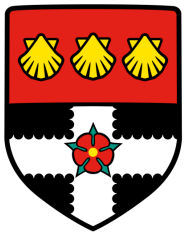

University of Reading

Department of Computer Science

Routers for LLM: A Framework for Model Selection and Tool Invocation

Ruben J. Lopes

Supervisor: Dr. Xiaomin Chen

A report submitted in partial fulfilment of the requirements of
the University of Reading for the degree of
Bachelor of Science in *Computer Science*

May 9, 2025

Declaration

I, Ruben J. Lopes of the Department of Computer Science, University of Reading, confirm that this is my own work and figures, tables, equations, code snippets, artworks, and illustrations in this report are original and have not been taken from any other person's work, except where the works of others have been explicitly acknowledged, quoted, and referenced. I understand that if failing to do so will be considered a case of plagiarism. Plagiarism is a form of academic misconduct and will be penalised accordingly.

I give consent to a copy of my report being shared with future students as an exemplar.

I give consent for my work to be made available more widely to members of UoR and public with interest in teaching, learning and research.

Ruben J. Lopes
May 9, 2025

Abstract

In the current landscape of large language models, users are confronted with a plethora of models and tools each offering a unique blend of specialisation and generality. This project proposes the development of a dynamic middleware *router* designed to automatically assign user queries to the most appropriate model or tool within a multi agent system. By using zero shot Natural Language Inference models, the router will evaluate incoming prompts against criteria such as task specificity and computational efficiency, and *route* the prompt to the most effective model and/or allow specific tools relevant that the model could use.

The proposed framework is underpinned by three core routing mechanisms:

- Firstly, it will direct queries to cost effective yet sufficiently capable models, a concept that builds on existing work in semantic routing Ong et al. (2025).
- Secondly, it incorporates a tool routing system that automatic invocation of specialised functions, thus streamlining user interaction and reducing inefficiencies currently inherent in systems like OpenAI's and Open Web UI. Furthermore this could also reduce inefficiencies in the recent reasoning models addressing the observed dichotomy between underthinking with complex prompts and overthinking with simpler queries when reasoning is manually toggled which can be costly and could cause hallucination.
- Thirdly, while the primary focus remains on model and tool routing, this work will preliminarily explore the potential application of the routing architecture as a security mechanism. Initial investigations will examine the theoretical feasibility of leveraging the router's natural language understanding capabilities to identify adversarial prompts. This includes a preliminary assessment of detection capabilities for prompt engineering attempts, potential jailbreaking patterns, and anomalous tool usage requests. However, given the rapidly evolving nature of LLM security threats and the complexity of implementing robust safeguards, comprehensive security features remain outside the core scope of this research. Although this aspect represents a promising direction for potential future work, particularly as the field of LLM security continues to grow.

By integrating these mechanisms, the research aims to pioneer a more efficient, modular, and secure distributed AI architecture. This architecture not only optimises resource allocation but also reinforces system integrity against emerging adversarial threats, thereby contributing novel insights into the development of next generation LLM deployment strategies.

Acknowledgements

An acknowledgements section is optional. You may like to acknowledge the support and help of your supervisor(s), friends, or any other person(s), department(s), institute(s), etc. If you have been provided specific facility from department/school acknowledged so.

Contents

List of Figures

List of Tables

List of Abbreviations

SMPCS	School of Mathematical, Physical and Computational Sciences
AI	Artificial Intelligence
GPT	Generative Pre-trained Transformer
MAS	Multi-agent systems
MCP	Multimodal Context Processing
MoE	Mixture of Experts
NLI	Natural Language Inference
NLP	Natural Language Processing
KQML	Knowledge Query and Manipulation Language
FIPA	Foundation for Intelligent Physical Agents
ACL	Agent Communications Language

Chapter 1

Introduction

In the past few years the landscape of large language models has expanded dramatically, with many domain specific as well as general purpose agents emerging across domains such as healthcare (like Med-PaLM 2 and BioGPT), coding (like CodeLlama and GitHub Copilot), and research (like Claude Opus and GPT 4o). Organisations that provide inference as a service now face complex trade offs between cost, latency, and capability; for example, GPT 4.5 can cost up to $75 per million tokens compared with just $0.15 for gemini 2.5 flash (llmpricing, n.d.). Although these models could be vastly different in terms of capability, the problem organisations face is determining *when* to deploy premium models versus more cost effective alternatives for a given task or prompt. This suggests a need for intelligent routing systems that can analyse incoming prompts and direct them to the most appropriate model based on task complexity, required capabilities, and cost considerations (artificialanalysis, n.d.).

Inspired by lower level (transformer embedded) "router" such as the one employed by mistral for their Mixtral (MoE) model the goal of this was to allow for a more distributed, higher level prompt based routing between a verity of models with varying levels of cost and complexity.

1.1 Problem statement

The proliferation of large language models has created a complex ecosystem where selecting the optimal model for a given task has become increasingly challenging.

Existing multi agent routing systems reveal several shortcomings. First, many current routers rely on **manual configuration**. For example, Both Open AI's as well as Open WebUI's chat interface require explicitly toggling of tools/selection of agents from users, and listing models to use or skip, on a per chat basis. Second, LLM based routers can suffer from reasoning **inefficiencies**. Recent studies identify *"underthinking"* (prematurely abandoning good reasoning paths) and *"overthinking"* (generating excessive, unnecessary steps) in modern LLMs. For instance, in a study Wang et al. (2025), the authors find that top reasoning models often switch thoughts too quickly an *"underthinking"* effect that hurts accuracy. Conversely, Kumar et al. (2025) demonstrate how even simple queries can be made to "overthink" (spending many tokens on irrelevant chains of thought) without improving answers . *Overthinking* is particularly problematic in the context of function calling, where excessive reasoning can lead to unnecessary API calls and increased costs or worse, hallucinations. This is especially relevant for models like GPT 4o and Claude 3 Opus, which are designed to handle complex reasoning tasks but can rack up significant costs if not used sparingly. The recent introduction of function calling in LLMs has further complicated this landscape, as users must now navigate a myriad of specialised tools and functions. This complexity can lead to ineffi-

cient routing decisions, where users may inadvertently select more expensive or less suitable models for their tasks. Finally, prompt interpretation remains imperfect: ambiguous or poorly phrased queries may be misrouted or require multiple LLM calls to resolve intent, leading to inefficiency.

Organisations and users face several key problems:

1. **Cost Efficiency Trade offs**: High capability models like GPT 4o and Claude 3 Opus provide powerful capabilities but at significantly higher costs than simpler models. Without intelligent routing, organisations and users may unnecessarily infer to expensive models for tasks that could be adequately handled by more cost effective alternatives.

2. **Selection Complexity**: With the dawn of function calling and Multimodal Context Processing (MCP), most chat systems offer numerous specialised tools and functions, but determining which tools are appropriate for a given query often requires manual specification by users or developers.

3. **Computational Resource Allocation**: Indiscriminate routing of all queries to high performance models can lead to inefficient resource allocation, increased latency, and higher operational costs for LLM providers and users.

1.2 Research Objectives

The premise of this research is to investigate whether pre existing Natural Language Inference models such as Facebook's bart-large-mnli could be used as drop in replacements to perform automated model selection and tool selection and potentially even using it as a security mechanism to detect adversarial prompts. Furthermore, I will examine the effectiveness of finetuning existing NLI models with specialised datasets designed for routing tasks.

The specific research objectives include:

- Creating a LLM Router library that can be deploy to existing systems with ease.
- Experimenting with Pretrained NLI models such as bart-large-mnli for both tool routing and model selection.
- Fine tuning the NLI model with existing datasets to improve its performance.
- Evaluating and assessing the accuracy the effectiveness using a set of prompts.
- Incorporate it with an existing Chatbot UI platform such as OpenWebUI.

Natural Language Inference (NLI) is a subfield of Natural Language Processing (NLP) that focuses on determining the relationship between sets of sentences. This is essential for what we are trying to achieve, buy using the prompt as the premise and the model descriptions as the hypothesis. By leveraging NLI techniques, we can create a more efficient and effective routing system that can automatically select the most appropriate model or tool for a given task.

1.3 Ethical Considerations

With LLMs being increasingly integrated into all aspects of our lives, it is crucial to consider the ethical implications of their use. This raises several important questions, such as:

- How do we ensure that LLMs are used responsibly and ethically?
- What are the potential consequences of using LLMs in sensitive areas such as healthcare, finance, and education?
- Its impact on the environment?
- The privacy and security implications of using non open weights models or proprietary models that are only accessible through APIs?

Although Routers are a set of tools that tries to improve the efficiency of LLMs and make them more context aware it is important to consider the ethical implications of LLMs and build tools such as Router that can help mitigate some of problems associated with LLMs.

1.3.1 Social Implications of of LLMs

One major concern is the potential for LLMs to spread misinformation. As these models are trained on vast amounts of data from the internet, they may inadvertently learn and propagate false or misleading information. This can have serious consequences, especially in areas such as healthcare, politics, and social issues, where accurate information is critical (Strubell et al., 2019).

