Multi-Model Routing for Energy-Efficient LLM Code Generation

Jia-Jie Michael Chan

Delft University of Technology

TU Delft

Multi-Model Routing for Energy-Efficient LLM Code Generation

by

Jia-Jie Michael Chan

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday April 28, 2026 at 15:30.

Student number:	4953770	
Project duration:	September 1, 2025 – April 28, 2026	
Thesis committee:	Arie van Deursen	Thesis advisor
	Luis Miranda da Cruz,	Daily supervisor
	Enrique Barba Roque	Daily co-supervisor
	Jie Yang	External examiner

Cover:	Canadarm 2 Robotic Arm Grapples SpaceX Dragon by NASA under CC BY-NC 2.0 (Modified)
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

I would like to sincerely thank Enrique for his continuous guidance throughout the process of this thesis over the past year and making time for me every week. I would also like to thank Luis for his support and expertise despite his busy schedule and wish him and his family all the best in the future.

Jia-Jie Michael Chan
Delft, April 2026

Summary

The introduction of large language models (LLMs) has transformed the way software is written. With the help of LLM powered code generation the productivity of software engineers has increased all over the world. However, these models are also computationally expensive. The ubiquitous use of these models has raised significant sustainability concerns.

LLM routing aims to reduce the usage of more complex models by routing easier tasks to smaller models. However, existing research on routing primarily focuses on monetary savings and the potential for routing from a sustainability perspective has yet to be explored.

In this thesis we propose an energy-aware LLM routing framework to measure, train and evaluate various routers. We implement our framework and conduct experiments to quantify the energy efficiency of routing and to examine the trade-offs between accuracy and energy consumption. Furthermore, we analyze the overhead introduced by the various routing components. Our results show that routing can reduce energy consumption by up to 15.3% on the HumanEval and MBPP dataset with minimal overhead when compared to a interpolated baseline. However, overall energy savings were found to decrease significantly as we aim for accuracy targets near the stronger model. These findings show that LLM routing is a viable strategy to reduce energy consumption of LLM code generation in scenarios where achieving maximum performance is not crucial.

Contents

1 Introduction

1.1. Introduction

Since the introduction of large language models (LLMs), they have become increasingly useful in many fields including software development. As a result, the use of generative AI has increased significantly, with OpenAI reporting over 900 million active weekly users for ChatGPT alone [27].

However, their massive size and frequent use have also led to a very high energy consumption. The estimated annual energy consumption for GPT4-o inference alone was approximately 391.5-463.3 GWh [16]. To put this into perspective, the average energy consumption of a U.S. household was approximately 10,791 kWh in 2023 [41], which means that the amount of energy consumed by GPT-4-o would roughly correspond to the electricity use of more than 36,000 U.S. households. This raises serious sustainability concerns.

Although these models boast impressive performance on challenging tasks, not every task will require a state-of-the-art model. The current uniform deployment of large models means that even for simple tasks, the most powerful model is used. This is energy-inefficient when a smaller and more efficient model could have performed the same task as well. This highlights the need for dynamic model selection strategies that balance accuracy and energy costs.

LLM model routing is a relatively recent approach that addresses the problem of inefficient inference by dynamically selecting the most suitable model among a pool of pretrained candidate models of varying sizes. The objective is to minimize overall cost without compromising accuracy by directing simpler inputs to smaller models while reserving larger models for more complex cases.

This thesis explores the viability of LLM routing for energy-efficient code generation. To do this, we present two different router designs and propose a framework for the evaluation and training of various energy-aware routers. We apply this framework and quantify the potential energy savings of this approach by conducting experiments using open-source and locally hosted models for code generation tasks.

1.2. Problem Statement

Although recent advances in model routing have successfully improved efficiency by dynamically routing queries to models of different sizes, current approaches primarily focus on monetary cost and accuracy. For example, MixLLM [43] and MetaLLM [26] reduce inference costs by routing simpler inputs to smaller models, but their research objectives are tied to the financial cost rather than energy usage. Few studies mention energy consumption as a routing metric, and even fewer incorporate energy measurements into routing decisions. One of the potential reasons for this gap is the lack of available datasets with energy consumption data, which limits the ability to train and evaluate energy-aware routing strategies effectively.

1.3. Research Questions

The goal of this thesis is to investigate how routing strategies can be leveraged to improve the energy efficiency of large language models (LLMs) in the context of Python code generation. To achieve this, we focus on three key research questions.

(RQ.1) Research Question 1

How effective is predictive routing in improving the energy efficiency of LLM code generation?

Our first question focuses on understanding whether predictive routing models can meaningfully reduce the energy consumed by large language models when generating code. To do this, we will implement several routers and compare their performance and energy consumption with a single model baseline.

(RQ.2) Research Question 2

What is the trade-off between energy consumption and accuracy?

Our second question looks at how routing balances energy consumption and overall accuracy. We investigate for which performance targets routing is efficient to identify in which scenarios routing could be desired. We aim to answer this question by creating routers at various levels of accuracy.

(RQ.3) Research Question 3

What is the energy overhead of the different router components?

Finally, we determine the overhead that is introduced from the various router components to determine whether the benefits of routing outweigh the costs and to identify the components that can be scaled and those that require caution. To investigate this, we individually measure the energy consumption of each different router component.

Through these research questions, we aim to learn more about the potential benefits and weaknesses of energy-aware LLM routing for code generation tasks.

1.4. Scope

This thesis focuses on evaluating routing strategies for large language models. We limit our domain specifically code generation given a natural language input prompt or code snippet. The experiments are restricted to Python programming tasks. Furthermore, the candidate model pool for our router is restricted to a binary setting, where our router selects between a weaker and a stronger model. While it is common to consider larger pools of models, limiting the problem to a binary setting reduces the computational and hardware requirements for this thesis.

Our primary evaluation metrics are accuracy, determined by functional correctness, and energy consumption during inference. We do not consider training, networking or evaluation costs. Furthermore, our scope is limited to single GPU systems.

1.5. Contributions

This thesis investigates the effectiveness of two routing strategies in the context of code generation. The main contributions of this work are as follows:

- We propose two different routing strategies to reduce energy consumption for LLM code generation.
- We provide insight into the trade-offs between router accuracy and energy consumption.
- We provide an experimental framework and dataset that enables the training and evaluation of routing strategies for energy-efficient code generation by adapting the RouterBench [14] framework to energy consumption.

- We analyze the overhead costs of routing systems and determine the scalability of the router.

1.6. Document Outline

This thesis is divided into 8 Chapters. Chapter 1, the current Chapter, provides the motivation behind this work and our research questions. Chapter 2 explains the background upon which this thesis is built. In Chapter 3 we discuss prior related work on model routing. Chapter 4 covers the methodology and presents the base framework of our research. Chapter 5 presents our implementation details of the framework and our experiment. The results of the experiment can be found in Chapter 6. In Chapter 7 we discuss the implications of the results, the potential shortcomings of our research, and future work. Finally, we conclude our thesis in Chapter 8.

2

Background

In this Chapter we cover the background of some of the tools used for our thesis. We first briefly explain large language models and the process of code generation using them, we then cover the primary datasets used for code generation as well as some of their drawbacks and finally we explain the various ways of measuring energy consumption.

2.1. Transformers and LLMs

Since the reveal of the transformer architecture in 2017 by Vaswani et al. [42], natural language processing has forever changed. This new architecture differed from all previous neural networks with the introduction of self-attention layers. This has allowed for each token in a sequence to be influenced by all other tokens previously found in the sequence. By computing the attention weights, the model determines how important each token is compared to the others in the sentence.

Large language models implement the transformer architecture to learn next token prediction [31]. Given a sequence of tokens, LLMs predict the probabilities for the next token given the combined output of all previous tokens in the sequence. To generate entire sentences autoregression is used. Starting from the initial input prompt, the next token predicted and the new result is appended to the prompt to then be used as input for the next iteration. This is repeated until a stop token is generated or the token limit is reached. This autoregression process is can be computationally expensive. To avoid some recomputation, key-value (KV) caching is employed to store some intermediate representations from the computation of previous tokens.

2.2. Embedding

Embedding is the process of representing text as a feature vector. Unlike traditional tabulated data with clear numerical and categorical features, representing raw text as a sensible list of numbers of fixed length is challenging, especially given that text may vary in length considerably. Since the input data for large language models are natural language prompts, our router will operate on the same data. As such, embedding is a crucial step for our router to enable informed routing decisions.

Early implementations of embeddings includes Bag of Words and One-Hot vectors. This involved creating a vocabulary and simply assigning a vector corresponding to each word in the list. However, these implementations did not take into account the semantics of each word which resulted in very sparse vectors.

In 2013, Mikolov et al. introduced Word2Vec [25] as a new way of representing natural language as a vector. Rather than naively assigning a vector to each word, Word2Vec trains a model to predict words given the other words located nearby. As a result, words that appear frequently in the same context and are semantically similar are given similar vectors. However Word2Vec does not take into account that word semantics may change depending on the context.

With the released BERT [10] in 2018, contextual embeddings were now possible. By leveraging the

attention layers of the transformer architecture introduced a year earlier, words are now given embeddings based on the semantics of the word within the full surrounding context. As a result the embeddings now capture the actual meaning of the word as was intended. However, the training of such embedding models requires a significant amount of text to accurately capture the semantics of each word in context. BERT was trained on over 3 billion words sourced from Wikipedia and BooksCorpus.

While the cost of training transformer based embedding models is high, there exist many pre-trained models. In this thesis we leverage such pre-trained embedding models. They form the basis for our router's ability to comprehend code related prompts.

2.3. Code Generation with LLMs

With the release of the first large language models it was also possible to generate code. After all, source code itself is just readable text instructions structured in a way such that it can be easily compiled into a program. However, since they were not specifically trained on code the results were mixed. Codex [6] improved upon this by finetuning a GPT language model on public GitHub code. A production version of this model was subsequently used to power GitHub Copilot, a widely adopted code autocompletion tool.

2.3.1. Datasets

To train and evaluate a router we require data. In this section we will cover some of the datasets commonly used to evaluate LLMs code generation and discuss some of the benefits and drawbacks of those datasets.

HumanEval

HumanEval is a benchmark introduced by OpenAI in July 2021 to evaluate large language models trained on code [6]. It consists of 164 hand-written Python programming problems with verified unit tests used for automatic function evaluation. Each task includes a function signature with docstring that describes the required behavior of the function. It also includes the library imports that the model may use. The model is tasked to complete the function and is then tested against the hidden test cases.

MBPP

Mostly Basic Programming Problems (MBPP) was released one month after HumanEval in August 2021 by Google [2]. However, their approach was to crowd-source a large set of programming questions while only a smaller set of questions were edited and verified by hand by the authors. As such MBPP contains a total of 974 Python problems of which 500 are dedicated to the test set. The remaining 474 questions are used for training and verification. MBPP tasks contain a short plain text instruction for the model, followed by the actual test case assert statements that the model has to pass. Unlike HumanEval where the test cases are hidden, MBPP requires the model to infer information from the test case to determine the function signature and exact behavior.

Limitations

While MBPP and HumanEval are widely used for evaluating code generation models, several studies have highlighted their limitations and potential shortcomings, which should be considered when using these benchmarks.

A known limitation of the original benchmarks is the relative simplicity of the coding tasks. HumanEval Pro and MBPP Pro were introduced in 2025 by Yu et al [47]. which feature more complex code generation problems that require models to reason and reuse previously generated code. o1-mini for example scored over 96% on the original HumanEval benchmark, but saw performance drop to around 76% on HumanEval Pro, showing that simple function completion tests may not generalize to more complex tasks.

A second concern was the robustness of the unit tests. Analysis of execution traces on these benchmarks reveal that the provided test suites often have high statement coverage but low branch and mutation coverage [21], meaning they may fail to catch many logical errors and edge cases in the generated code. As a result the model may pass all provided tests without truly fully capturing the intended behavior of the function, leading to inflated performance metrics. EvalPlus [20] introduced MBPP+ and

HumanEval+, a smaller subset of the original MBPP and HumanEval benchmarks with significantly increased test coverage.

However, these derivations of HumanEval and MBPP contain significantly fewer samples and are not commonly used in prior work, which affects comparability.

