in computer vision and information retrieval that extends cleanly to tabular result sets. Let G be the multiset of rows in the ground-truth table and P the multiset produced by the agent. The row-level IoU is:

$$\text{IoU}(G, P) = \frac{|G \cap P|}{|G \cup P|} \quad (1)$$

where intersection and union are computed in multiset semantics so that duplicate rows contribute proportionally rather than being collapsed. The score lies in $[0, 1]$: a value of 1 indicates an exact match; values below 1 reflect either missing rows (the agent retrieved too few), spurious rows (it retrieved too many), or value mismatches. Crucially, the metric is order-independent — row permutations do not affect the score — which makes it robust to `ORDER BY` differences across semantically equivalent queries. Before applying the formula, a normalisation pass is required to absorb surface-level representational differences (numeric precision, date formatting, column naming) that do not reflect genuine retrieval errors; the specific normalisation choices are an implementation detail described in Section 3.9.

Textual analysis evaluation. Standard n-gram overlap metrics such as BLEU or ROUGE measure surface similarity between generated and reference text, but are poorly suited to evaluating numerical reasoning: a response that correctly identifies the top-performing region but uses different phrasing scores low, while one that uses the right words with wrong numbers scores high.

LLM-as-judge evaluation [15] addresses this by using a capable language model as an automatic evaluator. Two main paradigms exist: *pairwise comparison*, where the judge ranks two candidate outputs against each other, and *pointwise scoring*, where the judge assigns an absolute score to a single output against a rubric. Pointwise scoring is better suited to benchmarking scenarios where many configurations must be compared, since it does not require all pairs to be evaluated explicitly. Zheng et al. showed that strong LLMs used as judges achieve over 80% agreement with human evaluators on open-ended tasks, making the approach a practical substitute for human annotation at scale.

The main limitations are well-documented: judges tend to favour longer or more fluent responses regardless of factual content (verbosity bias), may prefer outputs that resemble their own generation style (self-enhancement bias), and can be sensitive to the order in which options are presented or to minor changes in the evaluation prompt. These effects should be controlled by fixing the judge model and temperature across all evaluated configurations, and by designing rubrics that anchor scores to explicit, verifiable criteria rather than overall quality impressions.

Visualization evaluation. Chart quality cannot be assessed through text comparison alone: the generated code must first be executable, and the resulting chart must match the reference in terms of chart type, axis assignments, data grouping, and any explicit styling requirements. Evaluation therefore combines a deterministic check — whether the plotting code runs without errors — with semantic criteria assessed by an LLM judge comparing the agent’s chart configuration and code against the ground-truth equivalents.

Failure diagnostics. Aggregate scores across a benchmark conceal step-specific failure patterns. An agent may achieve high text scores while consistently failing on SQL joins, because the analysis step produces plausible text even from partially incorrect data. Evaluation pipelines should therefore produce per-item diagnostic

output that identifies at which step correctness was lost, enabling targeted analysis of where additional inference compute or prompt improvements are most needed.

Connection to this work. The benchmark evaluates agent outputs across three dimensions: an IoU-based table similarity score for data retrieval, and LLM-as-judge scores for textual analysis and visualization quality. For best-of- N candidate selection, a parallel set of reference-free evaluators is used [14], so that ground-truth information does not influence generation or candidate ranking.

2.6. Efficiency-Aware Benchmarking of Local LLM Agents

As LLM-based systems move from research prototypes to deployed applications, reporting quality in isolation is increasingly insufficient. The energy and carbon cost of training large models has been documented extensively [16, 17], but inference-time costs — which accumulate across every request in a deployed system — have received comparatively less attention, particularly in the context of multi-step agentic pipelines.

For locally-hosted models, inference energy is especially tractable to measure: the GPU is under direct experimental control, execution is sequential, and software tools such as **CodeCarbon** [2] can attribute CPU, GPU, and RAM energy consumption to specific code regions with per-run granularity. This makes local deployment a natural setting for controlled energy-performance experiments that would be opaque when calling a cloud API. In modular agents, run-level energy totals are informative but coarse. Different workflow nodes can have very different cost profiles: a SQL generation step with best-of- N sampling and a thinking model may consume an order of magnitude more GPU energy than a lightweight routing step. Per-step energy attribution is therefore more actionable than a single run-level figure, because it identifies which node dominates cost and where optimisation effort is best directed.

Energy-aware evaluation should go beyond reporting the highest-quality configuration and identify Pareto-efficient points on the quality-energy frontier: configurations that preserve most of the quality gain at substantially lower energy or latency. Useful derived metrics for this analysis include quality per kWh, quality gain per second relative to the baseline, and delta quality versus delta energy compared to a reference run.

Connection to this work. Each benchmark run records elapsed time, total run-level energy, and per-step LLM energy broken down by CPU, GPU, and RAM via CodeCarbon. The experimental design covers baseline model comparisons, step-specific hyperparameter sensitivity, maximum-token ladders, and compute expansion through best-of- N and CoT refinement — each representing a distinct form of inference-time investment. The final analysis distinguishes configurations that maximise output quality from those that achieve comparable quality at lower energy cost, providing Pareto-efficient deployment recommendations under latency or sustainability constraints.

2.7. Synthesis and Research Gap

Existing research provides the building blocks for LLM-based data-analysis agents: text-to-SQL methods for database grounding, graph-based orchestration for multi-step tool use, model families with different inference-time behaviour, prompting and self-consistency strategies for improving output quality, semantic evaluation methods for generated text and code, and software tools for estimating inference energy consumption. However, these aspects are typically studied in isolation: text-to-SQL benchmarks focus on query accuracy without considering downstream steps; orchestration frameworks are evaluated on task completion without energy accounting; and energy studies tend to target training rather than multi-step inference. This thesis combines all of these dimensions in a single reproducible setting: a modular data-analysis agent is evaluated across data retrieval correctness, textual analysis quality, and visualization quality simultaneously, while latency and per-step energy consumption are recorded for every configuration. Step-isolated experiments then identify which model families and which inference settings — sampling temperature, top- p , top- k , token budget, best-of- N sampling, and Chain-of-Thought refinement — improve the quality-energy trade-off, and at what cost.

3. Application: Sales Agent

This section describes the Sales Agent implemented for the thesis. Section 3.1 presents the system overview, while Sections 3.2 and 3.3 detail the system architecture and the LangGraph workflow that orchestrates its steps. Section 3.4 describes the individual agent tools and steps used for data lookup, textual analysis, and visualization, and Section 3.5 explains the LLM integration and prompting techniques. Section 3.6 presents the YAML-based configuration system, and Section 3.7 describes the multi-table dataset support. Finally,

Section 3.8 presents the benchmark design and Section 3.9 details the evaluation, output, and energy-monitoring pipeline used to profile the agent.

3.1. System Overview

The Sales Agent is an LLM-powered conversational system designed to enable business analysts to query and explore retail sales data using natural language, without requiring knowledge of SQL or any underlying data infrastructure. Given a question expressed in plain English, the agent autonomously determines the appropriate processing steps, retrieves the relevant data, produces a textual analysis, and optionally generates a chart to support data-driven decision making.

The system operates on a synthetic retail dataset composed of three relational tables stored in Parquet format: *sales*, which records daily store-level transactions with pricing and promotion information; *products*, which contains the product catalog including category, brand, unit price, and origin; and *stores*, which holds store metadata such as location, type, size, and staffing. The tables are linked through shared keys (*Store_Number* and *SKU_Coded*), allowing the agent to answer questions that span multiple entities through SQL JOIN operations. The agent is orchestrated via LangGraph [1], a framework for building stateful, graph-based LLM pipelines. At each invocation, the user’s natural language question is routed through a directed graph of processing nodes: a *tool selection* node decides which tools to activate, a *lookup* node generates and executes a SQL query via DuckDB to retrieve structured data, an *analysis* node produces a natural language summary of the results, and a *visualization* node generates Python chart code. The LLM backbone is configurable: the system supports both locally-hosted models via Ollama and cloud-based models via the OpenAI API, making it suitable for deployment in both resource-constrained and high-performance environments.

Figure 1 illustrates the high-level architecture of the Sales Agent, showing the flow from user input to final output across the agent’s processing pipeline.

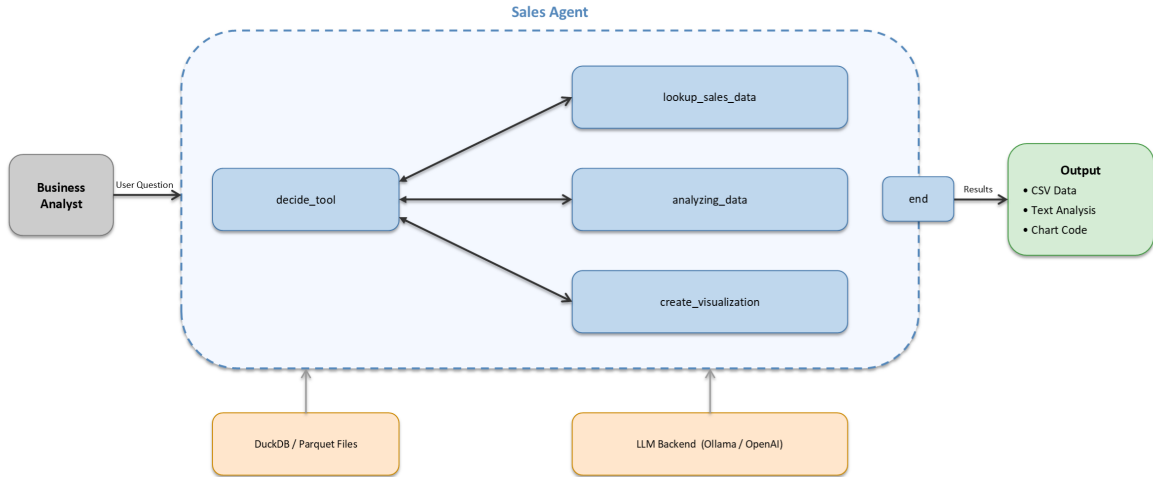


Figure 1: High-level architecture of the Sales Agent.

The output of the system consists of three complementary components: a structured CSV table containing the query results, a natural language analysis interpreting the data in the context of the original question, and an executable Python script that renders a visualization of the findings. This multi-modal output is designed to cover the full analytical workflow of a business analyst, from raw data retrieval to presentation-ready charts.

3.2. System Architecture

The Sales Agent is implemented as a Python package organised into six modules, each with a clearly delimited responsibility. Figure 2 shows the class diagram of the system.

SalesDataAgent is the main entry point of the system. It builds the LangGraph execution graph, wires the processing nodes together, and exposes a single `run()` method that accepts a natural language prompt and returns the final result dictionary. It holds references to the **AgentConfig**, the **DatabaseSchema**, and the **RunCache**, and coordinates the end-to-end execution including tracing and energy monitoring.

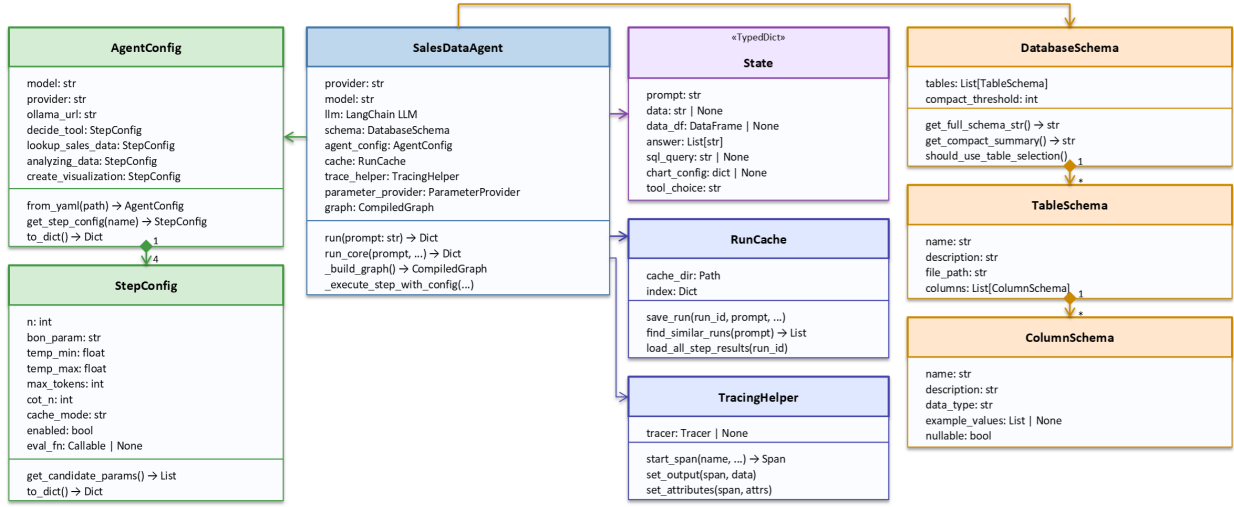


Figure 2: Class diagram of the Sales Agent.

AgentConfig and **StepConfig** form the configuration layer. **AgentConfig** holds global settings — the LLM provider (ollama or openai), the model name, and the Ollama server URL — together with one **StepConfig** instance per processing step. **StepConfig** encapsulates all per-step hyperparameters used by the experiments: number of best-of- n candidates, temperature range, top- p , top- k , maximum tokens, caching policy, and optional evaluation and selection callables. Both classes support serialisation to and from dictionaries and YAML files, enabling fully reproducible experiment configurations.

State is a **TypedDict** that acts as the shared memory flowing through the LangGraph nodes. It carries the user prompt, the retrieved data (as both a CSV string and a Pandas **DataFrame**), the generated SQL query, the chart configuration, the tool choice, per-step timing and evaluation scores, and the ground-truth scores accumulated during benchmarking.

DatabaseSchema, **TableSchema**, and **ColumnSchema** define the multi-table data model exposed to the LLM. **DatabaseSchema** wraps a list of **TableSchema** objects and provides two rendering modes: a compact summary (table names and descriptions only) used when the number of tables exceeds a configurable threshold, and a full schema string including all column names, types, descriptions, and example values used during SQL generation.

RunCache manages persistent caching of agent runs on disk. Each completed run is stored under a unique **run_id** directory containing the per-step results for all n best-of- n candidates. A JSON index file enables fast lookup of lexically similar past runs by prompt word overlap (Jaccard similarity over the prompt token sets), allowing the agent to reuse previously computed results and avoid redundant LLM calls.

TracingHelper provides optional observability integration via OpenTelemetry [10], with compatibility for the Arize Phoenix platform [11]. The integration is implemented in the codebase: when enabled, it records structured spans for each processing step, capturing inputs, outputs, latencies, and evaluation scores for post-hoc analysis. However, Phoenix was not used during the experiments reported in this thesis; the experimental analysis relies on the benchmark CSV outputs and CodeCarbon measurements.

The system depends on the following key external libraries:

- **LangGraph** / **LangChain** — graph-based LLM pipeline orchestration and provider-agnostic LLM interfaces (OpenAI, Ollama);
- **DuckDB** — in-process analytical SQL engine used to query Parquet files;
- **Pandas** / **PyArrow** — tabular data manipulation and Parquet I/O;
- **Matplotlib** — chart rendering for the visualization step;
- **CodeCarbon** — energy consumption tracking during agent execution;
- **OpenTelemetry** / **Arize Phoenix** — optional tracing and observability integration; implemented in the framework but not used for the reported benchmark runs;
- **NLTK** — natural language utilities available for optional n-gram-based text scoring (e.g. BLEU), not used by the LLM-as-judge metrics adopted in the experiments;

- **PyYAML** — parsing of experiment configuration files;

3.3. LangGraph Workflow

The execution logic of the Sales Agent is modelled as a directed cyclic graph built with LangGraph’s **StateGraph** API. The graph has four processing nodes and a single terminal node (**END**), as illustrated in Figure 3.

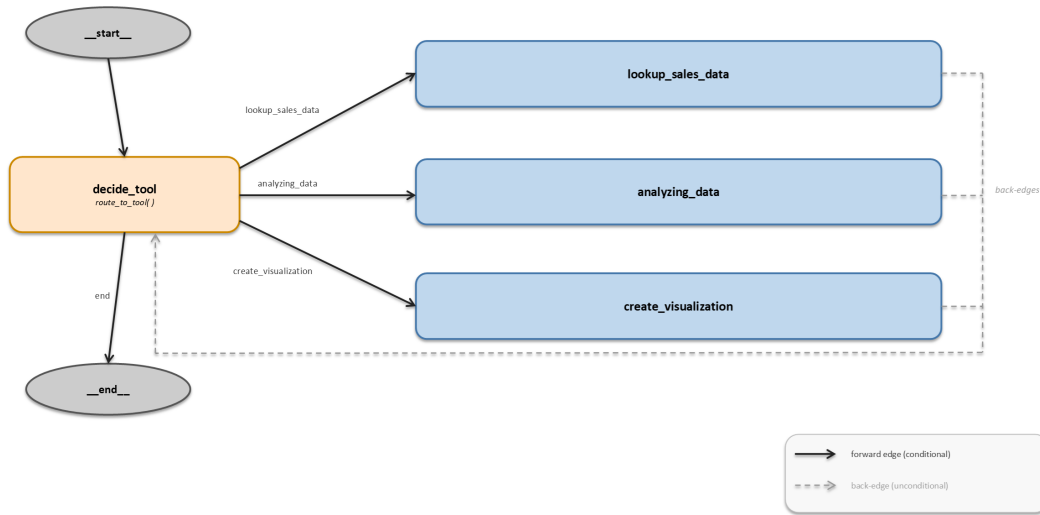


Figure 3: LangGraph execution graph of the Sales Agent.

The graph is defined as follows:

- **Entry point:** `decide_tool` — every execution starts here.
- **Conditional edges from `decide_tool`:** the routing function `route_to_tool` reads the `tool_choice` field set by the node and dispatches to one of `lookup_sales_data`, `analyzing_data`, `create_visualization`, or `END`.
- **Back-edges:** after each processing node completes, an unconditional edge returns control to `decide_tool`, forming a loop that continues until the LLM decides to terminate.

The `decide_tool` node. At each iteration, this node invokes the LLM with a structured prompt describing the four available tools, the current state of the conversation (user question, answers generated so far, last tool used), and a Chain-of-Thought reasoning template. The LLM reasons through five steps — analysing the user request, checking progress, identifying what is missing, verifying decision rules, and making the final choice — and outputs a single tool name. The routing is governed by three hard rules: (1) `lookup_sales_data` must run before any analysis or visualization; (2) no tool may be selected more than once per execution; (3) `end` is selected when both the textual analysis and the visualization answers have been generated, or when all tools relevant to the user’s request have been executed. Fuzzy string matching is applied to the raw LLM output to tolerate minor formatting variations.

The `lookup_sales_data` node. This node generates a SQL query from the user prompt and the database schema, executes it against the Parquet files via DuckDB, and stores the result in the **State** as both a CSV string and a Pandas **DataFrame**.

The `analyzing_data` node. Given the retrieved data and the original question, this node prompts the LLM to produce a natural language analysis. The generated text is appended to the **answer** list in the **State**.

The `create_visualization` node. This node first asks the LLM to produce a structured chart configuration (chart type, axes, title, colour scheme), then generates executable Python code using Matplotlib that renders the chart from the retrieved data. Both the configuration and the code are stored in the **State** and appended to the **answer** list.

Figure 4 shows the sequence diagram for a full execution of the Sales Agent on a question that requires data retrieval, analysis, and visualization. The notation follows a deliberately simplified variant of UML 2 sequence

diagrams: standard lifelines and message arrows are retained, combined fragments (`loop` and `opt`) mark the iteration boundary and the optional table-selection step, and dashed arrows denote return values. Activation bars, actor stereotypes (e.g. stick figures or boundary symbols), and explicit *create/destroy* messages have been omitted. Given the density of the diagram — nine lifelines and approximately 35 messages across four iterations — these elements would add visual noise without improving comprehension; the colour coding of the participant boxes already conveys their roles at a glance. This choice prioritises readability over strict UML compliance, which is standard practice in academic and industrial documentation when the primary goal is to communicate flow rather than to specify a system for code generation.

The system also supports two restricted operating modes configured via the `agent_mode` parameter in the YAML file: `lookup_only` skips both analysis and visualization, returning only the raw query result; `analysis` skips visualization and returns data plus text analysis. These modes reduce latency and LLM cost when the full output is not required.

3.4. Agent Tools and Steps

Each of the three processing nodes of the Sales Agent encapsulates a distinct analytical capability. This section describes their internal logic in detail.

3.4.1 Lookup Tool

The `lookup_sales_data` step is responsible for translating the user’s natural language question into a SQL query and executing it against the Parquet dataset via DuckDB.

The step opens a fresh in-process DuckDB connection for each invocation and registers all Parquet files as named SQL tables. It then builds the schema context — either the full column-level description or, when the number of tables exceeds `compact_threshold`, the output of the two-step table selection procedure described in Section 3.7. The LLM receives the schema context together with the user prompt and a structured SQL-generation prompt containing DuckDB-specific instructions, common mistakes to avoid, and few-shot examples. It then produces a SQL query. The query is executed via `duckdb.execute()`, and the result is stored in the `State` as both a CSV string (`data`) and a Pandas `DataFrame` (`data_df`).

Before passing the retrieved data to the analysis step, the system enriches it with pre-computed aggregate statistics: for every numeric column, the sum, minimum, maximum, and mean are appended to the CSV string as a dedicated summary block. This prevents the LLM from having to perform arithmetic over potentially hundreds of rows, improving both accuracy and consistency of the generated analysis.

3.4.2 Analysis Tool

The `analyzing_data` step takes the enriched CSV data and the original user prompt and asks the LLM to produce a concise natural language interpretation of the results. The prompt instructs the model to answer using only the values present in the data, to cite specific numbers, and to avoid speculation. A Chain-of-Thought template guides the model through understanding the question, examining the data, identifying key insights, and formulating a clear answer. The generated text is appended to the `answer` list in the `State`.

3.4.3 Visualization Tool

The `create_visualization` step is a two-phase process. In the first phase, the LLM is asked to extract a structured chart configuration from the data and the user’s question: it outputs a JSON-like dictionary specifying the chart type (bar, line, scatter, etc.), the columns to use for the x and y axes, the chart title, and any grouping or colour dimensions. In the second phase, a second LLM call receives the chart configuration and the `DataFrame` and generates executable Python code using Matplotlib that renders the requested chart.

Once generated, the code is immediately validated by executing it in an isolated namespace using Python’s built-in `exec()` function. The Matplotlib backend is switched to `Agg` (non-interactive, headless) prior to execution to avoid graphical environment dependencies. If the code raises an exception, the error message is stored in the `State` and can be fed back to the LLM by the CoT refinement mechanism (described in Section 3.5) for iterative correction. The validated code and the chart configuration are both appended to the `answer` list.

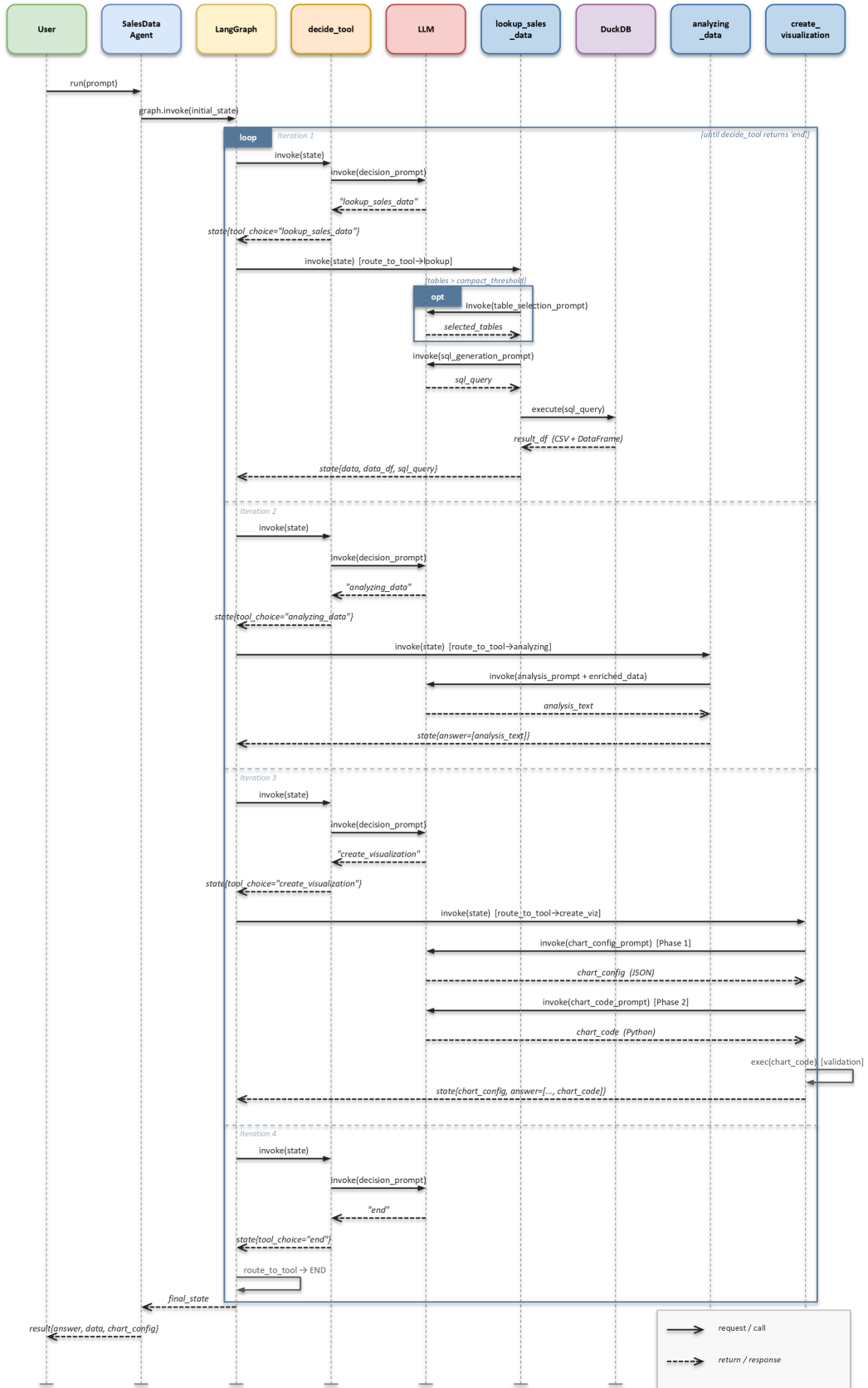


Figure 4: Sequence diagram of a full Sales Agent execution.

3.5. LLM Integration and Prompting Techniques

3.5.1 Supported LLM Backends

The Sales Agent is designed to be provider-agnostic. Two backends are supported and can be selected via the `provider` field in `AgentConfig`:

- **OpenAI** — cloud-based models accessed through the OpenAI API (e.g. `gpt-4o-mini`, `gpt-4o`). Authentication is handled via an API key passed through the configuration file or environment variable.
- **Ollama** — locally-hosted open-weight models served via the Ollama runtime (e.g. `llama3.2:3b`, `qwen2.5:7b`). The server URL is configurable, enabling deployment on remote GPU machines.

Both backends are accessed through the LangChain abstraction layer (`langchain-openai` and `langchain-ollama`), so the step functions are completely agnostic to the underlying provider. Per-step provider and model overrides are also supported: it is possible, for example, to use a large cloud model for SQL generation and a smaller local model for routing.

3.5.2 Prompt Structure

All prompts in the Sales Agent share a common skeleton. Each prompt opens with a role declaration that assigns a persona to the model (e.g. *“You are an expert SQL developer specialising in DuckDB queries”*), followed by a Markdown-sectioned body that separates concerns into labeled blocks: `## TASK` (what to produce), `## AVAILABLE DATA / ## USER QUESTION` (dynamic context injected via template variables at runtime), `## INSTRUCTIONS` (generation rules), and `## OUTPUT FORMAT` (exact specification of what the model should return). Keeping the static scaffold and the dynamic content clearly separated simplifies both prompt maintenance and output parsing.

