Comparing Retrieval-Augmented Generation Pipelines for Energy-Aware Code Refactoring

Youssef Oubelmouh **Pierre Chhieng** **Amir Salah Eddine Daoudi**
Devoteam
{youssef.oubelmouh, pierre.chhieng, amir.salah.eddine.daoudi}@devoteam.com

Abstract

Choosing a Natural Language Generation (NLG) pipeline is difficult when the generated output must be functionally correct, measurably effective, and inexpensive to produce, all at once. We propose an evaluation framework for comparing NLG pipelines under these joint constraints, together with the Inference-Adjusted Gain (IAG), an architecture-agnostic metric combining output quality and inference cost. We instantiate the framework on energy-aware code refactoring: five systems (LLM-only, Retrieval-Augmented Generation (RAG), Knowledge-Augmented Generation (KAG), Agentic RAG, and Token-Efficient Agentic RAG (TeaRAG)) share the same 1,368-entry knowledge base (KB), named GreenKB, backbone LLM, and validation protocol across 11 Python programs, including three real GitHub projects. Only TeaRAG passes the functional gate on all three GitHub projects, with confirmed energy reductions on two; the third remains within measurement noise. Every other system passes at most one project. A rollback ablation traces this robustness mainly to orchestration rather than retrieval alone. On three synthetic scripts, each run over five seeds, TeaRAG again achieves the highest IAG: it cuts energy use by 96 % on average, against 88 % for KAG, at a higher token cost (3.6k vs. 2.2k tokens). One synthetic case is unstable across seeds for every system, leaving the margin over KAG narrow.

1 Introduction

The choice of NLG pipeline architecture matters more as LLM-based systems move into resource-sensitive settings. Four families have emerged: single-pass RAG (Lewis et al., 2021), KAG (Liang et al., 2024), iterative Agentic RAG (Asai et al., 2023; Yan et al., 2024), and token-bounded pipelines such as TeaRAG (Zhang et al., 2025). Each makes a different trade-off between how much it retrieves, how much it reasons, and how many tokens it spends doing so. Practitioners still have no principled way to compare them on tasks where correctness, output quality, and cost all matter at once. An excellent output that costs too much to generate brings no benefit in deployment. We address this with an **evaluation framework** for comparing pipelines under these joint constraints.

We instantiate this framework on *energy-aware code refactoring*: given Python code, produce a functionally equivalent variant that uses less energy. The task is a useful testbed because all three dimensions can be operationalised quantitatively: automated tests assess functional correctness, CodeCarbon estimates execution energy on the evaluation host, and token counts capture inference cost (Niu et al., 2024). RAPL/DRAM counters (Khan et al., 2018) are used only to construct the physically measured subset of GreenKB. Because all systems share the same 1,368-entry GreenKB, backbone LLM, and external validation protocol, observed differences primarily reflect pipeline design (retrieval strategy and orchestration) rather than knowledge content. This is not a pure retrieval ablation, however, because the systems also differ in control flow and LLM-call structure. Prior work uses KB-guided refactoring as a tool (Wu et al., 2026), but does not ask which retrieval *architecture* works best when correctness, quality, and cost all matter. We address this gap through a controlled pilot study comparing five configurations on 11 programs.

In this study, output quality refers to the energy reduction achieved by a functionally valid refactoring. We address four research questions: (**RQ1**) Which architecture achieves the best combination of functional pass rate and output quality? (**RQ2**) Which minimises inference cost at comparable quality? (**RQ3**) Does structured knowledge (KAG) improve quality over flat similarity retrieval on tasks requiring relational reasoning? (**RQ4**) Does performance vary across anti-pattern

Proceedings of the 19th International Natural Language Generation Conference: Long Papers, pages 1396–1406
October 17-21, 2026 ©2026 Association for Computational Linguistics

families? We predict that (**H1**) KAG outperforms RAG where its taxonomy covers the target pattern, (**H2**) TeaRAG achieves lower token cost and latency than Agentic RAG at comparable output quality, and (**H3**) no single architecture dominates all pattern types.

Our **contributions** are:

- A **unified framework** for comparing NLG pipeline architectures under joint correctness, quality, and efficiency constraints, applicable to any constrained NLG task.
- The **IAG** metric, an architecture-agnostic score combining output quality and generation cost.
- **GreenKB**, a 1,368-entry knowledge base we built by merging heuristic and physically-measured anti-pattern entries.
- **Controlled adaptations** of KAG, Agentic RAG, and TeaRAG to this task, evaluated in a pilot study on 11 programs with an analysis of how each architecture fails.

2 Related Work

NLG pipeline architectures. The dominant way to ground LLM-based NLG in domain knowledge is RAG (Lewis et al., 2021), which retrieves supporting text by semantic similarity. Similarity alone, however, ignores relational structure and the preconditions a transformation must satisfy. KAG (Liang et al., 2024) replaces simple retrieval with concept graphs to reason across multiple steps, relationships, and constraints. Agentic RAG (Asai et al., 2023; Yan et al., 2024) instead runs several retrieve-and-reason cycles, typically orchestrated with LangGraph (LangChain Inc., 2024) or LlamaIndex (Liu, 2022). Each extra cycle adds token cost. TeaRAG (Zhang et al., 2025) reduces this cost by compressing the context and limiting the reasoning steps. It was originally designed for open-domain question answering. To our knowledge, no prior study has compared all four families on a single constrained-generation task under a controlled protocol.

