Splitting the Work: Energy-Efficient Code Generation with a Planner-Worker LLM Architecture

Yuchen Sun (6575641) Fedor Baryshnikov (5814197) Tom Clark (5275040)
Tobias Veselka (4451848) Yuting Dong (5244447)

TU Delft
CS4575 Sustainable Software Engineering (2025/26 Q3)

April 2, 2026

Abstract

As AI-powered coding assistants become indispensable in modern software development workflows (1), the environmental cost of their underlying Large Language Models (LLMs) has emerged as a significant concern, with inference accounting for the majority of their lifecycle energy consumption (2). This paper proposes a modular Planner-Worker architecture designed to reduce energy expenditure by decomposing complex programming tasks into isolated subtasks. By employing a 12B parameter Gemma 3 model as a planner (3) and 4B models as workers (3), our framework achieved a 58.1% reduction in mean total energy consumption compared to a single-model baseline (4).

Experimental evaluations using a structured manual rubric indicate that this approach maintains high output quality, with a negligible 0.2-point difference in performance scores (16.0/20 vs. 16.2/20), demonstrating that energy savings do not necessarily compromise user experience. The system's decoupled design ensures high extensibility, allowing developers to seamlessly substitute components with specialized or more efficient models (5) as the LLM ecosystem continues to evolve. These results provide a scalable and repeatable path toward sustainable AI-assisted software engineering.

1 Introduction

While AI coding assistants like GitHub Copilot have achieved mass adoption, their environmental footprint remains a growing concern. Since the release of ChatGPT in late 2022, AI-powered coding assistants have moved from experimental novelties to indispensable tools in developers' daily workflows. According to the Stack Overflow 2025 Developer Survey (1), 84% of developers now use or plan to use AI tools in their development process, a significant increase from 76% in 2024. Furthermore, 51% of professional developers report using AI tools on a daily basis. The JetBrains State of Developer Ecosystem 2025 survey corroborates these findings, reporting that 85% of developers regularly use AI tools for coding and development, with 62% relying on at least one AI coding assistant or agent (6). In practical terms, approximately 41% of all code written today is AI-generated or AI-assisted, and platforms like GitHub Copilot have surpassed 20 million users (7).

This surge in AI-assisted development, however, carries a significant and often overlooked environmental cost. The International Energy Agency (IEA) estimates that global data centre electricity consumption reached approximately 415 terawatt-hours (TWh) in 2024, accounting for roughly 1.5% of global electricity consumption (2). This figure is projected to more than double, reaching 945 TWh by 2030 which equivalent to the current total annual electricity demand of Japan. The IEA identifies AI as the most important driver of this growth, with AI-focused accelerated servers projected to grow at 30% annually. In the United States alone, data centres consumed 183 TWh in 2024, representing over 4% of the country's total electricity consumption (8). Training a single large model such as GPT-4 is estimated to consume approximately 50 GWh of electricity (9), and crucially, inference, the ongoing process of running and responding to user queries, can account for up to 90% of a model's total lifecycle energy consumption (10).

As the AI model ecosystem has matured, users now face an ever-expanding selection of models that vary dramatically in both capability and energy cost. Major providers such as OpenAI, Anthropic, and Google each offer model families spanning from lightweight variants (e.g., GPT-4o mini, Claude Haiku, Gemini Flash) to flagship models (e.g., GPT-5.2, Claude Opus 4.6, Gemini Pro), with API pricing, a rough proxy for computational cost, differing by up to two orders of magnitude (5). The prevailing behaviour among developers is to default to the most powerful available model for all tasks, regardless of whether the task demands such capability. The Stack Overflow survey reports that ChatGPT and GitHub Copilot - both of which default to or prominently feature their most capable models are used by 82% and 68% of developers respectively (1). This "one size fits all" approach means that simple coding tasks such as generating boilerplate code, adding docstrings, or writing basic utility

functions are processed with the same energy expenditure as complex architectural decisions or multi-file debugging sessions.

2 This paper's contribution

The aim of this paper is to address the substantial energy consumption associated with employing LLMs to solve programming tasks of varying complexity. We propose and evaluate a decomposed Planner–Worker framework, in which a large "planner" LLM structures and decomposes tasks, while smaller "worker" LLMs execute them. This pipeline demonstrates strong potential for reducing energy usage while maintaining response quality comparable to that of a single large model (see Section 3).

Beyond reducing per-prompt energy consumption, our approach aims to provide programmers with more consistent and reliable responses. By reformulating prompts, the planner LLM enables smaller models to generate higher-quality outputs, thereby reducing the need for iterative querying. This not only improves efficiency but also saves both computational energy and the user's time.

In addition to its technical contributions, this work introduces a transparent mechanism for reporting user energy consumption. Specifically, we provide users with clear metrics expressed in Joules, alongside contextualised comparisons that situate their usage within a broader context, promoting greater awareness of the environmental impact their LLM usage.

The remainder of this paper is organised as follows. Section 3 reviews related work on LLM energy consumption, task decomposition, and code generation benchmarks. Section 4 describes our system architecture, including planner-worker separation, execution modes, model selection, and energy measurement setup. Section 5 presents our experimental results of the research question addressing energy consumption difference between Single Big Model and our Planner-Worker Strategy. Section 6 discusses practical implications, threats to validity, and future work. Section 7 concludes.