2.4. Energy Consumption of LLMs

Prior work has shown that the energy consumption of large language models (LLMs) is highly task-dependent [1], with model size and output token length being a primary indicator for energy usage. However, several other factors were also found to influence energy efficiency [3]. These factors include input token length, batch size, and the size of the KV cache. Furthermore, the availability of sufficient GPU memory for the model itself is of utmost importance. When GPU memory is insufficient, parts may be offloaded to the CPU, leading to significant reductions in efficiency and substantial increases in energy consumption.

Given that output token length is a primary indicator for energy consumption, limiting the number of output tokens for downstream tasks is an efficient way of improving energy efficiency. Solovyena et al. [35] identify that ”babbling” behavior occurs in several large language models on code generation tasks. This leads to excessive token generation and energy consumption after the downstream task has already been completed. To solve this they devise an algorithm that continually attempts to run the provided test cases after each generated line, such that the generation may immediately be halted when passing the downstream task. However, this approach has several downsides. First, in real deployment settings, test cases will rarely be available to stop the token generation early, and second, when the generated code is incorrect, it will not stop the model from ”babbling”.

2.5. Measuring Energy

The process of measuring and identifying the energy consumption of software and LLM's has gained increasing attention as energy efficiency becomes more of a concern. However, the process of measuring energy can vary significantly between researchers. This section addresses the main metrics and methods.

2.5.1. Energy and Power Metrics

In this thesis, we will primarily be encountering the metrics Power and Energy, in watts (W) and joules (J) respectively. Watts are the number of joules per second. For the cost of model inference given a task, we are interested in the total amount of energy consumed in joules as inference is not a continuous process. However, many tools only sample the power draw at certain points in time. To obtain the energy consumed we can calculate the average power measured during the task interval and multiply by the amount of time the task took. However, this relies on the power draw to remain relatively consistent to be accurate [23]. A more suitable approach is to approximate the integral using the trapezoidal rule. A visualization can be found in Figure 2.1.

Given power measurement samples P at fixed intervals Δt, we can compute the energy by taking the sum of all measurement points in the interval and substract half of the first and last measurements. We then multiply by the sampling interval.

$$[H]E \approx \Delta t \left[\left(\sum_{i=0}^{n} P_i \right) - \frac{1}{2} P_0 - \frac{1}{2} P_n \right] \tag{2.1}$$

However, the start and end points of the interval of interest may not align exactly with measurement points. In such cases we can linearly interpolate the edges using the first measurement sample right before and after the interval.

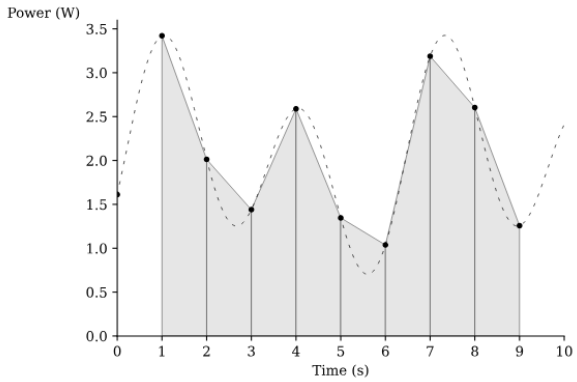


Figure 2.1: Trapezoidal rule integral approximation, image source: Cruz et al. [23]

2.5.2. FLOPS as proxy

Floating point operations (FLOPS) form the backbone of machine learning models. As a result, the number of FLOPS required to perform a task is a common metric for measuring the computational costs of a given model. As such, it could potentially be used as a proxy for energy when direct measurements are not feasible. However, while computational cost and energy consumption may be related, prior research has found that this relationship is not always clear [28, 19].

2.5.3. Energy Profilers

One way of measuring energy is through actual power meters. While they are the most accurate [9], they are not always practical. One of the challenges of using such power meters is aligning your experiment. Instead we can use software based energy profilers that approximate energy usage of the system. These profilers rely on internal hardware telemetry and is therefore highly hardware and operating system dependent, with various hardware manufacturers implementing their own software libraries such as RAPL (Intel), MSR (AMD) and NVML (Nvidia). This makes finding the right energy measurement tool not always a straightforward process.

EnergiBridge

EnergiBridge [32] is a program that provides a unified interface for the measuring and tracking of energy consumption within a system across different hardware platforms. Energibridge provides various energy related metrics for both CPU and GPU as well as general system information such as RAM and VRAM usage. It is designed to make energy measurement more accessible by providing a single API that abstracts away the different platform-specific underlying libraries such as RAPL for Intel, MSR for AMD, and NVML for NVIDIA. EnergiBridge is especially well suited for experimental studies as it allows researchers to collect accurate energy data with a single interface. The exact metrics provided by EnergiBridge depends on the system configuration as the different underlying libraries provide different metrics.

3

Related Work

This chapter introduces some of the existing routing models used to reduce the monetary costs of routing. We first introduce the 2 different types of routers commonly found in prior work. Afterwards, we touch upon some techniques used to improve routing such as finetuning. Finally, we cover a benchmarking framework that forms the basis for our work.

3.1. Routers

Recent work has explored routing mechanisms that leverage multiple large language models in order to optimize trade-offs between various metrics such as cost, latency, and performance. However, energy consumption trade-offs remain largely unexplored. Prior work on LLM routing typically falls into two categories: predictive and non-predictive (cascading) routers.

3.1.1. Cascading Router

Cascading routers improve efficiency by querying models sequentially, starting from smaller or less expensive models and moving up to larger models until the response is satisfactory. This early stopping mechanism can reduce the average computational cost by avoiding unnecessary use of the large models for simpler inputs. However, the sequential querying process introduces overhead for each previously inferred model in the chain. When larger models are ultimately needed, the total energy consumption may increase due to the cumulative cost of running multiple models in order. The challenge for cascading routers is how to determine whether a response was satisfactory.

FrugalGPT

FrugalGPT [5] is a monetary cost-aware routing framework designed for various natural language tasks. The paper identifies that popular models have heterogeneous pricing structures that can differ by two orders of magnitude. To mitigate this, the authors propose various cost saving strategies such as prompt adaptation, LLM approximation and LLM cascade. LLM cascade sequentially prompts a list of LLM API's until the generation is considered sufficiently reliable or the list is exhausted. To do this LLM cascade uses a generation scoring function. This function generates a reliability score given a prompt and response pair. If the score passes a threshold τ_i, the response of that model is selected. This scoring function was trained using a simple regression model that learns the correctness of a response given an input response pair. The model is trained in a supervised setting using correctness labels. However, the authors do not go into detail regarding how these were sourced.

The study shows that FrugalGPT can achieve up to a 98% cost reduction compared to always using the most expensive model and is able to improve performance by 4% over the largest model at equal cost. However, such significant improvements are likely only possible due to the vastly different pricing structures. Such pricing structures do not directly translate to significant gaps in energy consumption. Furthermore, ML as a Service (MLaaS) providers usually do not report energy metrics.

3.1.2. Predictive Routers

Predictive routing methods address the inefficiencies of ranking and cascading by learning to directly select the most appropriate model for each input before inference. Rather than running multiple models or querying sequentially, these systems train a router or policy network to predict which model will provide a satisfactory response at minimal cost, based on features of the input or preliminary model outputs.

OptiRoute

OptiRoute [29] is a dynamic model routing engine designed to balance multiple performance indicators, such as cost, accuracy, and latency, alongside non-functional requirements including helpfulness, honesty, and harmlessness, according to user-defined preferences. The system employs a hybrid approach that combines explicit user preference settings with a task analyzer and a routing engine.

The task analyzer consists of a finetuned small transformer model. Given an input query, the task analyzer classifies three implicit features: task type, complexity level, and domain. This smaller model was finetuned using query logs from a production LLM cloud service provider, leveraging a combination of human annotations and semi-supervised learning.

The extracted implicit features, together with the user-specified preferences, form a feature vector that is passed to a kNN based routing engine. The router then selects the model whose stored characteristics most closely match the constructed feature vector.

However, the paper provides limited detail on how the model-specific characteristics stored in the model database are obtained or quantified, leaving some ambiguity regarding the derivation and validation of these model feature representations.

MetaLLM

MetaLLM [26] formulates routing as a multi-armed contextual bandit problem. It dynamically learns a routing policy through online feedback, gradually improving its allocation of inputs to models based on observed reward signals that trade off accuracy and cost. MetaLLM does this by embedding the input using an out-of-the-box Sentence-BERT transformer.

MixLLM

MixLLM [43] similarly formulates the routing problem as a multi-armed contextual bandit problem. However, rather than embedding input vectors using a out-of-the-box Sentence-BERT model, MixLLM obtains their embeddings from a finetuned BERT encoder based on tag data. Furthermore, rather than using a single bandit model like MetaLLM, MixLLM utilizes LLM-specific predictors. This ensures on the fly scalability, as the addition of new models does not require retraining of the other arms. While bandit based methods are effective in online settings, we have full offline supervision. As such, standard supervised learning methods are more suitable.

3.1.3. Embedding & Finetuning

Several works leverage LLM query embeddings to enable input-aware routing of LLMs. These approaches try to capture semantic properties of the input prompt through the domain knowledge of medium-sized pre-trained transformers such as BERT [10] and use them to predict which model is most suitable for a given query [43, 7, 29, 26]. White such embeddings can be used out-of-the-box, some approaches also opt to finetune the transformer to cluster similar prompts.

MixLLM [43] finetuned a BERT model with InsTag [22] generated tags. InsTag is a model based on Llama2-13B [40] designed to generate instruction tags for natural language queries. These tags capture the semantic intent and complexity of the query. While the underlying Llama model is of significant size, the model is only used during the finetuning process of BERT. Therefore at inference time, the router does not provide significant overhead.

AdaptiveLLM [7] has a different approach. They reason that the complexity of the task may be correlated with the chain-of-thought length of the LLM solving the task. By prompting various LLM's on various tasks, the authors obtain pairs of queries and chain of thought lengths. This data was then used to cluster the queries based on their chain-of-thought length in different clusters representing task complexity. Triplet finetuning was then performed on BERT. However, we do suspect that some data

leakage may have occurred during the evaluation process. Tasks used for the finetuning process of BERT were also present in the evaluation test set. Nevertheless, AdaptiveLLM provides an interesting approach to difficulty estimation.

Finetuning embedding models for routing is an interesting approach to leverage larger transformer models without incurring additional overhead costs during inference. However, the finetuning process often requires a substantial amount of data.

3.1.4. RouterBench

RouterBench [14] introduces a benchmark for the evaluation LLM routing strategies across various tasks and models. It proposes a framework to compare routing methods based on quality and cost metrics. Routerbench combines several datasets from several different domains such as math, coding and writing with a candidate model pool of various sizes and API costs.

RouterBench deals with various different routing strategies. Directly querying the candidate models for the same prompt repeatedly to train and evaluate different routing strategies can become prohibitively expensive. Both in terms of financial cost and time. To address this, RouterBench proposes precomputation of all model-query pairs and storing all model responses and cost. While the upfront cost is high, these precomputed response and cost data can be reused for the training and evaluation process of different routing strategies, as the model responses are independent from router behavior. The decoupling of model inference from routing provides an efficient and fast way to evaluate different routing strategies.

This work is closely inspired by RouterBench. However, RouterBench provides a fixed candidate model pool and dataset, with emphasis on monetary costs. As such, the dataset cannot be used directly for energy-aware routing.

RouterBench also introduces an effective baseline routing strategy called the Zero Router, which is based on the non-decreasing convex hull. This approach selects models along the optimal quality-cost frontier such that each chosen model represents an efficient trade-off between quality and cost. As a result, the Zero Router provides a consistent and interpretable baseline that any routing strategy can be compared against and scales to any candidate model pool size.

One concern with routers trained on mixed-domain datasets such as RouterBench is that they may rely on category-based heuristics. Prior research [14] has shown that many routers exhibit this behavior, often directing most coding and math-related queries to the stronger model by default. Such behavior can interfere with routing for our specific downstream task, which focuses exclusively on code generation. This potentially limits the router's ability to make meaningful decisions in this domain.

4

Methodology

In this Chapter we go over our proposed framework. We first cover the data collection process which will be used in the training and evaluation of our routers. We then introduce our 2 different router designs and finally we discuss the evaluation metric to determine the capabilities of our router.

4.1. Proposed Framework

Our framework consists of two main steps. First. we have the data construction process in which we measure the energy consumption and evaluate the correctness of the generations. This is used to form a new augmented dataset with energy and correctness data. In the second step we train and evaluate routers using the augmented data.