Hallucination is another significant issue associated with LLMs spurning false information. This phenomenon occurs when a model generates text that is factually incorrect or nonsensical, leading to confusion and misinformation (Bender et al., 2021). Using Routers can potentially help mitigate this issue by directing queries to the most appropriate model or tool, reducing the likelihood of routing a complex query to a model that may not be able to handle it effectively. For example, if a user asks a complex question about a medical condition, the Router can direct the query to a specialised medical model or tool rather than a general purpose LLM. This can help ensure that users receive accurate and relevant information while also reducing the risk of hallucination. Adding an extra layer of safety such as a NLI based Security Guard that can help identify and filter out potentially harmful or misleading content.

1.3.2 Empact of LLMs on the Environment

With large and complex models like GPT 4 and Claude 3, the environmental impact of LLMs is a growing concern. According to a study by (Strubell et al., 2019), the energy consumption associated with training a single LLM can be equivalent to the lifetime emissions of five cars. The energy consumption associated with not only training these models but also running them for inference can be significant. This raises questions about the sustainability of using LLMs in various applications, especially when considering the carbon footprint associated with their operation. Routers have shown to be effective in reducing the number of tokens needed for a given task, which can help reduce the overall energy consumption associated with running LLMs. By directing queries to the most appropriate model or tool, Routers can help ensure that users receive accurate and relevant information while also reducing the energy consumption associated with running LLMs.

1.3.3 Impact of closed weights and sensored models

The use of closed weighted or proprietary models that are only accessible through APIs raises several privacy and national security concerns. These models may be subject to restrictions on what content they can generate or how they can be used, which can limit researchers ability to study and understand their behaviour.

For example, the online version of deepDeepSeek R1 is censor to avoid certain topics such as the Tiananmen Square protests (Sankaran, n.d.), the model actively avoids generating content related to this topic even though it fully understands the context of the question as shown in its thinking process when self hosted(Ramos, n.d.). This lack of transparency can hinder efforts to ensure that these models are used responsibly and ethically. Additionally, the reliance on proprietary models can create barriers. Here, the use of Routers can help redirect sensitive queries to self hosted models or tools, reducing the reliance on proprietary APIs and ensuring that users have more control over their data and the models they use. This can help mitigate some of the ethical concerns associated with using closed weighted models and ensure that users have more control over their data and the models they use.

> *All of the files related to this project are available in the GitHub repository:*
> *https://github.com/ru4en/fyp-llm-routers.git*

Chapter 2

Literature Review

2.1 Large Language Models: Current Landscape

Large scale LLMs continue to grow in parameter count and capability, intensifying the trade off between performance and computational cost. Models such as OpenAI's GPT 4 and Google's Gemini 2.5 Pro deliver top tier results, but at significantly higher inference costs often 20 to 40 times more than comparable alternatives (openai, n.d.). With many state of the art models being closed source (only accessible through an API), a new wave of open weight and open source models has emerged. These models make it easier for individuals and companies to self host, potentially lowering operational costs. For organisations offering inference as a service, open models are particularly advantageous not only for cost efficiency, but also for addressing privacy and security concerns associated with sending user prompts to third party providers.

2.2 Multi Agent Systems and Distributed AI Architecture

MAS have been a subject of research and development since the 1980s. While traditional MAS research established fundamental principles by using agent communication protocols such as KQML and FIPA ACL, the emergence of Large Language Models has transformed how these systems operate in practice.

In December 2023, Mistral AI introduced Mixtral 8x7B, a model that employs a Sparse Mixture of Experts architecture suggesting a promising approach which only activates a subset of a large model's "experts" per query. This gave them the edge over other models such as Llama 2 70B on most benchmarks where inference was 6 times faster and even *"matches or outperforms GPT 3.5 on most benchmarks"* Hu et al. (2024). While Mixtral applies routing at the model architecture level rather than through a separate system level orchestration, it demonstrated the potential for such a middle layer. This approach highlights how advanced modular designs can enhance performance. Even though the computational requirements for inference remain high for many GPUs, it was significantly less expensive to run compared to similar sized dense models. This increased demand for performance optimisation while leveraging existing models remains the core reason to research systems that deploy sophisticated multi model and multi agent systems.

2.3 Semantic Routing Mechanisms

Several recent projects provide router like middleware to manage multi model access. One such example is **OpenRouter.ai**, which provides a model route of sorts, unified into one API that

provides model inference behind an endpoint, dynamically routing requests across providers to optimise cost and availability. On the open source side, **RouteLLM** formalises LLM routing as a machine learning problem. RouteLLM learns from preference data such as chatbot arena rankings to send easy queries to cheap models and hard ones to big models. Their results show that such learned routers *"can significantly reduce costs without compromising quality, with cost reductions of over 85% on MT Bench while still achieving 95% of GPT 4's performance"* lmsys.org (n.d.). Another routing mechanism, Router Bench, shows promise with over 405,000 inference outcomes from representative LLMs, measuring routers on metrics such as dollar per token cost, latency, and accuracy Hu et al. (2024).

On the tool routing side of things, most work focuses on enabling LLMs to call tools rather than on how to choose them automatically. Landmark papers like **Toolformer** Schick et al. (2023) demonstrate how LLMs can learn to invoke tools. At the interface level, OpenAI's **Function Calling** and "built in tools"features have begun to infer tool usage directly from user prompts. For example, "google XYZ for me"automatically triggers a web search tool without explicit selection. In parallel, **LangChain** implements lightweight embedding based matching to decide when and which tools to invoke. Despite these advances, there remains a gap in formal publications on tool routing per se, especially in live inference settings.

2.4 Routing Approaches

Whilst looking for alternatives, some of the current decision making mechanisms used by LLM services are:

- **Rule based routing:** This relies on a predefined set of heuristic rules or configuration files to map incoming queries to specific LLMs or tools. For example, simple keyword matching or regular expressions might be used. A terminal tool could apply a rule such as `/bash/` to detect a Bash code block, then execute it in a virtual shell with appropriate safety checks. This approach offers full transparency and is reliable liveperson (n.d.). Each routing decision is directly traceable to an explicit rule, making the system's behaviour predictable and explainable **?**. However, because it relies solely on hard patterns, it often lacks contextual understanding. For example, a prompt like *"Could you make me a simple Snake game in Pygame?"* may not activate a development tool if the trigger is based only on a regular expression like `/python/`, which searches for explicit Python code blocks.

- **Prompt based routing:** This involves invoking a language model with a crafted system prompt. For example: `SYSTEM: "Determine whether the following prompt <USER_PROMPT> contains Bash. If so, return only the shell commands."` The model's response is passed to the relevant tool or agent. If a shell command is detected, it may be executed and its output returned to the user after post processing. A simple approach for model selection is to prompt a compact but capable model, such as TinyLlama, with the query and a list of available models, and ask it to select the most appropriate one. LLM based routers benefit from broad general knowledge and the ability to process complex inputs. However, they introduce higher computational overhead, latency, and occasional unreliability, making them expensive and potentially fragile.

- **Similarity Clustering based Routing:** This method leverages unsupervised clustering algorithms such as K-means to group historical user queries in a semantic embedding

space, thereby identifying clusters of similar requests. By operating on semantic similarity rather than rigid rules, this method affords greater contextual sensitivity, enabling more flexible task appropriate routing while retaining the predictability of cluster level performance. This method is effective if the quality of the data collected is very high Varangot-Reille et al. (2025).

- **NLI based (zero-shot) routing:** This is the approach we will implement. It employs a pre trained Natural Language Inference model, such as BART-Large-MNLI, to perform zero shot intent classification. The prompt is treated as the premise, while tool or agent descriptions are framed as hypotheses. The tool or agent with the highest scoring hypothesis is selected. This approach requires no additional training but is sensitive to the phrasing and calibration of the hypotheses. The quality of results thus depends heavily on how well these descriptions are constructed. This gives us both the reliability of Rule based routing whilst allowing the prompt to be flexible for some level of context relation.

2.5 Research Gap Analysis

As highlighted previously, multi agent routing has been successfully implemented both as closed source (**OpenRouter.ai**) as well as in open source libraries such as **RouteLLM**. These systems effectively distribute queries across multiple language models based on their respective capabilities. However, current approaches exhibit several limitations that warrant further investigation.

First, existing routing mechanisms predominantly rely on sophisticated architectures requiring substantial computational resources. For instance in the paper, Jiang et al. (2023) where a transformer based models with over 1 billion parameters for their routing decisions is suggested, while commercial solutions like OpenRouter.ai utilise proprietary embedding models that necessitate dedicated GPU infrastructure. These resource intensive requirements create significant barriers to deployment in resource constrained environments or edge computing scenarios.

Second, the dominant routing paradigms typically demand extensive training data encompassing diverse query model pairs with performance metrics. This data acquisition process is both time consuming and expensive, often requiring thousands of labeled examples to achieve acceptable routing accuracy. Such data requirements impede rapid adaptation to newly released language models or specialised domain applications where labeled data is scarce.

Third, while preliminary research has begun exploring NLI models for routing tasks, there remains a significant knowledge gap regarding their efficacy in production environments. The potential of NLI models specifically their ability to determine semantic relationships between user queries and model capability descriptions has not been thoroughly examined in the context of multi agent routing systems.