3 Related Work

3.1 Energy Consumption Dynamics in LLM Inference

The foundational premise of our research is supported by a growing body of evidence indicating that the energy cost of a single large-scale LLM is disproportionately higher than that of multiple small LLMs working collaboratively on the same query, despite generating outputs of similar quality. This section explores the specific mechanical inspirations derived from current literature to validate our approach. A critical distinction in LLM energy profiles lies in the prefill stage versus the decoding stage. Research from TokenPowerBench suggests that by utilizing a mid-tier LLM as a planner, we can generate a sophisticated execution plan at a relatively low initial energy cost (11). Once this plan is established, smaller worker models execute specific segments, effectively managing the decoding stage. Although the decoding stage is inherently more energy-intensive per token due to repeated memory access and sequential computation, the use of small-scale LLMs ensures that the cumulative energy draw remains predictable and significantly lower than that of a flagship model. This phenomenon is further explained by hardware utilization patterns identified in the study "On-Device or Remote?" (12), which notes that larger models demand higher GPU power states and more intensive memory bandwidth, whereas smaller models can operate within more efficient thermal and power envelopes. Empirical data from recent benchmarks clarifies this scale difference: for instance, a single inference with a 405B parameter model consumes approximately 6,706 Joules, while an 8B variant of the same family consumes only 114 Joules (11). This 59-fold difference in energy expenditure of a model 50 times larger reinforces our project's proposal that routing sub-tasks to smaller models is a viable path toward sustainable AI.

3.2 Task Decomposition in LLM Systems

The strategic shift from monolithic model usage to a decomposed "Planner-Worker" workflow is driven by the need for both accuracy and energy efficiency. Our methodology draws direct inspiration from the FrugalGPT framework and the concept of Speculative Cascades, both of which demonstrate that a "one-size-fits-all" approach to LLMs is inherently inefficient. FrugalGPT pioneered the use of LLM cascading, proving that routing queries to a sequence of models—starting with the smallest—can match the performance of flagship models like GPT-4 while reducing operational costs by up to 98% (5). Similarly, the Speculative Cascades approach highlights that modular systems can achieve high-quality results by utilizing specialized models for different stages of a request. Our project aligns with these findings by combining different scales of LLMs to create a more energy-aware application. By assigning complex reasoning to a capable planner and implementation to efficient workers, we minimize the "energy tax" associated with high-parameter models. Furthermore, evidence from Quality Assurance in Modular Systems confirms that such multi-model routing can reduce cumulative energy consumption by 30-40% without compromising the technical accuracy of the code produced (13). The data indicates that as model ecosystems mature, the potential for optimization through intelligent routing is vast, suggesting that our direction not only addresses current environmental concerns but also possesses significant room for further refinement as smaller models become increasingly capable.

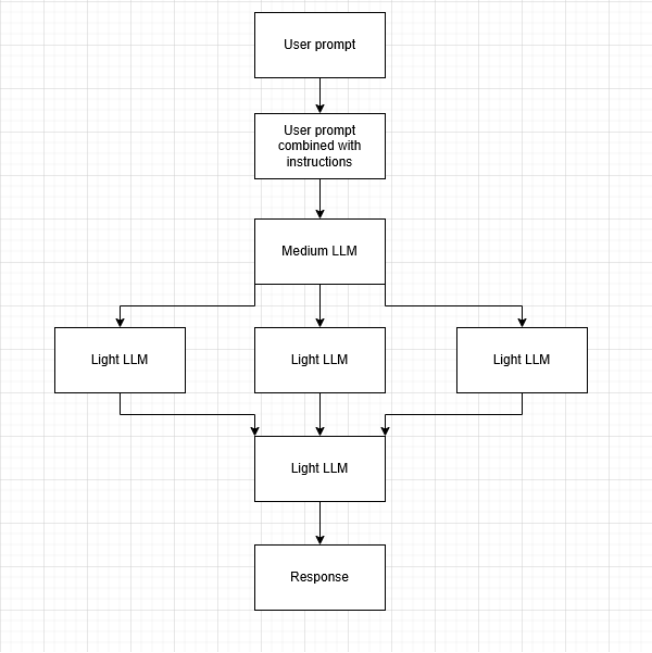

Figure 1: System pipeline

4 Methodology

4.1 System Architecture Overview

The proposed system involves two models: a "medium" and "light" LLM model. Upon receiving a user's input prompt, the medium model is instructed to split the task into few smaller tasks. These tasks are then completed by the smaller LLM models. After recombine all responses, the final answer will be return to the user. Notably, the medium model is significantly smaller than state-of-the-art large models (e.g., 27B+ parameters), allowing for a more energy-efficient orchestration of the pipeline.

4.2 Medium LLM

Our software uses Gemma3's 12 Billion parameter model as the medium LLM. In our software's pipeline, the LLM will split a task into multiple smaller tasks for the small LLM to perform.