3.5.3 Chain-of-Thought Prompting

Three of the four agent prompts — tool routing, data analysis, and chart configuration — augment the shared structure above with a `## CHAIN OF THOUGHT REASONING` section [12]. This section defines a sequence of named steps (e.g. *Step 1: Understanding the Question*, *Step 2: Examining the Data Structure*, ...) that the model must work through and expose in its output before producing the final answer. By externalising intermediate reasoning, the model is less likely to skip steps or make implicit assumptions, and the output is easier to inspect and validate.

The SQL generation prompt deliberately omits the CoT reasoning section. During development, smaller open-weight models tended to lose track of critical constraints — DuckDB-specific syntax rules, column qualification in JOINS, and the required plain-text output format — when required to produce explicit step-by-step reasoning before the query. The additional text generation appeared to exhaust the model’s effective attention on the earlier instruction blocks. The SQL prompt therefore replaces the CoT section with a `## COMMON MISTAKES TO AVOID` block listing wrong/right DuckDB patterns, and a `## EXAMPLES` block of few-shot query pairs, while the `## OUTPUT FORMAT` section explicitly instructs the model to return only the plain SQL with no explanations or markdown. This change consistently reduced syntax errors and instruction-following failures across the small models tested.

3.5.4 Best-of- N Sampling

For steps where output quality is critical — SQL generation and data analysis in particular — the agent supports best-of- N sampling. When `StepConfig.n` > 1, the step is executed N times with different values of the sampling parameter specified by `bon_param` (temperature, top- p , or top- k), linearly interpolated between `temp_min` and `temp_max` (or the corresponding bounds for top- p and top- k). Each of the N candidate outputs is independently scored by the `eval_fn` assigned to that step, and the candidate with the highest score is selected via `selection_fn` (defaulting to `argmax`). All N results are saved to the `RunCache` to allow re-selection with different evaluation criteria without re-invoking the LLM.

3.5.5 Iterative CoT Refinement

For steps with `StepConfig.cot_n` > 1, the agent applies iterative Chain-of-Thought refinement. After the initial LLM call, the output is fed back into the same prompt through a `CoTRefinementLLM` wrapper. This wrapper appends a refinement block to every subsequent call, showing the model its previous response and asking it to either reproduce it exactly when it is correct and complete — leaving the parts that are already correct unchanged and revising only the errors — or produce an improved version otherwise. The loop runs for at most `cot_n` total iterations, but stops early as soon as the output converges — i.e. when the textual similarity between

two consecutive responses reaches or exceeds 0.95, measured as the character-level `difflib.SequenceMatcher` ratio (defined as $2M/T$, where T is the total number of characters in the two responses and M is the number matched by the longest contiguous matching subsequences) — avoiding redundant refinement calls once the model has settled on a stable answer.

Two refinement modes are available. In the standard mode, the model receives its previous output and can voluntarily refine it. In error-aware mode, triggered whenever a step reports an execution error — either a SQL query that fails to execute in the lookup step or visualization code that raises an exception during validation — the error message is explicitly included in the refinement block with a directive to fix it. This self-correction loop allows the agent to recover from syntactically or semantically incorrect code without user intervention.

3.6. Configuration System

The Sales Agent exposes a hierarchical configuration system that controls every aspect of the agent’s behaviour without requiring code changes. All behavioural settings are defined in a single YAML file, loaded at startup via `AgentConfig.from_yaml()`, which additionally discovers the per-table `*_schema.yaml` files referenced by the `schema` section.

3.6.1 YAML Structure

The configuration file is divided into six top-level sections:

- **agent** — global LLM settings: provider (`openai` or `ollama`), model name, Ollama server URL, and OpenAI API key;
- **steps** — per-step hyperparameters, one sub-section per node (`decide_tool`, `lookup_sales_data`, `analyzing_data`, `create_visualization`);
- **schema** — path to the data directory containing the `*_schema.yaml` files and the `compact_threshold` value;
- **run** — execution parameters: prompt, visualization goal, operating mode, run identifier, output directory, caching options, and CodeCarbon flag;
- **ground_truth** — optional ground-truth references for tracking evaluation scores during benchmarking (described in Section 3.8);
- **tracing** — optional OpenTelemetry/Phoenix endpoint for observability.

3.6.2 Per-Step Parameters

Each step exposes the parameters listed in Table 1. A step can also override the global LLM provider and model, enabling heterogeneous deployments where, for example, SQL generation uses a larger cloud model while routing uses a lightweight local model. The default values shown reflect the baseline configurations used in the experiments (Section 4): Gemma 4 models are run at temperature 1.0, which is the recommended setting for thinking models; Mistral Small 3.2 is run at temperature 0.15, appropriate for an instruction-following model.

Parameter	Default (Gemma 4)	Default (Mistral)	Description
<code>n</code>	1	1	Number of best-of- n candidates
<code>bon_param</code>	<code>temperature</code>	<code>temperature</code>	Parameter varied across candidates
<code>temp_min</code> / <code>temp_max</code>	1.0 / 1.0	0.15 / 0.15	Temperature range for sampling
<code>top_p_min</code> / <code>top_p_max</code>	0.95 / 0.95	0.95 / 0.95	Top- p nucleus sampling range
<code>top_k_min</code> / <code>top_k_max</code>	64 / 64	64 / 64	Top- k sampling range (Ollama only)
<code>repeat_penalty</code>	1.1	1.1	Repetition penalty applied to already-seen tokens; > 1 discourages repetition (Ollama only)
<code>repeat_last_n</code>	64	64	Lookback window (tokens) over which the repetition penalty is applied; 0 disables it (Ollama only)
<code>max_tokens</code>	5000 / 7000	5000 / 7000	Maximum tokens per LLM generation (5000 for lookup, 7000 for analysis and visualization)
<code>cot_n</code>	1	1	Maximum CoT refinement iterations

Table 1: Configurable parameters of `StepConfig`.

3.6.3 Runtime Overrides

The `run.step_overrides` field allows temporary parameter changes for a single execution without modifying the base configuration: before the run starts, the agent deep-copies its configuration and applies the overrides on the copy. For example, setting `analyzing_data.n: 3` together with `temp_max: 1.3` enables best-of-3 sampling for the analysis step — widening its temperature range — in that run only.

A separate, complementary mechanism is the pluggable `ParameterProvider` abstraction, queried before each step to return the effective `StepConfig`. The default `DefaultProvider` returns the configuration unchanged and is the provider in effect for all experiments reported in this thesis, which run non-interactively. The codebase also ships an optional `TerminalProvider` which, when the agent is launched from an interactive terminal (i.e. when standard input is a TTY), displays each step’s parameters and lets the user accept or override them one by one before that step executes, falling back to the defaults otherwise. This interactive facility is intended for manual, exploratory use and is not exercised by the benchmark runs reported here.

3.7. Multi-Table Support and Dataset

3.7.1 Dataset Description

The Sales Agent operates on a synthetic retail dataset composed of three relational tables stored as Parquet files. The dataset was generated programmatically to simulate a realistic grocery retail scenario spanning multiple stores, product categories, and time periods. Table 2 summarises the three tables and their key columns.

Table	Key columns	Description
sales	Store_Number, SKU_Coded, Product_Class_Code, Sold_Date, Qty_Sold, Total_Sale_Value, On_Promo	Daily store-level transactions with pricing and promotion data
products	SKU_Coded, Product_Class_Code, category, subcategory, product_name, brand, unit_price_usd, weight_kg, is_organic, country_of_origin	Product catalog with category, brand, pricing, and origin details
stores	Store_Number, store_name, city, region, store_type, area_sqm, num_employees	Store metadata including location, type, and operational details

Table 2: Dataset tables and their primary columns.

The three tables are linked through two foreign keys: `Store_Number` connects `sales` to `stores`, and `SKU_Coded` connects `sales` to `products`. Both tables also carry the `Product_Class_Code` column — a category-level classification code shared by multiple SKUs — which can be used as an additional grouping dimension alongside the join. This relational structure allows the agent to answer questions that span multiple dimensions simultaneously — for example, aggregating revenue by product category and region, or filtering promotions by store type.

3.7.2 Schema Representation

Each table is described by a `TableSchema` dataclass that stores the table name, a human-readable description, the path to the Parquet file, and an ordered list of `ColumnSchema` objects. Each `ColumnSchema` carries the column name, a natural language description, the SQL data type, optional example values, and a nullability flag. These descriptions are the primary source of information that the LLM uses to generate accurate SQL queries — column names alone are often ambiguous, while descriptions such as “*Whether item was on promotion (1=yes, 0=no)*” or “*Total revenue from this transaction in USD*” provide the semantic context needed to map natural language questions to the correct columns.

Schema definitions are stored in versioned `*_schema.yaml` files in the data directory, one per table. The `AgentConfig.from_yaml()` loader discovers all schema files automatically by glob pattern, constructs the corresponding `TableSchema` objects, and assembles them into a `DatabaseSchema` instance.

3.7.3 Two-Step Table Selection

When the number of tables in the schema exceeds the `compact_threshold` (default: 5), the `lookup_sales_data` step switches to a two-step table selection strategy to keep the LLM context window manageable.

In the first step, the LLM receives only a compact summary of the schema — table names and one-line descriptions — together with the user’s question, and outputs a comma-separated list of table names that are relevant to answering it. The selection prompt also uses Chain-of-Thought reasoning, guiding the model to map the question’s entities to tables, identify required joins, and verify completeness.

In the second step, only the full column-level schema of the selected tables is included in the SQL generation prompt, reducing the total token count and avoiding distraction from irrelevant tables. This mechanism makes the agent scalable to databases with tens of tables without degrading SQL generation quality.

For the current three-table dataset the threshold is not reached and the full schema is always passed directly, but the mechanism is fully operational and activates automatically as the schema grows.

3.8. Benchmark Design

3.8.1 Benchmark Structure

To evaluate the Sales Agent systematically, a benchmark of 20 question-answer pairs was constructed manually. Each item in `benchmark_dataset.json` represents a realistic analytical question that a business analyst might pose. The full list of benchmark prompts, grouped by difficulty level, is provided in Appendix A. Each item is composed of the following fields:

- `prompt` — the natural language question given as input to the agent;
- `visualization_goal` — an optional directive specifying the desired chart type and axes, passed alongside the prompt;
- `gt_sql` — the reference SQL query that correctly answers the question on the current schema;
- `gt_data` — the expected query result as a CSV string, obtained by executing `gt_sql` on the dataset;
- `gt_analysis` — a reference natural language interpretation of the result, used to evaluate the quality of the agent’s textual output;
- `gt_chart_config` — the expected chart configuration dictionary (chart type, axes, title);
- `gt_chart_code` — reference Python/Matplotlib code that renders the correct visualization from `gt_data`;
- `explicit_requirements` — a structured dictionary of optional visual constraints (colour scheme, title format, label format, grid, markers) used to tighten the visualization evaluation;
- `difficulty` — an integer from 1 to 4 reflecting the SQL complexity required to answer the question correctly.

3.8.2 Difficulty Levels

The difficulty score is assigned according to a rubric defined in `gt_scoring_criteria.md` and reflects the complexity of the SQL reasoning required, not the wording of the prompt. The four levels are:

- **Level 1** — single-table queries with simple filters, one `GROUP BY`, and basic aggregation (e.g. top stores by revenue);
- **Level 2** — single-table queries with time bucketing via `DATE_TRUNC`, multiple aggregates, or top- N over time slices (e.g. monthly revenue and units sold for a full year);
- **Level 3** — queries requiring at least one join to `products` or `stores`, grouping on business dimensions external to `sales`, or one subquery (e.g. revenue by brand, region, or store type);
- **Level 4** — multi-step queries combining joins with nested aggregation, ratio or share metrics, `CASE WHEN` logic, or year-over-year comparisons (e.g. promo revenue share by category, region-category YoY growth).

Scores were assigned by inspecting the ground-truth SQL query rather than the natural-language wording of the prompt, following a set of prioritised rules: a single-table flat query is classified as level 1 or 2 depending on whether it involves time bucketing or multiple aggregates; any join to `products` or `stores` raises the level to at least 3; and the presence of a subquery, a derived table, a ratio or share metric, or a year-over-year comparison raises it to 4. Prompt length, the presence of a visualization request, and the number of result rows do not influence the score.

The 20 benchmark items are distributed across difficulty levels as shown in Table 3.

Difficulty	Items	Example prompt
1	4	Return the top 5 stores by total revenue
2	5	Show the 12 months of 2023 with total revenue and units sold
3	7	Show total revenue by product brand for 2023 as a bar chart
4	4	Compare average monthly revenue between store regions for 2022 and 2023
Total	20	

Table 3: Distribution of benchmark items by difficulty level.

3.8.3 Ground Truth Generation

Ground truth entries were produced through a semi-automated process: SQL queries were written manually and validated by executing them against the dataset; the resulting CSV was used directly as `gt_data`; reference analyses and chart code were generated with the assistance of a large language model and then reviewed and corrected by the authors. This human-in-the-loop approach ensures that the ground truth is both accurate and representative of the expected output quality.

The ground truth is used exclusively for performance tracking and is never exposed to the agent during execution. During benchmarking, `gt_eval_fn` callables are attached to each step and compute scores by comparing the agent’s output to the reference, but these scores do not influence the best-of- n selection, which relies solely on the non-GT evaluators described in Section 3.9.

3.9. Evaluation Metrics, Output, and Energy Monitoring

3.9.1 Evaluation Metrics

The Sales Agent is evaluated across three complementary dimensions, one per processing step.

Ground-truth table similarity (`csv_iou`). The quality of the data retrieval step is measured by comparing the agent’s result `DataFrame` with the ground-truth CSV. The output column is still named `csv_iou` for continuity with earlier experiments, but the final benchmark scorer is a graded ground-truth table-similarity metric rather than a brittle exact row-IoU score. This change is important because many correct SQL answers differ from the reference only in harmless surface choices: column aliases, column order, boolean labels, numeric formatting, or long-versus-wide layout.

Failed SQL executions, missing dataframes, and empty results receive score 0. Otherwise, both tables are first normalised with `normalize_dataframe_values`: float-encoded integers are cast to their integer representation, numeric values are rounded, dates are rendered as YYYY-MM-DD, and string cells are stripped of leading and trailing whitespace. Column names are lowercased before benchmark evaluation; during agent execution, result columns can also be aligned to the ground-truth column order when ground-truth columns are available.

The scorer then computes the best of two comparisons. The direct comparison aligns columns by a weighted combination of normalised column-name similarity and value similarity, so aliases such as `revenue` and `total_sales_value`, or swapped numeric columns with matching values, can still receive high credit. Rows are matched without assuming order, using a soft value similarity that gives full credit for equal values, month/month-start date equivalence, near-equal numeric values within tolerance, and common binary category equivalents such as `true/false`, `1/0`, `Organic/Non-Organic`, and `Promo/Non-Promo`. The final direct score combines table structure and soft row-value agreement. A second comparison detects common long-versus-wide binary grouped outputs, such as `month`, `promo_flag`, `revenue` versus `month`, `promo_revenue`, `non_promo_revenue`, and the final `csv_iou` value is the maximum of the direct and long-wide scores.

Text Score (`text_score`). The quality of the natural language analysis is evaluated using an LLM-as-judge approach [15]. A separate LLM call (temperature 0, `gpt-5.4`) receives the generated analysis alongside the ground-truth reference text and rates two criteria on a 1–5 scale:

- **Factual Accuracy** — do the key numerical values and conclusions in the generated text match those in the reference, regardless of phrasing or sentence structure?
- **Coverage** — does the generated text address all the main points and conclusions present in the reference?

The two raw scores are averaged and the result is linearly normalised from [1, 5] to [0, 1]:

$$\text{text_score} = \frac{\frac{s_{\text{accuracy}} + s_{\text{coverage}}}{2} - 1}{4} \quad (2)$$

The two criteria are combined with equal weight because Factual Accuracy and Coverage capture two independently necessary and non-substitutable properties of a good analysis—correctness of the reported values and completeness with respect to the reference—so neither is privileged a priori. The averaged rubric score is then rescaled from the judge’s native [1, 5] range to [0, 1] through an affine transformation that preserves the ordering and relative spacing of the scores while placing `text_score` on the same [0, 1] scale as `csv_iou` and `vis_score`. This makes the three step metrics directly comparable and lets them be averaged into a single `quality_mean` (the arithmetic mean of the three component score means, Equation 4) without any dimension dominating purely because of a wider numeric range. Using a strong LLM as a judge to rate generated outputs on a discrete scale is an established evaluation approach whose scores have been shown to align well with human judgement [15].

Visualization Score (`vis_score`). The quality of the generated chart is evaluated by a second LLM judge (gpt-5.4, temperature 0.2) that receives the agent’s chart configuration dictionary, the corresponding Matplotlib code, the ground-truth equivalents, and any explicit styling requirements attached to the benchmark item. The judge rates four criteria on a 1–5 scale:

- **Axis Correctness** — do the x and y axes reference the same data columns as the ground truth (case-insensitive)? Handles the three valid multi-series encodings (`y_axis`, `y_axes`, and `y_axis+group_by`) as equivalent when the same columns are ultimately visualized.
- **Chart Type Correctness** — does the chart type (`bar`, `line`, `scatter`, `area`) match the reference? Variations within a type (e.g. grouped vs. stacked bar) are acceptable.
- **Functional Equivalence** — would the generated Matplotlib code produce a visually equivalent chart? Code style, variable names, and import statements are ignored; only the rendered visual output matters.
- **Explicit Requirements Compliance** — applies only when the benchmark item specifies styling constraints (colour scheme, title format, label format, grid, markers); if all requirements are absent this criterion is marked N/A and assigned the maximum score.

Each raw score $s_i \in [1, 5]$ is normalised as $(s_i - 1)/4$ and the final score is a weighted sum:

$$\text{vis_score} = 0.25 \hat{s}_{\text{axis}} + 0.25 \hat{s}_{\text{type}} + w_f \hat{s}_{\text{func}} + w_e \hat{s}_{\text{expl}} \quad (3)$$

where $w_f = 0.40$, $w_e = 0.10$ when explicit requirements are present, and $w_f = 0.50$, $w_e = 0$ otherwise.

The weights always sum to one, keeping `vis_score` on a fixed $[0, 1]$ scale, and rank the criteria by how directly they affect whether the chart conveys the intended information. Axis Correctness and Chart Type Correctness receive an equal 0.25 each as the two fundamental structural decisions of a plot—*what* is plotted and *how*. Functional Equivalence carries the largest single weight because it is the most comprehensive criterion, capturing whether the fully rendered chart is visually equivalent to the reference rather than any isolated attribute. Explicit Requirements Compliance is deliberately capped at 0.10: styling constraints such as colour scheme or title format are cosmetic refinements, so this cap ensures that an otherwise correct chart cannot be heavily penalised for a purely stylistic deviation. When the benchmark item specifies no explicit requirements the criterion is not applicable, and rather than renormalising all four weights—which would also inflate the structural criteria—its 0.10 share is reassigned to Functional Equivalence, the most encompassing of the remaining criteria. This keeps the relative emphasis on substantive correctness stable and the resulting `vis_score` directly comparable across items with and without explicit styling constraints. Relying on an LLM judge to rate the generated chart against the reference follows the same established LLM-as-judge practice [15], whereas the specific criteria and their weights are a design choice tuned to the priorities of this benchmark.

3.9.2 No-GT Evaluators for Best-of- N Selection

In addition to the ground-truth evaluators used for tracking, the system provides three evaluators that operate without ground truth, used by the best-of- N selection mechanism during live execution.

For the lookup step, a *consensus-based* evaluator computes the pairwise IoU between all N candidate DataFrames and assigns each candidate a score equal to its average IoU against the others. The candidate with the highest average agreement — i.e. the one most consistent with the rest — is selected. This self-consistency approach [14] requires no reference answer and exploits the redundancy of multiple independent samples.

For the analysis and visualization steps, the no-GT evaluator uses an LLM judge that scores the output against the retrieved data and the original question, rather than against a reference text.

3.9.3 Energy Consumption Monitoring

As already mentioned, energy consumption is tracked using CodeCarbon [2], an open-source library that measures CPU, GPU, and RAM energy draw during code execution and converts it to an estimated CO₂ equivalent based on the carbon intensity of the local power grid.

Tracking is enabled by setting `enable_codecarbon: true` in the run configuration. When enabled, the system operates two independent trackers:

- **Step-level trackers** — a dedicated `EmissionsTracker` is started and stopped around the entire LLM activity of each processing step (`lookup_sales_data`, `analyzing_data`, `create_visualization`). Each tracker accumulates the energy of *all* LLM invocations within that step, including best-of- N candidates, CoT refinement iterations, and evaluation judge calls. The five CodeCarbon fields — total energy (kWh), CPU energy, GPU energy, RAM energy, and CO₂ emissions (kg) — are recorded independently per step, yielding 15 per-step energy columns in the output CSV, following the naming pattern `{step}_llm_{field}` where `{field}` is one of `energy_kwh`, `cpu_energy_kwh`, `gpu_energy_kwh`, `ram_energy_kwh`, `emissions_co2`.

Column	Description
test_case_id, prompt, difficulty	Item identifier, question text, difficulty level
gen_sql	SQL query generated by the agent
csv_iou, text_score, vis_score	GT-based scores for each step
csv_iou_reasoning, text_score_reasoning, vis_score_reasoning	Diagnostic message explaining the mismatch populated only when the corresponding GT score < 1.0
csv_eval_score, text_eval_score, vis_eval_score	No-GT scores used by the BoN selector
elapsed_sec	Total wall-clock time for the full run
lookup/analyzing/vis_time_sec	Per-step total wall-clock time
lookup/analyzing/vis_llm_time_sec	Time spent in LLM calls per step (sum of BoN candidates, CoT iterations, and judge calls)
{step}_llm_energy_kwh, {step}_llm_cpu gpu ram_energy_kwh, {step}_llm_emissions_co2	Per-step energy of the LLM calls — total, CPU, GPU and RAM energy (kWh) and CO ₂ emissions (kg) — for each of the 3 steps (5 fields × 3 steps, as described in the Energy Consumption Monitoring subsection above); None when CodeCarbon is disabled
energy_consumed_kwh, cpu/gpu/ram_energy_kwh, emissions_kg_co2	Run-level total energy and CO ₂ covering the full agent invocation; None when CodeCarbon is disabled

Table 4: Main columns of the output CSV.

- **Run-level tracker** — a second `EmissionsTracker` wraps the complete agent invocation, capturing the total footprint including orchestration, data I/O, and all steps. Its output is written to five run-level columns (`energy_consumed_kwh`, `cpu_energy_kwh`, `gpu_energy_kwh`, `ram_energy_kwh`, `emissions_kg_co2`). When `enable_codecarbon` is false, all 20 energy columns are set to `None`.

3.9.4 Output CSV

At the end of a benchmark run, all per-item results are aggregated into a single output CSV file. Each row corresponds to one benchmark item and contains the columns summarised in Table 4.

4. Experiment Design

This section defines the experimental protocol used to evaluate the Sales Agent. It first describes the common execution environment and then details five experiments, each isolating a distinct aspect of model behaviour or configuration:

- **Test 01 — Baseline Model Comparison** (Section 4.1): runs each candidate model with its Ollama-recommended default configuration on the full 20-prompt benchmark to establish a quality and efficiency reference.
- **Test 02 — Per-Agent Sensitivity Analysis** (Section 4.2): sweeps five static sampling hyperparameters one at a time (One-Factor-At-a-Time, OFAT) across each agent step to quantify their individual effect on pipeline quality.

- **Test 03 — max_tokens Ladder per Agent** (Section 4.3): varies the output token budget for each agent step across four discrete levels to identify the minimum safe cap and any quality sensitivity to the budget.
- **Test 04 — Compute Expansion (CoT Depth Ladder)** (Section 4.4): applies Chain-of-Thought refinement at depths 1–4 on the lookup and visualization steps; a targeted follow-up (Test 04b) checks whether Best-of-N temperature sampling provides genuine lookup benefit through candidate selection.
- **Test 05 — Final Pareto Confirmation** (Section 4.5): evaluates the most promising combined configurations from Tests 02–04 on the full benchmark to establish the quality-versus-energy Pareto frontier among the evaluated candidate configurations.

The campaign was carried out across two hardware environments. Tests 01, 02, and 05 were conducted on a dedicated remote Linux server used exclusively for inference workloads; its hardware and software specifications are reported in Table 5. Tests 03, 04, and 04b were instead executed on a separate Linux server whose specifications are reported in Table 6; this split was adopted to run the two groups of experiments in parallel and shorten the overall data-collection time. In both environments the GPU is the primary compute resource: all LLM inference is offloaded to it via Ollama. The difference in hardware does not affect the validity of the conclusions drawn from Tests 03, 04, and 04b, because those tests measure only *relative* effects: each sweep varies a single parameter across a fixed set of configurations that are all executed on the same machine, under identical software and model conditions. Absolute latency and energy figures are therefore not directly comparable across the two servers, but the rankings, sensitivity trends, and relative improvements that those tests are designed to capture remain unaffected by the hardware on which they are run.

Component	Specification
Server model	Dell PowerEdge R760
GPU model	NVIDIA L40S
GPU VRAM	45 GiB
CPU model	Intel Xeon Gold 6426Y ($\times 2$)
CPU cores / threads	32 / 64
System RAM	503 GiB
Operating system	Ubuntu 24.04.1 LTS
Linux kernel	6.8.0-117-generic
GPU driver	580.159.03

Table 5: Remote server hardware and software specifications (Tests 01, 02, 05).

Component	Specification
Machine model	Supermicro SYS-120GQ-TNRT
GPU model	NVIDIA A40 ($\times 2$)
GPU VRAM	45 GiB ($\times 2$)
CPU model	Intel Xeon Gold 5317 @ 3.00 GHz ($\times 2$)
CPU cores / threads	24 / 48
System RAM	1.0 TiB
Operating system	Rocky Linux 8.10 (Green Obsidian)
Linux kernel	4.18.0-553.8.1.el8_10.x86_64
GPU driver	555.42.02

Table 6: Secondary server hardware and software specifications (Tests 03, 04, 04b).

In addition to the hardware profile, the main software and model provenance metadata needed to reproduce the experiments are summarised in Table 7. The Python and CodeCarbon entries refer to the benchmark

execution environment used by the project. The Ollama version, CUDA runtime compatibility, and model digest/quantization entries are host-side properties: they must be recorded on the machine that serves the models, because they are determined by the installed Ollama binary and by the model artefacts present in the local Ollama cache.

Item	Value or provenance command
Python runtime	Python 3.12.13, executed from the Docker image based on <code>python:3.12-slim</code> ; verified with <code>python -version</code> .
CodeCarbon	CodeCarbon 3.2.1; verified with <code>python -c "import codecarbon; print(codecarbon.__version__)"</code> .
Ollama server	Host-side Ollama service; the exact version is recorded with <code>ollama -version</code> on the inference server.
Model digest and quantization	Recorded per evaluated model with <code>ollama show <model> -verbose</code> , where <code><model></code> is one of <code>gemma4:e4b</code> , <code>gemma4:26b</code> , or <code>mistral-small13.2:24b</code> . This command reports the local model digest and quantization used by Ollama.
CUDA runtime / driver compatibility	The DataAgent Docker image does not bundle a CUDA toolkit; GPU inference is performed by the host Ollama service through the NVIDIA driver. The driver and CUDA compatibility are recorded with <code>nvidia-smi</code> on the inference server.

Table 7: Software and model provenance metadata required to reproduce the experiments.