This research aims to address these gaps by investigating the viability of lightweight, pre trained NLI models as efficient routing mechanisms.

2.6 Critical Analysis of the Literature

The literature on routing mechanisms for LLMs and multi agent systems has made significant strides in recent years, particularly with many models using LLM based routing. With LMSYS Orgs lmsys.org (n.d.) and Hu et al. (2024) being the most similar to the research I am conducting. However, there are still several areas that require further exploration and

improvement. Firstly, the paper only focuses on routing two models, where one is a cheap model and the other is a more expensive model. Although this is a good start, it does not fully explore the potential of routing multiple models that we have available. Furthermore, the paper doeset explore the potential of using LLM leaderboards to evaluate the performance of the models in a more comprehensive way. This could be a valuable addition to the research, as it would provide a more complete picture of the performance of the models and their suitability for different tasks. Secondly, the idea is to use pipeline from the transformers library this will make sure most of the lower level details are handled by the library rather than having to implement them from scratch. Furthermore, the largest motivation for this research is to expand on the work done by lmsys.org (n.d.) and explore the use of these `Routers` as a more generalised middleware for not only routing LLMs but also managing tools; this stems from a persistent annoyance (or perhaps simple laziness) encountered when working with fragmented tool interfaces in typical LLM-based systems.

Chapter 3

Methodology

3.1 Research Design

This study employs an experimental research design to develop and evaluate a routing system for existing large language model interfaces. The approach draws from both software engineering methodologies and machine learning research practices to create a systematic framework for development and testing. The research follows an iterative development methodology, beginning with the selection of a foundational NLI and progressing through the creation of increasingly sophisticated routing mechanisms. The ultimate goal is to produce a modular, extensible, and user-friendly Python library that can be integrated into various AI applications.

The methodology consists of eight distinct phases:

1. Base NLI model selection
2. System Architecture Design
3. Generic Router prototype development
4. Library Development And Architecture Design
5. Evaluation framework creation
    (a) Automated Testing Script
    (b) User Interaction CLI Tools
6. Synthetic dataset generation (for testing)
7. Plugin integration with existing AI systems
8. Fine-tuning of base NLI model

Each phase builds upon the previous ones, with continuous evaluation and refinement throughout the process. This iterative approach allows us to incorporate findings from earlier stages into subsequent development, creating a feedback loop that strengthens the overall system design.

3.2 Selection of the Base NLI Model

The initial phase involves selecting an appropriate foundation model and model architecture to serve as the cognitive engine for the routing system. This selection process considers several critical factors that directly impact the viability and performance of the resulting system.

We evaluated possible models against the following criteria:

- **Classification Performance:** The model must demonstrate strong capabilities in text classification and categorisation tasks.
- **Inference Speed:** Given that routing decisions must occur with minimal latency to maintain system responsiveness, we established maximum acceptable response time thresholds based on human perception studies. Models exceeding these thresholds were eliminated from consideration regardless of their performance on other metrics.
- **Licensing Considerations:** Only models with permissive licensing terms suitable for both research and potential commercial applications were considered.

  For the experimental evaluation, we selected the open weights model: `facebook/bart-large-mnli`.

This model was selected for its strong performance on zero-shot classification tasks, particularly in the context of NLI; BART-Large-MNLI is a transformer-based model that has been pre trained on a large corpus of text and fine-tuned for NLI tasks, making it well suited for the routing system's requirements. The model's architecture allows it to effectively understand and classify complex prompts, making it an ideal candidate for the routing system Lewis et al. (2019).

Some of the key features of the BART-Large-MNLI model include:

- **Transformer Architecture:** BART-Large-MNLI is based on the transformer architecture, which has proven to be highly effective for a wide range of natural language processing tasks. This architecture allows the model to capture complex relationships between words and phrases in text, making it well-suited for understanding and classifying prompts.
- **Pre-trained on Large Datasets:** The model has been pre-trained on a large corpus of text, enabling it to leverage a wealth of knowledge and context when processing prompts. This pre-training helps the model generalise well to various tasks and domains.
- **Fine-tuned for NLI Tasks:** BART-Large-MNLI has been specifically fine-tuned for natural language inference tasks, which involve determining the relationship between a premise and a hypothesis. This fine-tuning makes the model particularly adept at understanding the nuances of language and context, allowing it to classify prompts effectively.

3.3 System Architecture

The architecture of the entire system is designed for the routers to sit between the user and the LLMs. The library is designed to be modular, allowing for easy integration with existing AI chat systems. In the following sections, I will describe the architecture of the system as a whole, and then the individual components of the system. The architecture is designed to be modular, allowing for easy integration with existing AI chat systems. The system should be designed to be extensible, allowing for easy addition of new models and tools in the future.

Overview of the Routing Architecture

Both the Model Router and Tool Router are designed use the same base structure, where the user prompt is passed to the router library, which then based on the prompt and the routing mechanism, selects the appropriate model or tool from the list of available models or tools. The selected model or tool is then used to invoke the model or tool and return the result to the user.

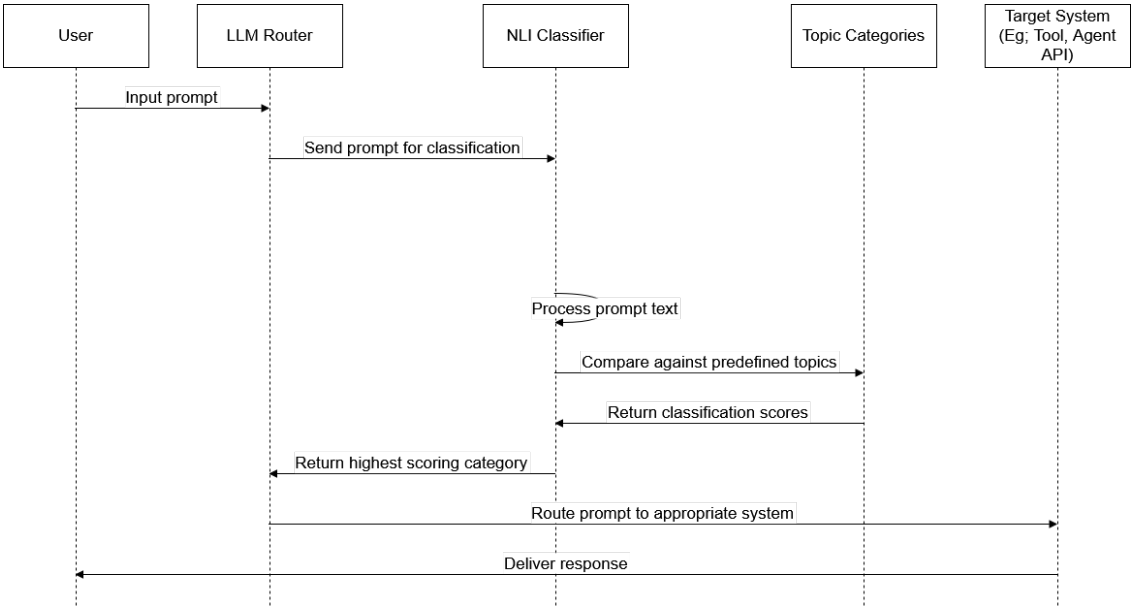

Figure 3.1: Sequence Diagram

NLI classification uses a zero shot approach, where the prompt is treated as the *premise* and the model or tool description is treated as the *hypothesis*. The NLI model then scores each hypothesis against the premise, and returned. The model or tool with the highest score is withing a set threshold is selected and top n is returned.
In the figure above, the NLI model is used to classify the prompt into a category, from a list of categories which is then returned back to the library with a confidence score. The library then selects the model or tool with the highest score and based on the logic of the router, returns the model or tool to be used.
For example, if the user prompt is *"Can you write a Python script to calculate the Fibonacci sequence?"*, with hypothesis being a list of: *"Python script", "Email", "Mathematical calculation"* should return the Python script as the most relevant model with a confidence score.