4.2.1 Task 1

Upon receiving a user's prompt, the system gives the medium LLM the following instruction:

> You are the control LLM. Your task is to split the following code into up to 3 separate tasks. Write each task as a clear, unambiguous instruction. Each task must be self-contained and should not require information from the other tasks to be completed. Do keep in mind that you will have to reassemble the final answer from the 3 smaller parts, so try to split in a way that makes it easy to reassemble. Do not provide any features that are not explicitly requested in the original user message. Do not make up any functions without explaining how they work with code. Split the tasks with a newline character. Do not add anything else in your response. The code to split is:

The goal of the instruction above is the following:

1. Instruct the medium LLM to split the task into three smaller tasks
2. Each task should be clear and concise
3. Each task should be isolated and not dependent on other tasks
4. The tasks should be easy to reassemble into a single task
5. Each task should be on a new line
6. The response should contain nothing but the tasks

4.2.2 Task 2

Upon the light LLM's completion of their individual tasks, another light LLM is invoked to combine the responses of the previous models into a single response. The following prompt is given to the light LLM to guide it in combining the results:

> You are the control LLM. Your task is to take the 3 smaller parts generated by the smaller LLMs and reassemble them into a single, coherent response to the original user message. The 3 parts are:

Upon finalising the combination, the response is presented to the user.

4.3 Light LLM

Our software uses Gemma3's 4 Billion parameter model as the light LLM. The role of the light LLM is to perform a small segment of the original task. Due to the nature of a more complex problem being broken down into smaller, more concise and manageable chunks, a small, energy efficient LLM is often sufficient and more energy efficient for this task.

The light LLM is prompted with the following instruction:

> You are a smaller LLM. Your task is to generate a response for the given part of the code, based on the instructions provided. In your response, do not include any explanations or reasoning, just provide the answer for the given part as a single JSON string, without any additional text. The instructions are:

Upon performing the task, the response of the light LLM is passed on to another light LLM for recombination with the following prompt:

You are the control LLM. Your task is to take the 3 smaller parts generated by the smaller LLMs and reassemble them into a single, coherent response to the original user message. Return your answer as normal text and not a JSON string. Do not include any statements relating to the fact that you are recombining parts, just provide the final answer as if you were responding to the original user message. The 3 parts are:

4.4 System Setup

Despite enterprise hardware offering superior for large-scale computations in terms of energy consumption (14), benchmarks were run using consumer grade hardware because of local availability of the latter.

System details	
GPU	NVIDIA GeForce RTX 3080ti 16GB
CPU	i9-11900H
RAM	2 x 16GB DDR4

Table 1: System details used for benchmarking

EnergiBridge measurements were done under nominal conditions, where background activity was minimized to more accurately calculate the real energy consumption to be displayed to the user in the frontend of the tool.

4.5 Model Selection

As detailed in subsection 4.2 and subsection 4.3, the models selected for the tool are Gemma3 4 and 12 billion models. The initial selection of the Gemma models did not intend to select the absolute optimal models for the specific use case, but to use models of several sizes of the same family for its implementation and validation. Consequently, the architecture remains model-agnostic, allowing users to integrate models of their own liking.

5 Results and Analysis

5.1 Metrics

Before presenting the results, we briefly describe the metrics we use throughout our experiments. Table 2 lists them alongside their units. All energy-related metrics are collected using EnergiBridge.

Table 2: Metrics

Metric	Unit
Total Energy Consumed	J (Joules)
CPU Package Energy	J
GPU Energy	J
Average Power Draw	W (Watts)
Energy per Output Token	J/token

Total Energy Consumed is our primary metric. It is the sum of CPU package energy, DRAM energy, and GPU energy over an entire inference run. This single number tells us how much energy the system used in total to answer one prompt.

CPU Package Energy is the energy used by the processor and its on-die components. We get this by taking the difference in EnergiBridge's `PACKAGE_ENERGY` reading between the start and end of each run. It shows how much of the total energy cost comes from the CPU side.

GPU Energy is the energy consumed by the graphics card. Since LLM inference is mostly a GPU task, this is usually the largest part of the total. EnergiBridge records instantaneous GPU power (`GPU0_POWER`) at regular intervals, and we compute the total by adding up power × time across all intervals.

Average Power Draw is simply total energy divided by execution time. It tells us how hard the hardware is working at any given moment. Two runs might use the same total energy, but one could be running at high power for a short time while the other runs at low power for a long time; this metric helps us tell the difference.

Energy per Output Token divides the total energy by the number of tokens the model produced. This is useful because different strategies may produce different amounts of text. By looking at the cost per token rather than the cost per run, we can make a fairer comparison even when output lengths differ.