Constrained NLG and joint evaluation. The tension between quality and cost is not specific to code. Data-to-text systems trade faithfulness against fluency, summarisers trade factual consistency against length, and dialogue systems trade response quality against latency. Yet evaluation has mostly scored one dimension at a time, with BLEU, BERTScore, or task success. Green AI (Strubell et al., 2019; Schwartz et al., 2020) argues energy cost should be first-class alongside accuracy, but this is not yet integrated into joint NLG evaluation: benchmarks for structured generation (Huang et al., 2024; Niu et al., 2024) still score quality and cost separately. Our **IAG** metric fills this gap. We chose code refactoring because it makes all three dimensions measurable, which turns it into a controlled testbed for comparing architectures.

LLMs for constrained code generation and energy measurement. Code refactoring (Fowler et al., 1999) is a natural example of constrained generation. The output must behave exactly like the input, which tests can verify; it should use less energy, which can be measured using monitoring tools; and it must keep the original function signature. LLM-generated code is systematically less efficient than human code (Huang et al., 2024), and KB-guided generation beats LLM-only (Wu et al., 2026; Anupam et al., 2026), but prior work treats the pipeline as a monolithic tool rather than evaluating the retrieval architecture. Reliably assessing whether an energy code smell (Green Code Initiative, 2024; Pereira et al., 2017; Masanet et al., 2020; Green Software Foundation, 2023) has been removed requires careful isolation from test-framework and operating-system noise, plus direct hardware counters where the host exposes them and a software estimator such as CodeCarbon (Courty et al., 2023) where it does not (Section 5.2).

3 Approach

3.1 Task Definition

Each system receives the content of one Python file and must return a functionally equivalent variant that consumes less energy. The input ranges from a single function, for the CodeNet and synthetic cases, to a full multi-function module, for the real-world GitHub repositories (Section 4.1). When a functional test suite exists, we use it only to validate the output afterwards (Section 5); the pipeline never sees it. We treat the task as constrained structured generation: every modified function must preserve its signature and input/output behaviour, and satisfy all available test assertions. Table 1 summarises the five configurations we evaluate; Figure 1 aligns their pipeline stages side by side. All five systems call the same backbone LLM, Mistral-Small-4 (a 119B mixture-of-experts model, `mistral-small-4-119b-2603`),

served through NVIDIA NIM.

ID	Name	What it does
S0	Baseline	Unmodified code; serves as the energy reference
S1	LLM-only	The LLM detects and fixes on its own, with no retrieval
S2	RAG	Retrieves similar KB entries (FAISS), then fixes in one pass
S3	KAG	Retrieves via the concept graph and injects its constraints
S4	Agentic RAG	Repeats retrieve-and-reason cycles until a judge is satisfied
S5	TeaRAG	Same cycles, but with selective retrieval and context compression

Table 1: The five systems compared, plus the S0 baseline. All share the same backbone and software setup (Section 5.2).

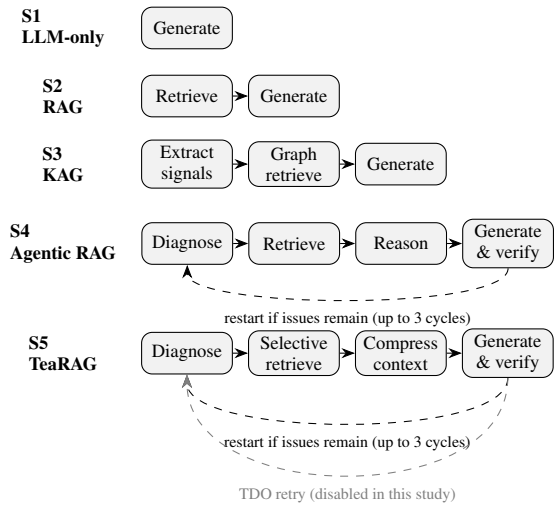


Figure 1: The five pipelines, stage by stage. *Test-Driven Optimization* (TDO) reruns the unit tests after each fix and retries on failure; disabled here so that all systems face the same validation gate (Section 3.5).

3.2 RAG (System S2)

A first LLM call detects active anti-patterns. A prebuilt FAISS index (NVIDIA `nv-embedqa-e5-v5` embeddings, chunk size 800, overlap 80) then retrieves the four most similar KB entries for each detected pattern; this step needs no LLM call. A second, final LLM call receives all retrieved entries in a single prompt and produces the fixed code.

3.3 KAG (System S3)

KAG replaces flat similarity search with a structured knowledge representation. Each entry is annotated with an anti-pattern concept node (7-category taxonomy, listed in Appendix A), observable symptoms, precondition predicates, and expected effect magnitude (from its measured energy reduction, Section 4.2). Retrieval proceeds in three steps: extract structural signals from the input code, project them onto concept nodes, and retrieve examples together with their preconditions and transformation constraints. Generation is thus grounded in relational reasoning rather than surface similarity. Unlike RAG, KAG requires no separate LLM call for detection. This reduces its call count but also introduces a confound in the inference-cost comparison, since cost differences cannot be attributed to retrieval alone.

3.4 Agentic RAG (System S4)

Agentic RAG is a LangGraph pipeline with four stages: diagnose the code, retrieve evidence from the KB with plain FAISS, reason over that evidence, and generate the fix. An LLM judge then checks whether issues remain; if so, the pipeline restarts from the diagnosis stage, for at most three cycles. Unlike TeaRAG, nothing limits how many tokens each cycle may spend.