Ollama runs as a persistent background service on the host machine, exposing a local REST endpoint at `localhost:11434`. All three models evaluated in the thesis — `gemma4:e4b`, `gemma4:26b`, and `mistral-small13.2:24b` — were downloaded from the Ollama model library and served locally, with no external network access during inference. An OpenAI API key is required only for the ground-truth judge, which calls `gpt-5.4` to score both the natural-language analysis step and the visualization step.

All benchmark runs are launched from deterministic YAML manifests and write their outputs to a persistent `runs/` directory. For clean energy attribution, model/configuration/repeat combinations are executed in series: only one benchmark process runs on the measured GPU at a time. This means that each CodeCarbon measurement corresponds to a single atomic agent run rather than to a concurrent workload sharing the same accelerator. The orchestration scripts are used only to automate this ordered sequence and to resume incomplete runs; they do not parallelise configurations whose energy values are later compared.

Before executing the tests in the definitive form described below, a lighter version of the same experiments was first run — with fewer repetitions and a smaller number of sampled parameter configurations — to explore preliminary results and guide the design of the full test suite. For clarity and conciseness, these preliminary runs are not reported here, as they constitute a reduced variant of the tests described in the following subsections.

4.1. Test 01 - Baseline Model Comparison

Test 01 establishes the quality and energy baseline for the three candidate models evaluated in this thesis: `gemma4:e4b`, `gemma4:26b`, and `mistral-small13.2:24b`. Each model runs with a fixed default configuration so that any observed differences in output quality, latency, and energy consumption are attributable solely to the model itself rather than to configuration choices. Rather than choosing these defaults arbitrarily, the baseline parameters for each model are taken directly from the recommended settings published on the respective Ollama model pages. The resulting per-model configurations are summarised in Table 8.

Parameter	gemma4:e4b	gemma4:26b	mistral-small3.2:24b
Temperature	1.0	1.0	0.15
top- p	0.95	0.95	0.95
top- k	64	64	64
repeat_penalty	1.1	1.1	1.1
repeat_last_n	64	64	64
BoN candidates (n)	1	1	1
CoT iterations (cot_n)	1	1	1
max_tokens — lookup	5000	5000	5000
max_tokens — analysis	7000	7000	7000
max_tokens — visualization	7000	7000	7000

Table 8: Baseline hyperparameter configuration for Test 01.

The evaluation is performed on the full 20-prompt benchmark, stratified across four SQL difficulty levels. For each prompt, three complementary quality scores are recorded: the graded ground-truth table-similarity score (`csv_iou`) for data retrieval accuracy, the LLM-as-judge score (`text_score`) for the natural-language analysis, and the LLM-as-judge score (`vis_score`) for visualization correctness when a chart is expected. In addition, per-step execution time and energy consumption are tracked via CodeCarbon. Each model configuration is repeated three times to allow variance estimation.

The results of this test inform the selection of the reference baseline for all subsequent tests and provide the first evidence of whether model size, the presence of an internal chain-of-thought reasoning mechanism, and the mixture-of-experts architecture lead to meaningful accuracy or efficiency differences on this agentic workload.

4.2. Test 02 - Per-Agent Sensitivity Analysis (OFAT)

Test 02 adopts a One-Factor-At-a-Time (OFAT) protocol to quantify how individual static sampling hyperparameters affect each agent step in isolation. In each phase, only one agent step’s parameters are varied while the other two steps are kept at the model’s baseline configuration. The three phases target, in order, the `lookup_sales_data` step (primary metric: `csv_iou`), the `analyzing_data` step (primary metric: `text_score`), and the `create_visualization` step (primary metric: `vis_score`). In addition to the step-specific metric, the full-pipeline `quality_mean` is also tracked so that improvements or regressions in the varied step can be related to the overall agent output.

The experiment varies five static sampling parameters, one at a time. For each parameter the ten off-baseline values form a fixed, hand-designed grid rather than a uniform partition of the interval: the continuous parameters are sampled more densely near the model-specific baseline, so that sensitivity around the operating point is resolved more finely than in the tails, while the two integer parameters use a grid with increasing gaps to span a wide range with few points. The exact grids are:

- **temperature** — model-specific and denser near the baseline: Gemma 4 {0.50, 0.60, 0.70, 0.80, 0.90, 0.95, 1.05, 1.10, 1.20, 1.30} (baseline 1.0); Mistral Small {0.03, 0.05, 0.075, 0.10, 0.125, 0.175, 0.20, 0.25, 0.35, 0.45} (baseline 0.15);
- **top- p** — {0.70, 0.75, 0.80, 0.85, 0.88, 0.90, 0.92, 0.94, 0.98, 1.00}, same for all models (baseline 0.95);
- **top- k** — {16, 20, 24, 32, 40, 48, 56, 80, 96, 128}, same for all models (baseline 64);
- **repeat_penalty** — {0.90, 0.95, 1.00, 1.03, 1.06, 1.08, 1.12, 1.15, 1.20, 1.30}, same for all models (baseline 1.1);
- **repeat_last_n** — {16, 24, 32, 40, 48, 56, 80, 96, 128, 256}, same for all models (baseline 64).

No Best-of- N or Chain-of-Thought strategies are included in this test: $n = 1$ and $\text{cot_n} = 1$ are fixed throughout, so all configurations differ only in their static sampling settings. The grid is fully deterministic: for each step, one baseline configuration plus $5 \times 10 = 50$ off-baseline configurations are generated, yielding **51 configurations per step** and $51 \times 3 = 153$ total configurations per model. The experiment is run on a 15-prompt benchmark subset to reduce wall-clock time while retaining enough prompt diversity for reliable per-step sensitivity estimates, for two independent repeats per model, yielding $153 \times 15 = 2,295$ scored prompt runs per model per repeat.

The main goal of this test is to identify which static parameters and which agent steps offer the most leverage for quality improvement at low computational cost, before considering more expensive compute-expansion strategies

in Test 04.

4.3. Test 03 - max_tokens Ladder per Agent

Test 03 isolates the effect of the token budget on each agent step by sweeping four `max_tokens` values for one step at a time, keeping the other two steps at generous caps that are unlikely to be binding. The token levels tested for each step are:

- `lookup_sales_data`: 2500, 3000, 5000 (baseline), 8000
- `analyzing_data`: 4000, 5000, 7000 (baseline), 10000
- `create_visualization`: 4000, 5000, 7000 (baseline), 10000

The rationale is that `max_tokens` is an upper bound on the number of tokens the model can generate in a single call, not a fixed generation length. A too-low cap can cause structural failures — truncated SQL queries, incomplete analysis text, or missing chart code — while a cap that is already non-binding should add little measurable cost. The experiment therefore aims to identify the minimum safe cap for each step and model, to verify whether higher caps provide any quality benefit beyond that threshold, and to check empirically whether larger caps introduce a significant energy increase.

The test runs on a 10-prompt benchmark subset across all three models, for three repeats each, yielding $12 \times 10 = 120$ scored prompt runs per model per repeat. The subset is reduced from the 15 prompts used in Test 02 to limit total wall-clock time, while preserving coverage of all four SQL difficulty levels (D1: 2, D2: 3, D3: 3, D4: 2 prompts). Both step-specific metrics and the full-pipeline `quality_mean` are reported alongside per-step energy measurements, so that any accuracy recovery from a higher cap can be weighed against its compute cost.

4.4. Test 04 - Compute Expansion (CoT Depth Ladder)

Test 04 investigates whether Chain-of-Thought (CoT) refinement can improve accuracy over the static baselines established in Tests 02–03. `gemma4:26b` is excluded: its main weakness (visualization) was already resolved by the token-budget increase in Test 03, its SQL and text steps were near saturation, and its high baseline latency made compute expansion particularly costly.

Both `gemma4:e4b` and `mistral-small13.2:24b` are evaluated with the same protocol: a CoT depth ladder varying `cot_n` $\in \{1, 2, 3, 4\}$ on the lookup and visualization steps independently, while the analysis step is kept fixed at baseline. This yields 8 configurations per model ($4 \text{ lookup} \times 2 \text{ steps}$), run on the same 10-prompt subset as Test 03 for three repeats, for the same reason of limiting total run time while retaining full difficulty coverage.

An additional follow-up campaign (Test 04b) is run on `gemma4:e4b` to disentangle whether lookup gains come from CoT refinement, from a better static temperature, or from genuine Best-of-N candidate selection. It compares three static temperature settings ($t \in \{0.3, 0.6, 0.8\}$) against three Best-of-3 temperature ranges — narrow $[0.8, 1.0]$, medium $[0.6, 1.0]$, and wide $[0.3, 1.0]$ — plus a CoT $n = 2$ reference, across 8 lookup-only configurations and three repeats. The use of $n = 3$ is deliberate: with only two candidates the no-GT selector has too little ranking signal to test whether Best-of-N is actually choosing among alternatives.

4.5. Test 05 - Final Pareto Confirmation

Test 05 is the final evaluation stage. Its purpose is to confirm whether the best configurations identified in Tests 02–04 maintain their advantage on the full 20-prompt benchmark, and to establish the quality-versus-energy Pareto frontier among the evaluated model and candidate-policy combinations.

Unlike the preceding tests, which vary one hyperparameter dimension at a time, Test 05 evaluates combined agent policies: each configuration specifies the full set of per-step parameters assembled from the most promising candidates of the exploratory phase. A substantial part of the test consists of newly run combined policies designed specifically for this final confirmation stage, rather than only reusing rows from earlier experiments. Roughly 10–15 configurations are evaluated per model, each combining one or more of the interventions from Tests 02–04: lookup Chain-of-Thought (varying `cot_n` from 2 to 4), static lookup temperature tuning (`temp` 0.8), adjusted repetition controls, and step-specific token budgets.

The configuration families are model-specific:

- `gemma4:e4b` — combinations of lookup CoT (`cot_n` 2 or 4), low-temperature lookup (`temp` 0.8), and reduced token caps across downstream steps;
- `gemma4:26b` — visualization-focused repair candidates that keep the lookup and analysis steps fixed and generous while varying only the visualization step: larger visualization token budgets (10000–16000), top- k sampling (`top_k` 20), repetition-penalty variants (`repeat_penalty` 1.0 and 1.03), and a lower visualization temperature (`temp` 0.95), plus one most-aggressive candidate that additionally adds static

upstream settings;

- **mistral-small13.2:24b** — lookup CoT (`cot_n 2` or `3`) with efficient token budgets, and combined static parameter sets with varied token caps.

The final candidate set combines the Test 01 baselines, the newly measured combined Test 05 policies, and useful configurations originally explored in Tests 02–04. The reuse is intentional but selective: rather than re-running every configuration, results that remain directly comparable are reused as they are, while a subset of earlier configurations is re-run for this final stage. These re-runs do not change the configuration parameters; their sole purpose is to measure the candidates in the same execution environment (the same server) as the newly run combined policies, so that every point on the final Pareto frontier is compared under identical execution conditions rather than across measurements taken on different machines or runs.

The primary metrics are **quality_mean**, prompt-level end-to-end quality (which penalises missing downstream scores caused by upstream failures), completion rate, elapsed time, and energy per prompt.

Energy as the efficiency axis. Although execution time and energy are both recorded, they are treated as two readings of the same underlying cost rather than as independent axes. On fixed inference hardware the accelerator draws a near-constant average power during generation, so per-prompt energy is, to first order, a fixed multiple of wall-clock latency; the baseline campaign confirms this, with the per-model latency and energy rankings coinciding to within about five percentage points. The quality–cost trade-offs and candidate-set Pareto frontiers in Section 5 therefore use energy per prompt as the single representative efficiency axis, while absolute latency is still reported in the per-test tables for deployments bound by response-time limits. Energy is preferred as the headline axis because it simultaneously captures the sustainability dimension and, unlike wall-clock time, aggregates the CPU, GPU, and RAM contributions measured by CodeCarbon.

5. Experimental Results

This section presents the empirical results of the five experiments introduced in Chapter 4. Section 5.1 compares the baseline models, and Section 5.2 analyses per-agent parameter sensitivity. Section 5.3 examines the effect of the output token budget, while Section 5.4 studies inference-time compute expansion through Chain-of-Thought refinement and also probes the effectiveness of Best-of- N sampling on the lookup step. Finally, Section 5.5 reports the final Pareto confirmation, including model-level summaries, quality-energy frontiers, and prompt-difficulty analyses over the evaluated candidate configurations.

5.1. Test 01 - Baseline Model Comparison

Test 01 establishes the quality and efficiency baseline for the three candidate models before any parameter tuning is applied. The experimental design is described in Section 4.1: each model runs with its Ollama-recommended sampling defaults over all 20 benchmark prompts, with no agent step varied. Three repeats per model were collected, yielding 3 models $\times$ 3 repeats $\times$ 20 prompts = 180 benchmark rows.

5.1.1 Quality Metrics and End-to-End Interpretation

The benchmark runner reports a standard **quality_mean** defined as the arithmetic mean of the three component score means:

$$\text{quality_mean} = \frac{1}{3}(\overline{\text{csv_iou}} + \overline{\text{text_score}} + \overline{\text{vis_score}}) \quad (4)$$

where the averages are taken over the subset of prompts that produced a scorable output for each component. Each of the three component scores is by construction bounded to $[0, 1]$ (Section 3.9), so their arithmetic mean is itself in $[0, 1]$ and can be read directly as an overall quality fraction. This definition is consistent across all tests and is used for continuity, but it can mask pipeline failures: when the lookup step returns an empty or erroneous SQL result, the text and visualization agents are never reached, so the resulting missing scores are simply excluded from the denominators of the text and visualization means, leaving the official **quality_mean** artificially high.

To address this, two additional quality views are used throughout this test:

- **Component E2E quality** — the same three-component mean, but any expected score absent due to an upstream pipeline failure is counted as 0.0 rather than excluded. This is the primary accuracy metric for comparisons in this section.
- **Prompt E2E quality** — the per-prompt mean of the expected component scores, with missing expected outputs counted as 0.0. This view penalises each failed prompt as a single unit, regardless of how many components were expected from it.

5.1.2 Aggregate Results

Table 9 summarises the key metrics averaged over the three repeats. By the official `quality_mean` (Equation 4), `mistral-small13.2:24b` leads (0.900) with `gemma4:e4b` close behind (0.867), while `gemma4:26b` scores notably lower (0.792) due to a weak visualization component. The component E2E metric, which counts missing expected scores as zero, narrows the gap between the two leading models: Mistral reaches 0.827 and Gemma E4B reaches 0.821, while Gemma 26B trails at 0.772.

Model	Official	Comp. E2E	Prompt E2E	csv E2E	text E2E	vis E2E	Mean sec	Mean kWh
<code>mistral-small13.2:24b</code>	0.900	0.827	0.844	0.899	0.800	0.783	48.9	0.00601
<code>gemma4:e4b</code>	0.867	0.821	0.835	0.903	0.792	0.769	75.4	0.00883
<code>gemma4:26b</code>	0.792	0.772	0.814	0.928	0.890	0.500	136.0	0.01667

Table 9: Test 01 aggregate results per model averaged over three repeats (missing pipeline outputs counted as zero).

These aggregate values already define the baseline quality-energy ordering. `mistral-small13.2:24b` is the strongest baseline because it combines the highest strict quality with the lowest energy cost. `gemma4:e4b` reaches a nearly identical component E2E quality (0.821 vs. 0.827) but at $1.47\times$ the energy and $1.54\times$ the latency. `gemma4:26b` is Pareto-dominated at baseline: it records the lowest strict quality while consuming $2.77\times$ the energy of Mistral, with its visualization score (0.500) being the primary drag.

5.1.3 Radar Summary

Figure 5 summarizes the baseline profiles across overall quality, step-level quality, prompt difficulty, reliability, and energy efficiency. Quality and reliability axes are normalized with the fixed anchored scale $\text{clip}((x - 0.4)/0.6, 0, 1)$, where 0.4 represents a very weak result and 1.0 an excellent result. The energy-efficiency axis uses the anchored inverse scale $\text{clip}((0.017 - \text{kWh})/0.013, 0, 1)$.

The radar axes are defined as follows:

- **Overall:** strict prompt-level E2E quality, averaging expected SQL, text, and visualization slots per prompt; missing expected scores count as zero.
- **SQL:** strict `csv_iou` over prompts with data ground truth; missing expected data scores count as zero.
- **Text:** strict `text_score` over prompts with analysis ground truth; missing expected text scores count as zero.
- **Vis:** strict `vis_score` over prompts with visualization ground truth; missing expected visualization scores count as zero.
- **D1–2, D3, and D4:** strict prompt-level E2E quality averaged over prompts with difficulty 1–2, 3, and 4 respectively.
- **Completion:** completed expected GT score slots divided by all expected GT score slots.
- **No hard fail:** fraction of prompts with strict prompt-level quality greater than zero.
- **Energy eff.:** anchored inverse energy score from kWh per prompt; higher is better.

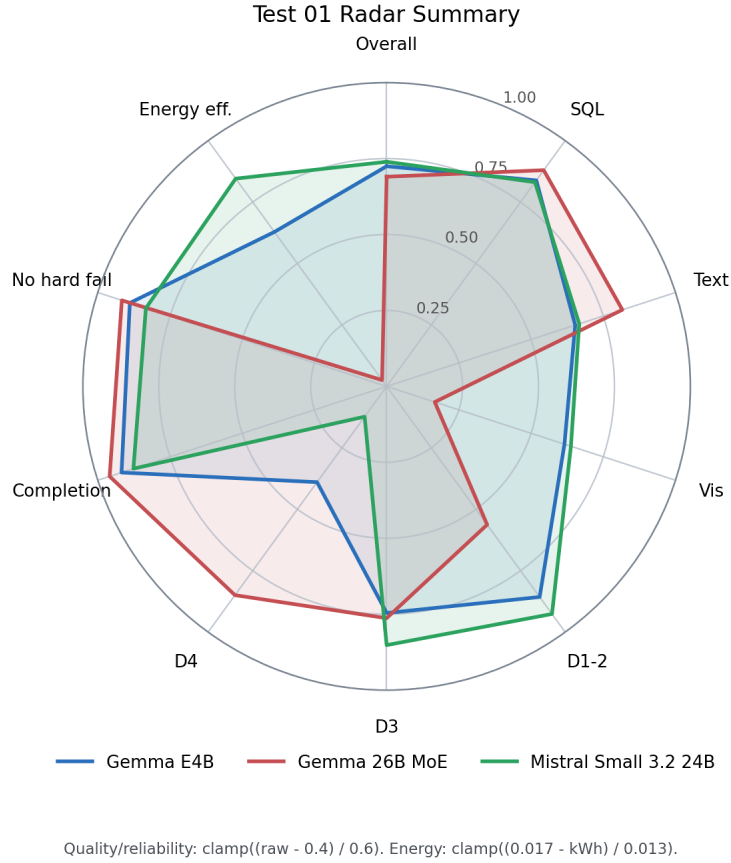


Figure 5: Test 01: radar summary of baseline model profiles. Quality and reliability axes use the fixed 0.4–1.0 scale; energy uses anchored inverse kWh per prompt.

Model	Overall	SQL	Text	Vis	D1–2	D3	D4	Compl.	No hard fail	kWh	Energy eff.
gemma4:e4b	0.835	0.903	0.792	0.769	0.914	0.848	0.634	0.951	0.933	0.00883	0.629
gemma4:26b	0.814	0.928	0.890	0.500	0.738	0.858	0.910	0.975	0.950	0.01667	0.026
mistral-small3.2:24b	0.844	0.899	0.800	0.783	0.956	0.911	0.474	0.926	0.900	0.00601	0.845

Table 10: Test 01: raw values used to build the radar summary.

The radar view makes the baseline trade-off compact. Mistral has the best energy-efficiency profile and is strongest on easy-to-medium prompts, but its difficulty-4 axis collapses. Gemma 26B has the opposite profile: strong SQL, text, completion, and hard-prompt behavior, but weak visualization and poor energy efficiency. Gemma E4B is the most balanced thinking baseline, although it is still more expensive than Mistral.

5.1.4 Repeat Stability

Figure 6 and Table 11 report the per-repeat metrics. Repeat variance is low enough for the model-level ranking to be stable: the largest quality range across repeats belongs to **gemma4:e4b** (0.077), while Mistral shows the tightest spread (range 0.021). Energy and latency are slightly more variable than scores because they depend on generated-token length and GPU runtime state.

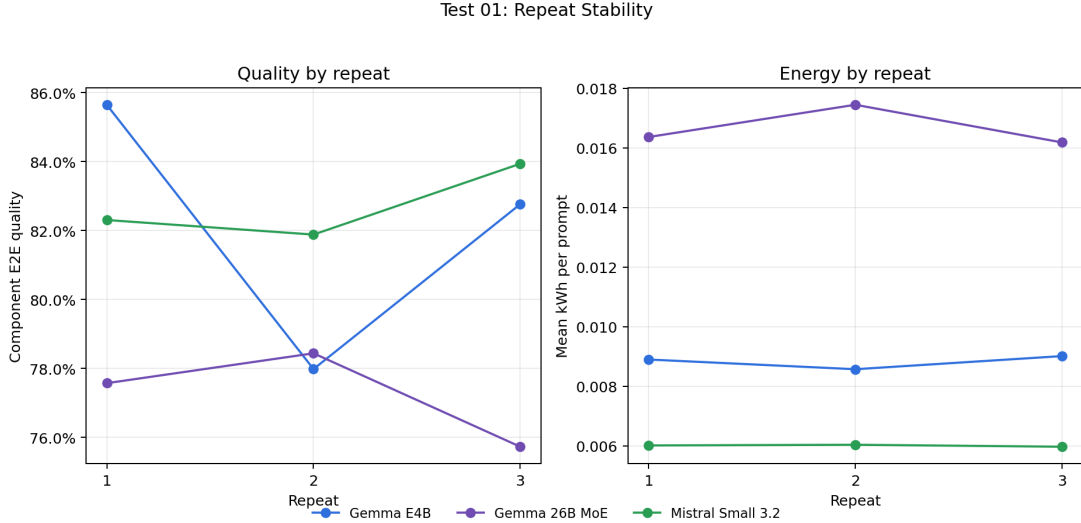


Figure 6: Test 01: component E2E quality per repeat by model.

Model	Repeat	Comp. E2E	Prompt E2E	Mean kWh	Mean sec
gemma4:e4b	rep 01	0.856	0.865	0.008899	76.7
gemma4:e4b	rep 02	0.780	0.804	0.008573	72.9
gemma4:e4b	rep 03	0.828	0.835	0.009015	76.5
gemma4:26b	rep 01	0.776	0.815	0.016366	134.0
gemma4:26b	rep 02	0.784	0.840	0.017449	142.0
gemma4:26b	rep 03	0.757	0.787	0.016185	131.9
mistral-small13.2:24b	rep 01	0.823	0.839	0.006016	49.0
mistral-small13.2:24b	rep 02	0.819	0.834	0.006039	49.0
mistral-small13.2:24b	rep 03	0.839	0.859	0.005973	48.6

Table 11: Test 01: per-repeat component E2E quality, prompt E2E quality, energy, and latency.

5.1.5 Pipeline Completion and Reliability

Table 12 reports the pipeline completion counts for each model. Across three repeats, all models produce complete CSV outputs for every prompt, but both text and visualization steps produce some missing expected scores concentrated on a small number of difficult prompts.

Model	Rows	csv complete	text complete	vis complete	Timeouts
gemma4:e4b	60	60	56	38	0
gemma4:26b	60	60	58	40	0
mistral-small13.2:24b	60	60	54	36	0

Table 12: Test 01: pipeline output counts per model over three repeats.

Both a CSV table and a text analysis are expected for every prompt, giving 60 expected slots per model (20 prompts $\times$ 3 repeats); a visualization, by contrast, is expected for only 42 slots, since just 14 of the 20 prompts request a chart (14 per repeat). All CSV slots complete, text completion is 54–58 of 60, and of the 42 visualization slots expected per model completion ranges from 85.7% for `mistral-small13.2:24b` (36 of 42) to 95.2% for `gemma4:26b` (40 of 42). Missing expected score slots are concentrated on difficulty-4 prompts:

`mistral-small13.2:24b` incurs 12 missing slots on prompts 4 and 17 (6 each, both scoring 0.000 across all repeats), while `gemma4:e4b` incurs 8 missing slots across prompts 17, 4, and 9.

5.1.6 Per-Agent Accuracy Breakdown

Figure 7 decomposes the component E2E quality into the three pipeline stages: CSV (data retrieval), Text (natural-language analysis), and Visualization. Missing expected scores are counted as zero in all bars.

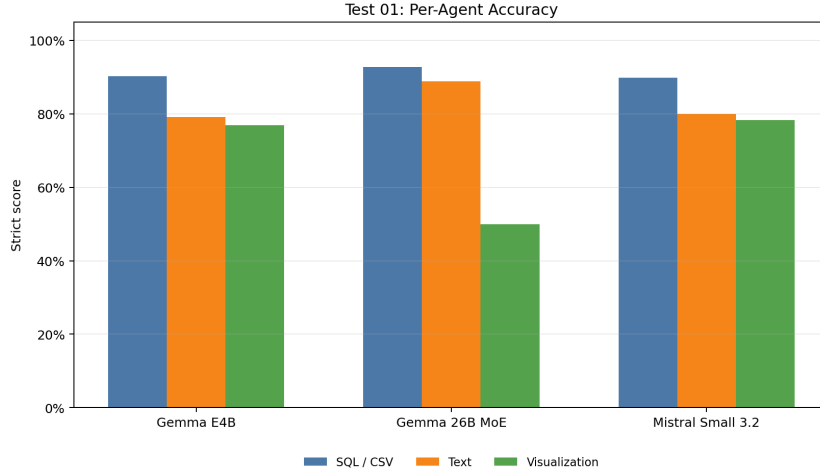


Figure 7: Test 01: per-agent E2E accuracy by model (missing outputs counted as zero).

The figure reveals distinct failure modes across the three models:

- `gemma4:26b` achieves the highest CSV accuracy (0.928) and the strongest text score (0.890), confirming its strength in structured data retrieval and natural-language reasoning. However, its visualization E2E score (0.500) is by far the lowest of the three, making visualization the primary drag on its aggregate quality.
- `mistral-small13.2:24b` achieves the best visualization score (0.783) and a solid text score (0.800), but its CSV accuracy (0.899) reflects complete SQL failures on difficulty-4 prompts that recur deterministically across all three repeats.
- `gemma4:e4b` shows the most balanced component profile: CSV (0.903), text (0.792), and visualization (0.769) are all within a similar range, with SQL failures on the same difficulty-4 prompts being the main source of quality loss.

5.1.7 Per-Model Error Analysis

gemma4:e4b. The main failures occur on prompts 17 and 4, both requiring a comparison of average monthly revenue across different store segmentation dimensions over 2022 and 2023. The generated SQL queries attempt the correct nested monthly aggregation logic but reference incorrect column aliases or date fields in the outer subquery, causing the lookup step to return empty results. A further partial failure occurs on prompt 9 (store cities exceeding a revenue threshold as a bar chart), where text and visualization outputs are produced but with reduced scores across repeats. These failure types motivate the lookup-step targeted interventions evaluated in Test 04.

gemma4:26b. The model does not fail on SQL. Its main weakness is concentrated in the visualization agent on two prompts:

- **Prompt 5** (promo vs. non-promo revenue as a bar chart): the chart type and data are approximately correct but the visualization configuration does not match the expected grouping structure (`mean quality` = 0.389, 2 missing slots).
- **Prompt 11** (monthly revenue split by promo flag as a grouped bar chart): similar grouping failure (`mean quality` = 0.500, 2 missing slots).

These failures suggest that the larger Gemma model requires targeted visualization tuning, as explored in Tests 02 and 03.

mistral-small13.2:24b. The model incurs the largest number of missing expected score slots (12), all concentrated on difficulty-4 prompts 4 and 17 (6 slots each). Both prompts require nested monthly revenue aggregation across store segmentation dimensions; both score exactly 0.000 quality across all three repeats, meaning the SQL failure recurs deterministically. Analysis completeness is a secondary weakness: several difficulty-1 through 3 prompts receive partial text scores due to incomplete analytical coverage or output schema mismatches.