5.2 Energy Consumption: Planner-Worker Framework vs. Single Heavy Model Baseline

RQ1: *Does our planner-worker approach consume less energy than directly using a single large model to both reason about and implement the same coding task?*

Experiment Setup. We ran both strategies on the same hardware with the same coding task - a complex programming question which involves writing a REST API for a simple management system from scratch. This prompt was made more difficult through the requirement of an asynchronous approach to be taken, in addition to a library being required - this adds complexity serves to create a task prompt that is challenging and requires a lengthy output. Throughout the rest of this paper, we refer to the single-model baseline as **Strategy 1** and our planner-worker approach as **Strategy 2**. EnergiBridge recorded CPU package energy (`PACKAGE_ENERGY`) and GPU power (`GPU0_POWER`) throughout each run, sampling roughly every 200 ms. We compute the total energy per run as:

$$E_{\text{total}} = E_{\text{CPU}} + E_{\text{GPU}} = \Delta\texttt{PACKAGE\_ENERGY} + \frac{\sum_i P_{\text{GPU},i} \cdot \Delta t_i}{10^6} \quad (1)$$

where P_{GPU} is in milliwatts and Δt in milliseconds. In Strategy 1, the 27b model (`gemma3:27b`) handles the entire task on its own: it reads the prompt, reasons about it, and writes all the code. In Strategy 2, the 12b model acts as a planner: it only produces a short outline (capped at 250 tokens), and then the 4b model (`gemma3:4b`) does the actual code writing based on that outline. All measured values are summarised in Table 3.

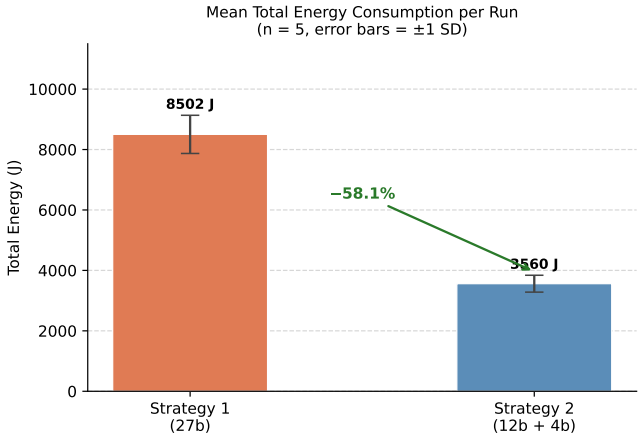

Figure 2: Mean total energy consumption per run for Strategy 1 and Strategy 2 (n=5, error bars = ±1 SD).

Total Energy. Looking at Table 3 (rows 1-6) and Figure 2, Strategy 1 used an average of **8501.5 J** per run, while Strategy 2 used only **3560.3 J** - a saving of about **58%**. What stands out is that the two strategies do not overlap at all: the best run under Strategy 1 (7560.2 J) still consumed more energy than the worst run under Strategy 2 (3943.7 J). This means the difference is not caused by lucky or unlucky runs; it is consistent across all repetitions. The median values are also very close to the means (S1: 8412 J, S2: 3565 J), which confirms there are no outliers pulling the averages in either direction.

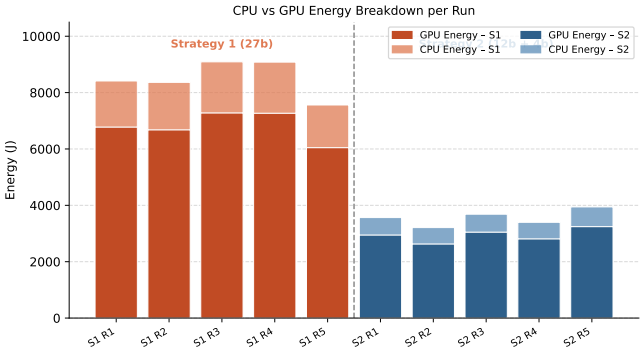

Figure 3: CPU vs. GPU energy breakdown for each individual run across both strategies.

CPU and GPU Breakdown. Figure 3 show where the energy actually goes. In Strategy 1, the GPU used about **6810 J**, making up roughly **80%** of the total, while the CPU used **1691 J**. In Strategy 2, GPU energy dropped to **2936 J** (↓56.9%) and CPU energy to **624 J** (↓63.1%). This makes sense: LLM inference is mostly a GPU task, and since the 4b worker model is much smaller than the 27b model, it simply needs less GPU power to run. The CPU saving is even larger in percentage terms, because a smaller model also requires less data movement and scheduling overhead.

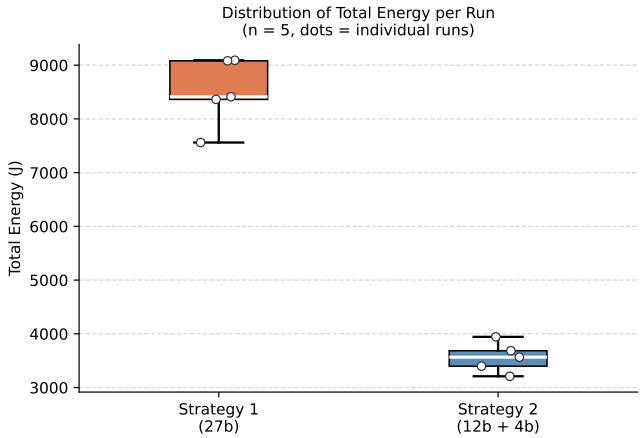

Figure 4: Distribution of total energy per run (n=5, dots = individual runs).