3.5 TeaRAG (System S5)

TeaRAG bounds both retrieval and reasoning within a four-step cycle. It first writes a short diagnosis: two or three sentences naming at most two plausible anti-patterns. It then retrieves selectively, using a hybrid FAISS-and-graph search restricted to entries matching the diagnosed patterns. The retrieved evidence is compressed to a fixed token budget. Finally, one LLM call produces the refactored code, and a judge checks whether issues remain; if any do, the cycle restarts from the diagnosis step, again for at most three iterations. TeaRAG is cheaper not because it iterates less, but because each iteration retrieves less and compresses what it retrieves (Section 6).

Production TeaRAG also includes the Test-Driven Optimization (TDO) loop of Figure 1, which reruns unit tests after each fix and retries on failure. We disable it here, so our gate only checks syntax, signature, and output equivalence; this explains the occasional CodeNet regressions a full TDO pass would catch. TeaRAG's optional extra project context is likewise left empty, for a fair comparison.

4 Experimental Resources

4.1 Evaluation Corpus

The evaluation corpus comprises 11 programs: 5 CodeNet, 3 GitHub, and 3 synthetic scripts. The *main corpus* of Tables 3 and 4 covers the 8 CodeNet and synthetic programs; the GitHub projects are evaluated separately (Section 6.5).

Five CodeNet programs. We took the five largest accepted Python submissions from the IBM CodeNet dataset (Puri et al., 2021) that still fit in the LLM's context window.[1] To check equivalence, we replay the original `stdin` input on the optimised code and compare output hashes. These competitive-programming submissions carry the classic algorithmic inefficiencies: redundant computation and quadratic loops.

Three real GitHub projects. We selected three Python repositories with SEART GitHub Search (Dabic et al., 2021). Each has at least 30 stars, between 500 and 10 000 lines of Python, and a distinct GitHub `topics` tag, so the three cover different domains: *ecc-pycrypto* (cryptography, ∼821 lines), *yandex-tracker-exporter* (ETL, ∼2 780 lines), and *itertable* (data processing, ∼2 474 lines).

Three synthetic scripts. We wrote three worst-case scripts ourselves. They serve both as quantitative benchmarks and as qualitative case studies (Section 6.6). Each script targets a different anti-pattern family: numpy algorithms, sorting and statistics, and Pandas I/O. We wrote them without looking at any system's retrieval index, so no architecture is favoured by construction. They do draw on the same broad families that GreenKB covers, however, so absolute energy reductions are probably higher than they would be on fully independent code.

4.2 Shared Knowledge Base (GreenKB)

All systems share **GreenKB**, a 1,368-entry unified knowledge base we built by fusing two complementary corpora:

ENG subset (568 entries, prefix `ENG_`). We built this subset from SonarQube/CAST rules, augmented with our own LLM-generated patterns (memory management, I/O, Pandas, parallelism, general design). Energy-gain estimates are *heuristic* (expert-produced, no physical measurement). This subset provides fine-grained anti-pattern labels absent from academic benchmarks.

ALG subset (800 entries, prefix `ALG_`). We built this subset from before/after Python pairs drawn from the HumanEval, MBPP, and Mercury problems (Austin et al., 2021; Du et al., 2024): Qwen3-32B produced one optimised version per problem, and we measured both versions with RAPL/DRAM hardware counters, in microjoules. Each pair therefore carries a physically measured *Energy Reduction* (ER):

$$\mathrm{ER} = \frac{E_{\mathrm{orig}} - E_{\mathrm{opt}}}{E_{\mathrm{orig}}} \tag{1}$$

where E_{orig} and E_{opt} are the measured energy of the original and optimised code. A positive ER means the optimised version uses less energy. This equation defines ER for the whole paper; only the energy meter changes (RAPL/DRAM here, CodeCarbon for the experiments of Section 5.2).

GreenKB: the merged knowledge base. GreenKB combines the two subsets: ENG brings the breadth of engineering patterns, ALG the reliability of physical measurement. They do not overlap and share the same schema, so FAISS and Neo4j index them uniformly, and all five systems draw on the same base.

For **KAG**, each entry is additionally indexed in Neo4j with the concept node, preconditions, and effect magnitude of Section 3.3; Cypher traversal then follows relational constraints across several hops, which flat FAISS cannot.

Table 2 shows the shared entry schema.

Field	Content
ID	Unique identifier (prefix `ENG_` or `ALG_`)
Language	Python
Category	Anti-pattern family label
Code before	Original Python function
Code after	Optimised, validated variant
Explanation	Transformation rationale
Energy impact	heuristic estimate / measured ER (μJ)
Source	ENG (heuristic) or ALG (measured)
Concept node	(KAG) anti-pattern category node
Preconditions	(KAG) validity predicates

Table 2: GreenKB entry schema. KAG additionally indexes concept nodes and preconditions in Neo4j. All other fields are shared.

[1] `small` split of `https://huggingface.co/datasets/windchimeran/codenet_python`, filtered to `code_size` $\leq$ 5 000 bytes to bound input length, ranked by size descending, top 5 overall (no per-problem deduplication).

5 Experimental Protocol

5.1 Functional Validation

Every candidate passes a three-step gate before benchmarking. First, a syntax check with Python's `ast.parse`. Second, signature preservation: the function must keep its name and argument count. Third, execution equivalence, which depends on the kind of program. For CodeNet programs, we pipe the original input to the optimised code in an isolated subprocess with a 10-second timeout, and require the output to match the original exactly. For the GitHub projects, we re-run the full pytest suite on the optimised codebase. The synthetic scripts call `random` without a seed, so their output differs from run to run by construction; for them, this last step only requires the script to run to completion in an isolated subprocess. A candidate failing any gate is rejected and excluded from energy benchmarking. Per-seed failures on the synthetic cases are handled separately, as described in Section 5.3 and Appendix C.