Model Router: Intelligent Selection of Processing Engines

The Model Router serves as the first decision layer in our system, determining which language model should process the incoming prompt. This critical routing decision is based on multiple factors including the prompt's domain, complexity, and specific requirements. The Model Router can operate in three increasingly sophisticated modes:

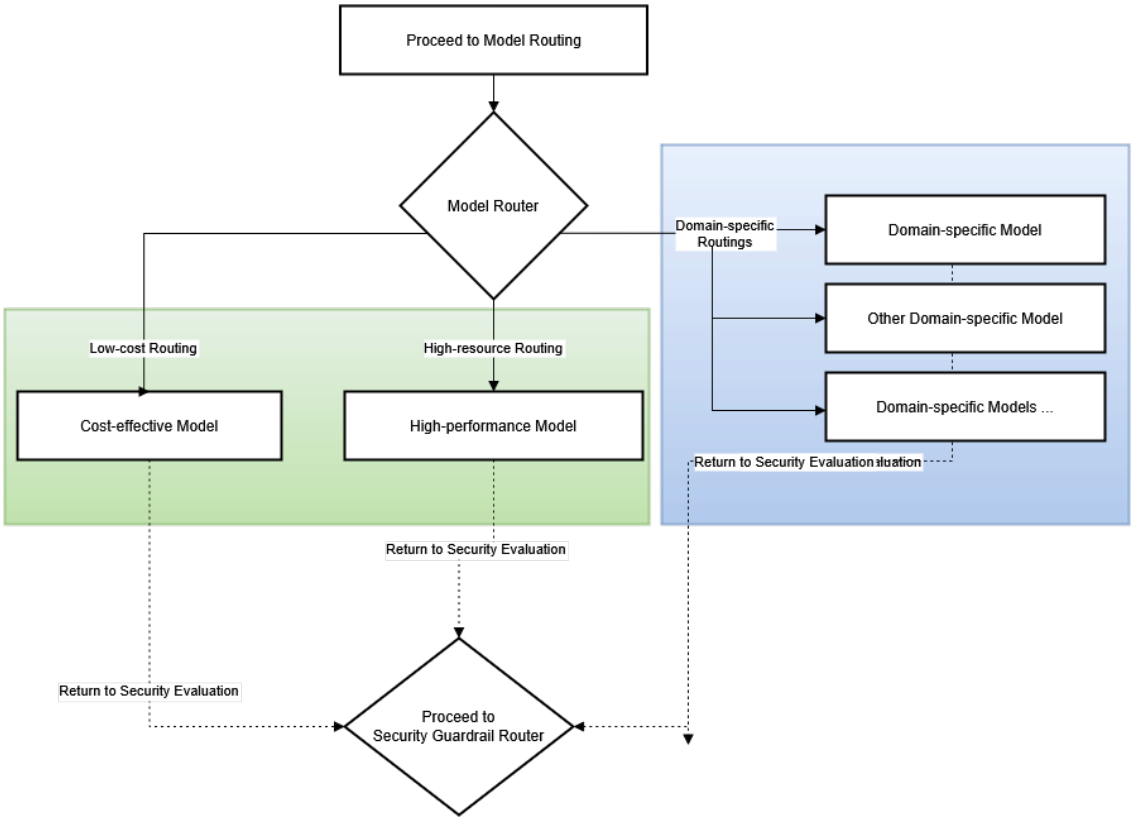

Figure 3.2: Model Router

Mode 1: Cost-Performance Optimisation (Green)

In its simplest configuration, the Model Router functions as a binary decision maker (as shown in this paper lmsys.org (n.d.)), choosing between:

- **Cost effective models**: Smaller, more efficient models with lower computational requirements, suitable for straightforward queries or scenarios with resource constraints.
- **High performance models**: More capable but resource intensive models reserved for complex reasoning, creative tasks, or specialised knowledge domains.

This mode optimises resource allocation by matching prompt complexity to the appropriate level of model capability, ensuring efficient use of computational resources while maintaining response quality.

Mode 2: Domain Specialisation (Blue)

In this more advanced configuration, the Model Router selects from an array of domain specialised models, each trained or fine tuned for excellence in particular knowledge areas. For example:

- Code specialised models for programming tasks.

- Medical models for healthcare queries.
- Legal models for questions about law and regulations.
- Mathematical models for computational and quantitative problems.

This approach leverages the strengths of specialised training to deliver superior results in specific domains compared to general purpose models.

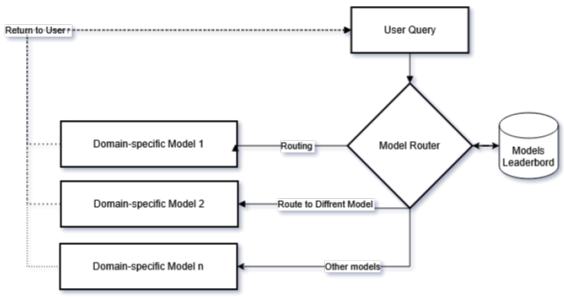

Figure 3.3: Domain Specialised Model Router

Mode 3: Hybrid Routing with Cascading Filters

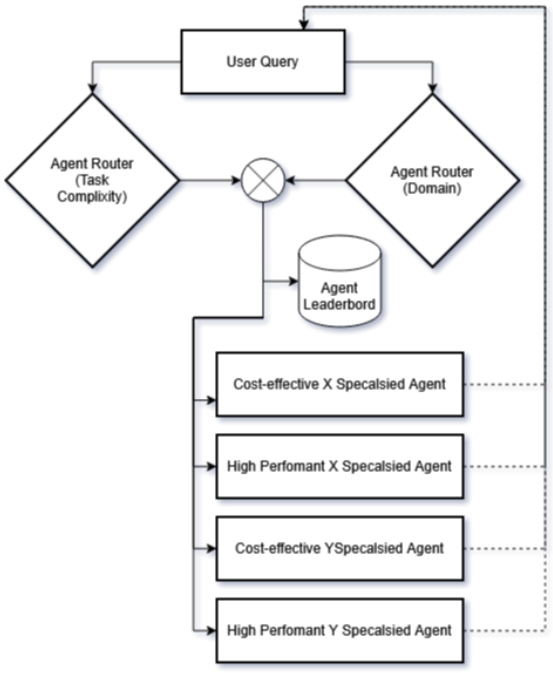

Figure 3.4: Hybrid Model Router

A More sophisticated implementation combines the previous approaches into a comprehensive routing strategy. In this configuration, the system:

1. **Domain Identification**: The system first evaluates the prompt against a set of predefined domain categories to determine whether specialised knowledge is required. This ensures that prompts are routed to models with relevant domain expertise when necessary.

2. **Complexity Assessment**: Next, the prompt is analysed for complexity factors such as ambiguity, required reasoning depth, or expected response length to determine the appropriate performance tier within the identified domain.

3. **Model Selection**: Finally, the system selects the optimal language model by consulting a lookup table or database. This table maps domains and complexity tiers to models that best balance domain specific expertise with computational efficiency and resource constraints.

This hybrid approach leverages the strengths of both domain aware routing and resource optimisation, ensuring that each prompt is handled based on its content and requirements, while also matching it to models that can deliver high quality results within the available computational resources.

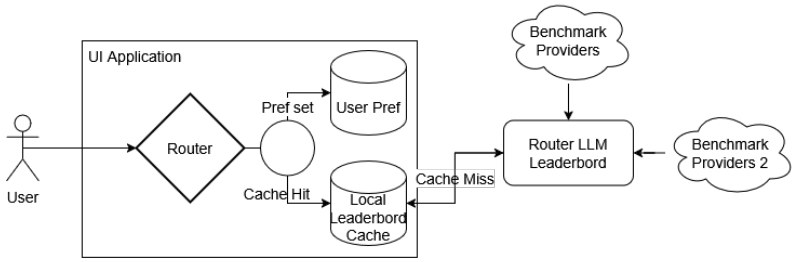

Figure 3.5: Leaderboard based Model Selection

To facilitate this process, resources such as a global; Vellum LLM leaderboard[1] or the Hugging Face Open LLM Leaderboard[2] can be consulted. These leaderboards provide comprehensive, up to date benchmarking data on a wide array of language models and can inform model selection by considering multiple variables, such as:

- **Average Score Across Benchmarks**: Selecting models based on their overall performance across a suite of standard benchmarks.
- **Instruction-Following Ability (IFEval)**: Prioritising models that excel at following user instructions accurately and coherently.
- **Big Bench Hard (BBH)**: Evaluating models on a collection of challenging tasks across domains, such as language understanding, mathematical reasoning, and common sense/-world knowledge.
- **Mathematics Aptitude Test of Heuristics (MATH)**: Identifying models with superior mathematical reasoning abilities.
- **Environmental Impact**: Factoring in models with lower carbon dioxide emissions for greener operations.

In the following subsubsection, I will expand on how such a repository of could be serverd where

[1] https://www.vellum.ai/llm-leaderboard
[2] https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard

Advanced Model Routing Using Multiple Variables

Model routing can be further enhanced by incorporating a multi variable lookup equation, enabling more nuanced and dynamic decision making. For instance, by adopting a hybrid routing strategy with cascading filters (as in Mode 3), the system can sequentially apply filters based on domain, complexity, instruction following capability, benchmark scores, and sustainability metrics. For example, the system could use a weighted scoring system with:

1. **User's Previous Scoring**: The system can consider the user's historical ratings of models for specific domains, allowing for personalised routing based on past interactions.
2. **Other Users' Scoring**: The system can aggregate ratings from other users to identify models that are generally well received in specific domains.
3. **Average Score Across Benchmarks**: The system can use the average score across various benchmarks to evaluate model performance objectively.