Stability. Figure 4 shows the distribution of total energy across all 5 runs for each strategy. The most striking observation is that the two boxes do not overlap at all - even the lowest point of Strategy 1 (around 7560 J) sits well above the highest point of Strategy 2 (around 3944 J). This gap of over 3600 J between the best-case S1 run and the worst-case S2 run confirms that the energy difference is not a matter of lucky or unlucky runs, but a consistent, repeatable result.

Looking more closely at the shape of each box, Strategy 2's distribution is noticeably tighter. Its IQR (the height of the box, covering the middle 50% of runs) is only **287.7 J**, compared to **717.9 J** for Strategy 1 (row 4 of Table 3) - a reduction of about 60%. The individual dots in Strategy 2 are clustered closely together between roughly 3200 and 3950 J, while Strategy 1's dots are spread across a wider range from about 7560 to 9090 J. The standard deviation follows the same pattern, dropping from 631.8 J to 279.2 J (row 3).

When we look at the coefficient of variation (which measures spread relative to the mean) both strategies are actually similar (S1: 7.43%, S2: 7.84%, row 17). This means Strategy 2 is not inherently more stable in a relative sense; its box just looks tighter because the numbers themselves are much smaller. In practical terms, though, the smaller absolute spread is what matters: if you want to predict how much energy a batch of 100 prompts will cost, Strategy 2 gives you a much narrower range to work with.

Energy per Token. To make a fair comparison even if the two strategies produce different amounts of text, we calcu-

Table 3: All measured metrics for Strategy 1 (single 27b model) and Strategy 2 (12b planner + 4b workers), averaged over 5 runs.

#	Metric	S1 (27b)	S2 (12b+4b)	Change	Note
1	Total Energy Mean (J)	8501.53	3560.34	↓58.1%	Main result; S2 uses less than half the energy
2	Total Energy Median (J)	8412.27	3565.41	↓57.6%	Close to the mean; no outlier issues
3	Energy Std Dev (J)	631.78	279.20	↓55.8%	S2 results are more consistent
4	Energy IQR (J)	717.86	287.67	↓59.9%	Middle 50% of runs are tightly grouped
5	Max Energy (J)	9090.98	3943.67	↓56.6%	Even the worst S2 run beats the S1 average
6	Min Energy (J)	7560.16	3209.99	↓57.5%	The best S1 run is still above the worst S2 run
7	CPU Package Energy (J)	1691.17	624.23	↓63.1%	Smaller model needs less from the CPU
8	GPU Energy (J)	6810.36	2936.11	↓56.9%	GPU does most of the work; biggest saving here
9	Average Power Draw (W)	97.37	86.95	↓10.7%	S2 puts less load on the hardware
10	Execution Time Mean (s)	87.29	40.94	↓53.1%	S2 finishes in about half the time
11	Execution Time Median (s)	86.64	40.42	↓53.3%	Consistent with the mean
12	Total Output Tokens	1966.0	1495.6	↓23.9%	Total output is also somewhat shorter
13	Energy per Token (J/tok)	4.324	2.381	↓44.9%	Each token is cheaper to produce in S2
14	Token Throughput (tok/s)	22.52	36.53	↑62.2%	S2 generates tokens faster
15	Small Model Token Share	0%	69.0%	–	Most of S2's output comes from the 4b model
16	Coeff. of Variation (%)	7.43%	7.84%	+5.5%	Both strategies fluctuate by a similar amount

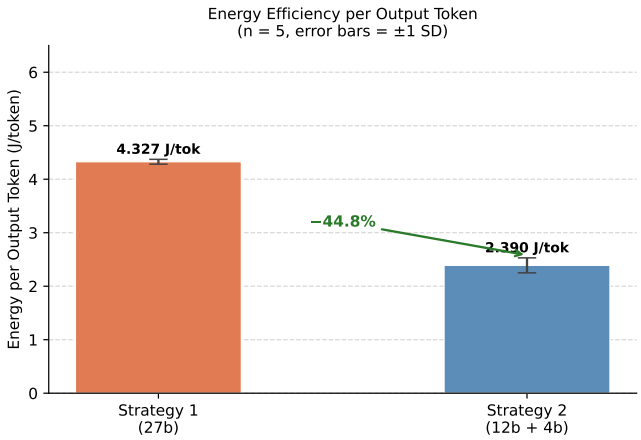

Figure 5: Energy efficiency per output token for both strategies (n=5, error bars = ±1 SD).

lated the energy cost per output token ((Figure 5 and row 14 of Table 3)). Strategy 1 generated an average of 1966 tokens at **4.327 J per token**. Strategy 2 generated 1495.6 tokens in total (across both models) at **2.390 J per token** - a **44.8%** reduction. This shows that the energy advantage is not just because S2 writes less text; each token is genuinely cheaper to produce.

Why Does This Work? The key is in rows 12 and 16. Under Strategy 1, the 27b model produces all 1966 tokens by itself. Under Strategy 2, the 12b model only produces **461.8 tokens** (the plan), which is a **76.5%** reduction. The remaining **69%** of the output is generated by the 4b worker, which uses far less energy per token. In other words, we are using the expensive model for a small but important job (thinking through the problem), and handing the bulk of the writing to a cheap model that can follow clear instructions. This is essentially the same idea behind FrugalGPT (5) - route work to the smallest model that can handle it - applied specifically to software engineering tasks.