5.2 Energy Measurement

All energy measurements in this paper come from CodeCarbon v3.2.8, under Python 3.13, which estimates CPU and RAM energy from execution time, CPU load, and Thermal Design Power (TDP); the RAPL/DRAM hardware counters that grounded the ALG subset (Section 4.2) are not exposed by our virtualised evaluation host. Because this estimate is noisier than direct counters, we report for GitHub the median of 10 repetitions with the interquartile range (IQR), the spread of the middle half of those repetitions, and treat any ER within the IQR of a same-code control as noise, not a gain (Section 6.5). Each benchmark series starts with one unmeasured warm-up run of at least 1,000 calls.

On real projects, each pytest invocation costs 1.3 to 3.5 seconds, which drowns out function-level differences of a few milliseconds. We therefore benchmark each optimised function in its own process and call it between 200,000 and 50 million times, depending on its runtime, until the signal stabilises.

For each system we report the energy of the generated code, its inference cost (tokens, LLM calls, latency), and its ER (Eq. 1), which tables give as a percentage.

5.3 The IAG Metric

To jointly assess output quality and generation efficiency, we introduce:

$$\mathrm{IAG} = \frac{\max(0,\, \overline{\mathrm{ER}}) \times 100}{1 + \log_{10}(T)} \tag{2}$$

Here T is the total number of tokens, generated plus retrieved. Both $\overline{\mathrm{ER}}$ and T are computed the same way: for each of the three synthetic cases we first take the median over the five generation seeds, then average the three per-case medians. Appendix B walks through the full computation.

When a seed fails the functional gate, we drop it from that case's median rather than counting it as zero. In practice only LLM-only loses a seed this way; counting that failure as zero would lower its IAG from 17.5 to 16.6, a tie with RAG, and would leave the top-two ranking unchanged (Appendix C). We use $\max(0, \overline{\mathrm{ER}})$ rather than the absolute value so that a system that increases energy (negative ER) scores zero, instead of being rewarded as if it had saved the same amount. Every mean ER in this paper is positive, so this choice changes no reported number. Replacing $\log_{10}$ with $\log_2$ leaves the top-two ranking unchanged. A linear penalty would not: it would mostly reward LLM-only for retrieving nothing at all.

GitHub IAG. For the GitHub projects, Eq. 2 uses the microbenchmark ER of Section 6.5. ER and T are the median over the three projects, and a project that fails the gate or stays within noise counts as zero (it is not dropped, unlike a failed seed in $\mathrm{IAG}_{\mathrm{syn}}$). In practice, a system needs a confirmed gain on at least two of the three projects to score above zero: with only three projects, $\mathrm{IAG}_{\mathrm{gh}}$ behaves more like a pass count than a fine-grained score.

For $\mathrm{IAG}_{\mathrm{syn}}$ we average across cases instead: two of the three synthetic cases sit at a ceiling, so a cross-case median would ignore the only case that separates the systems. It would also swap the top two (KAG 22.8, TeaRAG 21.8), one more reason to read the synthetic margin as narrow.

Generalising IAG beyond code. Nothing in Eq. 2 is code-specific: replace ER by any normalised quality gain (a ROUGE gain in summarisation, a task-success gain in dialogue, a fluency delta in data-to-text) and T its token cost, IAG transfers unchanged.

5.4 Statistical Analysis

The main corpus contains eight programs: the five CodeNet submissions and the three synthetic scripts. The two groups play different roles. The CodeNet programs run too fast for their energy gains to be measurable (a floor effect, Section 6), so they probe functional robustness only; all energy comparisons therefore rest on the three synthetic scripts. With so few programs, significance tests would be underpowered, so we treat every comparison as exploratory and report medians and IQRs rather than p-values.

For each synthetic case, every system generates and measures its fix five times, once per seed (seeds 42 to 46; see Limitations). We report the median over the five seeds for each case, and the mean of the three per-case medians (Tables 4 and 5, Section 5.3). CodeNet and GitHub results rest on a single seed.

6 Results

6.1 Functional Pass Rate (RQ1)

Only TeaRAG clears every syntax and signature gate. KAG matches it on the test gate, at 7 out of 8, and LLM-only fails mostly by breaking function signatures (Table 3).

System	Syntax	Sig.	Tests	Pass
S1 LLM-only	6/8	4/8	3/8	3/8
S2 RAG	7/8	7/8	5/8	5/8
S3 KAG	7/8	7/8	**7/8**	**7/8**
S4 Agentic RAG	7/8	6/8	5/8	5/8
S5 TeaRAG	**8/8**	**8/8**	**7/8**	**7/8**

Table 3: Functional pass rate on the eight programs of the main corpus (five CodeNet, three synthetic). GitHub results appear separately in Section 6.5.

Syntax uses `ast.parse`; Sig. checks signature preservation; Tests checks output hash (CodeNet) or isolated execution (synthetic, Section 5), a case passing if at least three of five seeds pass.