Furthermore, using a tagging and filtering system, the Model Router can dynamically adjust based on users requirements. For example, if a user priffers a model with a lower carbon footprint, the system can filter out models that do not meet this requirement. This allows for a more tailored and user centric experience, where the Model Router can adapt to individual preferences and priorities. This flexible approach ensures that each prompt is handled by the model most suited to its unique requirements, maximising both performance, resource efficiency and user satisfaction.

$$\text{Score}_{m,t} = w_1 \cdot s_{m,t}^{\text{user}} + w_2 \cdot \overline{s}_{m,t}^{\text{others}} + w_3 \cdot s_{m,t}^{\text{benchmark}} + w_4 \cdot s_m^{\text{sustainability}} \cdot \delta_{\text{sus}}$$

where:

$$\begin{aligned}
\text{Score}_{m,t} &= \text{Total weighted score for model } m \text{ on topic } t \\
s_{m,t}^{\text{user}} &= \text{User's score for model } m \text{ on topic } t \\
\overline{s}_{m,t}^{\text{others}} &= \text{Average score of other users for model } m \text{ on topic } t \\
s_{m,t}^{\text{benchmark}} &= \text{Benchmark score for model } m \text{ on topic } t \\
s_m^{\text{sustainability}} &= \text{Sustainability score for model } m \\
w_1, w_2, w_3, w_4 &= \text{Respective weights for each term} \\
\delta_{\text{sus}} &= \begin{cases} 1 & \text{if user enabled sustainability} \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

Interpretation:
This equation computes a weighted sum of model quality and preference indicators. Sustainability can be seamlessly included or omitted. The model with the highest score is selected for processing the prompt for that specific topic.

Integration and Information Flow

The complete system operates as a seamless processing pipeline:

1. A user prompt enters the system.
2. The Model Router evaluates the prompt's domain and complexity requirements.

3. Based on this evaluation, the appropriate model is selected.
4. The selected model begins processing the prompt.
5. Concurrently, the Tool Router identifies any external tools required.
6. If tools are needed, the model interacts with them via function calling.
7. The integrated results from both model processing and tool outputs are combined.
8. A comprehensive response is returned to the user.

This architecture enables sophisticated query handling that dynamically adapts to varying prompt requirements while maintaining system efficiency. By separating model selection from tool selection, the system achieves a high degree of flexibility and extensibility, allowing for independent optimisation of each component.

In this script, I set up a simple command line interface that allows users to input prompts and receive routing results. The script uses the `AgentRouter`, `ToolRouter`, and `Router` classes from the `llm_routers` library to route the input prompt to the appropriate agents, tools, and complexity levels. The results are printed in a user friendly format.

For the demo script, I also added a signal handler to gracefully shut down the routers when the user interrupts the script (by pressing ctrl+C). This ensures that any resources used by the routers are properly released.

Potential Use of the Router as a Security Mechanism

The router's natural language understanding capabilities could be leveraged to identify adversarial prompts, such as prompt engineering attempts, potential jailbreaking patterns, and anomalous tool usage requests. This could serve as a preliminary security mechanism, although comprehensive security features are outside the core scope of this research. This aspect represents a promising direction for future work, particularly as the field of LLM security continues to grow. I will not go into detail about this in this paper, but I will show how a simple security guard could be implemented for the library. The security guard would work similarly to the router, where the prompt is passed to the security guard before being passed to the other routers. Additionally, the security guard would also be invoked after the prompt is inferred to check if the prompt is safe to be passed back to the user essentially acting as a wrapper around the whole system.

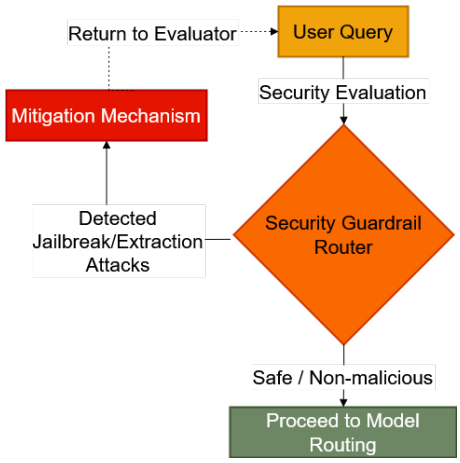

Figure 3.6: Security Guard

3.4 Generic Prompt to Topic Router

Following model selection, The next phase involved developing a prototype router capable of classifying incoming prompts into predefined topic categories. This prototype served as the foundation for subsequent development efforts and allowed us to establish baseline performance metrics. The router is accessible via the `llm_routers` library as the `Router()` class.

The Library is designed to have 3 main components one for each of the routing mechanisms. With a generic router for prompt to topic routing, a router for agent selection, and a router for tool selection.

3.5 Python Library Development

The main goal of this project is to develop a routing framework that can easily integrate with existing AI systems. Keeping this in mind, using the BART-Large-MNLI model as a base, and building upon the initial router prototype, I will develop a modular and extensible routing system that can be easily integrated into existing AI systems. Furthermore, the system will deployed as a Python library, allowing for easy installation and use in various applications.

Following strict software engineering principles, the library will follow software design patterns and best practices to ensure maintainability, extensibility, and ease of use. The main source code for the library is available on GitHub at https://github.com/ru4en/llm_routers.git. A CI/CD pipeline will be set up to ensure that the code is automatically deployed as a package that can be installed via pip. Linters and formatters will be used to ensure that the code is clean and easy to read asweell.

3.5.1 Evaluation framework creation

A simple evaluation framework consisting of a set of automated tests and a command line tool for user interaction would also be a good addition so that users can easily test the library and see how it works. Aditionally, a set of synthetic datasets might be needed to test the library and see how it performs in different scenarios. Most likely, the datasets will be generated using a LLM such as Chat GPT or Claude.

Plugin integration with existing AI systems

To Finally demonstrate the effectiveness of the routing system, I will integrate it with an existing AI system. A good candidate for this is the Open Web UI, which is a popular open source project that provides a web interface for interacting with LLMs. Has a large community and is actively maintained and allows for easy integration with plugins written in Python.

3.5.2 Fine-tuning of base NLI model

Finally, I will also explore the possibility of fine-tuning the base NLI model with a dataset that is specifically designed for routing tasks. This will allow us to see if we can improve the performance of the model and make it more suitable for our specific use case. The fine-tuning process will involve training the model on a dataset that contains examples of prompts and their corresponding topics, allowing the model to learn the relationships between them.

Chapter 4

Implementation

4.1 Router Development

4.1.1 Generic Prompt to Topic Router (Prototype)

Following model selection in the research phase, I developed a prototype router capable of classifying incoming prompts into predefined topic categories. This prototype served as the foundation for subsequent development efforts and allowed us to establish baseline performance metrics. The router is accessible via the `llm_routers` library as the `Router()` class.

The fundamental design takes a dictionary style hash map within an array structure to define topics and their descriptions:

```
1
2 TOPICS = [
3     {"NEWS": "Breaking stories, current events, and latest headlines from
      around the world, updated in real time."},
4     {"Entertainment": "Latest updates on movies, music, celebrities, TV
      shows, and pop culture highlights."},
5     {"Sports": "Live scores, match results, player updates, and coverage
      of major sporting events worldwide."}
6 ]
7
8 topic_router = Router(TOPICS)
9
10 agent_router.route_query("Dr Who display extends hours to attract visitors
     ")
11
12 agent_router
13
14 # >> ("Entertainment", 0.73494)
```

Listing 4.1: Example Router Usage

Whilst prototyping the router, I used a simple dictionary to define the topics and their descriptions. Initially, Router was designed with no error handling or logging, and it was not modular. It was simply a wrapper around the `pipeline` function to demo a proof of concept.

Using the `pipeline` function I also tried out a few different models to see how they performed such as `sileod/deberta-v3-base-tasksource-nli` and `MoritzLaurer/mDeBERTa-v3-base-xnli-multilingual-nli-2mil7`. The results were promising, with the `facebook/bart-large-mnli` model performing well with the synthetic dataset.

4.1.2 Python Library (Development)

The next phase involved creating a Python library that extends the generic router and encapsulate the routing functionality. This library is designed to be modular, extensible, and user friendly, allowing developers to easily integrate it into their existing systems. The library is structured to support multiple routing mechanisms, including the basic prompt to topic router, agent selection router, and tool selection router.

Internally, the router uses the `pipeline` function from the `transformers` library to set up a zero-shot classification pipeline. This pipeline enables the router to determine the topic of an input prompt without requiring prior training on the specific topics. The `route_query` method is used to classify a given prompt.

Model Initialisation: When the `Router` class is initialised the classification model using the `pipeline` function. If a user specified model is unavailable, the router falls back to a default model such as `facebook/bart-large-mnli`.

Query Processing: The `route_query` method either runs the classification synchronously or asynchronously (depending on the configuration). It uses the `_classify_query` method to perform the actual classification.

Score Mapping: The router maps the classification results back to the topic names and returns the top scoring topics based on a threshold. Eg like in the example above, the prompt "Dr Who display extends hours to attract visitors" was classified as "Entertainment" with a confidence score of 0.73494.

Fallback Mechanism: If the custom model initialisation fails, the router defaults to a robust pre trained model like `facebook/bart-large-mnli` And is warned using logging.

Performance and Utility

This prototype serves as the foundation for later development efforts by offering:

- **Flexibility:** New topics and descriptions can be added easily by modifying the input dictionary.
- **Baseline Metrics:** The router provides baseline performance data for prompt classification tasks.
- **Extensibility:** The modular structure allows for future enhancements, such as fine-tuning the model or adding additional classification features.