5.3 Answer Generation Quality

RQ2: *Does the Planner-Worker approach (Strategy 2) maintain acceptable output quality compared to directly prompting a single large model (Strategy 1)?*

Evaluation Rubric. Since we did not have an automated test suite for the benchmark task, we evaluated output quality by hand using a structured rubric. Two reviewers independently scored each of the ten outputs (five per strategy) against ten criteria that come directly from the prompt requirements. Each criterion is scored 0, 1, or 2, giving a maximum of 20 points per run. The full rubric is shown in Figure 6.

Results. Figure 7 and Figure 8 show the per-run scores for both strategies across all ten criteria.

Discussion. Overall, the two strategies produce very similar quality. Strategy 1 scored an average of **16.2/20** (σ = 2.05) and Strategy 2 scored **16.0/20** (σ = 1.87), a difference of just 0.2 points. That gap is well within one standard deviation of either distribution, so it is not meaningful in practice. Combined with the 58.1% energy reduction from Section 5.2, this supports our main claim: the planner-worker approach saves a lot of energy without noticeably hurting output quality.

#	Criterion	2 pts	1 pt	0 pts
C1	Database Setup	SQLite URL, create_engine, declarative_base, create_all, per-request session all correct	Functional but simplified session (global session or missing create_all)	Absent or fundamentally broken
C2	Task Model Fields	All 5 fields (id, title, description, completed, created_at) with correct SQLAlchemy types	All 5 fields named but type/column definitions incorrect	Fewer than 4 fields
C3	Create Endpoint	POST /tasks: instantiates Task, commits, returns result	Exists with correct intent but has bugs	Absent
C4	Read Endpoints	Both GET /tasks and GET /tasks/{id} present and correct	Only one present	Both absent
C5	Update Endpoint	PUT /tasks/{id}: partial update, 404, commit	Exists but has bugs	Absent
C6	Delete Endpoint	DELETE /tasks/{id}: removes record, 404, commit	Exists but has bugs	Absent
C7	Pydantic Validation	TaskCreate, TaskUpdate, TaskResponse defined and applied as response_model on all endpoints	Some models defined or coverage incomplete	No Pydantic models
C8	404 Error Handling	HTTPException(404) in all ID-based endpoints (read, update, delete)	Present in some but not all	None
C9	Code-to-Output Ratio	≥80% of response is code, no unrequested prose	Code primary but accompanied by substantial prose (40–70% code)	Prose dominates
C10	Prompt Instruction Adherence	Delivers exactly what prompt requested, no unrequested additions	Code provided but supplemented with explanations not requested	Does not deliver requested code

Figure 6: Output evaluation rubric. Each of the ten criteria is scored on a 0 to 2 scale based on how well the output meets the corresponding prompt requirement.

Run	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	Total
Run 1	2	2	2	2	2	2	2	2	1	1	18
Run 2	1	2	2	2	2	2	2	2	1	1	17
Run 3	2	2	2	2	2	2	2	2	1	1	18
Run 4	1	2	1	2	1	1	2	2	1	1	14
Run 5	2	2	1	2	1	1	1	2	1	1	14
Mean	1.6	2.0	1.6	2.0	1.6	1.6	1.8	2.0	1.0	1.0	**16.2**
SD	0.55	0.00	0.55	0.00	0.55	0.55	0.45	0.00	0.00	0.00	**2.05**

Figure 7: Per-run scores for Strategy 1 across all ten rubric criteria.

An additional experiment was conducted on the validity of the model's results, which can be found here. This experiment utilised a wide range of programming questions from ProgrammingHero's 100-plus-python-coding-problems-with-solutions dataset, which included tasks such as writing code to calculate functions, performing specific list operations and building simple logic-based games. Our results show that the model produced a result that was deemed valid 74% of the time.

Run	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	Total
Run 1	1	2	2	2	2	2	2	2	2	2	19
Run 2	0	1	1	2	2	2	2	2	2	2	16
Run 3	1	2	1	2	1	1	2	1	2	2	15
Run 4	0	1	1	2	2	2	2	2	2	2	16
Run 5	1	1	1	2	1	1	1	2	2	2	14
Mean	0.6	1.4	1.2	2.0	1.6	1.6	1.8	1.8	2.0	2.0	**16.0**
SD	0.55	0.55	0.45	0.00	0.55	0.55	0.45	0.45	0.00	0.00	**1.87**

Figure 8: Per-run scores for Strategy 2 across all ten rubric criteria.

Both strategies cover the functional requirements equally well. On the criteria that test whether the output actually implements what was asked read endpoints (C4), Pydantic validation (C7), and 404 error handling (C8), both strategies score the same or nearly the same (C4: 2.0/2.0; C7: 1.8/1.8; C8: 2.0 vs. 1.8). Both models consistently implemented all four CRUD operations in every run. This shows that the 4b worker, when given a clear plan from the 12b planner, understands the scope of the task just as well as the 27b model working alone.