6.2 Energy and Time Gains (RQ1)

On the three synthetic scripts, mean ER ranges from 69.9 % for LLM-only to 96.1 % for TeaRAG; Table 5 gives the per-case breakdown. Two cases, numpy-multi and sort/stats, are stable across seeds, with IQRs below 8 points. Pandas/IO is not: its IQRs range from 16.6 to 488 points (Limitations). On the five CodeNet programs, the test inputs are so small that every variant finishes too quickly to measure, and ER stays near zero for all systems, the same measurement floor that motivated our production-scale protocol (Section 5.2). These programs therefore probe functional robustness, not energy.

6.3 Inference Cost and IAG (RQ2)

Table 4 reports median token cost, latency, and IAG per system.

System	Tk.gen	Tk.ret	Calls	Lat.(s)	IAG_{syn}	IAG_{gh}
S1 LLM-only	1 996	0	2	19.7	17.5	0.0
S2 RAG	1 993	4 594	2	27.9	16.6	0.0
S3 KAG	**1 380**	**2 070**	1	21.8	20.3	0.0
S4 Agentic RAG	2 376	5 637	3.5	41.5	18.0	0.0
S5 TeaRAG	2 557	2 815	6	**19.2**	**21.1**	**6.1**

Table 4: Inference cost and IAG (Eq. 2).

Tk.gen and Tk.ret are generated and retrieved tokens, Calls is the number of LLM calls, and Lat. is latency. All four cost columns are medians over the eight programs of the main corpus. IAG_{syn} applies Eq. 2 to the three synthetic scripts (Section 5.3); IAG_{gh} applies it to the three GitHub projects (Section 6.5), where a project that fails the gate or stays within noise counts as zero. LLM-only retrieves nothing, hence its zero Tk.ret.

6.4 Results by Anti-Pattern Family (RQ3, RQ4)

Table 5 breaks ER down per synthetic case.

Case	S1	S2	S3	S4	S5
numpy multi	99.7	99.8	99.7	99.7	99.9
sort/stats	99.7	98.2	98.6	98.8	99.6
Pandas I/O	10.2*	31.8	66.9	56.7	88.8‡
Mean†	69.9	76.6	88.4	85.1	**96.1**

∗ Median over four seeds; the fifth failed to run. ‡ Unstable across seeds (Limitations). † Mean of the three case medians; this row feeds IAG_{syn} in Table 4.

Table 5: ER (%) per synthetic case: how much energy each system's fix saves. Each cell is the median over five generation seeds.

6.5 GitHub Project Validation

Table 6 shows, for each system and project, whether the refactored codebase still passes its test suite and how many tokens the refactoring cost. The IAG_{gh} column of Table 4 combines these outcomes with the microbenchmark ER measured below.

This section evaluates **functional robustness** and **token efficiency**. We do not report ER at the

	ecc-pycrypto		yandex-tracker		itertable	
Sys.	**Pass**	**Tokens**	**Pass**	**Tokens**	**Pass**	**Tokens**
S1	✓	1 430	×	14 062	×	1 549
S2	✓	4 373	×	35 504	×	4 821
S3	✓‡	2 310	×	23 770	×	1 695
S4	✓	9 354	×	50 703	×	14 834
S5	✓	4 950	✓	19 237	✓†	23 774

‡ Tests pass but ER is 1.1 % (within noise): the bit-shift pattern lies outside KAG's taxonomy. † Per-file rollback: 4 of 8 files keep their fix; ✓ means no regression (26 passed, 1 pre-existing failure), not a fully green suite. Token counts vary across runs (Limitations).

Table 6: GitHub validation: pass/fail and token cost (generated + retrieved) per refactoring.

pytest level: framework overhead dwarfs function-level differences (Section 5.2).

Production-scale microbenchmarks. The pytest-level floor could mean two things: no gains, or gains too small to measure. To decide, we benchmarked each modified function in isolation, at production-representative call counts, with CodeCarbon (Section 5.2). Every system went through the same standard pipeline, without TDO, and we benchmarked gate-failed candidates as well, unlike in IAG_{gh}. On ecc-pycrypto, over 50 million calls, the bit-shift fix saves 31.3 % for LLM-only, 25.8 % for RAG, 37.9 % for Agentic RAG, and 32.1 % for TeaRAG. KAG leaves the operation unchanged: an ER of 1.1 %, within noise. On itertable, over 200,000 calls, only the KAG and TeaRAG variants run at all (the others crash outright at this scale, despite their pytest-level results in Table 6) and both stay within noise (ER 6.3 % and 7.4 %). On yandex-tracker, only TeaRAG produces code that runs; over 300,000 calls per function, its three modified functions reach a median ER of 36.1 %, a confirmed gain. TeaRAG is thus the only architecture that passes all three projects. Its gains are confirmed on two of them, ecc-pycrypto and yandex-tracker, while itertable stays within noise. Its per-file rollback keeps 4 of itertable's 8 files, and on yandex-tracker it spends 1.8 to 2.6 times fewer tokens than RAG or Agentic RAG (Table 6).

Maintainability: energy versus readability. Saving energy can make code harder to read. To check this, we score every ecc-pycrypto output with `radon`'s maintainability index (MI), a 0–100 readability score where higher is better, averaged per file. The two systems that save the most energy also lose the most readability: TeaRAG's MI falls from 46.5 to 39.0 and Agentic RAG's to 38.9, because their fixes wrap the original code in helper functions. RAG barely moves (45.9), and KAG, which leaves the code essentially unchanged, keeps its MI intact. The yandex-tracker fixes show the same pattern, from 38.5 to 35.7. Readability is therefore a third dimension, independent of energy and token cost, that IAG does not capture.