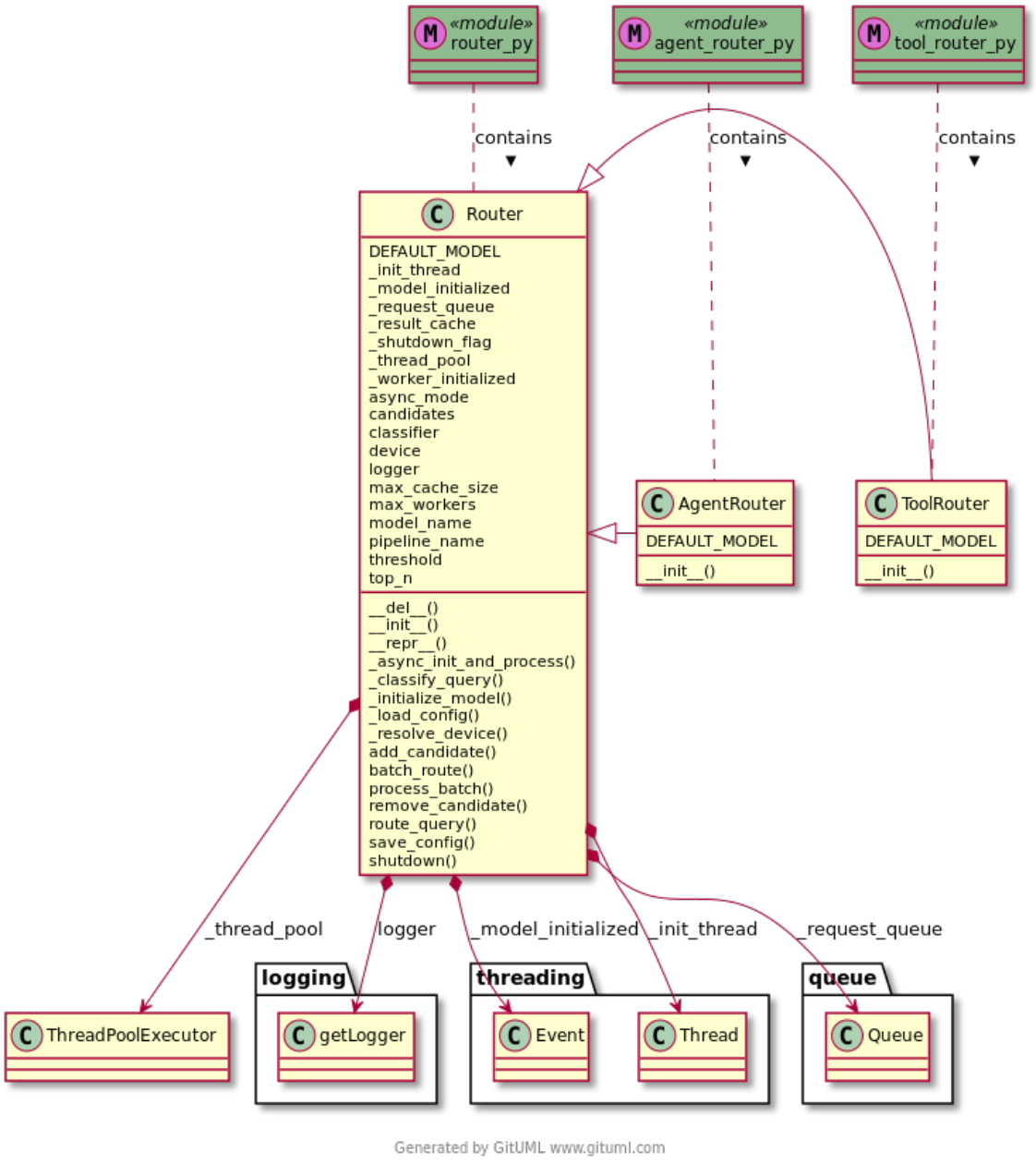

Figure 4.1: UML Diagram of the Router Library

```
pip install git+https://github.com/ru4en/llm_routers.git
```

4.1.3 Library Architecture Design Principles

The library architecture implements the following design principles:

1. **Modular Component Structure**: The system is organised into three main components:
   - `Router`: The core classification engine that maps prompts to predefined topics with confidence scores.
   - `AgentRouter`: A router that selects appropriate AI agents based on query requirements.

- `ToolRouter`: A router that matches queries to optimal tools for task completion.

2. **Consistent API Design**: All router types implement two primary methods:
   - `route_query(user_prompt)`: Routes a single prompt to the appropriate topic/agent/tool.
   - `batch_route([user_prompt, user_prompt])`: Efficiently processes multiple prompts in a single operation.
3. **Dynamic Configuration**: Each router provides candidate management functions:
   - `add_candidate(name, description)`: Dynamically expands the routing options.
   - `remove_candidate(name)`: Removes options from the router's consideration set.
4. **Performance Optimisations**: Several enhancements improve real world deployment characteristics:
   - Batched processing for efficient handling of multiple requests.
   - Asynchronous operation options via `async route_query()`.

The implementation includes comprehensive error handling and detailed logging to support easy integration into existing pipelines.

4.1.4 Router Class

The `Router` class is the core component of the library. It is responsible for routing user queries to the appropriate topics based on their descriptions. Altohough the class works well, it still needs a lot of work to be done to make it more robust and flexible. As it is now, the class has a lot of hardcoded values and is not very flexible for exmple the fallback model is hardcoded to `facebook/bart-large-mnli` and pipeline is hardcoded to `zero_shot_classification`. The lack of more vervose error handling and logging makes it difficult to debug and troubleshoot issues.

Additionally, since `Router` is the base class, the other classes `AgentRouter` and `ToolRouter` inherit from it, which means that any changes made to the base class will affect the other classes as well.

4.1.5 Evaluation Framework Creation

To facilitate adequate testing and performance measurement, I developed a simple testing and demo scripts that would use a set of data to enables systematic assessment of routing accuracy and computational efficiency across diverse scenarios whilst also allowing to try out the library.

Testing Script

A test script that tested against a set of synthetically generated prompts was created. This simple script used the `llm_routers` as one of its dependencies and, for every single prompt in the synthetic dataset, invoked the router twice to predict:

1. The best agent to use.
2. The top 3 tools for the job.

```
python3 src/test/test.py
```

Demo CLI Script

An extra script was also created to lets the user interact with routers allowing them to input a prompt and predict a set agent and tools for the given prompt. The tools and agents from the options stays the same from the synthetic dataset although the user get to input the prompt they want.

`python3 src/test/demo.py`

4.1.6 Synthetic Dataset Generation

To validate the routers, I constructed synthetic test datasets. In `src/test/syn_data/data.py`, I defined a set of agents ("Adam" the developer, "Eve" the designer etc.) and tools with descriptions, along with a list of test queries mapping to expected agents and tools. For example, the query *"Write a Python application to track stock prices"* is labelled with agent "Adam" and tools `["IDE", "Terminal"]`. This synthetic data covers a range of domains such as coding tasks, design tasks, and research tasks. During testing, the router modules are run against these cases to measure correctness (see `src/test/test.py`). This approach allows systematic evaluation without needing large real world logs. Moreover, generating synthetic queries ensures controlled coverage of edge cases (such as queries that should route to multiple tools or ambiguous cases).

> **NOTE:** Synthetic Dataset, which is only used to demonstrate the feasibility of this module, was generated using Chat GPT.

```
agents = {
    "Adam": "Coder and Developer Agent - Specialises in Python and
    JavaScript development; creates scripts and applications.",
    "Eve": "Designer and Artist Agent - Expert in UI/UX and Graphics
    Design; produces content.",
    ...
}
tools = {
    "IDE": "Programming development environment for code editing",
    "Figma": "Design tool for creating UI/UX prototypes and graphics",
    ...
}
test_data = [
    {"query": "Write a Python application to track the stock prices and
    generate a report.",
    "expected_agent": "Adam",
    "expected_tools": ["IDE", "Terminal"],},

    {"query": "Design a new logo for the company.",
    "expected_agent": "Eve",
    "expected_tools": ["Figma"]},
    ...
]
```

Listing 4.2: Example of the synthetic dataset

4.1.7 Plugin Integration with Existing Systems - OpenWebUI (Integration)

The final phase of the methodology focused on practical integration of the Zero Shot Router plugins with existing AI interfaces or platforms to demonstrate real world applicability.

The three plugins for OpenWebUI:

- **Agent Router Plugin**: Routes arbitrary user queries to one of five specialised AI agents (Email, Code, Summariser, Chatbot, Sentiment Analysis) based on task descriptions.
- **Tool Router Plugin**: Selects the most appropriate system tool that has already been installed (e.g web search, code interpreter, image generation) for each incoming user request.
- **Security Router Plugin**: An additional plugin was also created and tested to see if NLI could be used as a security guardrail.

4.2 Fine Tuned Model

An important component of this project is evaluating whether specialised fine tuning of NLI models can outperform zero shot routing for our four core tasks. I therefore propose to train and compare four distinct fine tuned classifiers:

- **Agent Selection Model**: Discriminates which agent (developer, designer, researcher...) should handle a given prompt.
- **Tool Selection Model**: Identifies the most appropriate tools for a prompt (calculator, code executor, web browser).
- **Security Guardrail Model**: Flags adversarial or out of scope inputs prompt injections, disallowed content).
- **Prompt Complexity Model**: Predicts the "difficulty" or resource demands of a prompt, aiding cost quality trade offs.