Strategy 1 is slightly better at low-level implementation details. The main difference shows up in C1 (Database Setup) and C2 (Task Model Fields), where Strategy 1 scores 1.6 and 2.0 compared to Strategy 2's 0.6 and 1.4. Strategy 1 more reliably produced correct SQLAlchemy session handling (e.g., using `Depends(get_db)` for dependency injection) and valid column type definitions. Strategy 2's 4b worker sometimes made mistakes with library-specific APIs: for example, using `engine.connect()` instead of a proper session object, or forgetting to import certain SQLAlchemy modules. This suggests that smaller models, even with a detailed plan to follow, are more likely to slip up on low-level library usage.

Strategy 2 follows the output format instructions more closely. Strategy 2 scored a perfect 2.0/2.0 on both C9 (Code-to-Output Ratio) and C10 (Prompt Instruction Adherence) in every single run. All of its outputs were pure code with no extra commentary. Strategy 1, on the other hand, added long "key improvements and explanations" sections to every output text that was never asked for and that inflates the token count without adding anything useful. Strategy 1 scored only 1.0/2.0 on both C9 and C10 across all runs. This is an interesting pattern: the smaller worker model sticks precisely to what it was told to do, while the bigger model tends to over-explain.

Strategy 1 is less consistent between runs. Strategy 1 produced the two highest individual scores (Runs 1 and 3, both 18/20), but also two of the lowest (Runs 4 and 5, both

14/20), caused by bugs like a shadowed class name and an incorrect standard library call. Strategy 2's scores are more tightly clustered (range: 14-19 vs. 14-18 for Strategy 1). This suggests that the planning step adds some structural consistency to the output, even if it does not eliminate all implementation errors.

Summary. Strategy 2 trades a small drop in low-level implementation correctness (C1, C2) for better format compliance (C9, C10), with everything else being essentially equal. The overall quality difference is negligible (0.2 points on a 20-point scale), while the energy saving is substantial (58.1%). This tells us that the planner-worker approach is a practical, energy-efficient alternative that does not compromise the usefulness of the generated code.

6 Discussion

6.1 Practical Implications

Our results show a 58% reduction in energy consumption for a single coding task. To understand what this means at scale, consider that AI coding assistant adoption has reached 91% according to the DX Q4 2025 report surveying 121,000 developers (15), and GitHub Copilot alone has surpassed 20 million users (16). The Stack Overflow 2025 survey confirms that 51% of professional developers use AI tools daily (1).

In our experiment, Strategy 2 saved approximately **4,941 J** per prompt. If a developer sends a modest 30 code-related prompts per working day, that amounts to roughly 148,000 J (0.041 kWh) saved daily, or about **10.3 kWh per developer per year**. If just 1 million developers adopted this approach, the cumulative saving would reach **10 GWh** annually, roughly the electricity consumption of 3,000 European households.

6.2 Limitations and future work

Our experiment was conducted using a single coding prompt. While this limits the generalisability of our findings, the task was not a trivial one: rather than asking the model to write a single function or sort a list, we asked it to implement a complete REST API endpoint with request validation, database interaction, and error handling. This is closer to what developers encounter in real-world work. Still, we cannot be sure that the 58% energy saving we observed would hold for other types of tasks, for example, pure algorithmic problems, large-scale refactoring, or multi-file debugging, and future work should evaluate the approach across a more diverse set of prompts.

We also tested only one model combination: Gemma 3 12b as the planner and Gemma 3 4b as the worker. Gemma 3 is a widely adopted open model family: at the time of writing, the Gemma family has surpassed 200 million downloads (17), and its 4B instruction-tuned variant matches the performance of the much larger Gemma 2 27B on several benchmarks. On LMArena, Gemma 3 27B achieved an Elo score of 1338, outperforming even Llama 3.1 405B (3). So our choice of model is far from obscure. However, different model families have different strengths and weaknesses, and the energy savings we observed may partly reflect how well Gemma 3 specifically handles the planner-worker splitt. Testing with other combinations - for instance, using a code-specialised model like Qwen2.5-Coder as the worker would help establish whether our results generalise beyond this particular pairing.

Another limitation is our approach to evaluating output quality. We defined a manual rubric to compare the outputs of Strategy 1 and Strategy 2, but this process was not automated. Without a standardised, execution-based benchmark such as HumanEval or MBPP with pass@1 scoring, our quality assessment is difficult to reproduce and may be influenced by subjective judgement. Future work should integrate automated evaluation to allow for repeatable and objective quality comparisons. Furthermore, EnergiBrdige reads the MSR registers for the energy calculations. Based on the background activity of the user, the readings will give out a slightly higher energy consumption than the actual consumption of the model alone; the actual results will even slightly better, because of the 'DC' value.

Finally, our planner always splits the user's task into exactly three sub-tasks, regardless of the task's complexity. For a complex task that involves many interdependent steps, three sub-tasks may not provide enough granularity, potentially forcing the worker to handle instructions that are still too broad. Conversely, for a simple task such as writing a single utility function, splitting into three parts is unnecessary overhead that adds extra planner and aggregation calls without meaningful benefit. While we attempted to get the heavy LLM to produce an empty instruction string when it did not deem further models necessary, we found that the heavy LLM did not consistently use this feature, and thus it was taken out. A more flexible approach would let the planner decide how many sub-tasks to create based on the complexity of the input, which we leave as future work.