6.6 Qualitative Case Studies

Cases 1 and 2: a shared ceiling. The first two scripts, four co-occurring numpy anti-patterns and a quadratic sort, are solved by all five systems: every median in the first two rows of Table 5 lies above 98 %, with stable seeds. Even KAG, which had failed the sort case in an earlier single-seed run over a signature mismatch, passes on all five seeds here. Once a pattern is easy to detect syntactically, retrieval architecture stops mattering.

Case 3 (Pandas I/O): the discriminating case. The five Pandas patterns do separate the systems. Agentic RAG reaches a median ER of 56.7 %, against 88.8 % for TeaRAG, which anchors to ALG entries and rewrites the computation with 3D numpy broadcasting. Both retrieve a similar amount of context (median 3,408 vs. 2,894 tokens), so the gap comes from what they retrieve, not from how much. This is also the case where every system varies most from seed to seed (Limitations), so we read this ranking as a tendency, not a fixed value.

7 Discussion

TeaRAG best combines pass rate and quality (RQ1). TeaRAG ties KAG on pass rate (7/8) but leads on ER (96.1 % vs. 88.4 %). On GitHub, TeaRAG alone passes all three functional gates and delivers confirmed energy reductions at production scale, reaching $IAG_{gh} = 6.1$. The others pass at most one of the three projects (multi-file or API-sensitive code), scoring $IAG_{gh} = 0$. Robustness and energy gain only matter together. A system that passes the tests but saves nothing, as KAG does on ecc-pycrypto, is not worth deploying; neither is one that saves energy but breaks the tests, as LLM-only does. TeaRAG is the only system that delivers both (see the rollback ablation below).

Ceiling effects leave H1 untested by Cases 1-2 (RQ3 inconclusive). On Cases 1 and 2, every system lands near a 99 % ceiling (Section 6.6): the

cases are too easy to tell structured retrieval apart from flat similarity. The only informative case, Pandas/IO, lies outside KAG's taxonomy on paper, and yet KAG's median ER of 66.9 % is second only to TeaRAG's 88.8 %, although two of its five seeds regress. This pilot therefore neither confirms nor refutes H1.

Agentic loops may need execution feedback. Agentic RAG's median ER on the Pandas case, 56.7 %, trails TeaRAG's 88.8 %. One plausible reason is that, without an execution gate inside the loop, multi-turn retrieval cannot tell a good transformation from a poor one, but we did not test such a variant directly. Both figures also vary substantially from seed to seed (Limitations).

Performance varies by pattern family: no universal winner (closing RQ4, H3 supported). Table 5 shows that no ranking holds across patterns. TeaRAG leads on numpy-multi; LLM-only edges it on sort/stats by 99.7 % to 99.6 %, a difference within noise; and KAG is the closest competitor on Pandas I/O. Combined with the maintainability results (TeaRAG's energy lead costs readability), no system wins on every axis or pattern: TeaRAG leads robustness and energy, KAG is cheapest, and readability trades against both.

Cross-file rollback determines robustness on real code. RAG, LLM-only, and Agentic RAG all fail itertable for the same reason: one bad rewrite among the eight files breaks the whole project, and none of them can roll a single file back. TeaRAG's per-file rollback turns the same situation into a pass, keeping four of the eight files. To check whether this is an orchestration effect rather than a retrieval effect, we ran a one-off ablation on yandex-tracker's eight candidate files: given the same per-file rollback, Agentic RAG passes too, keeping five of eight. The GitHub results therefore support TeaRAG's overall pipeline design, not its bounded retrieval alone.

Inference cost and the TeaRAG advantage (H2 supported). Compared with Agentic RAG, TeaRAG retrieves half the context (2,815 vs. 5,637 tokens) and answers in less than half the time (19.2 s vs. 41.5 s), with a better mean ER (96.1 % vs. 85.1 %). It is even slightly faster than LLM-only (19.2 s vs. 19.7 s) despite making three times as many calls, because each call is narrower in scope; it does generate a few more tokens than Agentic RAG (2,557 vs. 2,376). IAG_{syn} summarises the picture: 21.1 for TeaRAG, 18.0 for Agentic RAG, 16.6 for RAG. KAG comes within 0.8 point, at 20.3, and at a lower token cost than either, but only on the synthetic subset. On GitHub, only TeaRAG scores above zero (IAG_{gh} 6.1 against 0.0 for KAG).

Amortisation: is generating the fix worth it? IAG says nothing about whether a fix's savings ever repay the cost of generating it. TeaRAG's three yandex-tracker fixes save 26.8 J over 300,000 calls each (Section 6.5), and cost 19,237 tokens to produce. The savings therefore repay the generation cost only if producing one token costs less than about 1.4 mJ. Published estimates of per-token inference energy (Dauner and Socher, 2025) sit orders of magnitude above this threshold, so at the benchmarked volumes the fixes do not yet pay for themselves. The generation cost is paid once, however, while the savings grow with every call: energy-aware refactoring amortises on hot paths called far more often.

Transfer to other NLG tasks. Our framework needs three things from a task: a way to reject wrong outputs, a quality gain, and a token count. Code makes the first one easy, since a test either passes or fails. Other NLG tasks have their own way to reject outputs: a data-to-text description that invents or omits facts, a summary that contradicts its source, a dialogue turn that fails to complete the task. With such a check in place, IAG applies as described in Section 5.3. The one difference is that these checks are less exact than a test suite, so the pass rate becomes an estimate rather than a count.