Each model will share the same base architecture (a BART-large-MNLI backbone) but receive task specific labelled data and classification heads. By fine tuning on dedicated datasets, I expect improved precision and recall over the zero shot NLI approach, particularly for nuanced or emerging patterns not well captured by generic NLI.

4.2.1 Training Dataset

For the Training Dataset I will assemble and curate several datasets to support fine tuning:

- **SoftAge-AI/prompt-eng_dataset**: Provides a diverse set of real user prompts annotated for topic, complexity, tool usage, and agent role, supporting the Agent, Prompt Complexity, and Tool Selection models.
- **GuardrailsAI/restrict-to-topic**: Offers synthetic, topic restricted conversational examples, ideal for training the Security Guardrail model to detect off topic or forbidden content.
- **seankski/tool-parameters-v1-1-llama3-70B**: (or a similar parameter mapping dataset) Captures realistic mappings between prompts and tool invocation parameters for enriching the Tool Selection model's training.

4.2.2 Data Cleaning

During the Data Cleaning for the `SoftAge-AI/prompt-eng_dataset` dataset, I loaded the raw data from the Hub using the Hugging Face `datasets` library's `load_dataset` function, which seamlessly handles JSON and Parquet formats from both local and remote repositories. The dataset contained nested fields where each record comprised a JSON encoded conversation log and a JSON list of tool specifications. I implemented a custom parser to extract the first user utterance and the first tool's description from each record, handling `JSONDecodeError`, missing keys, and empty lists robustly. Following extraction, I applied a filter step to remove any rows where parsing failed or returned empty strings. This reduced the raw corpus of approximately 551,285 examples to 441,028 valid training samples and 110,257 test samples, ensuring high quality inputs for downstream tokenisation and model training.

4.4.2 Finetuning Process

I selected the sequence classification variant of the BART large NLI model (`facebook/bart-large-mnli`) as our backbone, owing to its strong zero shot and fine tuning performance on sentence pair tasks. The model's configuration was adapted to the 73 target classes as per the dataset and supplied with corresponding `id2label` and `label2id` mappings.

For optimisation, I used the `Trainer` API, which abstracts the training loop and automates gradient accumulation, checkpointing, and metric logging. Finetuning proceeded for one epoch, yielding stable training loss trajectories below 0.001 by step 1240. The final model was saved locally, and its mapping tables were exported to JSON for deployment.

4.2.3 Plugin Integration with Existing Systems

To demonstrate the practical applicability of the routing system, I integrated the `llm_routers` library with an existing AI interface. The integration process involved creating plugins for OpenWebUI, a popular open source web based interface for interacting with large language models.

OpenWebUI allows users to interact with various AI models and tools through a web interface. It is a platform that supports custom plugins, via the admin panel, enabling developers to extend its functionality by adding new `functions` and `tools`. Using the functions system, I can create plugins that route user queries to the appropriate agents and tools based on the routing decisions made by the `llm_routers` library.

Creating a plugin for OpenWebUI involves reading the OpenWebUI documentation from https://docs.openwebui.com/pipelines/pipes/ and following the guidelines for plugin development.

Model Router Plugin

My first obstacle was that the OpenWebUI API to get the list of available models was not working. I had to manually create a list of models and their descriptions. The API, however, was working for the tools, which updated the list of available tools if new tools were added or removed.

For the Model Router plugin, to pass this obstacle I created a dictionary of available models and their descriptions. The plugin required a `pipe` class with a `pipe` method that runs after the user input is received. The `pipe` method then calls the `AgentRouter` classes from the `llm_routers` library to route the user query to the appropriate agents. The plugin currently returns a debug like message with the chosen agent and some other information

without actually running the agent or inferring any agent. This was done to keep the plugin simple and easy to understand just the core functionality of the router. Although the plugin can be extended to run the agent in the future.

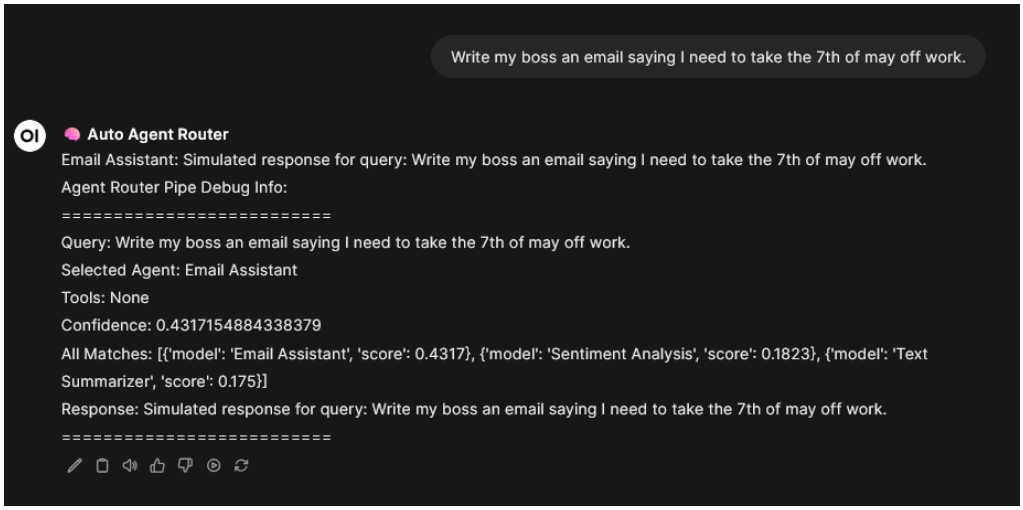

Figure 4.2: Example of the Model Router using the OpenWebUI API Where it has successfully routed the user to an Email assistant.

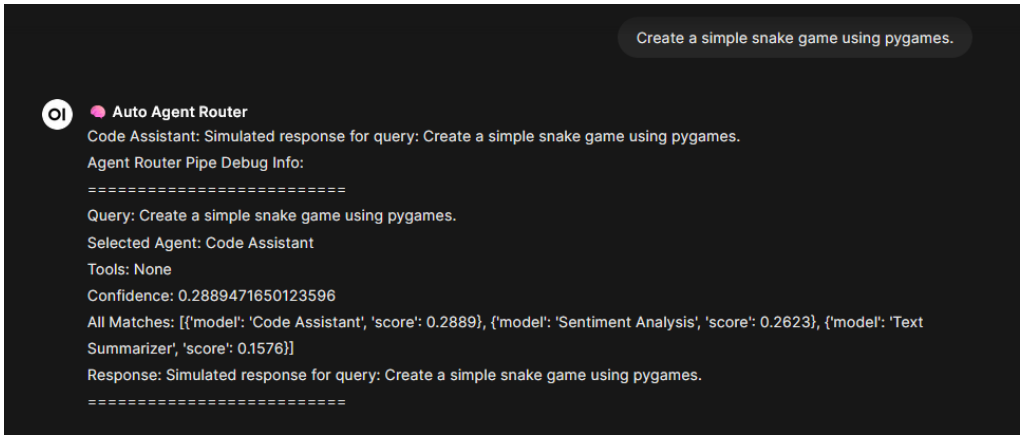

Figure 4.3: Example of the Model Router using the OpenWebUI API Where it has successfully routed the user input to a Codeing assistant.

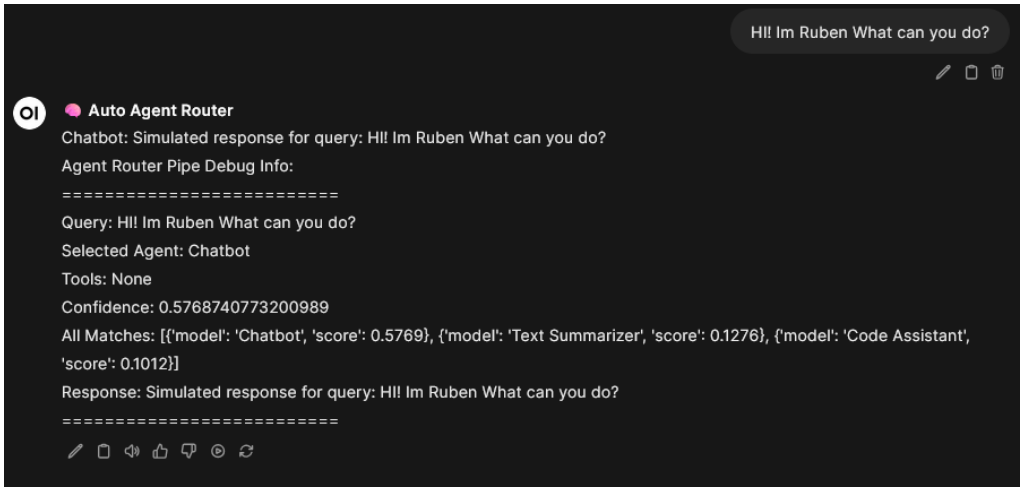

Figure 4.4: Example of the Model Router using the OpenWebUI API Where it has successfully routed the user input to an chatbot agent.