7 Conclusion

This paper addressed the growing environmental impact of AI-assisted software engineering by proposing a decomposed Planner-Worker architecture. Our experimental results demonstrate that the proposed framework effectively reduces the energy footprint of Large Language Model (LLM) inference without compromising the functional quality of the generated code.

The primary findings of this research are as follows:

- **Significant Energy Efficiency:** Strategy 2 (Planner-Worker) achieved a 58.1% reduction in mean total energy consumption compared to the single-model baseline. This saving is driven by a 56.9% decrease in GPU energy demand and a 63.1% reduction in CPU package energy.
- **Maintained Output Quality:** Despite the shift to

smaller worker models, response quality remained high. The difference in rubric-based scores between the single heavy model (16.2/20) and our strategy (16.0/20) was a negligible 0.2 points, falling well within one standard deviation.

- **Improved Latency:** Our approach cut execution time by approximately 53%, from 87.2 s to 40.9 s, effectively removing the traditional trade-off between sustainability and performance.

Furthermore, the modular nature of our system architecture provides high extensibility. By decoupling the planning phase from execution, the tool allows for seamless model substitution as the LLM landscape evolves. Future developers can easily replace the current Gemma 3 components with specialized models, such as code-specific variants or newer efficient architectures, to further optimize the energy-to-quality ratio. Ultimately, this work provides a practical and scalable path toward sustainable AI-assisted development, proving that intelligent task routing is a repeatable and consistent method for reducing the "energy tax" of modern software engineering tools.

References

[1] Stack Overflow, "2025 developer survey: AI," 2025. [Online]. Available: https://survey.stackoverflow.co/2025/ai/

[2] International Energy Agency, "Energy and AI," IEA, Paris, Tech. Rep., 2025. [Online]. Available: https://www.iea.org/reports/energy-and-ai

[3] Gemma Team, Google DeepMind, "Gemma 3 technical report," *arXiv preprint arXiv:2503.19786*, 2025.

[4] Z. Yang, K. Adamek, and W. Armour, "TokenPowerBench: Benchmarking the power consumption of LLM inference," *arXiv preprint arXiv:2512.03024*, 2025.

[5] L. Chen, M. Zaharia, and J. Zou, "FrugalGPT: How to use large language models while reducing cost and improving performance," *arXiv preprint arXiv:2305.05176*, 2023.

[6] JetBrains, "The state of developer ecosystem 2025: Coding in the age of AI," 2025. [Online]. Available: https://blog.jetbrains.com/research/2025/10/state-of-developer-ecosystem-2025/

[7] Second Talent, "AI coding assistant statistics & trends," 2025. [Online]. Available: https://www.secondtalent.com/resources/ai-coding-assistant-statistics/

[8] Pew Research Center, "What we know about energy use at US data centers amid the AI boom," 2025, available at: https://www.pewresearch.org/short-reads/2025/10/24/what-we-know-about-energy-use-at-us-data-centers-amid-the-ai-boom/.

[9] Z. Yang, K. Adamek, and W. Armour, "How hungry is AI? benchmarking energy, water, and carbon footprint of LLM inference," *arXiv preprint arXiv:2505.09598*, 2025.

[10] World Economic Forum, "How data centres can avoid doubling their energy use by 2030," 2025. [Online]. Available: https://www.weforum.org/stories/2025/12/data-centres-and-energy-demand/

[11] Z. Yang, K. Adamek, and W. Armour, "TokenPowerBench: Benchmarking the power consumption of LLM inference," *arXiv preprint arXiv:2512.03024*, 2025. [Online]. Available: https://arxiv.org/abs/2512.03024

[12] I. Malavolta, "On-device or remote? on the energy efficiency of fetching LLM-generated content," in *Proceedings of the 4th IEEE/ACM International Conference on AI Engineering – Software Engineering for AI (CAIN)*, 2025. [Online]. Available: http://www.ivanomalavolta.com/files/papers/CAIN_2025.pdf

[13] T. M. Ziller, "A dynamic routing approach for sustainable language model inference," Ph.D. dissertation, Technische Universität Wien, 2025.

[14] A. R. d. Almeida, "Idle consumer gpus as a complement to enterprise hardware for llm inference: Performance, cost and carbon analysis," in *Proceedings of the 2025 6th International Artificial Intelligence and Blockchain Conference*, 2025, pp. 23–30.

[15] DX, "AI-assisted engineering: Q4 impact report," 2025. [Online]. Available: https://getdx.com/blog/ai-assisted-engineering-q4-impact-report-2025/

[16] GitHub, "GitHub Copilot statistics and adoption trends," 2025, 20 million users reported as of July 2025; 150 million registered developers on GitHub. [Online]. Available: https://github.com

[17] Google, "Introducing Gemma 3 270M: The compact model for hyper-efficient AI," 2025, reports that Gemma downloads surpassed 200 million as of August 2025. [Online]. Available: https://developers.googleblog.com/en/introducing-gemma-3-270m/