8 Conclusion

We presented, to our knowledge, the first controlled comparison of four knowledge-grounded NLG pipeline architectures and an LLM-only baseline for energy-aware code refactoring under joint correctness, quality, and efficiency constraints. With one knowledge base, one backbone LLM, and one protocol shared across 11 programs, we find that performance is driven as much by orchestration as by retrieval strategy.

Three findings stand out. First, only TeaRAG passes all three GitHub gates, with confirmed energy gains on two projects; every other system passes at most one. The rollback ablation shows that this robustness comes from an orches-

tration mechanism that other pipelines could adopt; TeaRAG's own edge is its lower context use and latency. Second, over five seeds per synthetic case, TeaRAG obtains the best IAG_{syn} at 21.1, with KAG close behind at 20.3. Third, two of the three synthetic cases prove too easy to separate KAG from RAG, and the discriminating one, Pandas/IO, is unstable across seeds, leaving H1 open.

Both the findings and the IAG metric extend beyond code, to any knowledge-grounded, cost-constrained NLG system; future work includes other languages, larger codebases, and dynamic routing between KAG and TeaRAG.

Limitations

This pilot covers Python only, leans towards algorithmic patterns, and needs replication at larger scale. All results rest on a single backbone LLM. The synthetic gate only checks that the script runs (Section 5); a follow-up check at sub-function level found semantic divergences in 3 of the 75 generated fixes, none produced by TeaRAG. Our energy numbers are software estimates on isolated functions, not hardware measurements of whole programs: our virtualised host exposes no RAPL/-DRAM counters, and pytest-level gains drown in framework overhead (Section 5.2). As a safeguard, any ER smaller than the run-to-run spread of unchanged code counts as noise. Sub-second runs execute in a separate process to keep the tracker's overhead out of the reading. Finally, generation is nondeterministic even at a fixed seed, because NVIDIA NIM's GPU kernels vary between runs (Yuan et al., 2025); the Pandas/IO case is bimodal across seeds; GitHub and CodeNet results rest on a single seed; and the LLM judge used in the agentic loops was not itself validated.

Data and Code Availability

This work was conducted as part of an industrial R&D project. GreenKB, the per-case synthetic and CodeNet result files underlying Tables 3–6, the IAG computation script, and the five system implementations (RAG, KAG, Agentic RAG, TeaRAG, LLM-only) are available at `https://github.com/Youssef-Oub/inlg2026-rag-refactoring` and archived on Zenodo (`https://doi.org/10.5281/zenodo.22656427`) under an MIT licence (code) and CC BY 4.0 (data). The three GitHub projects used as real-world targets are third-party open-source repositories (Ethical Considerations) and are not redistributed.

Ethical Considerations

This work uses LLMs to generate code refactorings. Each output is evaluated using the applicable functional gate: pytest for GitHub projects, output-hash equivalence for CodeNet, and successful isolated execution for synthetic scripts. Because the synthetic execution gate is weaker, we do not treat it as proof of full semantic equivalence, as discussed in the Limitations. Energy measurements are comparative, controlled-environment values, not ground-truth carbon or deployment recommendations. We built GreenKB from publicly available benchmark datasets (MBPP, Mercury, HumanEval) and open-source static analysis rules (SonarQube/CAST) under their respective open licenses. The real project repositories (ecc-pycrypto, itertable, yandex-tracker-exporter) are MIT or Apache-licensed open-source projects selected without author involvement.

References

Sagnik Anupam, Alexander Shypula, and Osbert Bastani. 2026. Llm program optimization via retrieval augmented search.

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2023. Self-rag: Learning to retrieve, generate, and critique through self-reflection.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models.

Benoît Courty and 1 others. 2023. codecarbon. `https://github.com/mlco2/codecarbon`.

Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling projects in GitHub for MSR studies. In *Proceedings of the 18th International Conference on Mining Software Repositories (MSR)*, pages 560–564. IEEE.

Maximilian Dauner and Gudrun Socher. 2025. Energy costs of communicating with ai. *Frontiers in Communication*, 10:1572947.

Mingzhe Du, Luu A Tuan, Bin Ji, Qian Liu, and See-Kiong Ng. 2024. Mercury: A code efficiency benchmark for code large language models. *Advances in Neural Information Processing Systems*, 37:16601–16622.

Martin Fowler and 1 others. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

Green Code Initiative. 2024. creedengo-python. https://github.com/green-code-initiative/creedengo-python.

Green Software Foundation. 2023. Green software engineering: Principles for sustainable development. [Accessed 15-07-2026].

Dong Huang, Yuhao Qing, Weiyi Shang, Heming Cui, and Jie M. Zhang. 2024. EffiBench: Benchmarking the efficiency of automatically generated code. *Advances in Neural Information Processing Systems*, 37:11506–11544.

Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. 2018. Rapl in action: Experiences in using rapl for power measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2).

LangChain Inc. 2024. langgraph. https://github.com/langchain-ai/langgraph.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2021. Retrieval-augmented generation for knowledge-intensive nlp tasks.

Lei Liang, Mengshu Sun, Zhengke Gui, Zhongshu Zhu, Zhouyu Jiang, Ling Zhong, Yuan Qu, Peilong Zhao, Zhongpu Bo, Jin Yang, Huaidong Xiong, Lin Yuan, Jun Xu, Zaoyang Wang, Zhiqiang Zhang, Wen Zhang, Huajun Chen, Wenguang Chen, and Jun Zhou. 2024. Kag: Boosting llms in professional domains via knowledge augmented generation.