5.1.8 Summary

The main findings of Test 01 can be summarised as follows:

1. **Mistral Small is the Pareto baseline.** `mistral-small13.2:24b` achieves the highest component E2E quality (0.827), the lowest latency (48.9 s/prompt), and the lowest energy (0.00601 kWh/prompt). It serves as the primary efficiency reference for all subsequent tests.
2. **Gemma E4B is close on aggregate quality but more expensive.** With three repeats, `gemma4:e4b` reaches 0.821 component E2E quality — within 0.7% of Mistral — but at $1.47\times$ the energy and $1.54\times$ the latency. It remains a viable candidate when absolute quality is prioritised over efficiency.
3. **Gemma 26B is held back by visualization.** `gemma4:26b` achieves the strongest CSV and text component scores, but its visualization E2E score (0.500) and aggregate quality (0.772) are the weakest overall. Without targeted visualization tuning it is Pareto-dominated.
4. **Failure modes are step-specific.** Mistral and Gemma E4B mainly lose quality through recurrent lookup failures on hard aggregation prompts, while Gemma 26B mainly loses quality in the visualization agent. This motivates the agent-specific parameter sweeps in the following tests.

5.2. Test 02 - Per-Agent Sensitivity Analysis (OFAT)

Test 02 applies a One-Factor-At-a-Time (OFAT) protocol to quantify how individual static sampling hyperparameters affect each agent step in isolation. The experimental design is described in Section 4.2: each of the five parameters (`temperature`, `top_p`, `top_k`, `repeat_penalty`, `repeat_last_n`) is swept individually across ten values per step, producing 153 configurations per model. Each configuration is repeated twice over a 15-prompt subset of the benchmark, reduced from the full 20-prompt set to limit total run time while preserving the original difficulty distribution across all four SQL complexity levels (difficulty 1: 3 prompts, difficulty 2: 4 prompts, difficulty 3: 5 prompts, difficulty 4: 3 prompts).

5.2.1 Overall Resource Use

Table 13 reports the aggregate statistics averaged across all 153 configurations and repeats. The gap from Test 01 persists: `mistral-small13.2:24b` runs at 67.2 s and 0.00416 kWh per prompt while `gemma4:26b` requires 189.0 s and 0.01189 kWh. `gemma4:e4b` sits between them at 99.8 s and 0.00578 kWh per prompt. Note that the mean quality values here are lower than Test 01 because they average over all OFAT configurations — including suboptimal off-baseline settings — rather than reflecting only the best or baseline performance.

Model	Mean quality	Mean sec/prompt	Mean kWh/prompt
<code>gemma4:e4b</code>	0.8382	99.8	0.00578
<code>gemma4:26b</code>	0.8325	189.0	0.01189
<code>mistral-small13.2:24b</code>	0.8783	67.2	0.00416

Table 13: Test 02: aggregate resource use averaged across all 153 OFAT configurations.

The mean quality values here are not intended as model rankings because they average both good and deliberately poor off-baseline configurations. They do, however, reveal a useful cost ordering for the tuning campaign: `mistral-small13.2:24b` is the low-energy control, `gemma4:26b` is the expensive thinking MoE, and `gemma4:e4b` is the intermediate-cost small thinking model.

As a stability check, the matched repeats show small energy variance relative to the model-level cost gap. Quality variance is more visible, especially for `gemma4:26b`, so small deltas should not drive final parameter choices; the conclusions below focus on large, step-consistent effects.

5.2.2 Sensitivity Range by Agent Step

Figure 8 reports the best-minus-worst quality delta (“sensitivity range”) for each agent step and each model, summarising how much the choice of sampling parameters can move overall pipeline quality when varying a single step.

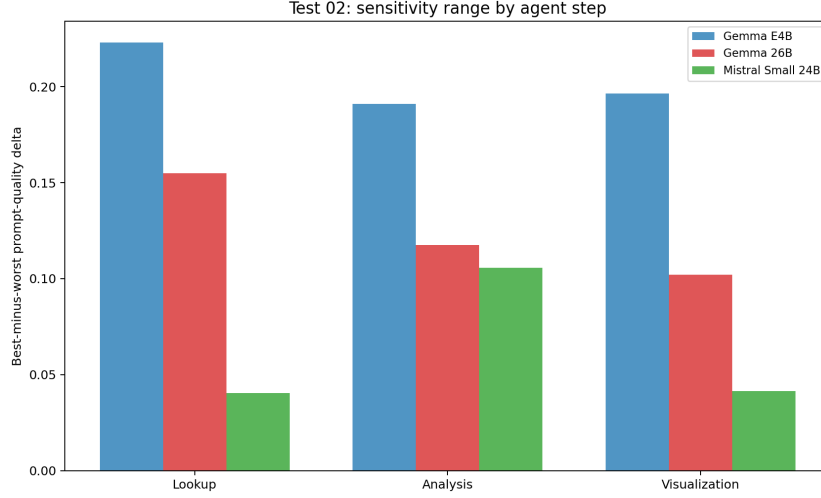


Figure 8: Test 02: sensitivity range (best-minus-worst prompt E2E quality delta) by step and model.

Model	Step	Baseline	Best Δ	Worst Δ	Range	Best config	Worst config
gemma4:e4b	Lookup	0.8062	+0.1168	−0.1061	0.223	top_p_0.94	repeat_penalty_1.2
gemma4:e4b	Analysis	0.8689	+0.0618	−0.1291	0.191	top_k_96	repeat_last_n_128
gemma4:e4b	Visualization	0.8480	+0.0842	−0.1121	0.196	top_p_0.7	temperature_0.7
gemma4:26b	Lookup	0.8480	+0.0319	−0.1229	0.155	repeat_last_n_96	top_k_56
gemma4:26b	Analysis	0.8675	+0.0065	−0.1111	0.118	top_k_24	top_p_0.75
gemma4:26b	Visualization	0.8121	+0.0679	−0.0342	0.102	repeat_penalty_1	top_p_0.8
mistral-small13.2:24b	Lookup	0.8861	+0.0042	−0.0362	0.040	repeat_penalty_1.3	repeat_last_n_80
mistral-small13.2:24b	Analysis	0.8861	+0.0069	−0.0986	0.106	top_k_56	repeat_penalty_1.2
mistral-small13.2:24b	Visualization	0.8889	+0.0014	−0.0401	0.042	repeat_last_n_56	temperature_0.1

Table 14: Test 02: sensitivity range and best/worst prompt E2E quality configuration per (model, step).

The most striking result visible in both the figure and the table is the model-level asymmetry. **gemma4:e4b** has the largest observed ranges in all three steps, with lookup spanning 0.223 quality points between the best and worst setting. This is too large to treat as a minor side effect: the small thinking model is strongly parameter-sensitive. For **gemma4:26b**, the lookup step still has a large sensitivity range of 0.155, while the visualization step offers the largest reliable upside (best $\Delta = +0.068$). For **mistral-small13.2:24b**, by contrast, the lookup and visualization ranges are just 0.040 and 0.042. The only step where Mistral shows meaningful sensitivity is analysis, where **repeat_penalty=1.2** causes a large quality drop (−0.099).

5.2.3 Quality Delta Heatmap by Parameter Family

Figure 9 shows, for each (step, parameter) pair, the best quality delta achieved over all tested values of that parameter, relative to the phase baseline. Green cells indicate that at least one value of that parameter improved pipeline quality; red cells indicate that the best achievable value still degraded it.



Figure 9: Test 02: best prompt E2E quality delta per (step, parameter) pair across all runs.

For `gemma4:e4b`, the largest signal is lookup tuning: `top_p` values close to the baseline, especially `top_p=0.94` and `top_p=0.98`, improve prompt E2E quality by more than 0.11 while adding little energy. This supports the later focus on E4B lookup repair.

The visualization block for `gemma4:26b` deserves particular attention. All five parameter families yield positive best deltas, ranging from `+0.043` (`top_p`) to `+0.068` (`repeat_penalty`). This is directly consistent with the Test 01 finding that Gemma 26B’s visualization step was its primary weakness: the step is not only weak at baseline but also the most responsive to tuning.

The red cell at Analysis/`temperature` for `gemma4:26b` reveals an important asymmetry: reducing temperature below 1.0 hurts the analysis agent’s text generation more than it helps it. The model’s thinking mechanism appears to rely on a degree of sampling diversity that is lost when temperature is lowered, a behaviour that does not apply to the non-thinking `mistral-small13.2:24b`.

5.2.4 Direct Step-Metric Delta Heatmap

Figure 10 shows the same structure as the quality heatmap but reports the best delta in the *direct* step metric (`csv_iou` for lookup, `text_score` for analysis, `vis_score` for visualization) rather than in overall pipeline quality. This separates the effect on the targeted step from any propagation into downstream steps.



Figure 10: Test 02: best direct step-metric delta per (step, parameter) pair.

Comparing Figures 9 and 10 reveals two useful details. First, the E4B direct metrics confirm that the quality gains are not only downstream noise: lookup $\text{top_p}=0.94$ raises csv_iou by about 0.139, and the best visualization settings also raise vis_score strongly. Second, for `gemma4:26b`, the analysis/temperature cell is negative in the direct metric heatmap, meaning that temperature changes can degrade the analysis text itself rather than merely propagating losses forward. Conversely, small positive quality deltas in Gemma 26B analysis are near the repeat noise floor and should not be over-interpreted as structural gains.

5.2.5 Quality-Energy Trade-off Across Configurations

Figure 11 plots every tested configuration as a point in the $(\Delta\text{energy}, \Delta\text{quality})$ plane, providing a simultaneous view of whether quality-improving configurations also save or cost energy. Here *quality* is the prompt-level end-to-end quality (Prompt E2E quality, Section 5.1.1), which counts missing downstream outputs as zero, and not the direct step metric (csv_iou , text_score , or vis_score).

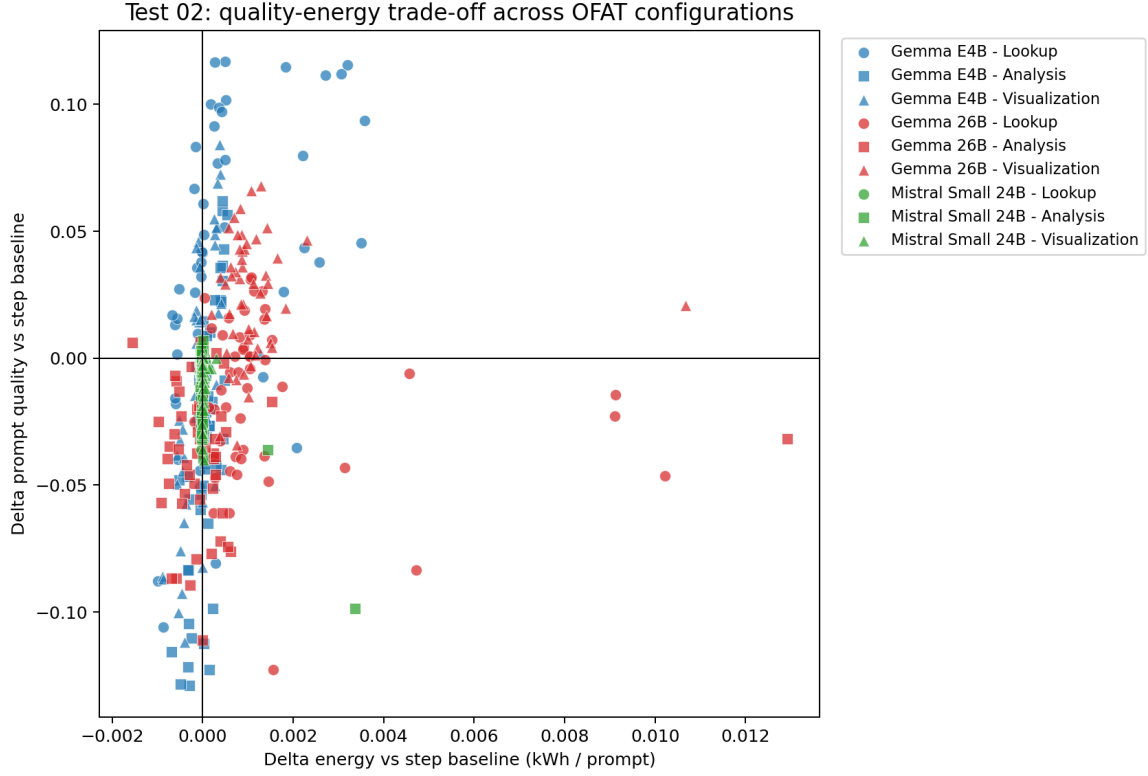


Figure 11: Test 02: prompt E2E quality delta vs. energy delta per OFAT configuration.

The three models occupy structurally different regions of the plot. `mistral-small13.2:24b` configurations form a narrow vertical band near $\Delta E = 0$: quality varies across parameter choices but energy consumption remains nearly constant, reflecting the model’s efficient architecture. The only clear outlier is `analysis_repeat_penalty_1p2` (-0.099 quality, $+0.00337$ kWh, $+90.4$ s), which simultaneously degrades text quality and inflates latency — an anomalous combination that suggests repeat-penalty settings interact poorly with Mistral’s internal sampling at the analysis step. For Mistral, parameter tuning is therefore primarily a quality concern; energy cost is largely insensitive to the chosen value.

`gemma4:e4b` occupies an intermediate-cost region. The strongest E4B quality gains are lookup configurations, many of which add only a few 10^{-4} kWh per prompt. This is the main new information from including E4B directly in the plot: the small thinking model has a practical static-tuning lever that is cheaper than the Gemma 26B visualization improvements.

`gemma4:26b` shows the opposite profile. Its configurations are scattered across a wide energy range, and the upper half of the scatter — quality-improving configurations — is dominated almost exclusively by visualization-step variants. Within the repeated Gemma 26B data, the strongest quality improvements belong to visualization variants, clustered in the upper-right quadrant (quality gain with a moderate energy increase). The best absolute improvement, `visualization_repeat_penalty_1`, delivers $+0.068$ quality at a cost of $+0.00128$ kWh and $+23.5$ s per prompt, although its direct `vis_score` is nearly unchanged. A more energy-conscious option is `visualization_repeat_penalty=1.03`: it yields $+0.051$ quality and the best direct `vis_score` gain ($+0.030$) for only $+0.00057$ kWh and $+12.9$ s — roughly half the energy overhead of the best-quality setting. These results indicate that the visualization step is the primary tuning lever for `gemma4:26b`: modest parameter changes here produce the largest and most consistent quality returns across the entire test.

The lower-right quadrant — quality loss with energy increase, the worst possible outcome — is populated by `gemma4:26b` lookup and analysis configurations, most notably `lookup_top_k_56` (-0.123 quality, $+0.00157$ kWh) and `lookup_top_k_48` (-0.083 quality, $+0.00471$ kWh). These represent configurations to actively avoid in production deployment.

5.2.6 Best and Worst Configurations

Table 15 lists the top quality-improving and top quality-degrading configurations across all models and all steps, along with the corresponding direct step-metric delta, energy delta, and latency delta.

Model	Step	Config	Δ quality	Δ metric	Δ kWh	Δ sec
<i>Top quality improvements</i>						
gemma4:e4b	Lookup	top_p_0.94	+0.1168	+0.1393	+0.00050	+7.4
gemma4:e4b	Lookup	top_p_0.98	+0.1165	+0.1254	+0.00028	+5.5
gemma4:e4b	Lookup	temperature_0.95	+0.1154	+0.0851	+0.00320	+83.4
gemma4:e4b	Lookup	temperature_0.8	+0.1148	+0.0773	+0.00183	+46.8
gemma4:e4b	Lookup	top_p_0.85	+0.1119	+0.0950	+0.00307	+79.4
<i>Largest quality losses</i>						
gemma4:e4b	Analysis	repeat_last_n_128	−0.1291	−0.1236	−0.00030	−4.7
gemma4:e4b	Analysis	top_p_0.7	−0.1285	−0.0852	−0.00049	−8.1
gemma4:26b	Lookup	top_k_56	−0.1229	0.0000	+0.00157	+34.4
gemma4:e4b	Analysis	temperature_1.05	−0.1229	−0.1250	+0.00015	+1.7

Table 15: Test 02: top prompt E2E quality-improving and quality-degrading OFAT configurations.

A notable feature of the improvement list is that the top rows belong to **gemma4:e4b** lookup. This does not overturn the Gemma 26B visualization result, but it shows that the small thinking model has a high-upside and relatively cheap tuning lever. In particular, **top_p=0.94** improves prompt quality by +0.117 and **csv_iou** by +0.139 for only +0.00050 kWh/prompt.

On the loss side, E4B analysis settings dominate the worst rows. This is consistent with the difficulty analysis below: when the small thinking model’s analysis generation is destabilized, the damage concentrates on difficult prompts and can be large. The Mistral **analysis_repeat_penalty=1.2** configuration remains an important misconfiguration even though it does not appear in the top four worst rows: it loses about 0.099 quality.

5.2.7 Per-Model Analysis

gemma4:e4b. **gemma4:e4b** is the most sensitive model in Test 02. Its range is 0.223 at lookup, 0.196 at visualization, and 0.191 at analysis. The largest useful signal is lookup tuning: **lookup_top_p=0.94** improves prompt E2E quality by +0.117 and **csv_iou** by +0.139 with only a small energy increase. The negative side is just as important: analysis settings such as **repeat_last_n=128** and low **top_p** can degrade quality by about 0.13. Thus E4B should be treated as a high-leverage but fragile small thinking model, motivating the later focus on E4B lookup repair.

gemma4:26b. The sensitivity profile of **gemma4:26b** is strongly step-specific and consistently asymmetric. The visualization step is unambiguously the highest-leverage target: every parameter family improves it, and the gains are large enough (up to +6.8% quality, +5.2% **vis_score**) to be practically meaningful and to more than offset the small-delta uncertainty identified by the matched-repeat check. The lookup step shows meaningful positive potential (+3.2% quality via **repeat_last_n_96**) but also a very steep downside risk: a single bad **top_k** value can collapse quality by −12.3%, likely by disrupting the structured SQL generation. The analysis step is the most fragile: temperature changes in particular push the model into degenerate outputs, and the safe operating window for this step is narrow around the baseline.

mistral-small13.2:24b. **mistral-small13.2:24b** is almost entirely insensitive to the parameters tested. The best achievable quality delta at any step is +0.007, which is within the small-delta uncertainty range. The model’s low baseline temperature (0.15) and the non-thinking architecture appear to make its output distribution far less responsive to sampling parameter perturbations than the thinking model. The practical implication is that static parameter tuning offers negligible quality improvement for Mistral, and that pursuing efficiency gains for this model requires structural changes rather than hyperparameter search. The sole exception is the **analysis_repeat_penalty_1.2** misconfiguration described above, which serves as a warning against assuming that insensitivity to improvement implies insensitivity to regression.

5.2.8 Accuracy by Prompt Difficulty

Figures 12 and 13 examine whether parameter sensitivity is uniform across prompt difficulty levels.

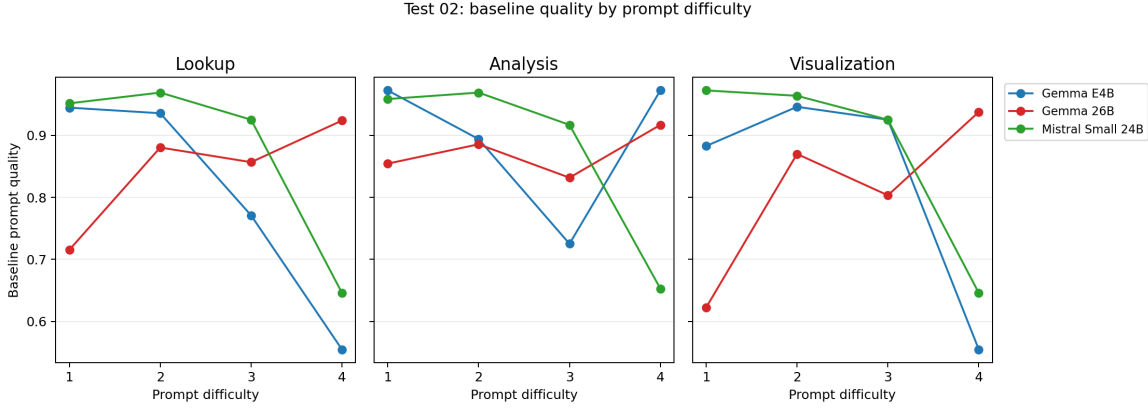


Figure 12: Test 02: baseline prompt-level end-to-end quality (Prompt E2E quality) by prompt difficulty for each model and agent step.

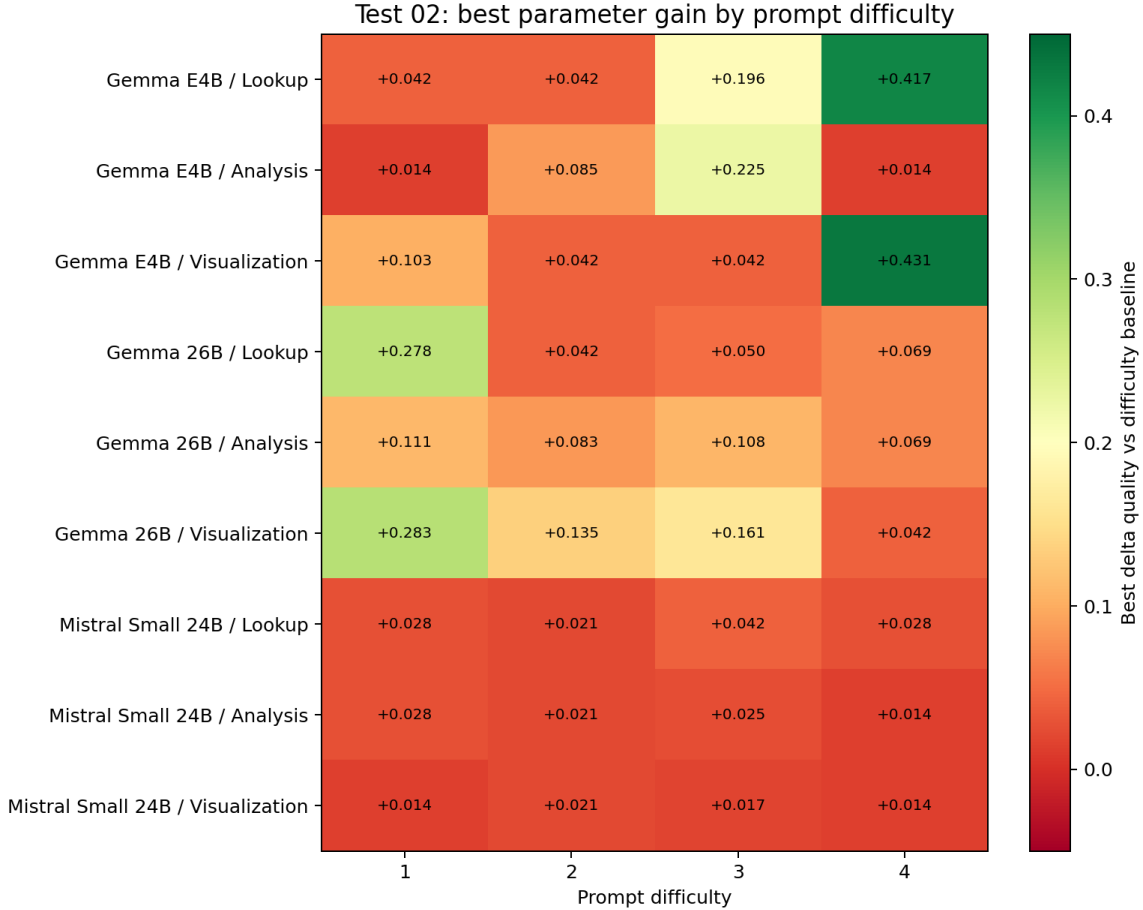


Figure 13: Test 02: best prompt E2E quality delta by prompt difficulty (heatmap).

Parameter sensitivity is not uniform across task difficulty. The largest observed swings belong to `gemma4:e4b`: visualization at difficulty 4 improves by +0.431 under `repeat_penalty=0.9`, lookup at difficulty 4 improves by +0.417 under `repeat_penalty=1.08`, and analysis at difficulty 4 can lose as much as -0.728 under `temperature=1.2`. The pattern is clear: hard prompts amplify the effect of parameter tuning. `gemma4:26b` shows the same mechanism, but at a smaller scale. It obtains large gains on weak slices, such as visualization at difficulty 1 (+0.246) and lookup at difficulty 1 (+0.194), while analysis at difficulty 4 can

degrade by -0.312 . `mistral-small13.2:24b` remains flatter across difficulty levels, confirming it as a robust baseline but also showing why static OFAT tuning finds little upside for this model.

5.2.9 Summary

Test 02 establishes that sensitivity to static sampling parameter tuning is a model-level property, not a step-level one:

1. **gemma4:e4b shows the widest sensitivity range of any model in the test.** Its lookup step alone spans 0.223 quality points between the best and worst static setting, exceeding any step of Gemma 26B or Mistral. Starting from the weakest lookup baseline (0.806), `lookup_top_p_0p94` delivers the strongest single-axis gain of the entire sweep at $+0.117$ quality over baseline with low energy overhead.
2. **gemma4:26b is highly tunable but fragile.** Sensitivity ranges reach 0.155 for lookup and 0.118 for analysis. The visualization step is the most rewarding target: all parameter families improve it, with the best configurations delivering up to $+6.8\%$ quality at low energy overhead.
3. **mistral-small13.2:24b is insensitive.** No parameter at any step moves quality by more than $+0.007$, confirming that this model’s efficiency advantage from Test 01 is robust to misconfiguration but also leaves little room for improvement through static hyperparameter tuning.
4. **The best repeated Gemma 26B gains still target visualization.** E4B lookup dominates the top-gain table, but Gemma 26B’s repeated evidence still points to visualization as its main tuning lever.
5. **Temperature reduction hurts Gemma 26B analysis.** The thinking model’s analysis quality degrades when temperature is lowered below the baseline of 1.0, suggesting that the model’s thinking process benefits from sampling diversity.
6. **Prompt difficulty magnifies tuning effects.** The largest gains and losses concentrate on hard prompts, especially for the thinking Gemma models.

5.3. Test 03 - max_tokens Ladder per Agent

Test 03 sweeps the `max_tokens` parameter for each agent step in isolation, using four discrete values per step. Unlike the sampling parameters tested in Test 02, `max_tokens` acts as a hard output-length ceiling: the model is forced to stop generating once the budget is exhausted, potentially truncating incomplete SQL queries, analysis paragraphs, or visualization code. Throughout the sweep the other sampling parameters (temperature, top- p , top- k) are kept at their baseline values, so that any observed effect can be attributed to the token budget alone; when one step’s cap is varied, the two non-varied steps are given a generous cap so that they never become the bottleneck. The test is run on a 10-prompt subset of the 20 benchmark prompts, evaluated over 12 configurations for three repeats per model (3 repeats $\times$ 12 configs $\times$ 10 prompts = 360 scored prompt runs per model).

5.3.1 Token Caps Tested

Table 16 lists the four `max_tokens` levels swept for each agent step. The levels are not arbitrary: each ladder brackets the step’s baseline cap—the models’ operating default already used in Tests 01 and 02 (underlined)—with two lower values, which probe the budget at which outputs begin to be truncated, and one higher value, which checks whether additional headroom yields any further gain. The full rationale for isolating `max_tokens` in this way is given in Section 4.3.