4.2. Data Construction

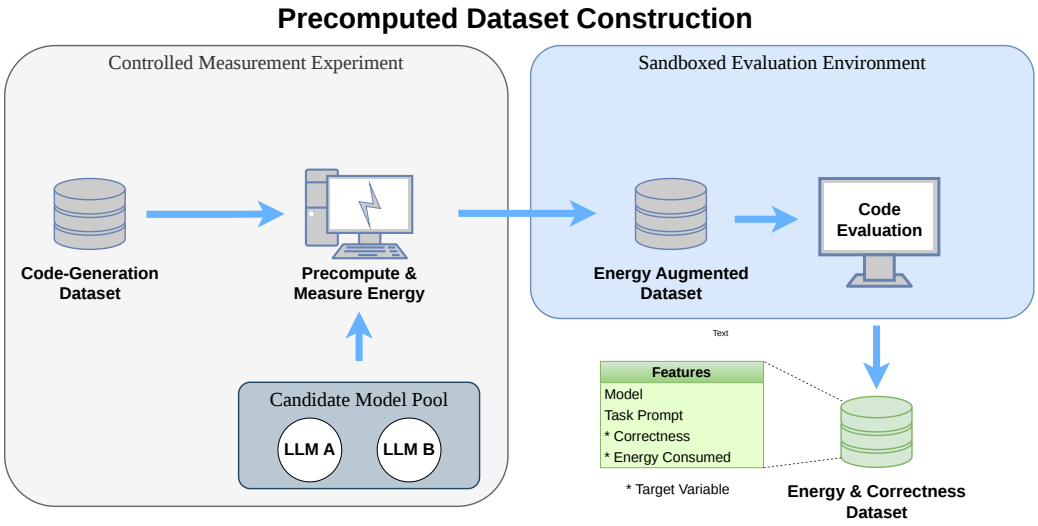


Figure 4.1: Data construction process for supervised energy-aware router training and evaluation. Each task prompt output is precomputed for every candidate model and then appended with functional correctness and energy consumption metrics.

To train and evaluate our routers for code generation efficiency, we require a dataset that includes code generation problems with model specific energy measurements. To construct this dataset, we follow a similar approach as RouterBench [14]. However, rather than introducing monetary API costs, we introduce energy consumption costs instead. In this approach, we augment existing code generation datasets with precomputed energy consumption and model responses. All candidate models are

evaluated on all tasks in the existing dataset, and for each task-model pair we record both the model response and the corresponding energy consumption. We can then evaluate model responses on correctness by directly executing the model response code in a sandbox environment against the provided test cases. A comprehensive overview of the router dataset construction process can be found in Figure 4.1.

This new dataset can then be used for supervised training and evaluation of the routers. Since all model output is precomputed and is independent of the actual router, the dataset can be efficiently used to train and evaluate various different routers without requiring additional inferences or energy measurements given that no new large language model is introduced to the pool. However, this approach does introduce the constraint that model generation stays consistent across multiple routers. For this reason the use of greedy decoding does not impose additional limitations. We can use greedy decoding for deterministic output, reducing the variance in energy consumption measurements.

Another advantage of this framework is that the overall evaluation process is separated from the model inference and energy measurement pipeline. This design allows us to measure model correctness in isolation within a sandbox environment, avoiding interference or overhead that could affect energy measurements. Sandbox environments are necessary because we automatically execute LLM-generated code without any human oversight, which is cause for significant security concerns. Especially when a standard Docker container alone may not provide sufficient isolation to mitigate all potential hazards [17, 30].

4.3. Candidate Model Pool

The selection of candidate models requires careful consideration, as the characteristics of the model pool directly influence the routing problem. Therefore, we consider various aspects of the candidate models to finally decide on the most suitable models for our router.

4.4. Router

The first step for our router is to represent the input prompts as a vector. For this we use a contextual embedding model. Each input prompt p is encoded into a vector of dimension d:

$$x = f(p) \in \mathbb{R}^d$$

where f is the embedding function and d is the output dimension of the embedding model.

The binary routing problem can then be formalized as a binary classification problem. Given the input embedding $x \in \mathbb{R}^d$, let $M = \{m_1, m_2\}$ represent the set of available models. The classifier learns the mapping function:

$$f : \mathbb{R}^d \to \{m_1, m_2\}.$$

For each prompt x, both model responses were evaluated offline, producing the following initial target variables

- a correctness indicator $c_i \in \{0, 1\}$ for model m_i
- an energy consumption value e_i in Joules

where $c_i = 1$ indicates that model m_i produces a correct response and $c_i = 0$ otherwise.

4.4.1. Router Design 1: Binary Classifier

The first routing strategy tackles the model selection problem directly as a supervised binary classification task. Given an input vector x, the router predicts which candidate model should be selected to generate the final response. To do this, we require labels for supervision.

Label Construction Using the observed outcomes of both models, a new target variable is derived. The routing label $y \in \{m_1, m_2\}$ is constructed that specifies which model should be selected for a given prompt. The labels are derived according to the following decision rule:

$$y = \begin{cases} m_1 & \text{if } c_1 = 1 \wedge c_2 = 0 \\ m_2 & \text{if } c_1 = 0 \wedge c_2 = 1 \\ \arg\min_{i \in \{1,2\}} e_i & \text{if } c_1 = 1 \wedge c_2 = 1 \\ \arg\min_{i \in \{1,2\}} e_i & \text{if } c_1 = 0 \wedge c_2 = 0 \end{cases}$$

If both models produce correct responses, the model with the lower energy consumption is preferred. If only one of the models produces a correct response, that model is preferred. This rule ensures that correctness is prioritized when selecting the model while energy is minimized when both models are sufficient.

In the case where both models fail to generate a correct answer, the router also selects the model with the lower energy consumption. This scenario represents a special case, however. One could argue that when both models are unable to produce a correct answer, selecting the lower-energy model may not be the most preferred strategy as this is typically the smaller and less capable model with a lower probability of solving the task. Nevertheless, since this formulation does not explicitly estimate the probability that a model can successfully solve a given task, we select the model with the lower energy cost in this case.

These labels are used exclusively to train our Classifier and are not used for evaluation.

Model Architecture The router itself is implemented as a classifier trained on prompt embeddings generated by our embedding model. Given a prompt embedding vector x, the classifier produces a probability distribution over the candidate models:

$$P(m_i \mid x), \quad i \in \{1, 2\}$$

The router selects the model with the highest predicted probability:

$$m_{\text{selected}} = \arg \max_{m_i \in \{m_1, m_2\}} P(m_i \mid x)$$

One advantage of this approach is that the router can be trained using fully supervised labels. These were constructed in such a way that the exact energy measurements played a minor role. For future dataset expansions, a simpler heuristic of assuming that smaller models generally consume less energy could be used to assign labels instead, thereby reducing the dependence on direct energy measurements during data collection.

An overview of the router can be found in Figure 4.2.

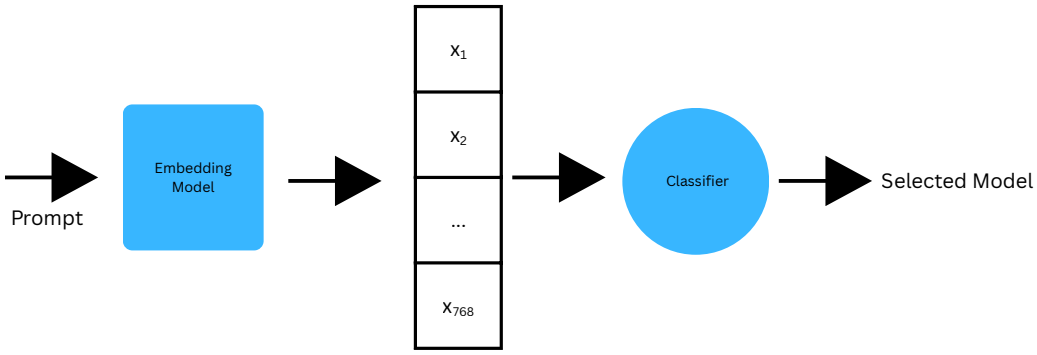


Figure 4.2: Classifier Router design

4.4.2. Router Design 2: Utility-Based Regressor

The second routing strategy formulates the model selection problem as a utility-based regression problem [39]. Utility-based regression models predict the expected utility of each candidate model. Instead of directly classifying which model to choose, the router estimates a utility score that captures the trade-off between correctness and energy consumption for each model, and then selects the model with the highest expected score. However, since energy can vary quite significantly between tasks while correctness is bound to true or false. We first compute the normalized energy defined as

$$\tilde{e}_i = \frac{e_i - \mu_E}{\sigma_E}$$

where μ_E and σ_E are the mean and standard deviation of the energy values computed over the training dataset.

We then define a utility score that balances accuracy and normalized energy, closely inspired by RouterBench [14]:

$$Utility_i = c_i - \lambda \tilde{e}_i$$

where the energy factor λ is a hyperparameter that controls the importance of energy consumption.

For each candidate model, we then train a regressor to predict its utility given the prompt embedding. Let $x \in \mathbb{R}^d$ be the embedding of a prompt. Then, for M candidate models, we define a set of regressors

$$g_m : \mathbb{R}^d \to \mathbb{R}, \quad m \in \{1, \ldots, M\}$$

where g_m predicts the utility of model m. In our setup with two candidate models, we have the two regressors g_1 and g_2. Given a prompt embedding x, the predicted utility for model m is

$$Utility_m = g_m(x)$$

At inference time, the router evaluates the predicted utility for each candidate model and selects the one with the highest score:

$$f(x) = \arg\max_{m_i \in M} g_i(x)$$

This formulation allows the router to consider both predicted performance and energy cost when selecting the most appropriate model for a given prompt. Introducing an energy factor λ enables fine-grained control over the router's behavior regarding energy-performance trade-offs, with a higher λ value resulting in a lower energy consumption at the cost of performance. Unlike the classification approach, this method can be easily extended to additional candidate models without retraining previously trained regressors, making it simple to scale. However, each new candidate model introduces an additional regressor that must be evaluated at runtime, increasing computational overhead during inference. The overall architecture closely resembles the LLM-specific contextual bandit design of MixLLM [43, 18], but operates in an offline setting without exploration.

An overview of the router can be found in Figure 4.3.

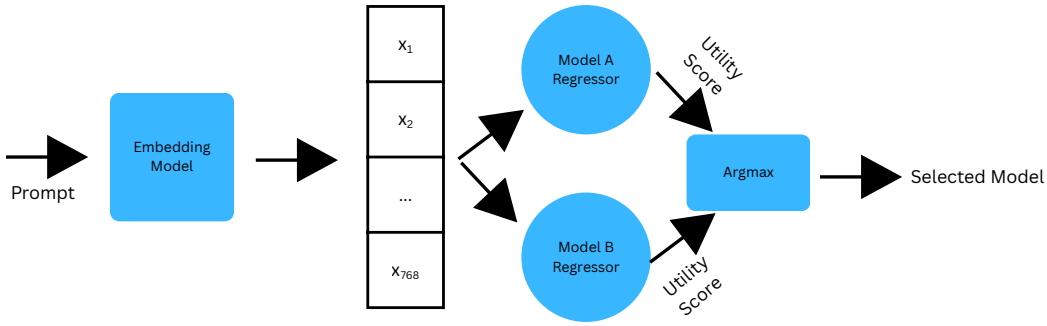


Figure 4.3: Regressor Router design

4.5. Evaluation Criteria

For our router evaluation we perform 5-fold group cross-validation, repeated 10 times with different random seeds to account for variation in train-test splits. For each fold, we train the router models as described in Section 4.4. Performance metrics (accuracy, energy) are averaged across all folds and repetitions weighted by fold size. Samples are grouped by unique task id such that all equivalent tasks end up in the same split. This is necessary because task entries are duplicated, with one entry for each candidate model. By ensuring equivalent tasks are in the same split, we avoid data leakage. Since not all tasks are equal in difficulty and length, cross-validation with repetitions helps to ensure unbiased and reproducible results.

To evaluate the effectiveness of our proposed routing strategies, we compare our routers with a baseline. However, since we have a trade-off between accuracy and energy consumption, this is not trivial, especially in non-binary settings. The construction of the baseline follows a similar methodology to RouterBench [14].

Let A_r and E_r correspond to the average accuracy and energy consumption of a routing strategy r over all folds and repetitions. Additionally, let (A_1, E_1) and (A_2, E_2) represent the accuracy and energy consumption of the two candidate models m_1 and m_2 on the same evaluation set.