Jerry Liu. 2022. llama_index. https://github.com/run-llama/llama_index.

Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. 2020. Recalibrating global data center energy-use estimates. *Science*, 367(6481):984–986.

Changan Niu, Ting Zhang, Chuanyi Li, Bin Luo, and Vincent Ng. 2024. On evaluating the efficiency of source code generated by llms. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, FORGE '24, page 103–107, New York, NY, USA. Association for Computing Machinery.

Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, pages 256–267, New York, NY, USA. Association for Computing Machinery.

Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks.

Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. 2020. Green ai. *Commun. ACM*, 63(12):54–63.

Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. 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, Florence, Italy. Association for Computational Linguistics.

Yue Wu, Minghao Han, Ruiyin Li, Peng Liang, Amjed Tahir, Zengyang Li, Qiong Feng, and Mojtaba Shahin. 2026. Fasterpy: An llm-based code execution efficiency optimization framework.

Shi-Qi Yan, Jia-Chen Gu, Yun Zhu, and Zhen-Hua Ling. 2024. Corrective retrieval augmented generation.

Jiayi Yuan, Hao Li, Xinheng Ding, Wenya Xie, Yu-Jhe Li, Wentian Zhao, Kun Wan, Jing Shi, Xia Hu, and Zirui Liu. 2025. Understanding and mitigating numerical sources of nondeterminism in llm inference.

Chao Zhang, Yuhao Wang, Derong Xu, Haoxin Zhang, Yuanjie Lyu, Yuhao Chen, Shuochen Liu, Tong Xu, Xiangyu Zhao, Yan Gao, Yao Hu, and Enhong Chen. 2025. Tearag: A token-efficient agentic retrieval-augmented generation framework.

A KAG Anti-Pattern Taxonomy

Table 7 lists the 7-category taxonomy used to structure the shared anti-pattern detection prompt (all five systems) and to build KAG's concept nodes (Section 3.3): KAG projects each detected anti-pattern onto one of these categories before retrieving examples, preconditions, and transformation constraints attached to that concept node.

Category	Scope
Algorithms	Sorting, search, traversal complexity
Data structures	List/dict/set/heap choice
Loops	Nested loops, redundant iteration
Memory	Unnecessary allocations, copies
I/O	File reads, network calls
Computation	Redundant/repeated arithmetic
Libraries	Suboptimal standard-library usage

Table 7: The 7-category anti-pattern taxonomy shared by the detection prompt and KAG's concept graph (Section 3.3).

B Per-Case Synthetic ER and Token Cost

Table 8 lists, for each synthetic case and each system, the median ER and the median total token count T over the five generation seeds. These are the raw values behind Table 5 and behind $\mathrm{IAG}_{\mathrm{syn}}$ in Table 4, and they suffice to reproduce both.

The recipe has two steps. First, average each system's three median ER values: for TeaRAG, the mean of 99.9, 99.6, and 88.8 is 96.1, which is the mean row of Table 5. Second, average the three median token counts the same way and apply Eq. 2: for TeaRAG the mean T is 3,560, so $\mathrm{IAG}_{\mathrm{syn}} = 96.1/(1+\log_{10} 3{,}560) \approx 21.1$, the value reported in Table 4. The same two steps give KAG its 20.3 (mean ER 88.4, mean T 2,236).

Case	Sys.	ER (%)	T
numpy multi	S1	99.7	904
	S2	99.8	4 047
	S3	99.7	2 116
	S4	99.7	4 076
	S5	99.9	2 784
sort/stats	S1	99.7	751
	S2	98.2	3 900
	S3	98.6	1 300
	S4	98.8	7 232
	S5	99.6	3 641
Pandas I/O	S1	10.2^{*}	1 364
	S2	31.8	4 688
	S3	66.9	3 292
	S4	56.7	4 792
	S5	$88.8^{\ddagger}$	4 256

Table 8: Median ER and median total tokens T for each synthetic case and system, over the five generation seeds. $*$ and ‡ as in Table 5.

C $\mathrm{IAG}_{\mathrm{syn}}$ When a Failed Seed Counts as Zero

When a seed fails the functional gate, Section 5.3 drops it from that case's median. One could object that dropping failures flatters the failing system. This appendix therefore recomputes $\mathrm{IAG}_{\mathrm{syn}}$ under the harsher rule: a failed seed stays in the median and counts as zero, as if the fix saved no energy at all. Table 9 compares the two rules, from the same per-seed data as Appendix B.

Only LLM-only ever fails the gate (one Pandas I/O seed). Under the zero rule its mean ER weakens from 69.9 % to 66.5 % and its $\mathrm{IAG}_{\mathrm{syn}}$ drops from 17.5 to 16.6, a tie with RAG instead of a small lead. Every other system keeps its score, since none of their seeds fail. The choice of rule therefore changes nothing above LLM-only: TeaRAG stays first and KAG second either way, and Agentic RAG stays ahead of LLM-only in both settings.

System	Failed seed dropped	Failed seed = 0
S1 LLM-only	17.5	16.6
S2 RAG	16.6	16.6
S3 KAG	20.3	20.3
S4 Agentic RAG	18.0	18.0
S5 TeaRAG	**21.1**	**21.1**

Table 9: $\mathrm{IAG}_{\mathrm{syn}}$ (Eq. 2) under the two treatments of a gate-failing seed. The left column is the rule used in Table 4; only LLM-only is affected, because it is the only system with a failed seed.