Agent step	max_tokens values
lookup_sales_data	2500 — 3000 — <u>5000</u> — 8000
analyzing_data	4000 — 5000 — <u>7000</u> — 10000
create_visualization	4000 — 5000 — <u>7000</u> — 10000

Table 16: Test 03: `max_tokens` values tested per agent step (baseline underlined).

5.3.2 Quality Ladder by Model and Step

Figure 14 shows how overall prompt E2E quality (Section 5.1.1) responds as the token cap increases for each (model, step) combination. The three panels reveal three qualitatively different response profiles.

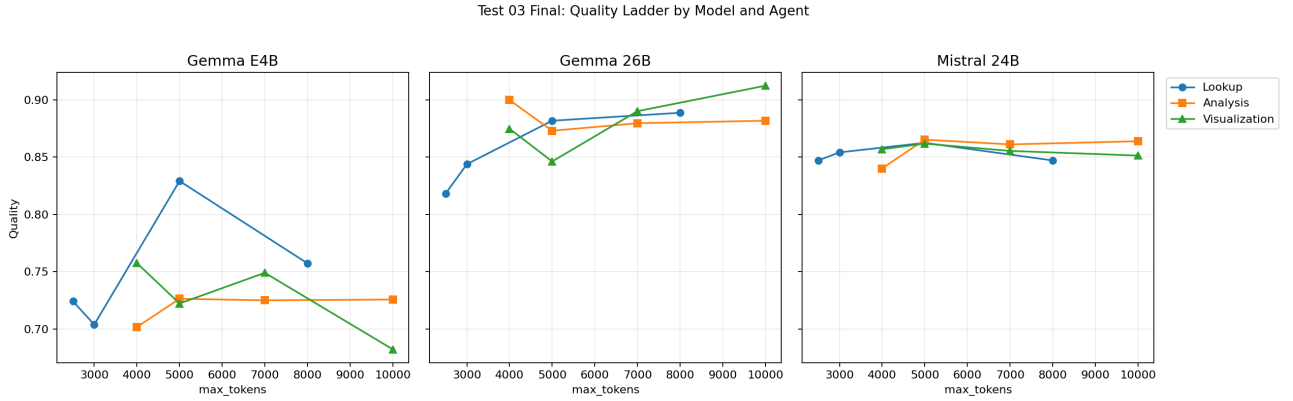


Figure 14: Test 03: prompt E2E quality vs. `max_tokens` by model and agent step.

The most striking feature of the Gemma E4B panel is the visualization line: quality improves from 4000 to 7000 and then falls from 0.75 to 0.69 at 10000. An analogous but less dramatic non-monotonicity appears in the lookup line, which peaks at 5000 (0.83) and decreases to 0.76 at 8000. These patterns suggest that beyond a certain budget the model shifts towards longer, more complex outputs that are harder to evaluate or that introduce structural errors.

The Gemma 26B analysis line shows the opposite direction: quality at the baseline 7000-token cap (0.880) is lower than at the tightest cap of 4000 (0.900), indicating that the model’s analysis generation benefits from the compression imposed by a shorter budget. This is the only (step, model) combination in the entire experiment where reducing the cap below the baseline yields a *non-trivial* positive quality effect: a handful of other pairs also peak below their baseline cap (gemma4:e4b visualization, +0.009; mistral-small13.2:24b visualization, +0.006, and analysis, +0.004; gemma4:e4b analysis, +0.001; see Table 17), but by margins small enough to be indistinguishable from measurement noise.

It must be noted, however, that even this +0.020 analysis gain is within the small-delta uncertainty observed for gemma4:26b, and that its direct `text_score` delta is 0.000 (Table 17); the analysis-cap effect should therefore be read as directional rather than as a robust, metric-supported gain. The same caution applies to the +0.022 visualization quality gain discussed below, which however is corroborated by a direct `vis_score` improvement of +0.054 that does clear the same practical threshold.

mistral-small13.2:24b shows a flat, plateau-like profile across all three steps: the total quality range is 0.025 for analysis, 0.015 for lookup, and only 0.010 for visualization, all within or near the small-delta uncertainty range. The model appears to generate complete, well-formed outputs even at moderate token caps, so additional headroom does not translate into measurable quality gains. This is a practically useful property: the cap can be set conservatively to limit resource use without risking quality regression.

5.3.3 Direct Step-Metric Ladder

Figure 15 isolates the effect on the direct agent metric (`csv_iou`, `text_score`, or `vis_score`) rather than on overall pipeline quality.



Figure 15: Test 03: direct step metric vs. `max_tokens` by model and agent step.

The Gemma 26B visualization result is the most actionable finding of this test. Increasing the visualization

budget from the baseline 7000 to 10000 tokens yields a **vis_score** gain of +0.054 and an overall quality gain of +0.022 at essentially zero additional energy cost (see Table 17). For Gemma 26B, this cap increase is a straightforward free improvement. It is also the only visualization curve that does not show a visible plateau inside the tested range: the direct **vis_score** continues to improve up to the largest tested value. This does not prove that still larger caps would continue to help, but it suggests that the 10000-token cap should be interpreted as a conservative upper value rather than as a measured saturation point for the Gemma 26B visualization agent.

For Gemma E4B, the collapse at 10000 tokens in visualization — visible both in the quality and the direct metric — is a safety-relevant finding: using the configuration that offers the most output space actually produces the worst chart code. The model appears to ramble or abandon its output structure when given excessive budget. The direct metrics of **mistral-small13.2:24b** mirror the quality plateau: **csv_iou** is virtually constant across all lookup caps (range 0.0002), and **vis_score** varies by at most 0.021. The token budget is simply not a binding constraint for this model.

5.3.4 Completion Ladder

Figure 16 shows how the pipeline completion rate changes with the token cap.

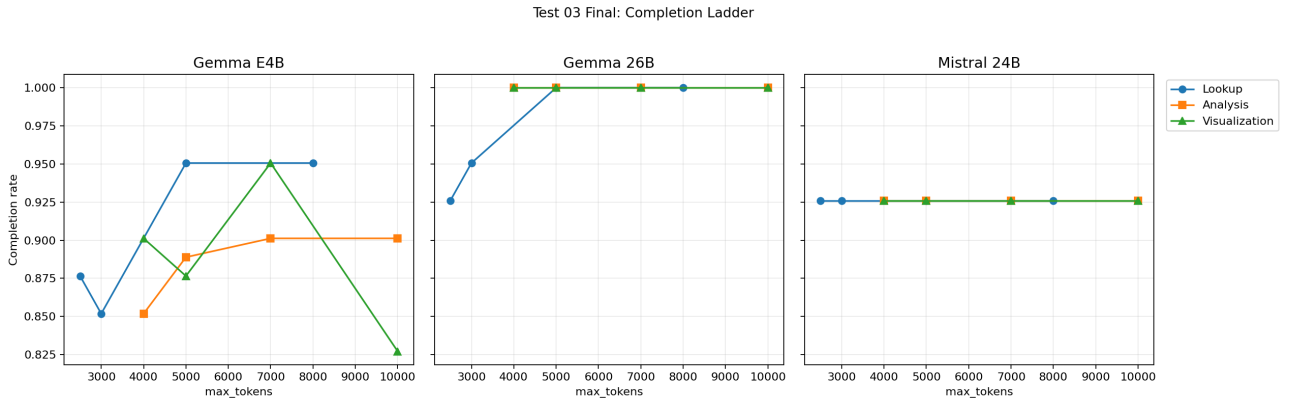


Figure 16: Test 03: full-pipeline completion rate vs. **max_tokens** by model.

The Gemma 26B lookup result has a direct mechanistic interpretation: at 2500 tokens the model is being cut off mid-SQL on some prompts, which produces syntactically incomplete queries that the database cannot execute. Raising the cap to 5000 tokens eliminates these truncations entirely. The baseline cap of 5000 for lookup is therefore not just optimal but necessary for this model.

For **gemma4:e4b**, the completion ladder reflects the same instability seen in the quality curves: the model achieves the lowest overall completion rate of the three (89.4%), and the visualization step is the main source of failures — the same step where the quality collapse at 10000 tokens was observed. A tight cap does not necessarily help either, since low budgets can trigger truncated chart code that fails at execution. The model sits in a narrow feasible window.

mistral-small13.2:24b maintains a stable completion rate across all tested caps (92.6% overall), with no step or cap value producing a notable drop. This further confirms that the model is robust to token budget choices and does not require careful cap tuning to avoid structural failures.

The completion curves also show an interesting similarity between the two Gemma thinking models. Despite the large difference in parameter count, the completion plateaus occur at comparable cap regions: once the lookup and visualization agents have enough room to finish their structured output, raising the nominal cap further does not substantially increase the number of completed score slots. A plausible explanation is that the effective reasoning budget actually used by the two Gemma models is of similar order on this benchmark; the larger model is not simply consuming proportionally more tokens to think. This remains an inference from completion behaviour, not a direct token-count measurement, but it is consistent with the small energy deltas observed when caps are raised.

5.3.5 Quality-Energy Trade-off

Figure 17 shows each cap configuration as a point in the (Δ energy, Δ quality) plane, relative to each model’s step baseline.

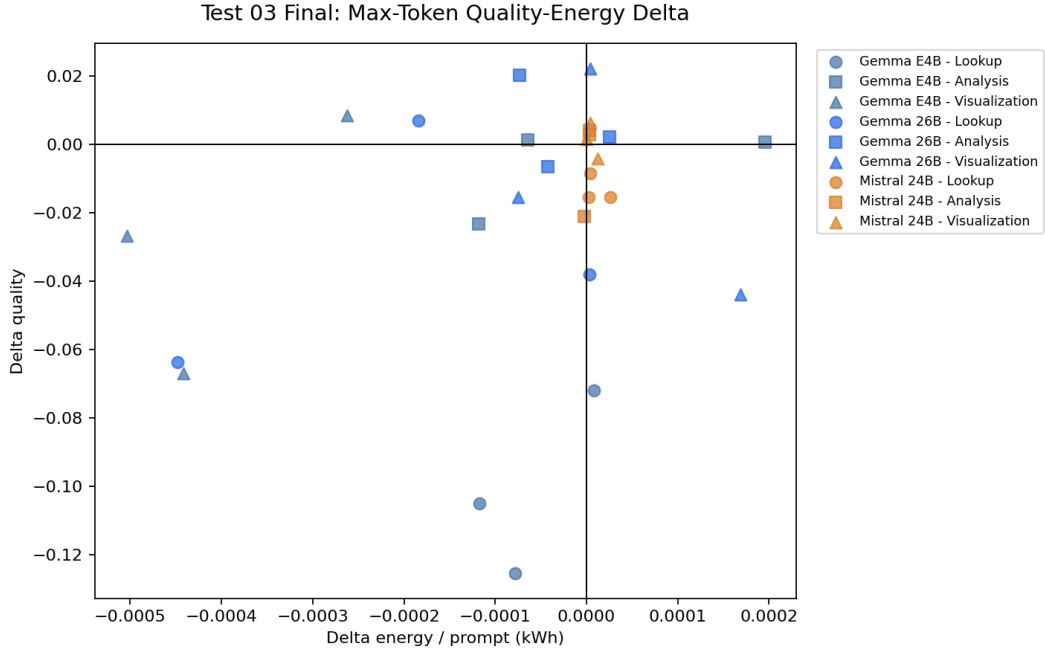


Figure 17: Test 03: prompt E2E quality delta vs. energy delta relative to per-step baseline cap.

The most notable feature of the scatter is that almost all points cluster near $\Delta E = 0$, regardless of model or step. Unlike sampling parameters (Test 02), token caps do not directly drive energy consumption: the model generates until its output is complete and the cap only becomes binding when the budget is too tight to finish. As a consequence, the available quality improvements are largely cost-free.

The clearest example is **gemma4:26b** visualization: raising the cap from 7000 to 10000 tokens yields +0.022 quality and +0.054 **vis_score** at $\Delta E \approx 0$. The same model’s analysis step moves in the opposite direction on the quality axis when the cap is raised, with a slight energy saving at lower caps — consistent with the compression effect discussed earlier.

This result changes the practical interpretation of **max_tokens**. The token cap is not itself a request to spend more compute: it only raises the ceiling available to the model. If the model finishes before hitting the ceiling, energy remains almost unchanged. For this reason, when a model-step pair reacts well to additional headroom, as Gemma 26B visualization does, the rational choice is to allocate the larger cap. The cost of allowing more tokens is small, while the cost of denying them can be a hard truncation or malformed output. Conversely, the Gemma E4B lookup and visualization curves show that generous caps must still be bounded: beyond the feasible window, additional headroom encourages longer but less reliable outputs.

The lower quadrants are populated by configurations where a cap that is too restrictive causes truncated or structurally broken outputs, particularly for **gemma4:e4b** at the visualization step. These are the only points where energy and quality move together downward, because a truncated generation is both shorter and useless. Table 17 summarises the per-step sensitivity range and the best-performing cap for each (model, step) pair. The merged view preserves both pieces of information needed for interpretation: how wide the cap-induced quality/metric variation is, and whether the best cap differs materially from the baseline.

Model	Step	Qual. range	Metric range	Baseline cap	Best cap	Δ quality	Δ metric	Worst cap
gemma4:e4b	Lookup	0.125	0.157	5,000	5,000	0.000	0.000	3,000
gemma4:e4b	Visualization	0.075	0.253	7,000	4,000	+0.009	0.000	10,000
gemma4:e4b	Analysis	0.025	0.021	7,000	5,000	+0.001	+0.017	4,000
gemma4:26b	Lookup	0.071	0.078	5,000	8,000	+0.007	+0.021	2,500
gemma4:26b	Visualization	0.066	0.149	7,000	10,000	+0.022	+0.054	5,000
gemma4:26b	Analysis	0.027	0.021	7,000	4,000	+0.020	0.000	5,000
mistral-small13.2:24b	Visualization	0.010	0.021	7,000	5,000	+0.006	0.000	10,000
mistral-small13.2:24b	Analysis	0.025	0.063	7,000	5,000	+0.004	+0.004	4,000
mistral-small13.2:24b	Lookup	0.015	0.000	5,000	5,000	0.000	0.000	2,500

Table 17: Test 03: **max_tokens** sensitivity range and best cap per (model, step).

5.3.6 Accuracy by Prompt Difficulty

Figures 18 and 19 examine whether the effect of `max_tokens` varies with prompt difficulty, from two complementary angles. Figure 18 reports the *sensitivity range* — the gap between the best and worst quality across all token caps in each (model, step, difficulty) slice — which measures how much the cap choice matters at all, in either direction. Figure 19 instead reports the *best-cap delta* — the quality gain of the best cap over the baseline cap (the best cap is annotated in each cell) — which measures how much can be gained by re-tuning the cap relative to its default. The two views can diverge: a slice may show a large sensitivity range yet a near-zero best-cap delta when the baseline cap is already optimal but an overly tight cap would sharply degrade quality.

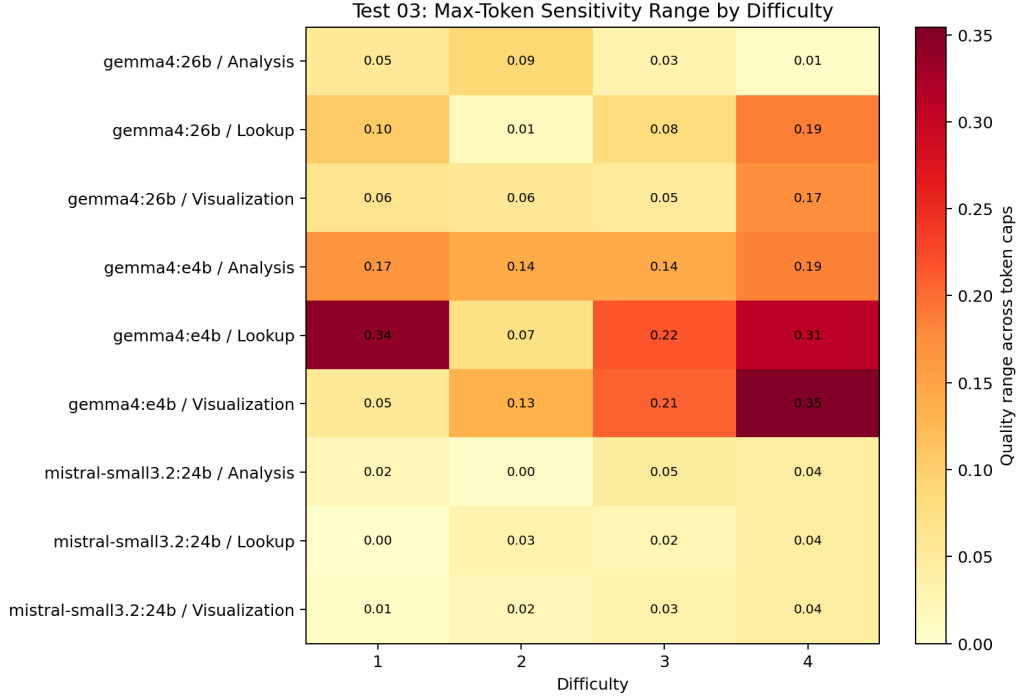


Figure 18: Test 03: `max_tokens` sensitivity by prompt difficulty for each model and step.

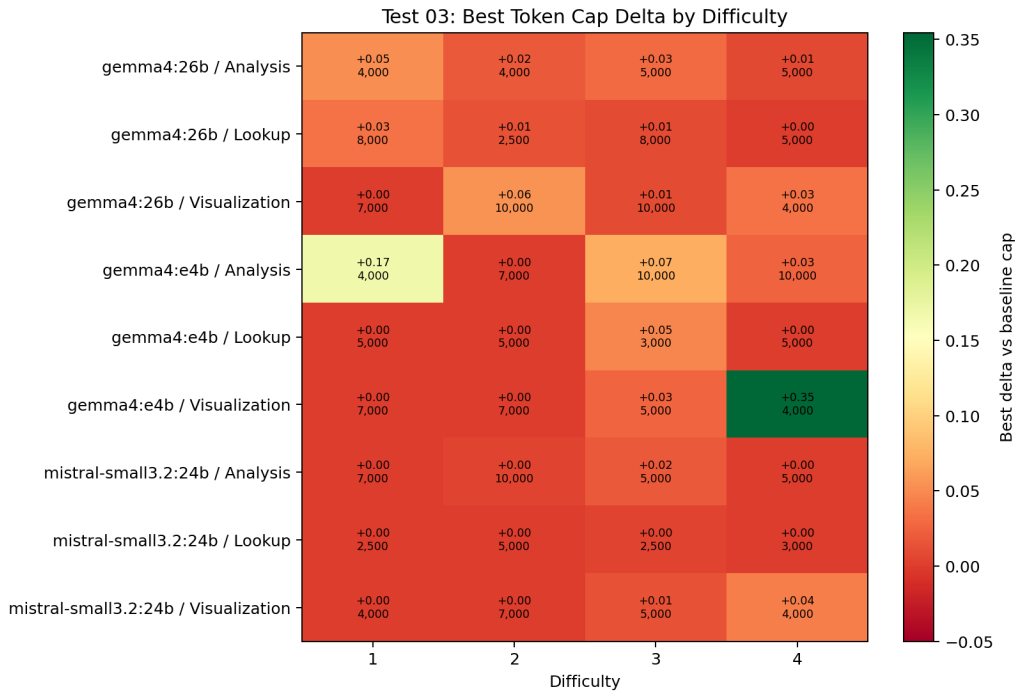


Figure 19: Test 03: best `max_tokens` cap delta by prompt difficulty.

The token cap behaves as a failure-prevention bound rather than a smooth quality dial, and its impact is strongest on slices where the default cap causes truncation or malformed outputs. The most striking case is **gemma4:e4b** visualization at difficulty 4: reducing the cap from the 7000-token baseline to 4000 raises quality by +0.355, because the model generates overly long or structurally incorrect chart code at the default budget. The two heatmap views can also diverge: at **gemma4:e4b** lookup difficulty 1 the default 5000-token cap is already optimal, so no larger cap improves quality (best-cap delta of +0.000, quality staying at 0.939), yet the slice is far from insensitive — dropping the cap to 3000 collapses quality (a sensitivity range of 0.342). These results support using generous but not blindly maximal caps per agent: a cap that is too tight causes discrete failures on complex prompts, while an excessively high cap rarely helps and can encourage verbose outputs that reduce quality.

5.3.7 Summary

1. **max_tokens is a per-agent safety bound, not a universal accuracy knob.** For the majority of (model, step) pairs the optimal cap is either the baseline or one step away, and quality changes are small. The exceptions are structural: where low caps cause truncation-induced failures, raising the budget has a large discrete effect.
2. **Gemma 26B visualization is genuinely token-constrained.** Raising the cap from 7000 to 10000 yields a +0.054 `vis_score` gain with negligible energy change. This is the clearest actionable finding of Test 03. Since no plateau is visible at the top of the tested ladder, future runs could probe a slightly larger visualization cap for this model.
3. **Gemma 26B analysis benefits from a tighter cap.** The 4000-token cap outperforms the 7000-token baseline by +0.020 quality. The compression forces the model to produce a more concise, less diffuse analysis response.
4. **Gemma E4B is non-monotone at both lookup and visualization.** Higher caps beyond 5000 (lookup) and 7000 (visualization) cause quality regressions, with the 10000-token visualization cap producing the worst `vis_score` (0.51) of any configuration in the test. This model should therefore use bounded, agent-specific caps rather than blindly generous ones.
5. **Mistral is insensitive to max_tokens.** The quality range does not exceed 0.025 at any step, and the baseline cap is optimal or tied-optimal at all three steps. Mistral can therefore be run with moderate caps without sacrificing accuracy.

5.4. Test 04 - Compute Expansion via Chain-of-Thought

Test 04 evaluates Chain-of-Thought (CoT) iterations as a compute expansion strategy for the lookup and visualization steps. Each configuration requests `cot_n` $\in \{1, 2, 3, 4\}$ refinement iterations, where `cot_n` = 1 is the single-call baseline. An early-stop mechanism halts the loop when the model signals that the previous output is already satisfactory, so the executed depth can be lower than the requested depth. The test was run on **gemma4:e4b** and **mistral-small13.2:24b** only (3 repeats $\times$ 8 configs $\times$ 10 prompts = 240 rows each). **gemma4:26b** was excluded because its main weakness (visualization) was already addressed by the token-budget increase in Test 03, its SQL and text steps were near saturation, and its per-prompt cost is already the highest of the three models, making additional compute iterations particularly expensive relative to the expected gain.

5.4.1 Aggregate Results

Table 18 reports the CoT aggregate results per (model, step, depth), and highlights a clear asymmetry between the two CoT targets. For the lookup step, CoT consistently helps: **gemma4:e4b** gains up to +0.203 prompt E2E quality at `cot_n`=4 (with `csv_iou` rising from 0.757 to 0.983), and **mistral-small13.2:24b** peaks at +0.063 at depth 3. The numerical peak for Gemma E4B is therefore depth 4, but this should not be interpreted as evidence that four full refinement rounds are usually needed. As shown later in Table 19, the requested depth 4 configuration executes only 2.50 iterations on average and reaches full depth in just 13.3% of lookup rows. Depth 3 already fixes the hardest lookup slice (difficulty 4, +0.524 quality), and depth 2 captures most of the aggregate gain at lower cost. For the visualization step the pattern is reversed: quality degrades at every depth above 1 for both models, with **gemma4:e4b** losing up to -0.115 at depth 3. The energy columns show that visualization CoT adds little overhead in absolute terms (at most +0.00059 kWh from depth 1 to depth 4), yet every unit of that energy is wasted because quality declines at each step. Lookup CoT carries a larger energy cost (up to +0.00193 kWh at depth 4 for **gemma4:e4b**), but the quality gains are substantial enough to justify it.

Model	Config	Step	cot_n	Quality	Δ quality	Δ metric	kWh/prompt
gemma4:e4b	lookup_cot_n1	Lookup	1	0.6939	0.0000	0.0000	0.00251
gemma4:e4b	lookup_cot_n2	Lookup	2	0.8795	+0.1856	+0.1968	0.00346
gemma4:e4b	lookup_cot_n3	Lookup	3	0.8594	+0.1655	+0.1860	0.00356
gemma4:e4b	lookup_cot_n4	Lookup	4	0.8972	+0.2033	+0.2261	0.00444
gemma4:e4b	vis_cot_n1	Vis.	1	0.7565	0.0000	0.0000	0.00280
gemma4:e4b	vis_cot_n2	Vis.	2	0.7149	-0.0417	-0.1131	0.00310
gemma4:e4b	vis_cot_n3	Vis.	3	0.6413	-0.1152	-0.1607	0.00312
gemma4:e4b	vis_cot_n4	Vis.	4	0.7438	-0.0128	-0.0536	0.00339
mistral-small13.2:24b	lookup_cot_n1	Lookup	1	0.8562	0.0000	0.0000	0.00236
mistral-small13.2:24b	lookup_cot_n2	Lookup	2	0.8958	+0.0396	+0.0333	0.00271
mistral-small13.2:24b	lookup_cot_n3	Lookup	3	0.9188	+0.0626	+0.0898	0.00293
mistral-small13.2:24b	lookup_cot_n4	Lookup	4	0.8944	+0.0382	+0.0333	0.00279
mistral-small13.2:24b	vis_cot_n1	Vis.	1	0.8597	0.0000	0.0000	0.00262
mistral-small13.2:24b	vis_cot_n2	Vis.	2	0.8542	-0.0056	-0.0119	0.00267
mistral-small13.2:24b	vis_cot_n3	Vis.	3	0.8401	-0.0196	-0.0060	0.00274
mistral-small13.2:24b	vis_cot_n4	Vis.	4	0.8264	-0.0333	-0.0060	0.00280

Table 18: Test 04: CoT aggregate results per (model, step, depth). *Delta quality* is the change in prompt E2E quality (Section 5.1.1) and *delta metric* the change in the step’s direct metric (*csv_iou* for lookup, *vis_score* for visualization), both relative to the *cot_n=1* baseline.

5.4.2 Accuracy vs. Requested CoT Depth

Figure 20 plots overall prompt E2E quality against the requested *cot_n* value for both steps and both models.

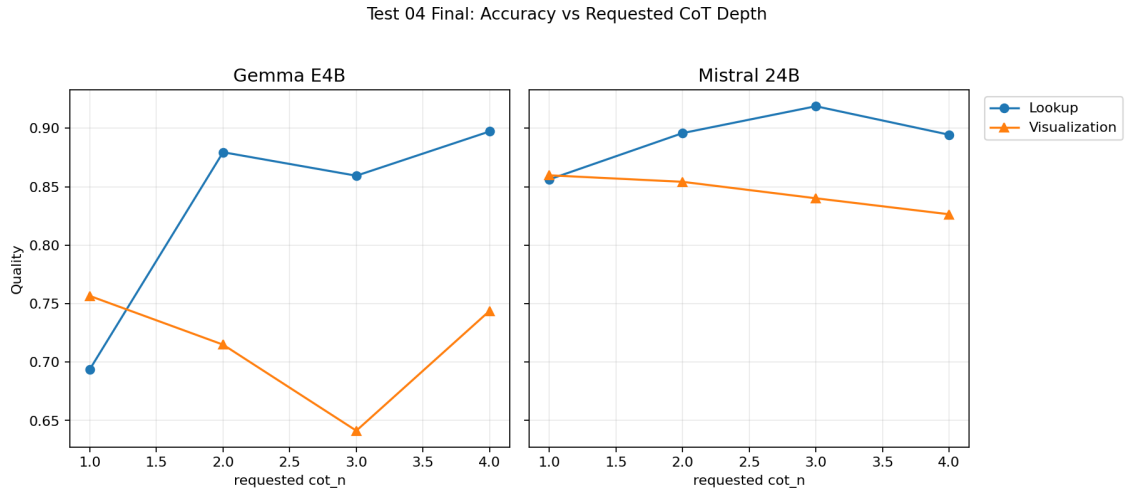


Figure 20: Test 04: prompt E2E quality as a function of requested *cot_n*.