Since our setup only includes two models, the baseline accuracy–energy trade-off line can be represented by a linear interpolation between the two endpoints. For a parameter $\alpha \in [0, 1]$, the interpolated baseline performance is defined as

$$A_\alpha = \alpha A_1 + (1 - \alpha)A_2,$$

$$E_\alpha = \alpha E_1 + (1 - \alpha)E_2.$$

This interpolation represents the expected performance to a hypothetical baseline router that randomly selects between the two models with probability α. This approach is equivalent to the non-decreasing convex hull baseline proposed in RouterBench applied to a binary setting.

To quantify the benefit of our router, we compare the energy consumption of the router to the interpolated baseline at the same level of accuracy. We solve for α:

$$\alpha = \frac{A_r - A_2}{A_1 - A_2}$$

Finally, the energy reduction is calculated:

$$EnergyReduction = \frac{E_\alpha - E_r}{E_\alpha},$$

where E_α is the interpolated energy corresponding to the router's accuracy $A_\alpha = A_r$.

5

Evaluation

In order to evaluate our framework we devise an experiment and implement our designs using real models and data. This chapter covers all of the implementation details for our experiment including the models and datasets used, energy measurement configurations, and various parameter settings.

5.1. Datasets Chosen

To train and evaluate the effectiveness of our routing strategy, we conduct experiments on two widely used code generation datasets: HumanEval [6] and MBPP [2]. The reason these datasets were selected was because they are commonly used in prior work and provide a sufficient number of samples to facilitate both training and evaluation. The two datasets combined provide us with 1138 python coding tasks. Another significant advantage of these benchmarks is the fact that they provide automatic unit tests for each task. This allows for more objective and interpretable evaluation of the LLM-generated code. This avoids relying on approximations or similarity metrics that can be difficult to interpret.

While these datasets have known limitations as discussed in Section 2.3.1, we choose to use the full MBPP and HumanEval dataset over the alternatives to maximize sample size and maintain comparability with prior work. Furthermore, since these datasets are widely used for benchmarking LLM's, they are more likely to be excluded and filtered from the training data of such models [15, 13]. This simplifies the process of finding models compatible with the dataset and lowers risks of data leakage.

5.2. Candidate Model Pool Selection

In this section we cover the two large language models used for the code generation process. We first cover the selection criteria and then select our models.

5.2.1. Model Selection Criteria

For our candidate models we consider the following aspects for inclusion.

Performance gap: Prior work [11] has shown that the benefits of routing depends on the performance gap between the candidate models. When models have a small performance gap, routers can achieve cost savings with minimal quality loss. Larger performance gaps can offer greater potential cost gains if the router is capable of accurately distinguishing easy from hard queries, but are more likely to degrade in performance compared to a single large router. However, the models are selected such that the smaller model has a lower cost with usually lower performance to facilitate cost-quality trade-off.

Instruction-following capability: Since our evaluation focuses on code generation under instruction-based prompts, candidate models must reliably follow prompt instructions. In our setting, not following the prompt instruction could result in a runaway or excessively verbose generation. This leads to inflated token counts and consequently higher measured energy consumption. Ensuring that models follow instructions is therefore particularly important.

Open-source availability: Models must be openly available to allow for local deployment in a controlled setting to facilitate precise energy measurement. Closed API-based models would prevent direct hardware based energy measurements and reproducible experiment conditions.

Hardware feasibility: The models must fit comfortably within the memory constraints of a single RTX 4090 (24GB VRAM). As suggested in prior work [3], high memory utilization can negatively impact performance and efficiency. To avoid such factors from effecting our measurements, we ensure that the selected models operate with sufficient memory headroom during inference.

5.2.2. Chosen Models

Based on the criteria outlined above in section 5.2.1, we select the following two models:

- DeepSeek-Coder-1.3B-Instruct
- Qwen2.5-Coder-3B-Instruct

DeepSeek-Coder-1.3B-Instruct is a 1.3 billion parameter large language model developed by DeepSeek AI and released in 2023 [13]. It is derived from DeepSeek-Coder-1.3B-Base and subsequently fine-tuned on approximately 2 billion tokens of instruction data. The underlying base model was trained from scratch on a 2 trillion token corpus, composed of 87% source code and 13% natural language data. The code training data was sourced from public GitHub repositories consisting of 87 different programming languages. In benchmark evaluations, DeepSeek-Coder-1.3B-Instruct achieves competitive results among open-source code models, including a reported pass@1 accuracy of 49.4% on the MBPP benchmark and 65.2% on the HumanEval benchmark.

Qwen2.5-Coder-3B-Instruct on the other hand was developed by Alibaba Cloud and was released in 2024 [15]. It inherits the base model (Qwen2.5) transformer design and tokenizer, and was then specialized for code through code-centric pretraining. Afterwards it was further specialized through instruction tuning. Similarly to DeepSeek-Coder, the code was sourced from public GitHub repositories. The final training data consisted of 5.2 trillion tokens, of which 70% code, 20% text and 10% math. Qwen2.5-Coder-3B-Instruct was able to outperform DeepSeek-Coder-1.3B-Instruct on both MBPP and HumanEval, achieving an accuracy of 73.6% and 84.1% respectively.

These models exhibit a meaningful difference in parameter count (1.3B vs. 3.0B parameters), which is expected to result in differences in energy consumption. Additionally Qwen2.5-Coder-3B-Instruct was shown to have a moderate performance gap over DeepSeek-Coder-1.3B-Instruct. This creates the potential for a non-trivial routing trade-off.

Since both models were evaluated on the MBPP and HumanEval benchmarks, their training data was decontaminated to remove any overlaps with these datasets. This was achieved by applying 10-gram filtering rules to ensure that no training examples from MBPP or HumanEval were included. As a result, both models are well-suited for our evaluation dataset.

Furthermore, both models are openly available and comfortably fit within the 24GB memory constraint of the RTX 4090 to be deployed locally. Memory utilization remains well below critical thresholds even while deployed simultaneously, avoiding potential performance degradation associated with near-capacity operation.

The two models are optimized for code generation tasks and have shown adequate performance on the MBPP and HumanEval benchmarks. Additionally, they have been instruction tuned. Instruction tuning improves prompt adherence by training models on structured instruction and response pairs, thereby reducing variability caused by failures to follow task specifications [46]. These factors increase the probability of the models adhering to the prompt instructions.

Instruction-tuned models have also been shown to perform effectively in zero-shot settings [45], which aligns with our evaluation protocol. However, prior work has also demonstrated that instruction-tuned models may exhibit substantial performance variance across semantically equivalent prompts [37]. This sensitivity shows the importance of maintaining a consistent prompting format throughout our experiments to ensure replicable results and stable energy measurements.

5.3. Generation

5.3.1. Prompting Strategy

The performance of large language models is highly sensitive to prompt design. As such, a structured prompting strategy was used to ensure consistent and reproducible model behavior across all tasks and models.

Code 5.1: Prompt template

````
"Please only respond with a codeblock and do not explain anything. Complete the following task:
```python
{prompt}
```"
````

We employ a zero-shot prompting strategy. The task prompt is embedded as shown in Code 5.1. The prompt explicitly instructs the models to respond only with the completed code block and to omit any explanations. We impose this constraint to minimize unnecessary token generation and reduce output variability. Furthermore, the prompt contains a structured response format for more consistent and reliable code extraction for both evaluation and practical use.

For the Qwen model, we further apply the official chat-based prompt template recommended by the developers [15]. This template separates system prompts from user prompts. In our case, we include the following system message: ”You are Qwen, created by Alibaba Cloud. You are a code generator and only respond with code without explanations.” This system level instruction further reinforces the output constraints already present in the user prompt.

5.3.2. Generation Parameters

For all tasks, code generation was performed using greedy decoding with no sampling. This results in deterministic generation which is a necessary constraint for our RouterBench inspired precomputation approach. Furthermore, the maximum token length for generated outputs was set to 1024 tokens. This limit was set to prevent runaway generations from consuming an excessive amount of tokens while still allowing sufficient length for typical function completions required for our use case. Finally, we employ several stopping criteria to stop generations early.

5.3.3. Early Stopping

We enforce early stopping rules to reduce the probability of runaway generations and to reduce post function generated code explanations. While post-processing approaches can also mitigate these problems, stopping generation early decreases the number of discarded tokens, reducing unnecessary energy costs. A key limitation, however, is that generation may only be stopped after the generated function is completed. As such, any conversational preambles (e.g. the model acknowledging our request), cannot be filtered out through such methods. The early stopping strategy consists of three main components:

Repeated Line Detection: The system monitors the most recent lines of generated code, and if any line appears three times within the last five lines, generation is halted. This prevents the model from endlessly repeating or alternating lines, which are common forms of runaway generation. However, this approach does not capture all cases of runaway generation, such as when a counter or other variable causes each line to be slightly different.

Code Block Completion Detection: Generation is stopped as soon as the model closes the code block. This ensures that each output contains only a complete code snippet without trailing tokens such as an explanation of the generated code.

Assertion and Test Case Filtering: We detect certain patterns commonly associated with post-function test case generations. Such generations are common in the MBPP dataset where assert statements are provided in the task prompt. This commonly results in the model repeating the assertions and generate new ones. This not only results in unnecessary token generation but also adds

additional code that could interfere with evaluation. This may happen when the model generates its own failing test cases, these would be counted as a failed code generation by the evaluation wrapper. Patterns detected include lines starting with ”assert” and ”# Test cases” When these patterns appear in the output, they are replaced with a code block closure and generation is stopped early.

5.4. Response Extraction & Evaluation

To evaluate the LLM-generated code completions, we first extract Python code blocks from the model completions. Since the prompts were designed to elicit responses in Python code blocks enclosed in triple backticks, extraction is performed using a simple regular expression that identifies these code blocks and isolates the Python code. In certain cases, the model output may not strictly follow the expected formatting. For example, Qwen sometimes places the ”python” keyword on the next line within the triple backticks rather than immediately adjacent to the opening backticks. In such situations the actual generated code may still be correct and for this reason the extractor is designed to handle such variations to improve robustness. However, we cannot guarantee correct extraction in all cases.

After extraction, we implement two dataset specific evaluation pipelines corresponding to MBPP and HumanEval tasks. In both cases, the extracted code is concatenated with the associated test cases from the dataset. The combined code is then executed in a controlled local environment. A completion is considered correct if the code executes without errors and passes all unit tests provided in the dataset.

As discussed in section 4.2, it is strongly recommended to perform the evaluation process in a safe environment in case of destructive code generation by the candidate models. For our experiments, code correctness evaluations were performed bare-metal on a separate disposable machine. However, they can also be evaluated on a virtual machine or other sandbox environment if a disposable machine is not available.

5.5. Router Models

In this Section we cover the embedding and prediction models selected for our router and discuss their parameter settings.

5.5.1. Embedding Model

For our contextual embedding model, we decided to use the CodeBERT model [12], a pre-trained transformer developed by Microsoft. CodeBERT is particularly well-suited for our task because it has been trained on both code and natural language datasets. This allows it to capture the semantics of both code and natural language present in prompts. Furthermore, CodeBERT with 125M parameters is relatively lightweight, reducing the computational overhead during embedding compared to larger language models, while still being large enough to produce high-quality embeddings. We chose to use CodeBERT out-of-the-box rather than finetuning it, given our limited dataset.

5.5.2. Prediction Models

The final component of our router is the prediction model. For our routers, we will cover 2 regression models and 2 classification models.

The first router is implemented as a logistic regression classifier trained on prompt embeddings generated using CodeBERT. Logistic regression is selected as it is lightweight and has shown strong performance in high-dimensional embedding spaces. Prior work has shown that logistic regression applied to various transformer embeddings can outperform both neural and non-neural classifiers on the LIARS dataset [24]. In our setup, each prompt is represented as a 768-dimensional CodeBERT embedding.

Our second router uses an XGBoost regressor. XGBoost is selected over linear regression as it is able to model non-linear relations and performed well in prior work when used on high dimensional embeddings [7].

For our third and fourth routers we use a KNN classifier and regressor. These were selected for their relative simplicity while also being suitable for embedding representations as semantically similar prompts should be geographically close in the embedding space.