The Gemma E4B lookup curve deserves particular attention: the *cot_n=1* to *cot_n=2* gain (+18.6 quality points) is the largest single-step improvement across all four tests for this model. It directly addresses the SQL-generation fragility diagnosed in Test 01 — iterative refinement corrects erroneous SQL through self-criticism, raising *csv_iou* from 0.757 at baseline to 0.954 at *cot_n=2*.

For visualization the pattern reverses: quality declines monotonically with depth for both models, reaching -0.115 at *cot_n=3* for *gemma4:e4b* and -0.033 at depth 4 for *mistral-small13.2:24b*. Self-criticism improves SQL queries but degrades chart code: the typical visualization failure is semantic — the Python runs fine, yet the chart does not match the requested plot (wrong grouping, wrong split, or subtly different semantics) — and the loop, lacking a reliable criterion for chart correctness, over-edits working code and introduces regressions. This is consistent with a broader caution for CoT: self-refinement helps only when the model can reliably verify the property being evaluated [13].

5.4.3 Quality-Energy Trade-off of CoT Configurations

Figure 21 maps all configurations into the $(\Delta \text{ energy}, \Delta \text{ quality})$ plane, where *quality* denotes the prompt E2E quality of Section 5.1.1.

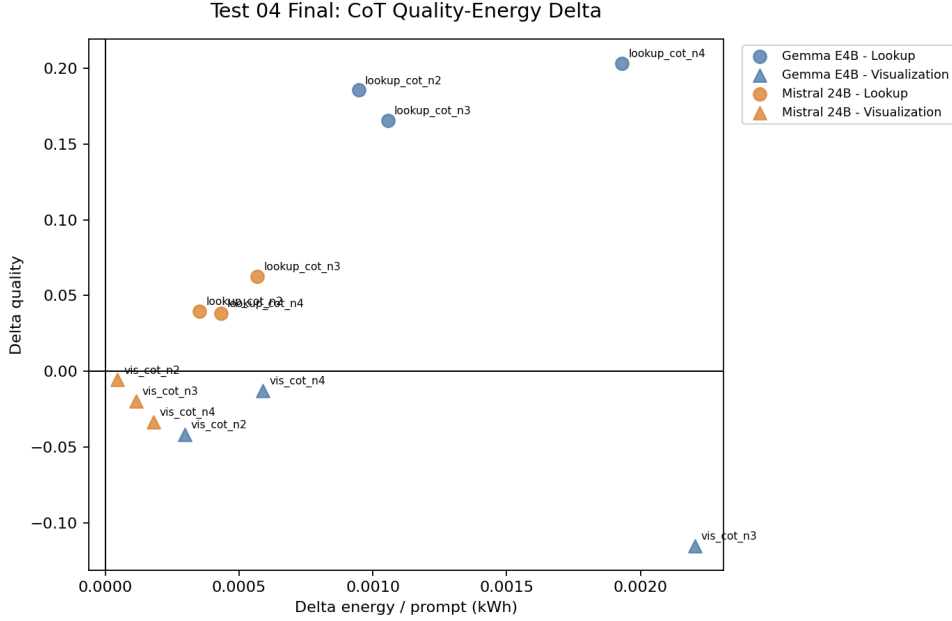


Figure 21: Test 04: prompt E2E quality delta vs. energy delta per CoT configuration.

The scatter divides cleanly into two clusters. Lookup CoT configurations land in the upper-right quadrant — quality and energy both rise, but the ratio is favourable: **gemma4:e4b** gains +0.203 at **cot_n=4** with moderate overhead, and **mistral-small13.2:24b** reaches +0.063 at depth 3 at much lower absolute cost. Visualization CoT configurations land in the lower-right quadrant: every extra iteration costs energy and degrades quality at once, making CoT a strictly dominated strategy for chart generation.

5.4.4 Direct Step-Metric Gain vs. Extra Energy

Figure 22 disaggregates the result to show direct step-metric change (**csv_iou** or **vis_score**) against the energy overhead.

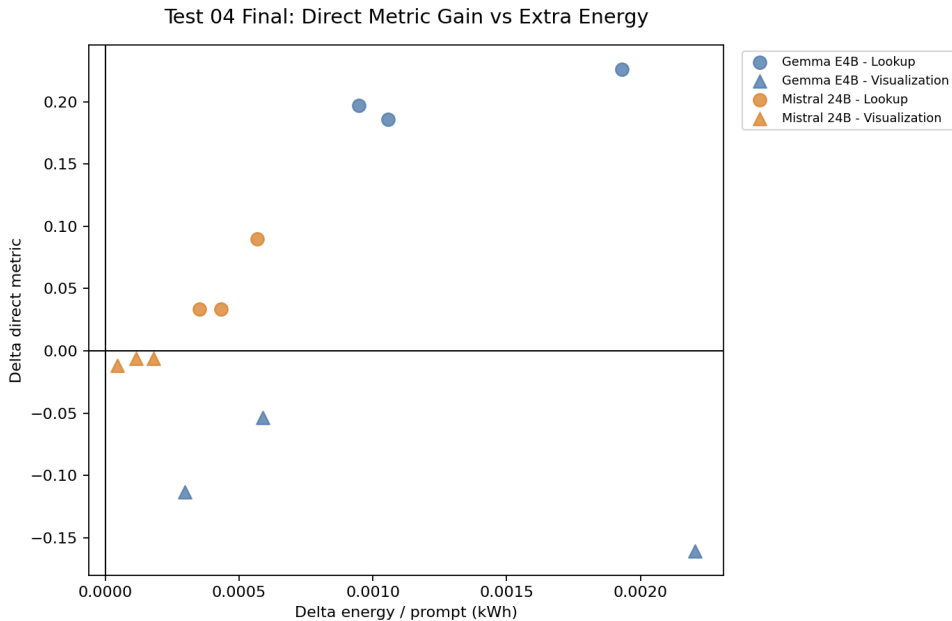


Figure 22: Test 04: direct step-metric delta vs. energy delta per CoT configuration.

The step-metric view reinforces the two-cluster picture of Figure 21 with greater resolution. The `csv_iou` gains for lookup CoT are substantial — up to +0.226 for `gemma4:e4b` at depth 4 (and already +0.197 at depth 2) — confirming that the overall quality improvement seen in the trade-off plot is genuinely driven by better data retrieval, not by indirect effects on other steps. Conversely, the `vis_score` losses for visualization CoT are equally unambiguous: the metric drops at every depth above 1 for both models, with no configuration managing to improve chart quality at any energy cost. The separation is sharper here than in the trade-off plot because the step metric isolates the causal step, removing the dilution from the two fixed steps.

5.4.5 Cost Multiplier

Figure 23 shows the time and energy multipliers relative to the `cot_n=1` baseline as a function of requested depth.

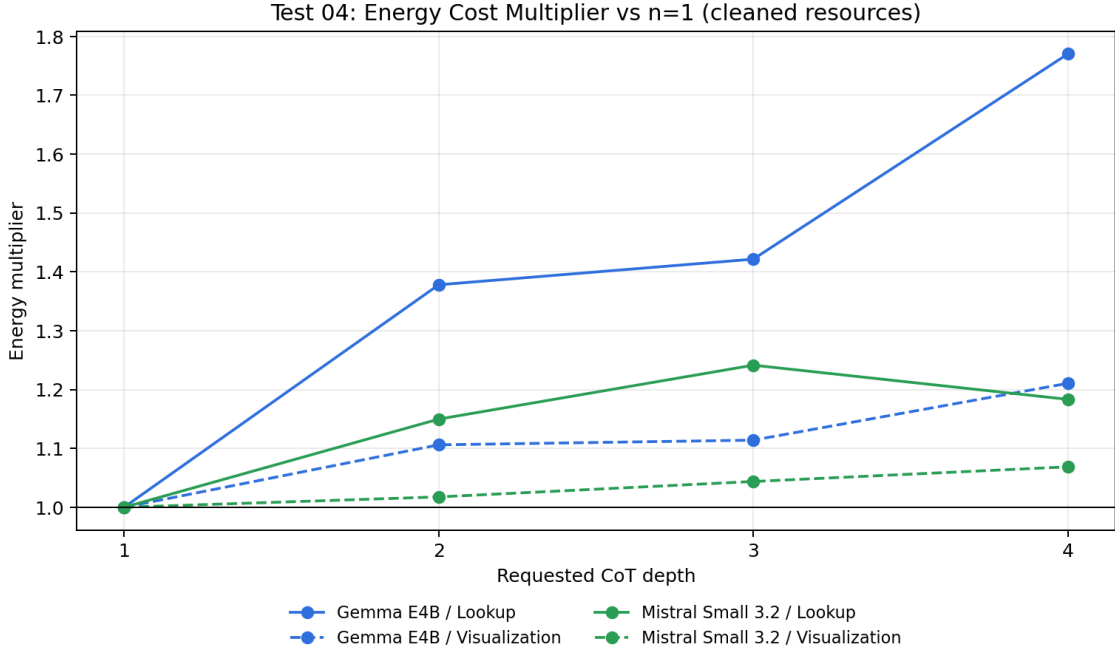


Figure 23: Test 04: time and energy multiplier vs. requested `cot_n` (baseline = $1\times$).

The cost multiplier grows incrementally with `cot_n`: each *executed* iteration adds a roughly constant amount of inference time and energy, so on a per-iteration basis the cost is predictable, with no unexpected spikes or diminishing overhead. The growth is, however, sub-linear in the *requested* depth rather than proportional to it. Because of the early-stop mechanism (Figure 24, Table 19) the executed depth saturates near 2, so requesting depth 3 or 4 only marginally increases the realised cost over depth 2: `gemma4:e4b` lookup energy rises from 0.00346 to 0.00444 kWh between requested depths 2 and 4 (+28%, not +100%), and `mistral-small3.2:24b` lookup energy at depth 4 (0.00279 kWh) is in fact below depth 3 (0.00293 kWh). CoT depth therefore remains a predictable compute lever, but the saturation of the executed depth makes the realised cost plateau rather than scale linearly with the requested value.

5.4.6 CoT Execution Depth and Early Stops

Figure 24 plots the mean *executed* (effective) CoT depth against the *requested* depth for each model-step pair; the corresponding step-execution and early-stop rates behind this behaviour are reported in Table 19.

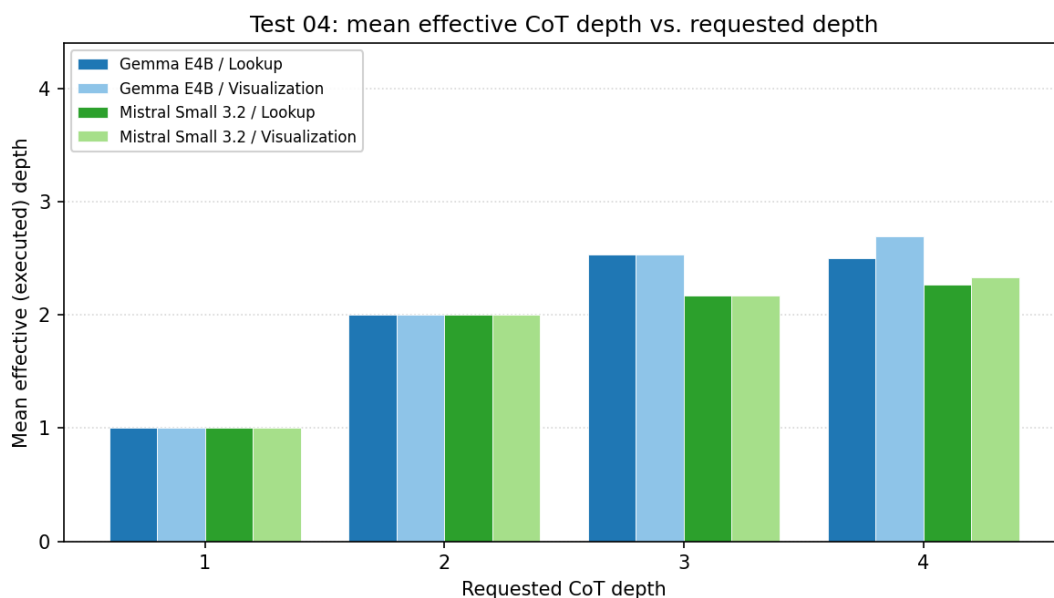


Figure 24: Test 04: mean effective (executed) CoT depth vs. requested `cot_n`.

As the figure shows, the executed depth tracks the request only up to `cot_n` 2 and then plateaus near 2–2.7, so the requested depth is largely an upper bound rather than the computation actually performed. This plateau has an important practical implication: for both models and the lookup step, `cot_n=3` and `cot_n=4` often reduce to `cot_n=2` in practice. The quality difference between `cot_n=2` and `cot_n=4` for Gemma E4B lookup (+0.185 vs. +0.203) is therefore partially a noise effect rather than a genuine depth benefit.

The early-stop and full-depth rates in the table are computed only over the runs where the CoT loop actually ran. This matters for visualization, whose step executes only for the prompts that request a chart: restricting to those runs measures how the refinement loop actually behaves, rather than diluting the rates with prompts that never trigger the step. Seen this way, high requested depths are even less attractive than they first appeared: for Gemma E4B lookup, `cot_n=4` stops early in 86.7 % of executed runs, and for Mistral lookup the early-stop rate is 90.0 %.

Table 19 summarises the execution diagnostics.

Model	Step	<code>cot_n</code>	Step exec. rate	Exec. mean	Early stop	Full depth
gemma4:e4b	Lookup	1	100.0 %	1.00	0.0 %	100.0 %
gemma4:e4b	Lookup	2	100.0 %	2.00	0.0 %	100.0 %
gemma4:e4b	Lookup	3	100.0 %	2.53	46.7 %	53.3 %
gemma4:e4b	Lookup	4	100.0 %	2.50	86.7 %	13.3 %
gemma4:e4b	Visualization	1	60.0 %	1.00	0.0 %	100.0 %
gemma4:e4b	Visualization	2	53.3 %	2.00	0.0 %	100.0 %
gemma4:e4b	Visualization	3	50.0 %	2.53	46.7 %	53.3 %
gemma4:e4b	Visualization	4	53.3 %	2.69	75.0 %	25.0 %
mistral-small3.2:24b	Lookup	1	100.0 %	1.00	0.0 %	100.0 %
mistral-small3.2:24b	Lookup	2	100.0 %	2.00	0.0 %	100.0 %
mistral-small3.2:24b	Lookup	3	100.0 %	2.17	83.3 %	16.7 %
mistral-small3.2:24b	Lookup	4	100.0 %	2.27	90.0 %	10.0 %
mistral-small3.2:24b	Visualization	1	60.0 %	1.00	0.0 %	100.0 %
mistral-small3.2:24b	Visualization	2	60.0 %	2.00	0.0 %	100.0 %
mistral-small3.2:24b	Visualization	3	60.0 %	2.17	83.3 %	16.7 %
mistral-small3.2:24b	Visualization	4	60.0 %	2.33	83.3 %	16.7 %

Table 19: Test 04: CoT execution diagnostics (rates computed among executed rows).

The visualization step execution rate sits at approximately 60 % because *decide_tool* only triggers this step when the prompt explicitly requests a chart; the remaining 40 % of prompts require only tabular data or a textual answer and never reach the visualization step. In some configurations the rate falls below 60 %: this occurs when the lookup setting produces incorrect or incomplete responses, causing the planner to terminate the pipeline before the visualization step is reached.

5.4.7 Accuracy by Prompt Difficulty

Figure 25 examines how CoT gains vary with prompt difficulty.

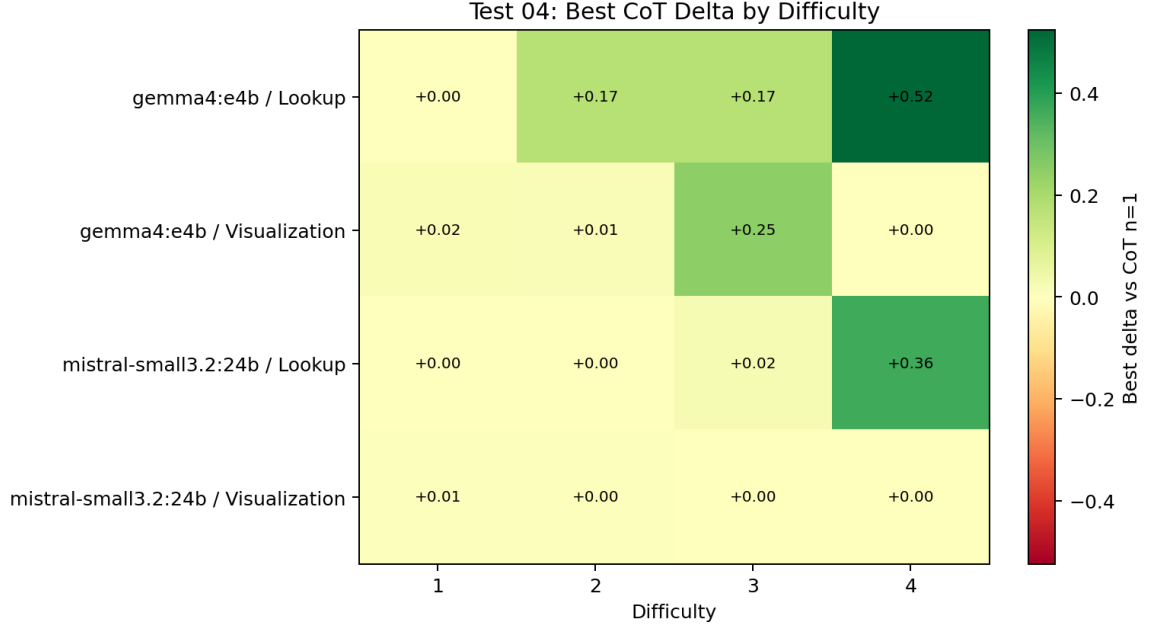


Figure 25: Test 04: best CoT prompt E2E quality delta by prompt difficulty.

CoT depth is most effective precisely where the baseline model is weakest. For **gemma4:e4b** lookup, difficulty-4 prompts benefit from a +0.524 prompt E2E quality gain at `cot_n=3` — the largest single-slice gain in the entire test — while difficulty-1 prompts show no gain from additional iterations. The same asymmetry holds for **mistral-small3.2:24b**: difficulty-4 lookup gains +0.365 at `cot_n=3`, while difficulties 1 through 3 show only marginal changes (+0.002 to +0.023). This pattern is consistent with the SQL-feedback mechanism of CoT: executable results can correct errors on complex nested queries, but add no signal when the baseline query is already correct. Visualization CoT shows the opposite trend: quality is flat or negative across all difficulty levels for both models, reinforcing the conclusion that visualization CoT should not be deployed.

5.4.8 Lookup Best-of-N Range Check (Test 04b)

Test 04b is a targeted follow-up run exclusively on **gemma4:e4b** that compares three families of lookup strategy on a common 10-prompt subset (3 repeats $\times$ 8 configs $\times$ 10 prompts = 240 scored rows): static temperature settings, Best-of-3 temperature ranges (BoN), and `lookup_cot_n2` as a reference.

Each **static temp** configuration fixes a single temperature for the lookup step ($n = 1$); the tested values 0.3, 0.6, and 0.8 are exactly the lower endpoints—the first candidate—of the wide, medium, and narrow ranges, so that a fixed temperature can be compared directly against the full range that starts from it. Each **BoN range** configuration instead generates three lookup candidates ($n = 3$), so that the pairwise no-GT selector faces a genuine multi-candidate ranking problem and then picks one of the three as the step output. The three candidate temperatures are neither identical nor drawn at random: they are placed at deterministic, evenly spaced points across the interval. Three ranges are tested—narrow [0.8, 1.0], medium [0.6, 1.0], and wide [0.3, 1.0]—giving, for example, candidate temperatures of 0.8/0.9/1.0, 0.6/0.8/1.0, and 0.3/0.65/1.0 respectively. Finally, `lookup_cot_n2` is included as a reference because it was the strongest lookup intervention identified in Test 04, providing a yardstick against which the Best-of-N strategies can be judged.

Config	Type	Quality	Δ quality	csv_iou	Completion	kWh/prompt
lookup_baseline	baseline	0.731	0.000	0.780	90.0 %	0.00676
lookup_temp_0.3	static temp	0.760	+0.029	0.822	83.3 %	0.00642
lookup_temp_0.6	static temp	0.768	+0.037	0.853	86.7 %	0.00649
lookup_temp_0.8	static temp	0.784	+0.053	0.802	83.3 %	0.00618
bon_temp_n3_narrow	BoN range	0.689	−0.042	0.735	83.3 %	0.00849
bon_temp_n3_medium	BoN range	0.785	+0.054	0.867	90.0 %	0.00893
bon_temp_n3_wide	BoN range	0.775	+0.044	0.834	86.7 %	0.00862
lookup_cot_n2	CoT depth	0.797	+0.067	0.871	96.7 %	0.00820

Table 20: Test 04b: aggregate results per lookup strategy, **gemma4:e4b**.

With three candidates the selector is not completely inert: the medium and wide ranges each choose a non-first candidate 4 times out of 30, while the narrow range does so only once. However, the selector still chooses the first candidate in 86.7–96.7% of runs, and the median selection margin is zero for all three ranges. This means that candidate diversity is present, but the local no-GT selector only weakly exploits it.

The best BoN setting is the medium range [0.6, 1.0], which reaches 0.785 prompt E2E quality and the strongest BoN csv_iou score (0.867). This confirms that range design matters: the narrow range is harmful, while the wide range adds diversity without matching the medium range. Yet the overall conclusion is still negative for deploying BoN as the default lookup repair. A single static `lookup_temp_0.8` setting is the best low-cost point, increasing quality by +0.053 while using slightly less measured energy than baseline. `lookup_cot_n2` is the best quality point (0.797 quality, 0.871 csv_iou) and dominates the strongest BoN range: it obtains higher quality, higher CSV similarity, higher completion, and lower energy than `bon_temp_n3_medium`. Thus Best-of-N remains useful as a diagnostic for lookup candidate diversity, but CoT with SQL execution feedback is the more effective compute-expansion strategy for **gemma4:e4b**.

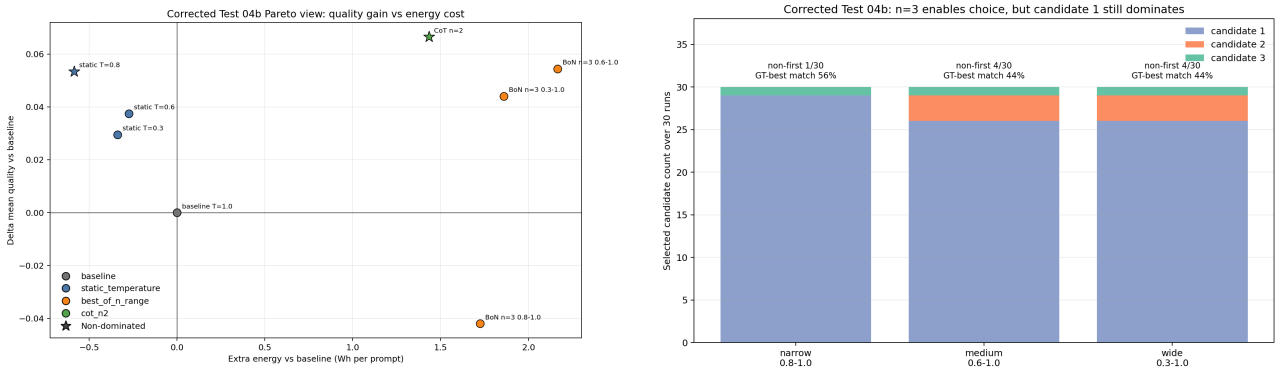


Figure 26: Test 04b Best-of-3 analysis: quality–energy (delta) view (left) and no-GT selector diagnostics (right).

As Figure 26 shows, the quality–energy (delta) view highlights static `lookup_temp_0.8` and `lookup_cot_n2` as the two most favourable lookup interventions: `lookup_temp_0.8` improves quality while slightly *lowering* energy relative to the baseline, whereas `lookup_cot_n2` reaches the highest quality but at a *higher* energy cost. The selector diagnostics, meanwhile, confirm that Best-of-N still chooses the first candidate in most runs.

5.4.9 Summary

1. **Lookup CoT is the most effective single intervention found for gemma4:e4b.** `cot_n=2` raises quality by +18.6 points and csv_iou by +19.7 points at a cost multiplier of 1.38 $\times$. This directly addresses the SQL-generation brittleness identified in Test 01.
2. **Lookup CoT also benefits Mistral, modestly.** `cot_n=3` yields the best Mistral result (+6.3 quality points, +9.0 csv_iou points), though the gain is three times smaller than for Gemma E4B.
3. **Visualization CoT is harmful for both models.** Every depth above 1 degrades `vis_score`, with Gemma E4B `cot_n=3` causing a −16.1 point direct metric regression. Visualization CoT should not be used.

4. **The effective CoT depth saturates near 2.** The early-stop mechanism makes `cot_n=3` and `cot_n=4` behave similarly to `cot_n=2` for the lookup step. Requesting depth 2 is therefore the most cost-efficient default for both models. For Gemma E4B, `cot_n=3` is useful as a hard-prompt repair setting because it fixes the difficulty-4 lookup slice, but `cot_n=4` should not be the default: it rarely reaches the fourth iteration and its extra aggregate gain is small relative to the additional cost and variance. For `mistral-small13.2:24b`, `cot_n=3` is the nominal quality peak (+0.063), but the gain over depth 2 is much smaller than the Gemma gain and most high-depth runs stop early. Depth 3 should therefore be reserved for maximum lookup quality, not for the energy-aware default.
5. **Lookup Best-of-N helps but is dominated by CoT (Test 04b).** On the `gemma4:e4b` lookup step, Best-of-3 candidate diversity does improve quality when the temperature range is well chosen — the medium range [0.6, 1.0] reaches +0.054 quality and the strongest BoN `csv_iou` (0.867), while the narrow range is harmful — but the local no-GT selector exploits this diversity only weakly, still picking the first candidate in 86.7–96.7% of runs. A single static `lookup_temp_0.8` is the best low-cost point (+0.053 quality below baseline energy), and `lookup_cot_n2` dominates the best BoN range on quality, CSV similarity, completion, and energy. Best-of-N is therefore a useful diagnostic for candidate diversity, but CoT with SQL-execution feedback remains the more effective compute-expansion strategy for lookup.

5.5. Test 05 - Final Pareto Confirmation

Test 05 evaluates the combined configurations described in Section 4.5. The candidate set includes every useful configuration selected for the final Pareto analysis, including configurations originally explored in Tests 02–04. These rows are not treated as a separate pool: they are part of Test 05 by design. Configurations whose measurements remain directly comparable are reused as they are, whereas a subset is re-run so that all candidates are measured in the same execution environment (the same server). The goal is to compare the best available candidate policies under identical execution conditions, rather than across measurements taken on different machines or runs.

5.5.1 Overall Ranking

Table 21 reports the highest-ranking configurations by strict prompt E2E quality. The strongest candidate is `mistral-small13.2:24b` with `lookup_cot_n2`: it reaches prompt quality 0.9395 with full completion while consuming only 0.00453 kWh per prompt. Gemma E4B remains extremely close: `lookup_cot_n2_temp_0p8_low_cost_tokens` reaches quality 0.9371 with full completion, but at higher energy. The top-ranked rows therefore show that the final recommendation is not simply a thinking-model preference: a larger non-thinking model with a repaired lookup step can be both cheaper and slightly more accurate on the evaluated full benchmark. Completion, however, is not uniform across the top-ranked Mistral rows: while the leading `lookup_cot_n2` candidate completes every expected output slot, several other strong Mistral configurations (`combined_cot_static`, `lookup_cot_n2_efficient_tokens`, and both `lookup_cot_n3` variants) reach only 98.1% completion, whereas every Gemma E4B candidate in the table completes all expected slots. Full completion is therefore one axis where the repaired Gemma E4B configurations remain more consistent than their Mistral counterparts.