5.5.3. Router Parameters

Model	Parameter	Value
XGBoost Regressor	n_estimators	200
	max_depth	4
	learning_rate	0.05
	subsample	0.8
	colsample_bytree	0.8
	random_state	42
Logistic Regression	max_iter	2000
KNN Classifier	n_neighbours	3
KNN Regressor	n_neighbours	3

Table 5.1: Router model hyperparameters

Hyperparameters can play an important role in controlling the complexity, accuracy, and generalization performance of machine learning models. While hyperparameter tuning is often preferred, in this work we prioritized performing more repetitions and cross-validation to obtain tighter confidence bounds on model performance. Table 5.1 summarizes the specific settings used for the routers. For the XGBoost router, the maximum tree depth is limited to four, which is intended to help reduce overfitting on the 768-dimensional CodeBERT embeddings. To compensate for the shallower tree depth, 200 estimators are used over the default 100. Row and feature subsampling is used to further reduce the risk of overfitting. We set a random_state variable for reproducibility.

All other hyperparameters for both models remain at their default values in their respective library implementations.

For the energy factor λ used in our regressors, we evaluate all values between 0 and 1 in increments of 0.1 to explore how different energy and accuracy trade-off settings affect overall router efficiency.

5.6. Measurement setup

We use EnergiBridge [32] to measure the energy consumption of the models. As discussed in Section 2.5.3, EnergiBridge provides a unified interface for energy measurement across different hardware platforms and abstracts low-level libraries such as RAPL, MSR, and NVML. Its straightforward interface makes it easy to integrate into experimental workflows while still leveraging real energy measurements rather than relying on metrics such as the number of floating point operations (FLOPS) as theoretical proxies. EnergiBridge provides a 200 millisecond measurement interval (5Hz) which should provide sufficient resolution to accurately capture energy usage during model inference in our setup.

All energy measurements were conducted on the Green Server, a dedicated workstation for energy measurements at the TU Delft. The server has the following hardware configuration:

- CPU: AMD Ryzen 9 7900X
- GPU: NVIDIA GeForce RTX 4090 (24GB VRAM GDDR6X)
- Memory: 64GB RAM DDR5

The system runs Ubuntu 22.04.3 LTS with Linux kernel version 6.2.0.

We collect the following metrics of interest:

- GPU Power Usage (mWatts)
- GPU Memory Usage (MiB)
- CPU Energy (Joules)

We will be relying on GPU Power for the majority of our research as the embedding and code generation models will be running on GPU using CUDA Version 12.2. We measure GPU Memory usage to verify that our models fit within the provided 24GB's of VRAM. We capture CPU Energy for our router models that may not have GPU implementations.

The experiments were conducted in a Docker container using Docker 24.0.5 to ensure reproducibility. Although the use of Docker containers is known to increase energy consumption compared to running on bare-metal Linux [33, 38, 44], we argue that in practical real-world deployments, containerization and orchestration are common and generally necessary. Therefore, it is reasonable to include the energy overhead of Docker in our measurements.

All measurements were performed under controlled conditions, with no other computationally intensive workloads running concurrently. Various energy consumption metrics were recorded every 200 milliseconds during the entire experiment with all models loaded into GPU memory beforehand. We isolated the energy consumed for model inference from other instructions by recording a timestamp immediately before the inference started and immediately after it finished. This defines our interval of interest. To convert GPU power to energy we use the trapezoidal rule as explained in Section 2.5.1 and apply linear interpolation for the start and end points.

The experiments were performed in two separate batches. The first batch consisted of the 500 test problems from MBPP, while the second batch included the remaining 474 training problems from MBPP together with all 164 tasks from HumanEval.

Each task inference was measured 10 times for each model. Within each batch, tasks were randomly shuffled prior to execution to minimize potential ordering effects as was recommended by some researchers [8]. Inference runs were executed one at a time, and a 15-second idle period was inserted between each run to allow the system to return to a stable baseline state. While 10 measurements per task may seem relatively few, we deem it to be sufficient for the evaluation of aggregated metrics across multiple tasks, since our hypothesis concerns overall router capabilities rather than the performance for any singular task. A 15-second wait time between experiments may also appear short. However, given that an average experiment lasts only 2 seconds, we expect it to be sufficient for the GPU to stabilize.

5.6.1. Overhead Measurements

While predictive routers do not incur multiple model inference costs the way that cascading routers do, they do still introduce additional overhead. In our case, the CodeBERT embedding model is particularly, as CodeBERT is still a 125 million parameter transformer model. While an order of magnitude smaller, these extra computations may still outweigh the benefits of routing. A second concern are the prediction models themselves, in particular XGBoost. XGBoost is a relatively computationally expensive model relative to other traditional machine learning models as many trees need to be evaluated. Furthermore, our router design requires an XGBoost prediction for each candidate model in our pool. The XGBoost overhead is therefore directly related to the scalability of our design. Finally, we also measure the energy consumption of our logistic regressor for completeness. Since logistic regression inference consists of only a single linear function followed by a sigmoid evaluation, we expect its energy consumption to be negligible. We collect 30 samples for each model and measure the energy required to embed and predict the entire HumanEval and MBPP dataset, totaling 1138 prompts. We measure GPU energy consumption for codeBERT embedding and CPU energy for our prediction models, as these models do not natively support GPU execution.

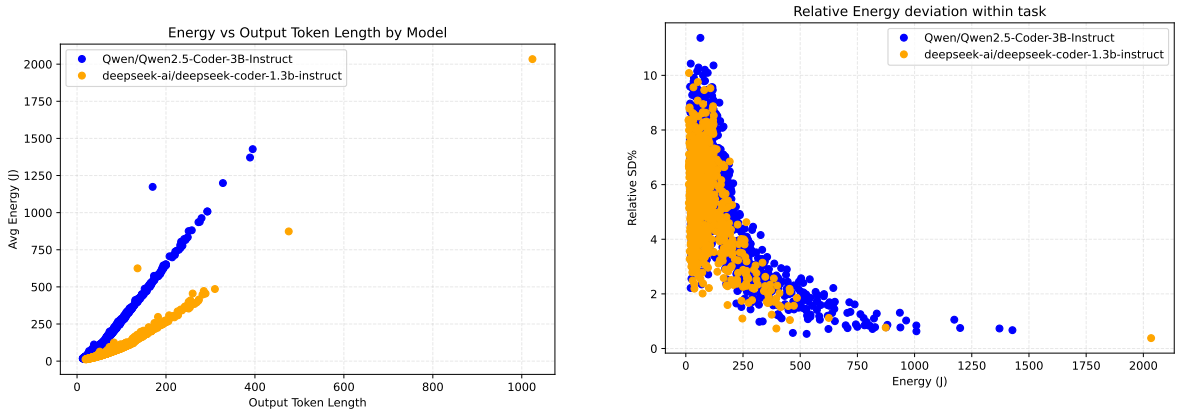
5.7. Reproduction Package

The implementation and reproduction package of our experiment is publicly available on GitHub [4]. Further instructions can be found in the repository README.

6 Results

This Chapter covers the results of the experiments described in Chapter 5 with the goal of being able to answer our initial research questions. We will first go over the results of our energy measurements and the obtained dataset. Afterwards, we present the results of our routers on that dataset and answer our three research questions.

6.1. Energy Measurement Results



(a) Energy consumption as a function of output token length.

(b) Relative measurement error. Error decreases as the energy consumed by the task increases.

Figure 6.1: Average energy consumption of task and relative error for 10 measurements per task

After conducting our energy measurements as explained in Section 5, we examine the data for relationships between output token length and energy consumption. Figure 6.1 shows the average energy consumption for each task, sorted by the number of output tokens produced. A clear linear trend can be observed, indicating that tasks requiring longer generations tend to consume more energy. This increase in energy consumption with output length appears more pronounced for Qwen-coder, which has a much steeper trend line than Deepseek-coder. This indicates a higher energy cost per generated token for Qwen-coder.

We also find two notable outliers from the overall linear trends. First, we observe that one instance of deepseek has reached the maximum output token limit of 1024 and was truncated due to a runaway generation, resulting in a much higher than expected energy measurement for the given task. Second, two data points deviate from the linear pattern, one for each model. We suspect the cause to be the same input prompt. Upon closer investigation, both points indeed correspond to the same MBPP task to ”Write a function to calculate a grid of hexagon coordinates where function returns a list of

lists containing 6 tuples of x, y point coordinates.". However, this is not the full input prompt that is given to the model. MBPP prompts are appended with the 3 unit test cases that the generated code is required to pass. We found that these test cases were associated with an unusually large input prompt when measured in tokens. The included test cases consisted of long hardcoded floating-point numbers of high precision, which substantially increased the input token count and consequently the energy consumption despite a modest output length. This is likely due to random floating point numbers being unlikely to share large sections of tokens and therefore having to be split into smaller tokens.

To compare the overall efficiency of both models we compute the amount of energy consumed per output token. A violin boxplot can be found in Figure 6.2. Qwen-coder has a median energy consumption of 1.896J per generated output token, with an average of 2.084J. Deepseek-coder on the other hand has an median and average energy consumption of 0.807J and 0.876J respectively. This shows that deepseek-coder is over twice as efficient compared to the larger qwen-coder model on a per token basis. However, this may not directly translate to the real efficiency gains when accounting for model behavior and tokenizer differences. In table 6.1 we find that the overall energy consumption per task is 160.79 Joules and 82.11 Joules for Qwen-coder and Deepseek-coder respectively. The actual energy consumption difference is smaller than expected when looking at per token efficiency metrics. This is likely due to the difference in tokens consumed per task. We observe that deepseek is slightly more verbose in their responses, generating 5.0% more characters on average. However, the model consumes 27.3% more tokens, suggesting that Deepseek-coder has less efficient tokenization compared to Qwen-oder.

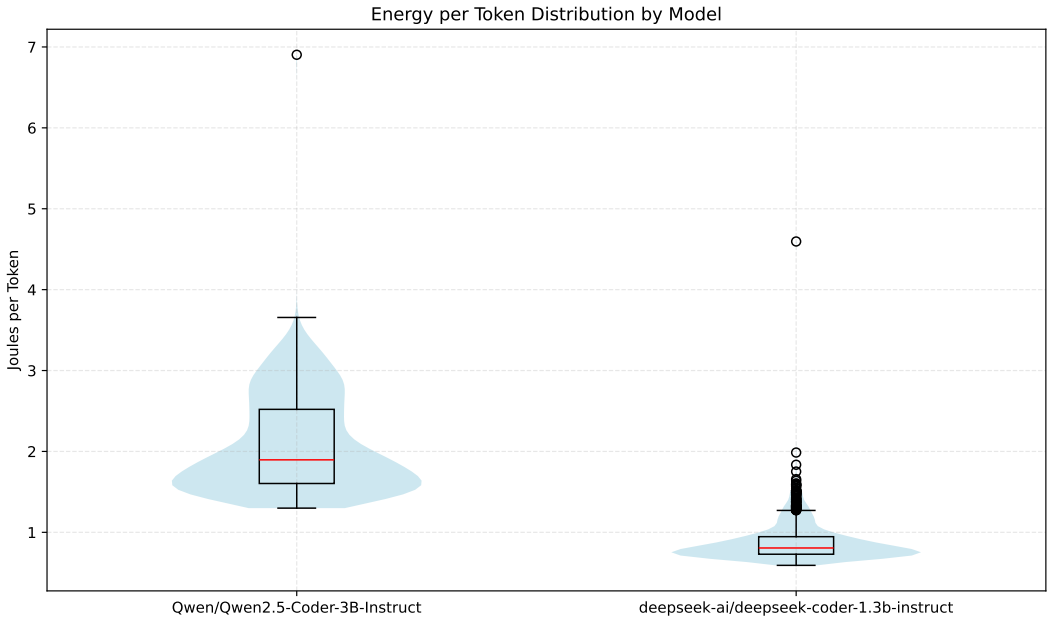


Figure 6.2: A violin boxplot of the energy consumption of a task per token by model. Qwen-coder has a median energy consumption of 1.896J per generated output token, with an average of 2.084J. Deepseek-coder on the other hand has an median and average energy consumption of 0.807J and 0.876J respectively.

Model	Avg. Energy (J)	Avg. Tokens	Avg. Characters
deepseek-ai/deepseek-coder-1.3b-instruct	82.10	81.65	237.82
Qwen/Qwen2.5-Coder-3B-Instruct	160.84	64.12	226.52

Table 6.1: Average energy consumption and token count per task for different models and their average completion length in number of characters.

Finally, we examine the highest amount of GPU Memory used at any point in time. We find that our

models utilize at most 22426 MiB of VRAM out of 24564 available. This ensures us that all of our models indeed fully fit on our GPU and that there was no CPU offloading.

6.1.1. Measurement Variance

To determine the accuracy of our measurements, we compute the variance of the measurements given the same task and model. We find that the relative standard deviation becomes lower as the task consumes more energy as found in Figure 6.1b. This is likely due to the measurements lasting longer and therefore providing more stable readings. We do not observe a significant difference between the two models with regards to measured variance. The measurement reliability significantly increases as measurements exceed 250 Joules with a relative standard deviation between 1% and 6%. Experiments under 250 Joules have a relative standard deviation varying between 2% all the way up to 14%.

We apply the Shapiro-Wilk test [34] to test for normality of our data points. We find that for 93.2% of our tasks normality may be assumed. While the remaining 6.8% does not pass the test, we consider it reasonable to assume normality given the large number of tests. We establish 95% confidence intervals based on one-sample Student's t-test [36] and 10 measurements. We find that the true average task energy consumption is likely within 10% for low energy measurements and within 4% for high energy measurements. While this is still quite high, it is sufficient for aggregated metrics across all tasks, especially given that tasks that consume more energy have higher precision are paired with higher weight in the calculation of aggregated energy metrics.

6.2. Model Evaluation & Dataset Results

	Deepseek-1.3B Incorrect	**Deepseek-1.3B** Correct
Qwen-3B Incorrect	308 (27.1%)	56 (4.9%)
Qwen-3B Correct	224 (19.7%)	550 (48.3%)

Table 6.2: Confusion matrix comparing correctness labels between Qwen/Qwen2.5-Coder-3B-Instruct and deepseek-ai/deepseek-coder-1.3b-instruct.

After evaluating the model generated code as described in Section 5.4, we obtain the confusion matrix shown in Table 6.2. The matrix shows that the two models are both correct or wrong in 858 of 1138 tasks (75.4%), of which 308 tasks (27.1%) are failed by both models. This leaves us with 550 tasks (48.3%) that are correctly solved by both models and can thus be optimized for energy-efficiency.

Furthermore. Qwen manages to correctly solve 224 tasks that DeepSeek fails as we would expect from the more powerful model. However, Deepseek manages to solve 56 tasks that Qwen does not. This indicates that while Qwen is the stronger model overall in terms of accuracy, it is not always superior. As such, the models may be able to complement each other for improved performance on these code generation tasks.

6.3. Labels & Oracle

We append the dataset with labels to train our Logistic Regression Classifier by applying our decision rule from Section 4.4.1. We find that the optimal routing directs 728 tasks (64.0%) to the smaller Deepseek model and routes 410 (36.0%) tasks to the larger Qwen model. This is surprising, given the confusion matrix results found in Section 6.2 Table 6.2 we would naively expect only 224 tasks to be routed to the larger Qwen model with our decision rules. However, it appears that in at least 186 cases the larger Qwen model manages to consume less energy than the Deepseek model, despite their difference in model size.

Furthermore, we can utilize these labels to define the Oracle router [14]. The Oracle router is a hypothetical router which selects the optimal model for each task as defined by these labels. By design, this router provides the theoretical upper bound achievable for this dataset prioritizing accuracy given the set of models. In essence, the Oracle router demonstrates how we can leverage complementing models to improve overall performance beyond what is possible with a single model.

6.4. Baseline Evaluation

Model	Accuracy (%)	Energy (J)
DeepSeek 1.3B	53.25	82.09
Qwen 3B	68.01	160.84
Oracle	72.93	94.01

Table 6.3: Accuracy and energy consumption of baseline models.

From our model evaluation results found in Section 6.2 and energy measurements in Section 6.1 we can derive the accuracy and energy consumption of our base models. The result can be found in Table 6.3. Deepseek-1.3B achieves an average accuracy of 53.25% with an average energy consumption of 82.09 Joules per task. Qwen-3B on the other hand has a higher accuracy of 68.01% but consumes 160.84 Joules per task. We can use these values to construct a linear function relating energy consumption to model accuracy trade-off. We compute that the baseline energy consumption increases by 5.33 Joules for each additional percentage point of accuracy. We derive the following linear equation as our baseline:

$$E(A) \approx 533.48 * A - 202.00J, \quad A \in [0.5325,\ 0.6801] \tag{6.1}$$

Furthermore, we find that the Oracle router achieves an accuracy of 72.93% and consumes an average of 94.01 Joules per task. We observe that the oracle far outperforms the strong model with an accuracy of 72.93% while still consuming 42% less energy.

6.5. RQ1. Router Results

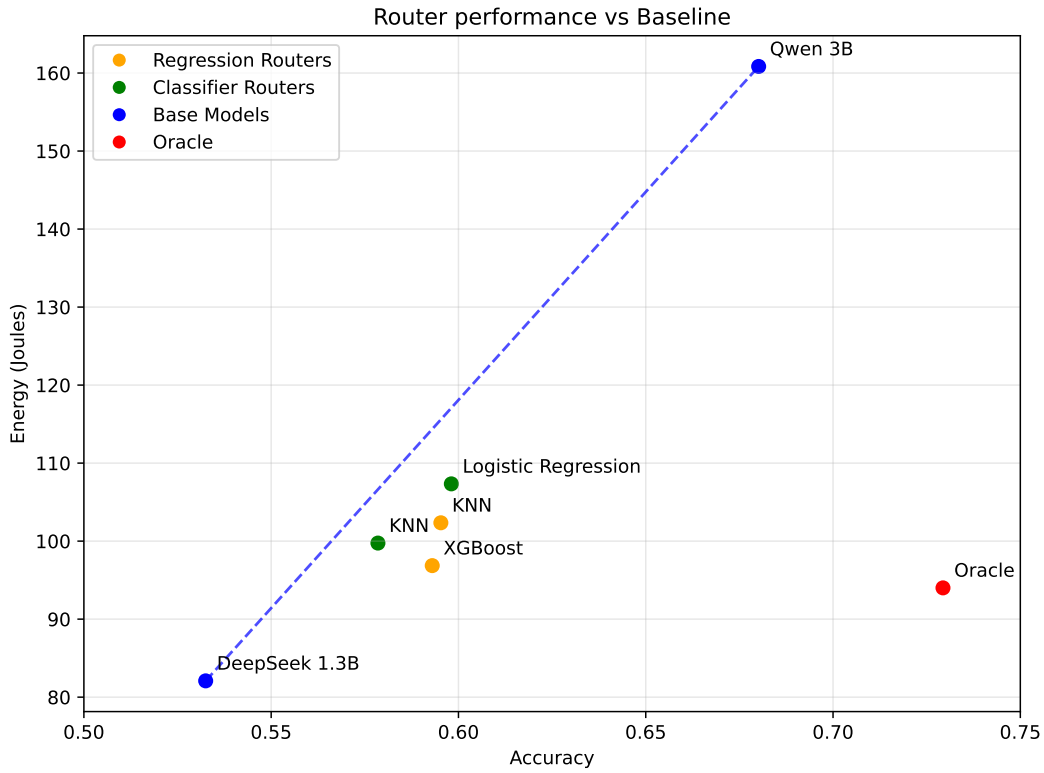


Figure 6.3: 10 Repetition average 5-fold router performance on the combined MBPP and HumanEval dataset. Our routers (yellow) consume less energy than the interpolated baseline (blue). The Oracle router (red) significantly outperforms all models, achieving an accuracy higher than any individual model with a slight increase in energy consumption over the small model

Model	Accuracy (%)	Energy (J)	Interp. Baseline (J)	Reduction (%)
Logistic Regression Classifier	59.81	107.34	177.08	8.32
KNN Classifier	57.85	99.75	106.62	6.44
XGBoost Regressor	59.30	96.86	114.36	15.30
KNN Regressor	59.53	102.35	115.58	11.45

Table 6.4: Performance comparison of routing models.

Figure 6.3 shows an overview of the performance of our routers compared to the interpolated baseline. After performing 10 repetitions our Logistic Regression Classifier Router achieves an accuracy of 59.81% while consuming 107.34 Joules per task. The interpolated baseline consumes 116.86 Joules for equivalent performance, resulting in a modest 8.32% reduction in energy consumption over the baseline. However, our XGBoost Regressor Router with energy factor parameter $\lambda = 0.5$ achieves a similar accuracy of 59.30% while only consuming 96.86 Joules per task. Resulting in a 15.3% energy reduction over the interpolated baseline of 114.36 Joules. This is however, still far from the theoretical upper-bound presented by the Oracle router. The KNN models perform worse relative to the alternative models using the same approach. The KNN Classifier achieves the lowest energy savings out of the 4 models, with a reduction of only 6.44%. The KNN Regressor on the other hand outperformed both of the classifier based models with a relative reduction of 11.45% at $\lambda = 0.5$. With these results we can answer our first research question:

RQ.1: How effective is predictive routing in improving the energy efficiency of LLM code generation?

Answer to RQ1

We implemented two predictive routers and compared them to an interpolated baseline. The results on MBPP and HumanEval show that predictive routing outperforms the interpolated baseline and reduce energy consumption by 15.3% and 8.32% for our XGBoost regressor and Logistic Regression classifier respectively. The KNN classifier and regressor achieved an energy reduction of 6.44% and 11.45% respectively. With these results we can conclude that predictive is quite effective in improving the overall energy efficiency of LLM code generation, with the regression based design being particularly promising.

6.6. RQ2. Energy Trade-offs

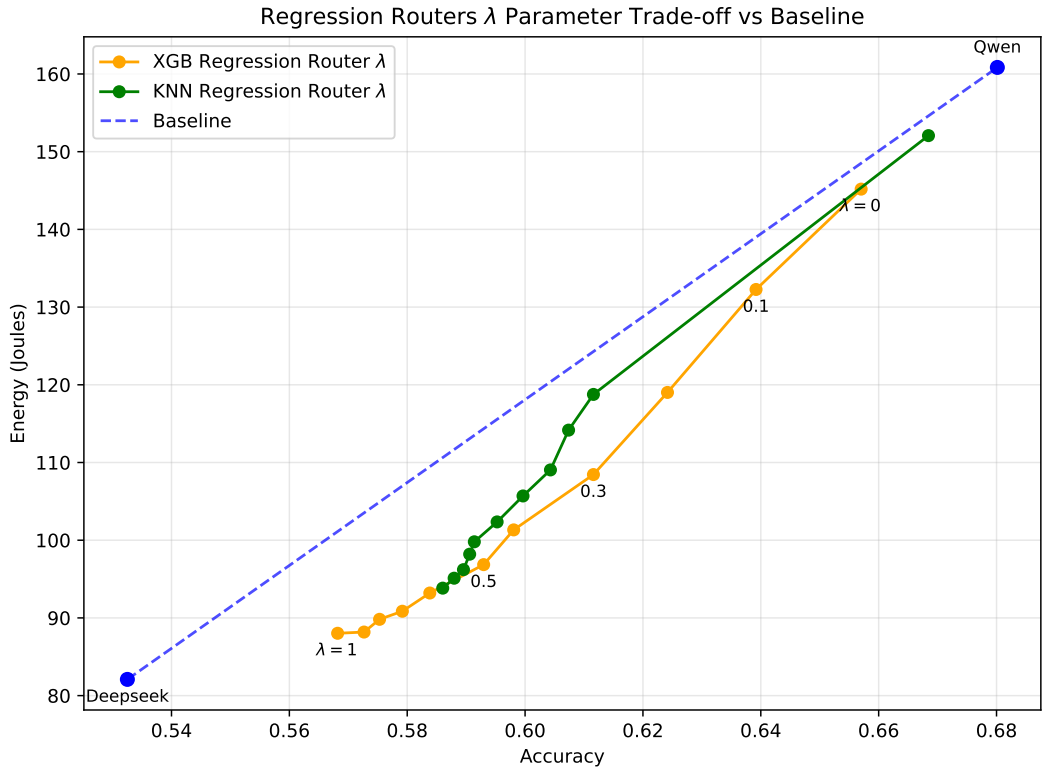


Figure 6.4: XGB Regression Router at various energy factor λ values

	XGBoost			KNN		
λ	Avg. Acc (%)	Energy (J)	Energy Red. (%)	Avg. Acc (%)	Energy (J)	Energy Red. (%)
0.0	65.70	145.18	2.24	66.85	152.07	1.64
0.1	63.92	132.27	4.84	61.16	118.76	4.44
0.2	62.42	119.02	9.13	60.74	114.17	6.44
0.3	61.16	108.45	12.74	60.43	109.05	9.42
0.4	59.81	101.33	13.44	59.96	105.69	10.36
0.5	59.30	96.86	15.29	59.53	102.35	11.43
0.6	58.38	93.20	14.86	59.14	99.80	12.07
0.7	57.92	90.86	15.07	59.06	98.20	13.15
0.8	57.53	89.82	14.39	58.95	96.21	14.49
0.9	57.27	88.18	14.81	58.80	95.11	14.83
1.0	56.82	88.02	12.95	58.60	93.83	15.19

Table 6.5: Average 5-fold regression router performance for different λ values.