Model	Config	Prompt E2E	Completion	kWh/prompt
<code>mistral-small13.2:24b</code>	<code>lookup_cot_n2</code>	0.9395	100.0%	0.00453
<code>gemma4:e4b</code>	<code>lookup_cot_n2_temp_0p8_low_cost_tokens</code>	0.9371	100.0%	0.00602
<code>gemma4:e4b</code>	<code>lookup_cot_n4_temp_0p8_low_cost_tokens</code>	0.9287	100.0%	0.00645
<code>mistral-small13.2:24b</code>	<code>combined_cot_static</code>	0.9206	98.1%	0.00449
<code>gemma4:e4b</code>	<code>lookup_cot_n4_temp_0p8</code>	0.9189	100.0%	0.00633
<code>gemma4:e4b</code>	<code>lookup_cot_n4_low_cost_tokens</code>	0.9176	100.0%	0.00636
<code>mistral-small13.2:24b</code>	<code>lookup_cot_n2_efficient_tokens</code>	0.9168	98.1%	0.00445
<code>gemma4:26b</code>	<code>vis_repeat_penalty_1p03_tokens12000</code>	0.9110	100.0%	0.01006
<code>gemma4:e4b</code>	<code>lookup_cot_n2_temp_0p8</code>	0.9109	100.0%	0.00609
<code>gemma4:e4b</code>	<code>lookup_cot_n4</code>	0.9046	100.0%	0.00640
<code>mistral-small13.2:24b</code>	<code>lookup_cot_n3_efficient_tokens</code>	0.9015	98.1%	0.00452
<code>mistral-small13.2:24b</code>	<code>lookup_cot_n3</code>	0.8983	98.1%	0.00455

Table 21: Test 05: strongest candidate configurations by prompt E2E quality.

The larger Gemma 26B MoE was evaluated with visualization-focused candidates that follow the Test 03 insight: the visualization step benefits from a larger token budget. The best 26B candidate, `vis_repeat_penalty_1p03_tokens12000`, raises the visualization score to 0.674 and reaches prompt quality 0.911 with full completion. This is a clear alignment improvement over the weaker visualization profile observed earlier, but it is not enough to enter the quality-energy frontier because the energy cost remains around 0.010 kWh per prompt.

5.5.2 Radar Summary

Figure 27 compares one runnable final representative per model: `lookup_cot_n2_temp_0p8_low_cost_tokens` for Gemma E4B, `vis_repeat_penalty_1p03_tokens12000` for Gemma 26B, and `lookup_cot_n2` for Mistral. The same configuration is used for all axes of a model, so the plot represents executable policies rather than per-axis oracle choices. Quality and reliability axes use the same fixed 0.4–1.0 normalization as Test 01, while energy uses the anchored inverse kWh scale. The radar axes follow the definitions introduced in the Test 01 radar summary.

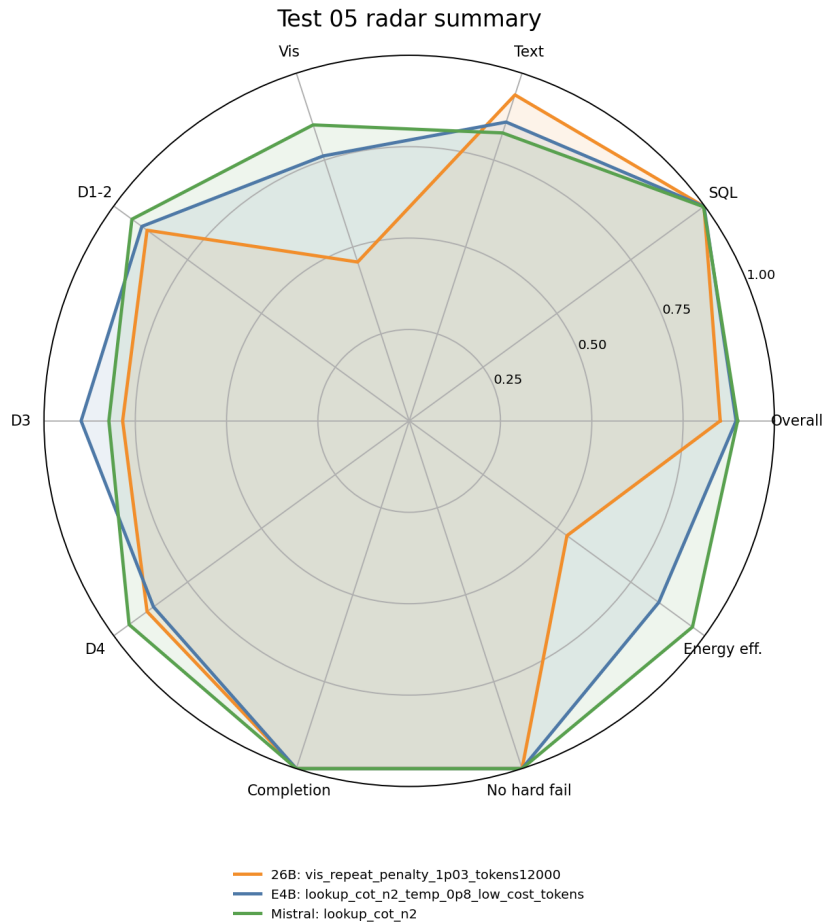


Figure 27: Test 05: radar summary of the selected final representative configuration for each model.

Model	Overall	SQL	Text	Vis	D1-2	D3	D4	Compl.	No hard fail	kWh	Energy eff.
<code>gemma4: e4b</code>	0.937	0.999	0.916	0.857	0.944	0.939	0.919	1.000	1.000	0.00602	0.845
<code>gemma4: 26b</code>	0.911	0.999	0.963	0.674	0.933	0.871	0.932	1.000	1.000	0.01006	0.534
<code>mistral-small13.2:24b</code>	0.940	0.998	0.897	0.911	0.962	0.893	0.969	1.000	1.000	0.00453	0.959

Table 22: Test 05: raw values used to build the final representative radar summary.

The radar view shows why Mistral is the final recommendation. Its representative configuration combines the strongest overall score, the best visualization score, difficulty-4 robustness, full completion, and the best energy efficiency. This full-completion result is specific to the representative `lookup_cot_n2` configuration, however, as noted in the ranking, several other strong Mistral candidates stop short of 100 % completion. Gemma E4B

remains very close on overall quality and is the strongest difficulty-3 profile, but it does not hold the high-quality endpoint, which Mistral occupies. Gemma 26B keeps excellent SQL, text, and reliability axes, and its visualization axis improves after the larger visualization-token setting, but the repair remains incomplete and its energy efficiency is lower.

5.5.3 Final Pareto Frontier

Figure 28 and Table 23 show the non-dominated configurations when maximising strict prompt E2E quality and minimising energy per prompt. This is a single unified frontier over the full Test 05 candidate set. The result is more decisive than the aggregate ranking: all non-dominated points belong to the Mistral family.

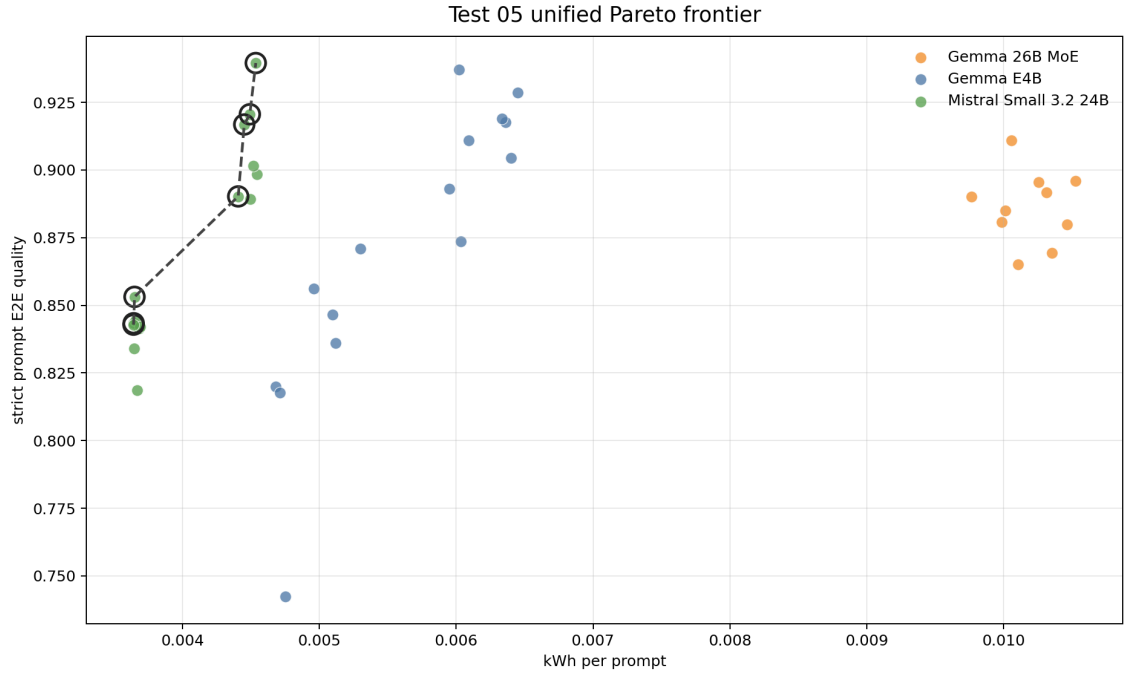


Figure 28: Test 05: unified Pareto frontier among the evaluated candidate configurations.

Model	Config	Prompt E2E	Completion	kWh/prompt
mistral-small3.2:24b	visualization_repeat_last_n_56	0.8429	92.6 %	0.00364
mistral-small3.2:24b	combined_static_best_efficient_tokens	0.8434	92.6 %	0.00364
mistral-small3.2:24b	max_tokens_quality	0.8531	92.6 %	0.00365
mistral-small3.2:24b	combined_cot_static_tokens	0.8902	96.3 %	0.00441
mistral-small3.2:24b	lookup_cot_n2_efficient_tokens	0.9168	98.1 %	0.00445
mistral-small3.2:24b	combined_cot_static	0.9206	98.1 %	0.00449
mistral-small3.2:24b	lookup_cot_n2	0.9395	100.0 %	0.00453

Table 23: Test 05: non-dominated candidate configurations for prompt E2E quality vs. energy.

Table 24 reports the per-step parameters that define these frontier configurations, listing only the settings that deviate from each model’s baseline.

Model	Config	Lookup	Analysis	Visualization
mistral-small13.2:24b	visualization_repeat_last_n_56	baseline	baseline	repeat_last_n=56
mistral-small13.2:24b	combined_static_best_efficient_tokens	repeat_penalty=1.3, max_tokens=3000	top_k=56, max_tokens=5000	repeat_last_n=56, max_tokens=5000
mistral-small13.2:24b	max_tokens_quality	baseline	max_tokens=5000	max_tokens=5000
mistral-small13.2:24b	combined_cot_static_tokens	cot_n=3, repeat_penalty=1.3, max_tokens=3000	top_k=56, max_tokens=5000	repeat_last_n=56, max_tokens=5000
mistral-small13.2:24b	lookup_cot_n2_efficient_tokens	cot_n=2, max_tokens=3000	max_tokens=5000	max_tokens=5000
mistral-small13.2:24b	combined_cot_static	cot_n=3, repeat_penalty=1.3	top_k=56	repeat_last_n=56
mistral-small13.2:24b	lookup_cot_n2	cot_n=2	baseline	baseline

Table 24: Test 05: per-step parameters of the non-dominated frontier configurations.

The frontier has a clear structure. Mistral occupies both the cheapest non-dominated region and the highest-quality endpoint. Static visualization and token-budget candidates define the low-energy frontier around 0.00364–0.00365 kWh per prompt. Adding lookup CoT then moves the frontier upward: `lookup_cot_n2_efficient_tokens` and `combined_cot_static` approach the 0.92 quality region, while `lookup_cot_n2` reaches the highest evaluated quality, full completion, and 0.00453 kWh per prompt. Gemma E4B is still practically close to this endpoint, but it is slightly lower in strict quality and more expensive. Gemma 26B improves with the visualization repair, but its roughly 0.010 kWh per-prompt cost keeps it dominated once energy is part of the decision criterion.

5.5.4 Pareto Frontier by Prompt Difficulty

The evaluated Pareto picture changes when the benchmark is split by difficulty. For each level the quality–energy frontier is recomputed over that difficulty’s benchmark rows, and one representative non-dominated configuration is selected from it. Table 25 reports the high-quality endpoint for each difficulty slice: the configuration that reaches the strongest strict prompt quality among the evaluated candidates while preserving full completion on that slice.

On difficulty 1 and 2 prompts, Mistral dominates the evaluated frontier: it is both cheap and highly accurate, reaching prompt quality close to 0.98 in the best configurations. On difficulty 3 the evaluated frontier becomes mixed: Mistral remains the most efficient, but Gemma E4B reaches the highest quality once lookup CoT and static lookup temperature are combined. Difficulty 4 is the most decisive slice: Mistral `lookup_cot_n2` reaches quality 0.969 with full completion at 0.00526 kWh per prompt, while the best Gemma E4B hard-prompt candidate reaches quality 0.919 at higher energy. This does not prove that Mistral will remain preferable for arbitrarily harder prompts; it only shows that, for the evaluated difficulty-4 benchmark slice, the lookup-CoT Mistral candidate is the best observed hard-prompt endpoint. Larger or more adversarial hard-prompt sets remain an important follow-up.

Difficulty	Best model/config	Role	Prompt E2E	Completion	kWh/prompt
1	mistral / baseline	high-quality endpoint	0.979	100.0 %	0.00333
2	mistral / lookup_cot_n3_efficient_tokens	high-quality endpoint	0.979	100.0 %	0.00481
3	gemma4:e4b / lookup_cot_n4_temp_0p8_low_cost_tokens	highest quality	0.946	100.0 %	0.00609
4	mistral / lookup_cot_n2	high-quality endpoint	0.969	100.0 %	0.00526

Table 25: Test 05: representative best configurations by prompt difficulty.

Table 26 reports the per-step parameters of these difficulty-specific best configurations, again listing only the settings that deviate from each model’s baseline.

Diff.	Model	Config	Lookup	Analysis	Visualization
1	mistral-small13.2:24b	baseline	baseline	baseline	baseline
2	mistral-small13.2:24b	lookup_cot_n3_efficient_tokens	cot_n=3, max_tokens=3000	max_tokens=5000	max_tokens=5000
3	gemma4:e4b	lookup_cot_n4_temp_0p8_low_cost_tokens	cot_n=4, temp=0.8	max_tokens=5000	max_tokens=4000
4	mistral-small13.2:24b	lookup_cot_n2	cot_n=2	baseline	baseline

Table 26: Test 05: per-step parameters of the representative best configurations by prompt difficulty.

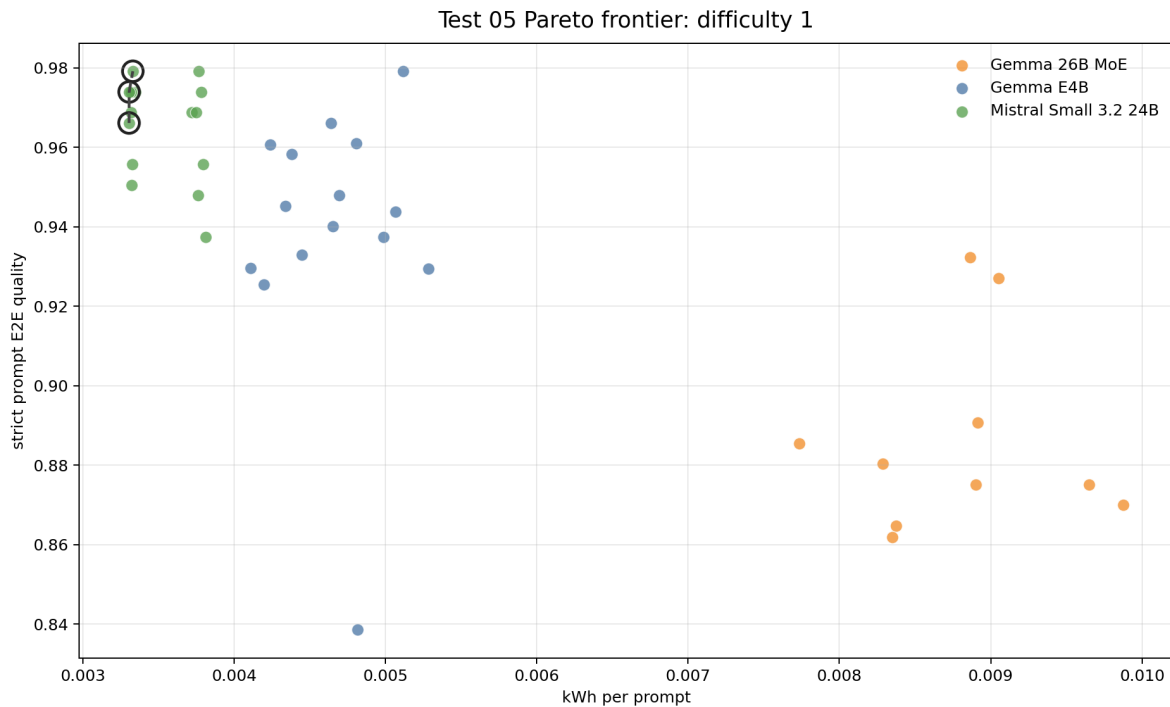


Figure 29: Test 05: difficulty 1 quality–energy Pareto frontier among the evaluated candidates.

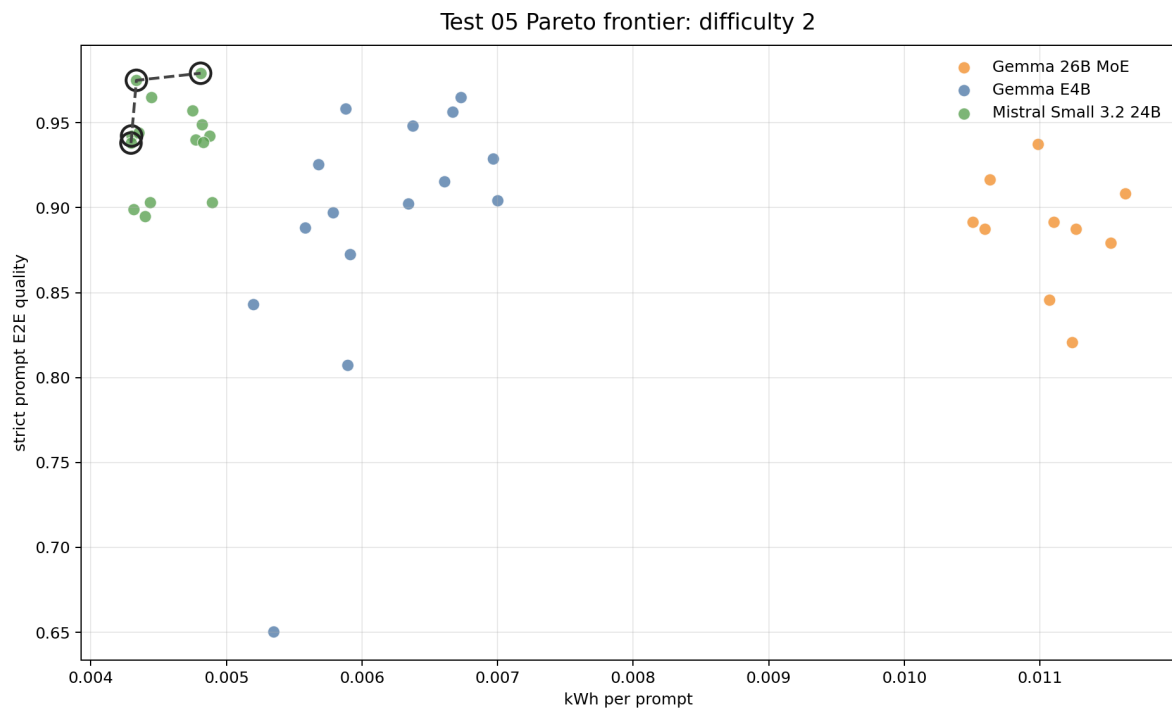


Figure 30: Test 05: difficulty 2 quality–energy Pareto frontier among the evaluated candidates.

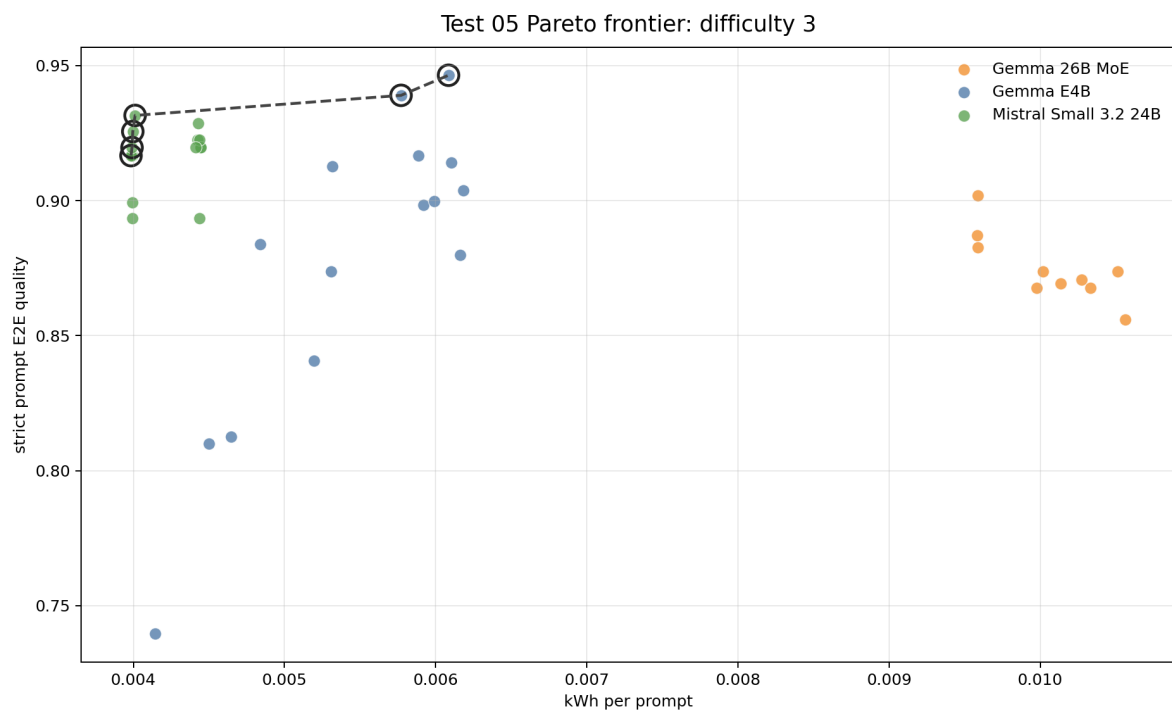


Figure 31: Test 05: difficulty 3 quality–energy Pareto frontier among the evaluated candidates.

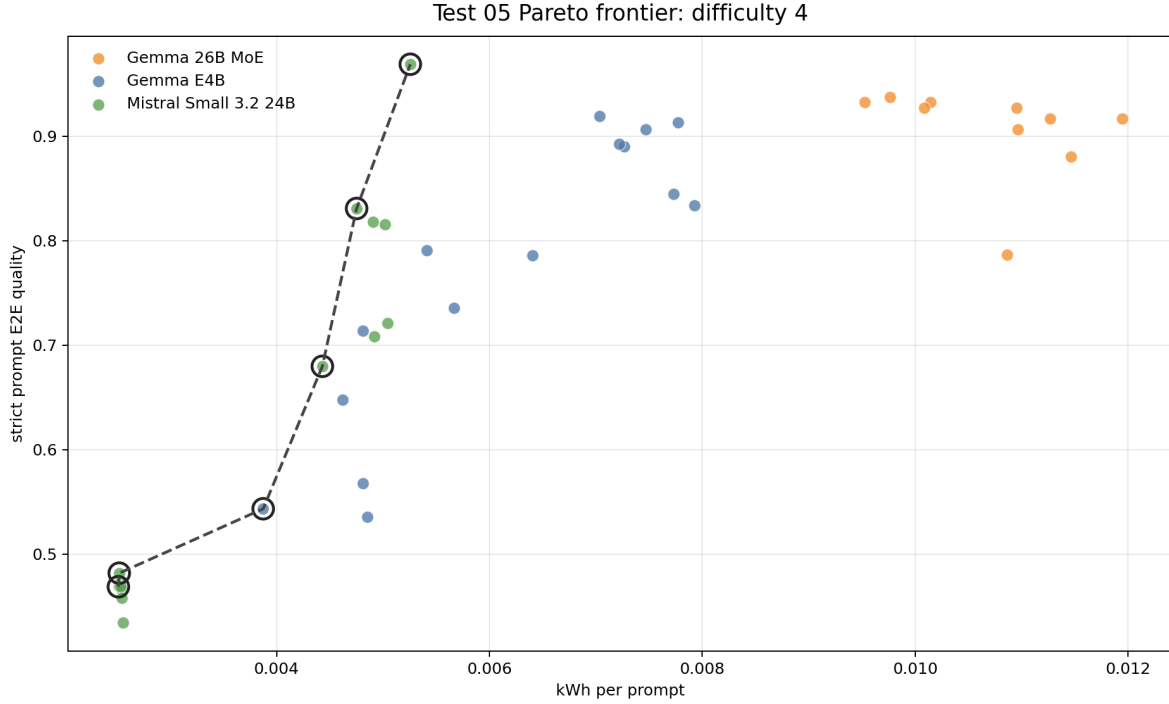


Figure 32: Test 05: difficulty 4 quality–energy Pareto frontier among the evaluated candidates.

The prompt-difficulty view is essential for interpreting the final Pareto front. A single aggregate score would suggest that Mistral and Gemma E4B are close competitors with different energy profiles. The difficulty split shows why the final Mistral recommendation is stronger than a purely low-cost argument: Mistral leads on difficulties 1, 2, and 4, while Gemma E4B remains the highest-quality endpoint on difficulty 3. The difficulty-4 result should be read with the scope of the benchmark in mind. It shows that Mistral `lookup_cot_n2` handles the evaluated hard prompts very well, but it does not eliminate the possibility that larger or more complex hard-prompt sets could again favor the thinking-model family. Figures 29–32 make the same transition visible as a sequence of frontiers: difficulties 1 and 2 are Mistral-led evaluated frontiers, difficulty 3 introduces Gemma E4B as the high-quality endpoint, and difficulty 4 returns to a Mistral endpoint for the evaluated hard-prompt slice.

5.5.5 Summary

1. **Test 05 is the final confirmation stage.** It compares the strongest end-to-end policies under identical execution conditions to determine the final quality–energy recommendation.
2. **The final Pareto front is not a pure model-size ranking.** The larger Gemma 26B MoE improves with visualization-focused tuning but remains dominated in the evaluated quality–energy frontier.
3. **Mistral defines the evaluated frontier.** Mistral configurations occupy both the low-energy region and the highest-quality endpoint once lookup CoT is added.
4. **Gemma E4B remains a close high-quality alternative.** The final E4B candidate combines lookup CoT, static lookup temperature, and lower downstream token caps, reaching quality only slightly below the best Mistral endpoint. Full completion is in fact one axis where E4B stays more consistent than several Mistral candidates, which leave a small fraction of expected outputs incomplete.
5. **Prompt difficulty determines which Pareto point is best.** Mistral is the natural choice for the evaluated easy, medium, and difficulty-4 prompts; Gemma E4B remains the best evaluated endpoint on difficulty 3, and broader hard-prompt testing is needed before generalising this ordering to more complex workloads.