While Section 6.5 covers the performance of the KNN and XGBoost regression routers at a fixed λ value of 0.5, this hyperparameter can be adjusted to suit different performance requirements. We evaluate our router for λ values ranging from 0 to 1, in 0.1 increments. The results can be found in Figure 6.4 and Table 6.5.

We find that our routers successfully adapt to a wide range of performance requirements as we alter the λ parameter, with the XGBoost router providing a wider and more consistent interval while generally outperforming the KNN router. Our second observation is that the trade-off is not linear. Instead, we find that routing seems to perform better at accuracies below the midpoint between the two routers.

The most energy-efficient configurations for both correspond to a performance of around 58% accuracy at which both routers are able to achieve energy reductions of roughly 15%. In the case of both models, efficiency gains drop significantly as performance approaches that of the larger Qwen model, this indicates that the most efficient settings heavily rely on the smaller Deepseek model. With that we can answer our second research question:

RQ.2: What is the trade-off between energy consumption and accuracy?

Answer to RQ2

We introduced a hyperparameter λ to our regression models and evaluated the models on λ settings from 0 to 1 in 0.1 increments. We found that the trade-off is not linear and that the efficiency of the models were significantly higher λ values and routing performed best when targeting an accuracy lower than the halfway point between the strong and weak model. The results show that routing is more suitable in cases where lower accuracy is acceptable and that efficiency drops significantly as we approach the larger model.

6.7. Overhead

Component	Total Energy (J)	Per Prompt Energy (J)	% of DeepSeek 1.3B
CodeBERT Embedding	405.15 ± 13.84	0.356 ± 0.01	0.43
Single XGBoost Prediction	5.65 ± 1.51	0.00496 ± 0.00132	0.01
Logistic Regression Prediction	0.79 ± 0.07	0.00069 ± 0.00006	0.00

Table 6.6: Energy overhead of different router components in Joules $\pm 1\sigma$. The total energy consumption is measured for 1138 prompts. We find that the overhead is less than 0.5% of a DeepSeek generation per prompt

We measured the overhead of various router components as described in Section 5.6.1. The results can be found in table 6.6. As expected, the energy consumption of these components differs by orders of magnitude. CodeBERT embeddings dominate the routing overhead, followed by XGBoost, with LR being minimal. However, all these overheads combined are still multiple orders of magnitude smaller than the energy cost of actual code generation by the inference models, indicating that our routing design overhead is negligible. As such, extending our XGBoost regressor router to multiple models should not significantly increase energy consumption. With this information we can now answer our final research question:

RQ.3: What is the energy overhead of the different router components?

Answer to RQ3

We measured the overhead of the embedding and router models and found that CodeBERT consumed around 405.15 Joules to embed the entire dataset while the XGBoost and Logistic Regression models consumed 5.65 and 0.79 Joules respectively for all the predictions. This shows us that the overhead for prediction is negligible and the embedding process constitutes for nearly all of the overhead. However, this is still insignificant relative to the cost of the code generation process. The total constitutes for less than 0.5% of the total energy consumption.

7 Discussion

In this Chapter we discuss the implications of the results and the overall generalizability of our work. Furthermore, we cover the various limitations and the potential avenues for future research.

7.1. Implications

Our results show that small reductions in energy consumption can be achieved through model routing for code generation on the HumanEval and MBPP dataset. Specifically, we have found that routing can result in energy savings of 8.32% for our classifier based approach and 15.3% using the regression method. The difference in the performance of these routers may be attributed to the fact that during the process of creating the labels for classification a significant amount of information was lost about the amount of energy consumed by the models. During the labeling process, only a binary comparison was used to determine which model consumed more or less energy, without considering the size of the difference. In contrast, the regression based method preserves more information about the actual energy cost differences. However, this likely also makes the former approach more robust to slight changes in energy measurement data and could potentially remain effective using simple heuristics rather than fine-grained energy measurements. Using such heuristics would result in a significantly simpler data acquisition process.

Although the overall energy saved was fairly limited, especially on a per task basis, these reductions can become increasingly more significant when handling larger volumes of inference requests. When applied on a larger scale, even modest percentage reductions can turn into substantial energy savings. Additionally, the ability of our router to reduce energy consumption for the specific downstream task of code generation is particularly relevant as LLM usage becomes increasingly more prevalent in the process of software development. However, the specificity of the task may also contribute to the relatively limited gains observed, as it would prevent the routers from being able to rely on simple category-based heuristics and instead require a deeper understanding of the prompt and underlying coding concepts. This makes routing within specific downstream tasks particularly challenging.

We have also demonstrated that our regression based routers are able to efficiently trade-off energy consumption and accuracy. This trade-off was shown to be the most effective at accuracy targets slightly below the midpoint between the smaller and larger models, while offering diminishing as performance approaches that of the stronger model. This implies that the proposed router design is particularly suitable for scenarios where always achieving the absolute highest accuracy is not critical, such as in coding assistants, but where energy efficiency may be a concern. In such settings, the router design allows for some control of this trade-off through the parameter λ during training. However, it is limited to the training phase only and does not support control of this parameter during the production phase of the model.

Furthermore we have found that the energy costs incurred by the overhead is minimal and that the benefits of predictive routing significantly outweigh the introduced overhead. We find that the models directly responsible for the routing decision making is especially lightweight and that the embedding

model is responsible for the vast majority of the overhead. However, even the embedding was at least two orders of magnitude lower than the actual code generation process. The limited impact of the routing decision making models on overall energy consumption is expected, as their complexity and size is negligible compared to that of transformer-based embedding and code generation models. In contrast, the embedding model consuming so much less energy despite being only one order of magnitude smaller than the generation models may not have been as obvious. However, this can easily be explained by the fact that the embedding process does not require autoregression and therefore involves only a single forward pass through the model, whereas code generation requires repeated next-token prediction until the code is complete. This implies that there is still potential for using larger models for embedding process, which may lead to improved representation quality. These findings also leave room for the routing decision making models to be larger and more complex. However, this comes with the requirement of additional data to train the larger models effectively.

The current code generation models used have a parameter count of 1.3B and 3.0B. This is relatively small compared to many of the current state-of-the-art models that would likely be used in most production environments. It is therefore important that our design is able to generalize to such settings and we expect that this approach can be scaled up to include larger sized models. That said, effectively introducing larger and more capable models is not a straightforward process. One of the challenges that arises when including large models into the routing framework in its current state, is that the difficulty of tasks in the training data is relative to model capability. As model performance increases, previously challenging tasks may become trivial, potentially leading to a very imbalanced dataset in which the vast majority of examples favor the smaller model and very few justify the use of the larger model. This can result in significant difficulties in training the router. Furthermore, increasing task complexity may also require larger embedding and routing models to adequately capture the more complex tasks. However, given that the potential absolute energy gains increases as we scale up the size of the generation models, the amount of overhead permitted for the router also increases. As such, we are able to scale router complexity. This does come with the limitation of likely requiring more training data to avoid overfitting. In short, our router designs are likely capable of generalizing to larger models given the presence of a larger and more comprehensive training dataset.

This work provides a starting framework for future research on energy-aware routing of large language models. The large performance gap between our proposed routers and the upper bound established by the Oracle indicates that there is still a lot of room for improvements, both in terms of energy efficiency and accuracy. Overall, the proposed framework is flexible and can be extended to serve as a basis for further research and the exploration of new energy-aware routing strategies.

7.2. Limitations

While a lot of care was taken in conducting this research, several factors could not be accounted for. In this section we cover the various limitations of our research.

7.2.1. Dataset limitations

The experiments in this work were conducted on the MBPP and HumanEval datasets. Both datasets are relatively small in size and consist primarily of short and self-contained programming tasks. As a result, they may not fully capture the complexity and diversity of tasks that are present in real-world software development scenarios. This may limit the extent to which the observed results could be generalized to a more complex code generation settings or other datasets.

Furthermore, as discussed in section 2.3.1, the utilization of MBPP and HumanEval comes with limitations regarding correctness. Both MBPP and HumanEval rely on unit tests to assess whether generated code satisfies the task requirements. Passing these tests can be considered a necessary test for correctness, as failing them indicates an incorrect solution. However, they are not a sufficient test, since the provided test cases may not cover all edge cases or fully capture the intended behavior. As a result, models may pass all tests while still producing incorrect or incomplete implementations. Our current metrics do not differentiate between such partially correct solutions and fully correct solutions. As a result partially correct solutions may be preferred by the router as they are also more likely to consume less energy as the generated code may contain fewer tokens with less robust edge case handling.

7.2.2. Prompt Sensitivity

Another limitation of the proposed routing framework is its sensitivity to the specific phrasing and structure of prompts. Large language models can produce significantly different outputs depending on subtle variations in the prompt, such as wording, formatting, or inclusion of examples. As a result, not only can the correctness of the output vary, the energy consumption is likely to be different as it is directly related to the output length.

The router itself is of particular concern. Unlike the LLMs, it is trained on a limited dataset and does not have extensive exposure to natural language. This makes it less capable of handling variations in prompt wording or structure, which can lead to suboptimal routing decisions and inconsistent energy savings and performance. While this sensitivity is especially pronounced for prompts that differ from those seen during training, the use of BERT embeddings mitigates this concern to some extent, as semantically similar prompts are likely to produce similar embeddings.

7.2.3. Hardware Differences

We conducted all measurements on the Green Server at TU Delft to ensure consistency. However, these results may not generalize to other hardware setups. Different CPUs, GPUs, and system environments can impact both the energy consumption and the inference speed of models. For instance, energy readings may differ between high-end server GPUs and more common consumer-grade hardware. Furthermore, different manufacturers use different telemetry and metrics which may not be consistent with others. Finally, this thesis also did not explore the effects of multi-GPU setups and the impact of parallelization on energy efficiency for model routing.

Although absolute energy values may differ on other hardware, the relative comparisons between routing strategies and the overall trends we observe are more robust to these differences. However, extending the dataset to include additional data or other models is not straightforward. Each new data sample would require new energy measurements on the exact same hardware setup, or more realistically, all energy measurements would need to be repeated to maintain consistency across the dataset. As such, we provide an adaptable framework to easily reproduce the results locally.

7.2.4. Code Quality and Efficiency

Our evaluation metrics do not account for code efficiency. LLM Generated functions can vary significantly, with some implementations being considerably more computationally expensive than others despite producing correct outputs. With our current metrics only the functional correctness is assessed. This is especially relevant in the context of energy consumption, as inefficient code may increase energy usage elsewhere in production. Therefore potentially negating any gains from more efficient inference.

Additionally, other aspects of code quality such as readability and maintainability are also not evaluated in our metrics. In fact, many desirable practices, such as including comments, docstrings or more descriptive variable and function names, may be implicitly penalized instead. These practices do not change the underlying functionality of the code but do increase the amount of tokens consumed, and therefore contribute to a higher measured energy consumption. As a result, the evaluation favors models that generate shorter and potentially harder to read code over a more maintainable implementation.

Furthermore we do not explore the consequences of incorrect code. In practice such errors may result in additional model inferences in an attempt to debug the code, leading to much higher energy consumption over using a more powerful model with higher accuracy. In more severe cases, incorrect implementations may even result in problems in production or security vulnerabilities. As such, the trade-off between accuracy and energy consumption can be significantly more complex and dependent on the context.

7.3. Future Work

This work proposed two routing strategies to reduce energy consumption for code generation. However, there still exist many other routing strategies that could be explored. Approaches such as fine-tuning the embedding model as was done in some prior work for cost-effective routing may also be effective for energy-aware routing. That said, the current lack of energy information makes it difficult to employ such methods. Future work could include the creation of a large dataset with energy consumption to

facilitate new research.

Besides new routing strategies, the applicability of routing in real world scenarios also still needs to be further investigated. For instance, in a real world deployment models are unlikely to be ran on a single GPU in an isolated setting. The introduction of a multi-GPU setting may complicate the routing process and load balancing could play a crucial role in the overall efficiency of the system.

Furthermore the compatibility of various prompt engineering strategies with model routing is still to be explored. During this research project we exclusively used a zero-shot setting. However, in practice Chain-of-Thought (CoT) and other prompting strategies are commonly used to improve model performance. Future work could look into methods for effectively applying such prompting strategies to routers.

Finally, one of our findings was the need for early stopping for our smaller router to prevent excessive token usage. However, our approach to solve this was dataset specific and would likely not generalize well to other settings. As such, a lot of research could still be conducted to detect and prevent unnecessarily verbose generations. An approach would be finetuning models for concise generations. We expect that concise generations could result in significant reductions in energy consumption in the future.

8

Conclusion

The usage of AI generated code for software development has been increasingly more common since the rise of Large Language Models and that is unlikely to change anytime soon. Although this has brought significant improvements in productivity, it has also brought many concerns. While many are worried about the quality of the output of such models the sustainability of using such models has often been overlooked.

This thesis investigated the possibilities of routing as a potential option to reduce the overall energy footprint. We have proposed a framework for the creation of energy-aware routers ranging from the data collection process all the way through to the training and evaluation of the models. This framework can form the basis for the creation of new routing strategies. We have also shown the effectiveness by applying the framework to the HumanEval and MBPP dataset. We used the capabilities of DeepSeek-Coder-1.3B and Qwen-Coder-3B combined with CodeBERT embeddings and have achieved reductions in energy consumption of up to 15.3%.

We have also investigated the trade-off of model routing and found that in all cases routing resulted in a lower accuracy than the strong model but consumed less energy. Our router was the most efficient when it achieved an accuracy just below the halfway point between the strong and weak model. We also observe a rapid decrease in efficiency as we aimed for accuracy targets near the strong router. This suggests that routing is more effective when high accuracy is not as critical.

The overhead of the router was measured and it was determined that the small decision making models introduced next to no overhead. The embedding process on the other hand did consume an average 0.356 Joules per task. However, this was still 2 orders of magnitude lower than the cost of generation.

While the overall results may not have been groundbreaking, the sustainability of large language models, or rather the lack thereof, is unlikely to be solved through a single solution. This thesis researched one piece of the puzzle that could eventually be part of a greater solution towards Green AI.

References

[1] Marta Adamska et al. *Green Prompting*. 2025. arXiv: 2503.10666 [cs.CL]. URL: https://arxiv.org/abs/2503.10666.

[2] Jacob Austin et al. *Program Synthesis with Large Language Models*. 2021. arXiv: 2108.07732 [cs.PL]. URL: https://arxiv.org/abs/2108.07732.

[3] Francisco Caravaca, Ángel Cuevas, and Rubén Cuevas. *From Prompts to Power: Measuring the Energy Footprint of LLM Inference*. 2025. arXiv: 2511.05597 [cs.AI]. URL: https://arxiv.org/abs/2511.05597.

[4] Jia-Jie Michael Chan. *GreenRouting*. https://github.com/MichaelChan20/GreenRouting. Accessed: 2026-04-28. 2026.

[5] Lingjiao Chen, Matei Zaharia, and James Zou. *FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance*. 2023. arXiv: 2305.05176 [cs.LG]. URL: https://arxiv.org/abs/2305.05176.

[6] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG]. URL: https://arxiv.org/abs/2107.03374.

[7] Junhang Cheng et al. *AdaptiveLLM: A Framework for Selecting Optimal Cost-Efficient LLM for Code-Generation Based on CoT Length*. 2025. arXiv: 2506.10525 [cs.SE]. URL: https://arxiv.org/abs/2506.10525.

[8] Luís Cruz. *Green Software Engineering Done Right: A Scientific Guide to Set Up Energy Efficiency Experiments*. http://luiscruz.github.io/2021/10/10/scientific-guide.html. Blog post accessed on 18/04/2026. 2021.

[9] Luís Cruz. *Tools to Measure Software Energy Consumption from your Computer*. https://luiscruz.github.io/2021/07/20/measuring-energy.html. Blog post accessed on 18/04/2026. 2021.

[10] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL]. URL: https://arxiv.org/abs/1810.04805.

[11] Dujian Ding et al. *Hybrid LLM: Cost-Efficient and Quality-Aware Query Routing*. 2024. arXiv: 2404.14618 [cs.LG]. URL: https://arxiv.org/abs/2404.14618.

[12] Zhangyin Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020. arXiv: 2002.08155 [cs.CL]. URL: https://arxiv.org/abs/2002.08155.

[13] Daya Guo et al. *DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence*. 2024. arXiv: 2401.14196 [cs.SE]. URL: https://arxiv.org/abs/2401.14196.

[14] Qitian Jason Hu et al. *RouterBench: A Benchmark for Multi-LLM Routing System*. 2024. arXiv: 2403.12031 [cs.LG]. URL: https://arxiv.org/abs/2403.12031.

[15] Binyuan Hui et al. "Qwen2. 5-Coder Technical Report". In: *arXiv preprint arXiv:2409.12186* (2024).

[16] Nidhal Jegham et al. *How Hungry is AI? Benchmarking Energy, Water, and Carbon Footprint of LLM Inference*. 2025. arXiv: 2505.09598 [cs.CY]. URL: https://arxiv.org/abs/2505.09598.

[17] Nicolas Lacasse. *Open-sourcing gVisor, a sandboxed container runtime*. https://cloud.google.com/blog/products/identity-security/open-sourcing-gvisor-a-sandboxed-container-runtime. 2018.

[18] John Langford and Tong Zhang. "The Epoch-Greedy algorithm for contextual multi-armed bandits". In: *Proceedings of the 21st International Conference on Neural Information Processing Systems*. NIPS'07. Vancouver, British Columbia, Canada: Curran Associates Inc., 2007, pp. 817–824. ISBN: 9781605603520.

[19] Sheng Li et al. *Searching for Fast Model Families on Datacenter Accelerators*. 2021. arXiv: 2102.05610 [cs.CV]. URL: https://arxiv.org/abs/2102.05610.

[20] Jiawei Liu et al. *Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation*. 2023. arXiv: 2305.01210 [cs.SE]. URL: https://arxiv.org/abs/2305.01210.

[21] Xiangyue Liu et al. "Evaluating the Test Adequacy of Benchmarks for LLMs on Code Generation". In: *Journal of Software: Evolution and Process* 37 (2025). URL: https://api.semanticscholar.org/CorpusID:279645154.

[22] Keming Lu et al. *#InsTag: Instruction Tagging for Analyzing Supervised Fine-tuning of Large Language Models*. 2023. arXiv: 2308.07074 [cs.CL]. URL: https://arxiv.org/abs/2308.07074.

[23] Philippe de Bekker Luís Cruz. *All you need to know about Energy Metrics in Software Engineering*. http://luiscruz.github.io/2023/05/13/energy-units.html. Blog post accessed on 18/04/2026. 2023.

[24] Sumit Mamtani and Abhijeet Bhure. *Pooling Attention: Evaluating Pretrained Transformer Embeddings for Deception Classification*. 2025. arXiv: 2511.22977 [cs.CL]. URL: https://arxiv.org/abs/2511.22977.

[25] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL]. URL: https://arxiv.org/abs/1301.3781.

[26] Quang H. Nguyen et al. *MetaLLM: A High-performant and Cost-efficient Dynamic Framework for Wrapping LLMs*. 2025. arXiv: 2407.10834 [cs.LG]. URL: https://arxiv.org/abs/2407.10834.

[27] OpenAI. *Scaling AI for Everyone*. Accessed: 2026-04-19. 2026. URL: https://openai.com/index/scaling-ai-for-everyone/.

[28] David Patterson. *The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink*. Feb. 2022. DOI: 10.36227/techrxiv.19139645.v1.

[29] Deepak Babu Piskala et al. "Dynamic LLM Routing and Selection based on User Preferences: Balancing Performance, Cost, and Ethics". In: *International Journal of Computer Applications* 186.51 (2024), pp. 1–7. DOI: 10.5120/ijca2024924172.

[30] Rafiqul Rabin et al. *SandboxEval: Towards Securing Test Environment for Untrusted Code*. 2025. arXiv: 2504.00018 [cs.CR]. URL: https://arxiv.org/abs/2504.00018.

[31] Alec Radford et al. "Improving language understanding by generative pre-training". In: (2018).

[32] June Sallou, Luís Cruz, and Thomas Durieux. *EnergiBridge: Empowering Software Sustainability through Cross-Platform Energy Measurement*. 2023. arXiv: 2312.13897 [cs.SE]. URL: https://arxiv.org/abs/2312.13897.

[33] Eddie Antonio Santos et al. *How does Docker affect energy consumption? Evaluating workloads in and out of Docker containers*. 2017. arXiv: 1705.01176 [cs.DC]. URL: https://arxiv.org/abs/1705.01176.

[34] S. S. SHAPIRO and M. B. WILK. "An analysis of variance test for normality (complete samples)†". In: *Biometrika* 52.3-4 (Dec. 1965), pp. 591–611. ISSN: 0006-3444. DOI: 10.1093/biomet/52.3-4.591. eprint: https://academic.oup.com/biomet/article-pdf/52/3-4/591/962907/52-3-4-591.pdf. URL: https://doi.org/10.1093/biomet/52.3-4.591.

[35] Lola Solovyeva and Fernando Castor. *Towards Green AI: Decoding the Energy of LLM Inference in Software Development*. 2026. arXiv: 2602.05712 [cs.SE]. URL: https://arxiv.org/abs/2602.05712.

[36] Student. "The probable error of a mean". In: *Biometrika* (1908), pp. 1–25.

[37] Jiuding Sun, Chantal Shaib, and Byron C. Wallace. *Evaluating the Zero-shot Robustness of Instruction-tuned Language Models*. 2023. arXiv: 2306.11270 [cs.CL]. URL: https://arxiv.org/abs/2306.11270.

[38] Senay Semu Tadesse, Francesco Malandrino, and Carla-Fabiana Chiasserini. "Energy Consumption Measurements in Docker". In: *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2. 2017, pp. 272–273. DOI: 10.1109/COMPSAC.2017.117.

[39] Luís Torgo and Rita Ribeiro. “Utility-Based Regression”. In: Sept. 2007, pp. 597–604. ISBN: 978-3-540-74975-2. DOI: 10.1007/978-3-540-74976-9_63.

[40] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. arXiv: 2307.09288 [cs.CL]. URL: https://arxiv.org/abs/2307.09288.

[41] U.S. Energy Information Administration. *How much electricity does an American home use?* Accessed: 2025-07-07. 2023. URL: https://www.eia.gov/tools/faqs/faq.php?id=97&t=3.

[42] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[43] Xinyuan Wang et al. *MixLLM: Dynamic Routing in Mixed Large Language Models*. 2025. arXiv: 2502.18482 [cs.CL]. URL: https://arxiv.org/abs/2502.18482.

[44] Mehul Warade et al. “Monitoring the Energy Consumption of Docker Containers”. In: *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. 2023, pp. 1703–1710. DOI: 10.1109/COMPSAC57700.2023.00263.

[45] Jason Wei et al. *Finetuned Language Models Are Zero-Shot Learners*. 2022. arXiv: 2109.01652 [cs.CL]. URL: https://arxiv.org/abs/2109.01652.

[46] Xuansheng Wu et al. “From Language Modeling to Instruction Following: Understanding the Behavior Shift in LLMs after Instruction Tuning”. In: *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. Ed. by Kevin Duh, Helena Gomez, and Steven Bethard. Mexico City, Mexico: Association for Computational Linguistics, June 2024, pp. 2341–2369. DOI: 10.18653/v1/2024.naacl-long.130. URL: https://aclanthology.org/2024.naacl-long.130/.

[47] Zhaojian Yu et al. “HumanEval Pro and MBPP Pro: Evaluating Large Language Models on Self-invoking Code Generation Task”. In: *Findings of the Association for Computational Linguistics: ACL 2025*. Ed. by Wanxiang Che et al. Vienna, Austria: Association for Computational Linguistics, July 2025, pp. 13253–13279. ISBN: 979-8-89176-256-5. DOI: 10.18653/v1/2025.findings-acl.686. URL: https://aclanthology.org/2025.findings-acl.686/.