6. Conclusions

This thesis investigated the accuracy–energy trade-off of a multi-step LLM data agent for sales-data analysis. This chapter summarises the findings with respect to the five research questions introduced in Section 1. The central result is that model choice alone is not sufficient to explain performance: the most important effects emerge from the interaction between model family, agent step, prompt difficulty, and inference-time parameters. The baseline comparison shows that `mistral-small3.2:24b` is the most efficient untuned model. It achieves

the best aggregate strict quality at the lowest energy cost, especially on easy and medium prompts. However, the prompt-difficulty analysis reveals a limitation that is hidden by the aggregate score: Mistral collapses on the hardest SQL aggregation prompts, where `gemma4:26b` performs best at baseline and `gemma4:e4b` can be repaired through targeted lookup interventions. This confirms the importance of reporting results by prompt difficulty rather than only as a single mean score.

The per-agent sensitivity experiments show that hyperparameter tuning must be step-specific. Gemma 26B is highly tunable but fragile: its visualization agent is weak at baseline and responds strongly to both static sampling parameters and larger token caps. In particular, raising the Gemma 26B visualization cap from 7000 to 10000 tokens improves `vis_score` with negligible energy cost, and the curve does not yet show a clear plateau. Gemma E4B behaves differently: adding too much token headroom can reduce quality, especially in lookup and visualization, so it requires bounded caps rather than simply larger ones. Mistral by contrast, is comparatively insensitive to token-cap changes and obtains only modest positive gains from static sampling adjustments, although poorly chosen settings can still cause noticeable degradation. This robustness makes it a safe baseline, but also means static hyperparameter search offers it little room for improvement.

The token-budget experiment therefore supports a practical rule: increasing `max_tokens` is cheap when the model does not actually consume the extra budget, so generous caps are useful for model-step pairs that benefit from additional headroom. They should not be treated as universally better, because some thinking-model agents become more verbose and less reliable when given too much room.

The compute-expansion experiment shows that CoT refinement is useful only when the feedback signal matches the failure mode. Lookup CoT is effective because SQL errors are executable and can be fed back to the model. For Gemma E4B, lookup CoT is the strongest single intervention in the thesis; for Mistral it also helps, but the gain is smaller. Visualization CoT is harmful for both models. The generated chart often fails semantically rather than syntactically, so code-execution feedback does not reveal whether the plot matches the user’s requested grouping, split, or chart semantics. Additional CoT iterations therefore tend to over-edit the output and degrade `vis_score`. The early-stop diagnostics reinforce this point: requested depths of 3 or 4 usually execute only around two iterations, so high `cot_n` should be interpreted as an upper bound rather than as the actual compute spent. Depth 2 is the safest default, while depth 3 can be reserved for hard lookup prompts when maximum accuracy is more important than energy. A targeted follow-up on the Gemma E4B lookup step also evaluated a best-of-N sampling strategy for SQL generation using three temperature-range candidates. This follow-up shows that candidate diversity can help, with the medium range [0.6, 1.0] outperforming the narrow and wide ranges, but the no-GT selector still chose the first candidate in most runs. A single well-chosen lookup temperature was therefore the best low-cost option, while CoT remained the better quality intervention because it corrects SQL using execution feedback rather than relying only on sampling diversity.

The final Pareto confirmation combines the best insights from the earlier tests. Among the evaluated candidate configurations, the non-dominated frontier is defined by Mistral candidates. The best final Mistral configuration combines lookup CoT with the model’s efficient generation profile, reaching the highest strict prompt quality with full completion at low energy. Gemma E4B remains a very close high-quality alternative, especially on difficulty 3 prompts, but it is slightly more expensive and slightly lower in strict aggregate quality. Gemma 26B improves when the visualization step is given more token headroom, confirming the Test 03 signal that its plotting agent was under-supported, but the improvement only aligns its performance partially and its energy cost keeps it off the final evaluated Pareto frontier. The fact that Mistral dominates the evaluated difficulty-4 slice should not be generalised to all possible hard analytical prompts: larger and more adversarial hard-prompt suites may still expose different trends and remain an important next step.

Overall, the thesis supports three practical recommendations for LLM data agents:

1. **Tune parameters per agent step rather than globally.**
2. **Allocate compute according to the failure mode:** token headroom for steps that are genuinely truncated, CoT for SQL lookup where feedback is verifiable, and conservative generation for visualization when the main error is semantic mismatch.
3. **Report accuracy and energy by prompt difficulty,** because the best model for simple tasks is not necessarily the best model for hard analytical questions.

References

- [1] LangChain AI. LangGraph: Build stateful, multi-actor applications with LLMs. <https://github.com/langchain-ai/langgraph>, 2024. Accessed: April 2025.
- [2] CodeCarbon Contributors. CodeCarbon: Track carbon emissions from machine learning. <https://github.com/mlco2/codecarbon>. Accessed: January 2026.
- [3] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct:

- Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.
- [4] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, 2018.
 - [5] Jinyang Li, Binyuan Hui, Ge Qu, Jiayi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can LLM already serve as a database interface? A BIG bench for large-scale database grounded text-to-SQLs. In *Advances in Neural Information Processing Systems*, volume 36, 2023.
 - [6] Mohammadreza Pourreza and Davood Rafiei. DIN-SQL: Decomposed in-context learning of text-to-SQL with self-correction. In *Advances in Neural Information Processing Systems*, volume 36, 2023.
 - [7] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarczyk, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations*, 2017.
 - [8] Google DeepMind. Gemma 4 technical report. <https://ai.google.dev/gemma>, 2025. Accessed: May 2025.
 - [9] Mistral AI. Mistral small 3.2. <https://mistral.ai/news/mistral-small-3-2>, 2025. Accessed: May 2025.
 - [10] OpenTelemetry Authors. OpenTelemetry: High-quality, ubiquitous, and portable telemetry to make robust, reliable software. <https://opentelemetry.io>, 2024. Accessed: April 2025.
 - [11] Arize AI. Phoenix: Open-source LLM observability. <https://phoenix.arize.com>, 2024. Accessed: April 2025.
 - [12] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc., 2022.
 - [13] Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations*, 2024.
 - [14] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations*, 2023.
 - [15] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-bench and chatbot arena. In *Advances in Neural Information Processing Systems*, volume 36. Curran Associates, Inc., 2023.
 - [16] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3645–3650, 2019.
 - [17] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. Green AI. *Communications of the ACM*, 63(12):54–63, 2020.

A. Benchmark Prompt Set

This appendix lists all 20 benchmark prompts used to evaluate the Sales Agent, grouped by difficulty level. Each entry reports: the natural language question given as input to the agent; the visualization directive (if any); the reference SQL query used to generate the ground-truth result set; and the reference analysis text used to evaluate the agent’s textual output.

Difficulty Level 1 — Single-table aggregation

Prompt 1. Difficulty: 1 Visualization goal: none.

“Return the top 5 stores by total revenue, identified by store number.”

Reference SQL:

```
SELECT Store_Number,
       SUM(Total_Sale_Value) AS total_revenue
FROM sales
GROUP BY Store_Number
ORDER BY total_revenue DESC
LIMIT 5
```

Reference analysis: Store 2970: \$836,341.33; Store 3300: \$619,660.17; Store 1320: \$592,832.07; Store 1650: \$580,443.01; Store 1210: \$508,393.77.

Prompt 2. Difficulty: 1 Visualization goal: bar chart comparing total revenue for promo and non-promo sales in 2023.

“Show promo vs non-promo revenue in 2023 as a bar chart.”

Reference SQL:

```
SELECT CASE WHEN On_Promo = 1 THEN 'Promo' ELSE 'Non-Promo' END
       AS promo_status,
       ROUND(SUM(Total_Sale_Value), 2) AS total_revenue
FROM sales
WHERE EXTRACT(YEAR FROM CAST(Sold_Date AS DATE)) = 2023
GROUP BY On_Promo
ORDER BY On_Promo ASC
```

Reference analysis: Non-promo sales generated \$5,316,412.07 in 2023, vs \$99,706.86 for promo sales ($\approx 1.8\%$ of total revenue).

Prompt 3. Difficulty: 1 Visualization goal: none.

“Return the top 10 product class codes by total units sold in 2023.”

Reference SQL:

```
SELECT Product_Class_Code,
       SUM(Qty_Sold) AS total_units
FROM sales
WHERE EXTRACT(YEAR FROM CAST(Sold_Date AS DATE)) = 2023
GROUP BY Product_Class_Code
ORDER BY total_units DESC, Product_Class_Code ASC
LIMIT 10
```

Reference analysis: Product class 22850 leads with 89,229 units sold in 2023, followed by 22975 (60,425) and 22800 (57,485). The top 10 ranges down to class 24375 with 11,358 units.

Prompt 4. Difficulty: 1 Visualization goal: bar chart of the top 8 stores ranked by total units sold in 2022.

“Show the top 8 stores by total units sold in 2022 as a bar chart, identifying each store by its store number.”

Reference SQL:

```
SELECT Store_Number,
       SUM(Qty_Sold) AS total_units
FROM sales
WHERE EXTRACT(YEAR FROM CAST(Sold_Date AS DATE)) = 2022
GROUP BY Store_Number
ORDER BY total_units DESC, Store_Number ASC
LIMIT 8
```

Reference analysis: Store 2970 leads with 23,580 units; the eighth-ranked store (1540) sold 14,517 units.

Difficulty Level 2 — Time bucketing and multi-aggregate

Prompt 5. Difficulty: 2 Visualization goal: line chart with date on the x -axis and daily sales value on the y -axis.

“Show me the sales in Nov 2021.”

Reference SQL:

```
SELECT Sold_Date,
       SUM(Total_Sale_Value) AS Total_Sales_Value
FROM sales
WHERE CAST(Sold_Date AS VARCHAR) LIKE '2021-11%'
GROUP BY Sold_Date
ORDER BY Sold_Date
```

Reference analysis: Sales in November 2021 show daily values ranging from \$11,030.39 to \$52,004.60, with a total of \$499,984.43 across 29 days.

Prompt 6. Difficulty: 2 Visualization goal: line chart showing monthly revenue trends for 2023.

“Show the 12 months of 2023 with total revenue and total units sold.”

Reference SQL:

```
SELECT DATE_TRUNC('month', Sold_Date) AS month_start,
       SUM(Total_Sale_Value)          AS total_revenue,
       SUM(Qty_Sold)                  AS total_units
FROM sales
WHERE EXTRACT(YEAR FROM Sold_Date) = 2023
GROUP BY DATE_TRUNC('month', Sold_Date)
ORDER BY month_start ASC
```

Reference analysis: The 12 months of 2023 show total revenue ranging from \$353,284.21 (July) to \$874,263.67 (December), with total units from 27,932 to 60,941.

Prompt 7. Difficulty: 2 Visualization goal: none.

“Return the top 10 sales days in 2023 with total revenue and total units sold.”

Reference SQL:

```
SELECT CAST(Sold_Date AS DATE)          AS Sold_Date,
       ROUND(SUM(Total_Sale_Value), 2)   AS total_revenue,
       SUM(Qty_Sold)                     AS total_units
FROM sales
WHERE EXTRACT(YEAR FROM CAST(Sold_Date AS DATE)) = 2023
GROUP BY CAST(Sold_Date AS DATE)
ORDER BY total_revenue DESC, Sold_Date ASC
LIMIT 10
```

Reference analysis: The highest-revenue day is 2023-12-22 with \$62,640.70 and 4,059 units; the tenth-ranked day is 2023-12-16 with \$34,630.77 and 2,435 units.

Prompt 8. Difficulty: 2 Visualization goal: line chart showing monthly revenue for product class code 22975 in 2022.

“Show monthly revenue for product class code 22975 in 2022 as a line chart.”

Reference SQL:

```
SELECT DATE_TRUNC('month', CAST(Sold_Date AS DATE)) AS month_start,
       ROUND(SUM(Total_Sale_Value), 2)              AS total_revenue
FROM sales
WHERE EXTRACT(YEAR FROM CAST(Sold_Date AS DATE)) = 2022
      AND Product_Class_Code = 22975
GROUP BY month_start
ORDER BY month_start ASC
```

Reference analysis: Monthly revenue for product class 22975 in 2022 ranges from \$73,802.56 (July) to \$122,319.24 (December), with a mid-year trough and a strong year-end peak.

Prompt 9. Difficulty: 2 Visualization goal: grouped bar chart showing monthly revenue in 2023 split between promo and non-promo sales.

“Show monthly revenue split by promo flag for 2023 as a grouped bar chart.”

Reference SQL:

```
SELECT DATE_TRUNC('month', CAST(Sold_Date AS DATE)) AS month_start,
       ROUND(SUM(CASE WHEN On_Promo = 1
                      THEN Total_Sale_Value ELSE 0 END), 2) AS promo_revenue,
       ROUND(SUM(CASE WHEN On_Promo = 0
                      THEN Total_Sale_Value ELSE 0 END), 2) AS non_promo_revenue
FROM sales
WHERE EXTRACT(YEAR FROM CAST(Sold_Date AS DATE)) = 2023
GROUP BY month_start
ORDER BY month_start ASC
```

Reference analysis: Monthly non-promo revenue peaks in December at \$835,289.24, with promo revenue highest in December (\$38,974.43) and November (\$35,099.91). February, March, June, and July had no promo sales.

Difficulty Level 3 — Joins and business-dimension grouping

Prompt 10. Difficulty: 3 Visualization goal: bar chart showing total revenue per brand for 2023, ordered from highest to lowest.

“Show total revenue by product brand for 2023 as a bar chart.”

Reference SQL:

```
SELECT p.brand,
       ROUND(SUM(s.Total_Sale_Value), 2) AS total_revenue
FROM sales s
JOIN products p ON s.SKU_Coded = p.SKU_Coded
WHERE YEAR(CAST(s.Sold_Date AS DATE)) = 2023
GROUP BY p.brand
ORDER BY total_revenue DESC
```

Reference analysis: In 2023, GoldenCrust leads with \$1,237,536.46, followed by CreamyFarm (\$1,116,963.60), CleanHome (\$985,063.32), CoolSip (\$942,234.79), and GlowUp (\$682,053.43). CrunchTime has the lowest revenue at \$452,267.33; total across all brands is \$5,416,118.93.

Prompt 11. Difficulty: 3 Visualization goal: bar chart of store cities with total 2022 revenue above \$200,000, ordered from highest to lowest.

“Show store cities where total revenue in 2022 exceeded 200,000 as a bar chart.”

Reference SQL:

```
SELECT st.city,
       ROUND(SUM(s.Total_Sale_Value), 2) AS total_revenue
FROM sales s
JOIN stores st ON s.Store_Number = st.Store_Number
WHERE EXTRACT(YEAR FROM CAST(s.Sold_Date AS DATE)) = 2022
GROUP BY st.city
HAVING SUM(s.Total_Sale_Value) > 200000
ORDER BY total_revenue DESC
```

Reference analysis: Seven cities exceeded \$200,000 in total revenue in 2022. Washington leads at \$342,948.68, followed by Memphis (\$255,559.74) and Austin (\$244,685.18). The remaining four — Dallas, Las Vegas, San Diego, and San Antonio — range between \$202,010.28 and \$223,011.84.

Prompt 12. Difficulty: 3 Visualization goal: bar chart showing total 2023 revenue by store type.

“Show total revenue by store type for 2023 as a bar chart.”

Reference SQL:

```
SELECT st.store_type,
       ROUND(SUM(s.Total_Sale_Value), 2) AS total_revenue
FROM sales s
JOIN stores st ON s.Store_Number = st.Store_Number
WHERE EXTRACT(YEAR FROM CAST(s.Sold_Date AS DATE)) = 2023
GROUP BY st.store_type
ORDER BY total_revenue DESC, st.store_type ASC
```

Reference analysis: Supermarket leads with \$1,938,262.70, followed by Discount (\$1,574,739.14), Hypermarket (\$1,188,793.49), and Convenience (\$714,323.60).

Prompt 13. Difficulty: 3 Visualization goal: bar chart showing total units sold by product category in 2023.

“Show total units sold by product category for 2023 as a bar chart.”

Reference SQL:

```
SELECT p.category,
       SUM(s.Qty_Sold) AS total_units
FROM sales s
JOIN products p ON s.SKU_Coded = p.SKU_Coded
WHERE EXTRACT(YEAR FROM CAST(s.Sold_Date AS DATE)) = 2023
GROUP BY p.category
ORDER BY total_units DESC, p.category ASC
```

Reference analysis: Dairy leads with 141,284 units, followed by Bakery (87,114), Beverages (71,131), Household (57,576), Personal Care (38,045), and Snacks (24,817).

Prompt 14. Difficulty: 3 Visualization goal: none.

“Return the top 3 brands by realized average selling price in 2023, where average price is computed as total revenue divided by total units sold, along with total units sold per brand.”

Reference SQL:

```
SELECT p.brand,
       ROUND(SUM(s.Total_Sale_Value) /
             NULLIF(SUM(s.Qty_Sold), 0), 4) AS avg_realized_unit_price,
       SUM(s.Qty_Sold) AS total_units
FROM sales s
JOIN products p ON s.SKU_Coded = p.SKU_Coded
WHERE EXTRACT(YEAR FROM CAST(s.Sold_Date AS DATE)) = 2023
GROUP BY p.brand
HAVING SUM(s.Qty_Sold) > 0
ORDER BY avg_realized_unit_price DESC, total_units DESC, p.brand ASC
LIMIT 3
```

Reference analysis: The top 3 brands by average realized price are CrunchTime (18.2241), GlowUp (17.9275), and CleanHome (17.1089).

Prompt 15. Difficulty: 3 Visualization goal: none.

“Return the top 8 cities by total revenue in 2023.”

Reference SQL:

```
SELECT st.city,
       ROUND(SUM(s.Total_Sale_Value), 2) AS total_revenue
FROM sales s
JOIN stores st ON s.Store_Number = st.Store_Number
WHERE EXTRACT(YEAR FROM CAST(s.Sold_Date AS DATE)) = 2023
GROUP BY st.city
ORDER BY total_revenue DESC, st.city ASC
LIMIT 8
```

Reference analysis: Washington leads with \$331,128.83; the eighth-ranked city (Seattle) generated \$181,115.33.

Prompt 16. Difficulty: 3 Visualization goal: grouped bar chart comparing 2023 revenue by region for organic and non-organic products.

“Show 2023 revenue by region comparing Organic and Non-Organic products as a grouped bar chart.”

Reference SQL:

```
SELECT st.region,
       CASE WHEN p.is_organic THEN 'Organic'
            ELSE 'Non-Organic' END   AS is_organic,
       ROUND(SUM(s.Total_Sale_Value), 2) AS total_revenue
FROM sales s
JOIN stores st ON s.Store_Number = st.Store_Number
JOIN products p ON s.SKU_Coded = p.SKU_Coded
WHERE EXTRACT(YEAR FROM CAST(s.Sold_Date AS DATE)) = 2023
GROUP BY st.region, is_organic
ORDER BY st.region ASC, is_organic ASC
```

Reference analysis: West leads with \$1,701,793.25 in Non-Organic revenue in 2023. Non-Organic revenue outpaces Organic by roughly 5:1 across all four regions.

Difficulty Level 4 — Multi-step aggregation and year-over-year

Prompt 17. Difficulty: 4 Visualization goal: grouped bar chart comparing average monthly revenue per region between 2022 and 2023.

“Compare average monthly revenue between store regions for 2022 and 2023.”

Reference SQL:

```
SELECT st.region,
       s.yr AS year,
       ROUND(AVG(s.monthly_rev), 2) AS avg_monthly_revenue
FROM (
  SELECT Store_Number,
         YEAR(CAST(Sold_Date AS DATE)) AS yr,
         DATE_TRUNC('month', CAST(Sold_Date AS DATE)) AS month,
         SUM(Total_Sale_Value) AS monthly_rev
  FROM sales
  WHERE YEAR(CAST(Sold_Date AS DATE)) IN (2022, 2023)
  GROUP BY Store_Number, yr, month
) s
JOIN stores st ON s.Store_Number = st.Store_Number
GROUP BY st.region, s.yr
ORDER BY st.region, s.yr
```

Reference analysis: Northeast has the highest average monthly revenue in both years (\$17,703.52 in 2022, \$14,970.11 in 2023), though it declined year-over-year. Midwest is lowest but shows growth from \$10,522.09 to \$10,821.44. South and West both show a slight decline from 2022 to 2023.

Prompt 18. Difficulty: 4 Visualization goal: grouped bar chart comparing average monthly revenue by store type between 2022 and 2023.

“Compare average monthly revenue between store types for 2022 and 2023.”

Reference SQL:

```
SELECT st.store_type,
       monthly.yr AS year,
       ROUND(AVG(monthly.monthly_revenue), 2) AS avg_monthly_revenue
FROM (
  SELECT s.Store_Number,
         EXTRACT(YEAR FROM CAST(s.Sold_Date AS DATE)) AS yr,
         DATE_TRUNC('month', CAST(s.Sold_Date AS DATE)) AS month_start,
         SUM(s.Total_Sale_Value) AS monthly_revenue
  FROM sales s

```

```

WHERE EXTRACT(YEAR FROM CAST(s.Sold_Date AS DATE)) IN (2022, 2023)
GROUP BY s.Store_Number, yr, month_start
) monthly
JOIN stores st ON monthly.Store_Number = st.Store_Number
GROUP BY st.store_type, monthly.yr
ORDER BY st.store_type ASC, monthly.yr ASC

```

Reference analysis: Convenience has the highest average monthly revenue in 2022 (\$15,914.67) and remains first in 2023 (\$14,881.74). All store types show a slight decline from 2022 to 2023.

Prompt 19. Difficulty: 4 Visualization goal: bar chart showing the promo revenue share as a percentage of total revenue for each product category in 2023.

“Show promo revenue share by product category for 2023 as a bar chart, expressing the share as a percentage of total revenue.”

Reference SQL:

```

SELECT p.category,
       ROUND(SUM(CASE WHEN s.On_Promo = 1
                       THEN s.Total_Sale_Value ELSE 0 END) /
             NULLIF(SUM(s.Total_Sale_Value), 0) * 100, 2) AS promo_revenue_share
FROM sales s
JOIN products p ON s.SKU_Coded = p.SKU_Coded
WHERE EXTRACT(YEAR FROM CAST(s.Sold_Date AS DATE)) = 2023
GROUP BY p.category
ORDER BY promo_revenue_share DESC, p.category ASC

```

Reference analysis: Snacks leads with 6.26% promo revenue share, followed by Dairy (5.00%) and Personal Care (1.90%). Bakery shows only 0.21%; Beverages and Household have no promo sales.

Prompt 20. Difficulty: 4 Visualization goal: none.

“Return the top 10 region-category combinations by year-over-year revenue growth from 2022 to 2023, showing revenue for both years and the growth expressed as a percentage rounded to 2 decimal places.”

Reference SQL:

```

WITH yearly AS (
  SELECT st.region, p.category,
         EXTRACT(YEAR FROM CAST(s.Sold_Date AS DATE)) AS yr,
         SUM(s.Total_Sale_Value) AS revenue
  FROM sales s
  JOIN stores st ON s.Store_Number = st.Store_Number
  JOIN products p ON s.SKU_Coded = p.SKU_Coded
  WHERE EXTRACT(YEAR FROM CAST(s.Sold_Date AS DATE)) IN (2022, 2023)
  GROUP BY st.region, p.category, yr
)
SELECT region, category,
       ROUND(SUM(CASE WHEN yr = 2022 THEN revenue ELSE 0 END), 2) AS revenue_2022,
       ROUND(SUM(CASE WHEN yr = 2023 THEN revenue ELSE 0 END), 2) AS revenue_2023,
       ROUND(
         ((SUM(CASE WHEN yr = 2023 THEN revenue ELSE 0 END) -
           SUM(CASE WHEN yr = 2022 THEN revenue ELSE 0 END)) /
          NULLIF(SUM(CASE WHEN yr = 2022 THEN revenue ELSE 0 END), 0)) * 100,
         2) AS yoy_growth_pct
FROM yearly
GROUP BY region, category
HAVING SUM(CASE WHEN yr = 2022 THEN revenue ELSE 0 END) > 0
ORDER BY yoy_growth_pct DESC, revenue_2023 DESC, region ASC, category ASC
LIMIT 10

```

Reference analysis: Top growth: Midwest/Dairy +61.89% (\$109,924.97 → \$177,962.09), followed by Northeast/Dairy +52.08% and West/Dairy +38.41%. The tenth entry is Midwest/Personal Care +0.60% (\$97,222.05 → \$97,802.39).

Abstract in lingua italiana

I Large Language Model (LLM) alimentano sempre più spesso interfacce in linguaggio naturale verso dati strutturati, ma il loro impiego in produzione solleva questioni che vanno oltre l'accuratezza, ovvero la latenza di inferenza e il costo energetico. Per studiare questi aspetti, la tesi realizza un *Sales Agent*: un sistema basato su LLM, orchestrato con LangGraph, che traduce una domanda di business in una query SQL, ne interpreta i risultati in forma testuale e, opzionalmente, genera un grafico. Una configurazione gerarchica in YAML espone i parametri di inferenza studiati (parametri di campionamento, best-of- N , budget di token e profondità del raffinamento Chain-of-Thought). Il contributo principale è un profiling sistematico che misura congiuntamente la qualità dell'output, il tempo di esecuzione e il consumo energetico per singolo step (tramite CodeCarbon). La qualità è valutata su 20 domande analitiche, stratificate su quattro livelli di difficoltà SQL, lungo tre dimensioni: un punteggio graduato di similarità tabellare rispetto alla ground truth per il recupero dei dati e due punteggi LLM-as-judge per l'analisi testuale e la visualizzazione. Cinque esperimenti a step isolato definiscono una baseline per tre modelli open-weight (Gemma 4 E4B, Gemma 4 26B, Mistral Small 3.2 24B) e poi variano uno step dell'agente alla volta, concludendo con una conferma di Pareto. Sono inoltre progettati per verificare un'assunzione comune nel deployment locale di LLM, ovvero che i modelli thinking più grandi siano necessariamente la soluzione migliore. I risultati mostrano invece che la sola scelta del modello non spiega le prestazioni: gli effetti dominanti emergono dall'interazione tra famiglia del modello, step dell'agente, difficoltà del prompt e impostazioni di inferenza. Mistral è il modello non ottimizzato più efficiente ma crolla sui prompt di aggregazione più difficili; tuttavia, una volta corretto il suo step di lookup con il raffinamento Chain-of-Thought, raggiunge la qualità strict più alta e definisce la frontiera di Pareto finale. Gemma 26B è altamente ottimizzabile ma energeticamente costoso; Gemma E4B, corretto con raffinamento Chain-of-Thought sullo step di lookup a temperatura più bassa, rimane un'alternativa ad alta qualità molto vicina ma è dominata una volta considerata l'energia. L'analisi di sensitività mostra inoltre che gli effetti degli iperparametri sono fortemente specifici per step, che budget di token più ampi aiutano solo le coppie modello-step realmente troncate, e che il raffinamento Chain-of-Thought conviene solo quando il suo feedback corrisponde al tipo di errore: è l'intervento più efficace per il lookup SQL verificabile, ma dannoso per la visualizzazione, dove gli errori sono semantici e non eseguibili. La tesi distilla tre raccomandazioni per agenti dati LLM attenti all'energia: ottimizzare gli iperparametri per singolo step dell'agente anziché globalmente, allocare il compute a tempo di inferenza in base al tipo di errore, e riportare accuratezza ed energia per livello di difficoltà del prompt, poiché il modello migliore per i compiti semplici non è il migliore per quelli difficili.

Parole chiave: Agenti dati basati su LLM, Energy-aware profiling, Trade-off qualità-energia, LLM open-weight locali, Valutazione LLM-as-judge