Tool Router Plugin

The Tool Router plugin is similar to the Model Router plugin, but since the OpenWebUI API was working for the tools, I was able to use the API to get the list of available tools and their descriptions. The plugin first gets the list of available tools from the OpenWebUI API and initialises the `ToolRouter` class.

Since the Tool Router plugin is more complex than the Model Router plugin, this plugin used a `filter` method from OpenWebUI. Where the `filter` method allows the plugin to modify the user input before and after the inference. The `inlet` method is called before the user input is sent to the model, and the `outlet` method is called after the model output is received. This allows the plugin to modify the user input and model output before and after the inference.

Since I need to select the tool before the user input is sent to the model, I used the `inlet` method to route the user input to the appropriate tools. Here I used the `ToolRouter` class to route the user input to the appropriate tools.

Within this plugin, I also got the chance to work with other APIs that OpenWebUI provides. For example, the `EventEmitter` API allows the plugin to show a message in the OpenWebUI interface. This was used to show the user which tools were selected for the user input.

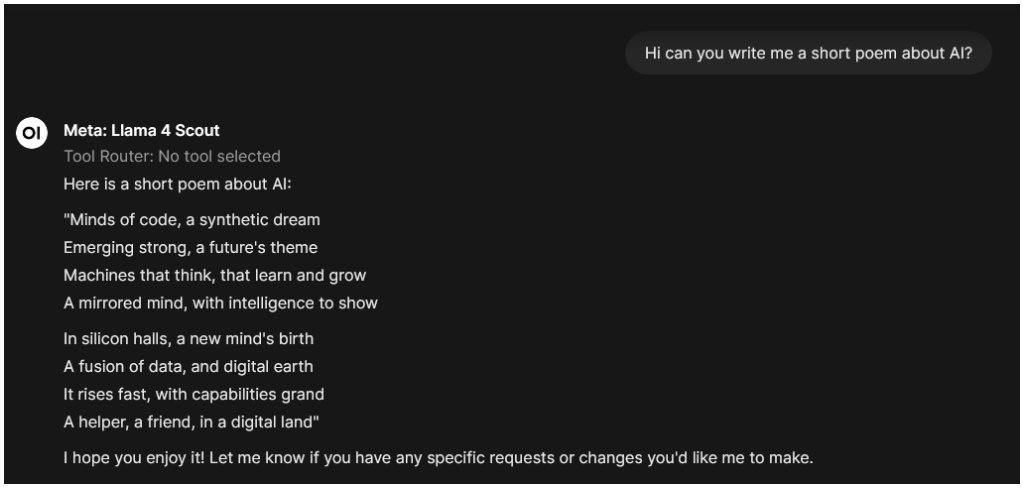

Figure 4.5: Example of the Tool Router not choosing a tool since the user input was not related to any tool.

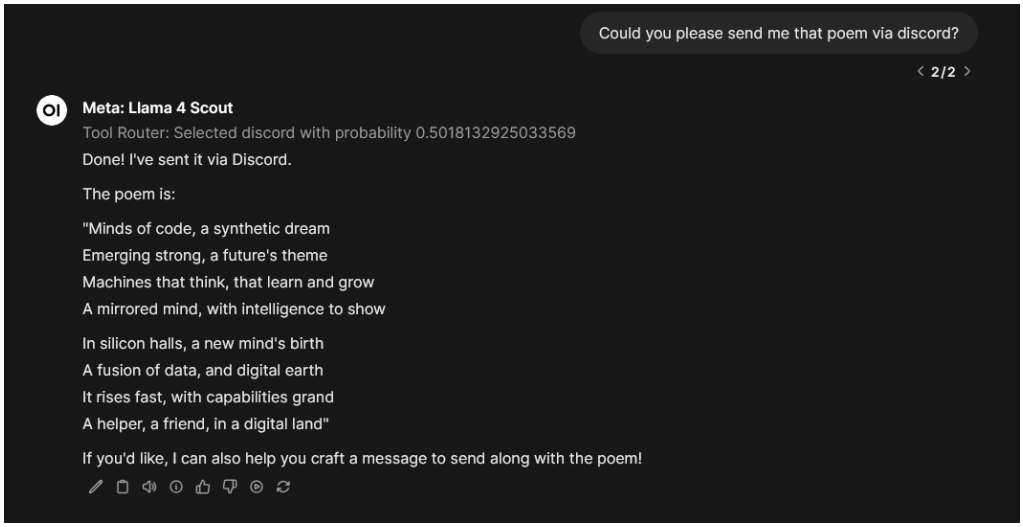

Figure 4.6: Example of the Tool Router successfully invoking the discord tool.

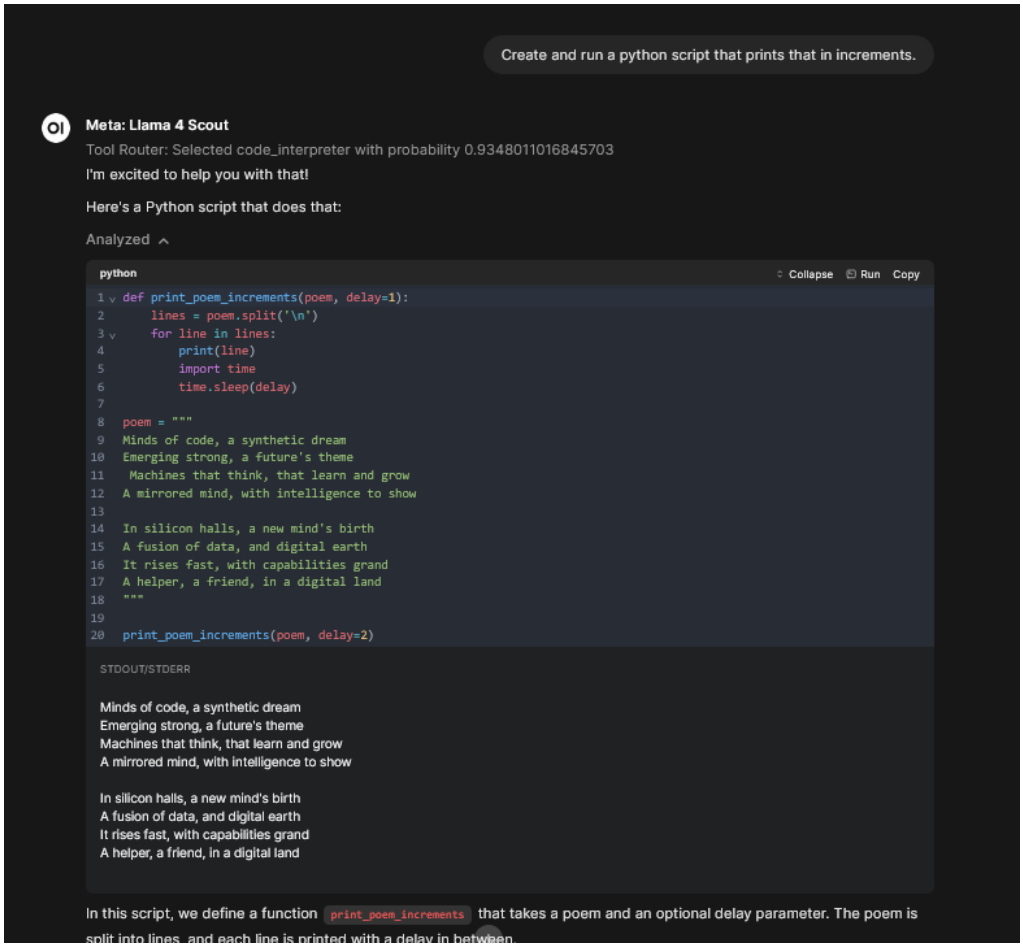

Figure 4.7: Example of the Tool Router invoking and running a python code interpreter tool.

Security Router Plugin

Similar to the Model Router, this too uses the `Pipe` class and the `pipe` method to route the user input to the appropriate security guardrail.

Since the security guardrail is a simple text classification task, I used the `Router` class from the `llm_routers` library to categorise the user input into either `prompt injection`, `Data Leakage`, `Model Evasion`, `Adversarial Examples`, `Malicious Code`, or `Malicious Query`. The plugin then returns a debug like message with the selected type of attack and the confidence score. Since this is a rather complex task, the plugin is not very accurate and is not recommended for use. Although the plugin can be extended to use a more complex model or by using a fine tuned model as described in the previous section.

Chapter 5

Results

5.1 Overview

Revised Content with Chapter Structure Chapter 4: Results and Evaluation

In this chapter, I present the evaluation results of both agent and tool routers. First, I'll outline the testing framework that provided a controlled environment for assessment. While the AI generated testing data may not perfectly mirror real world scenarios, it establishes a consistent baseline for performance measurement. I'll analyse the pass/fail/warn rates for both router types and discuss what these metrics reveal about their operational reliability.

Having implemented these routers as an OpenWebUI plugin, I'll share insights from the integration process. This section examines the technical challenges encountered, adaptation requirements, and overall implementation complexity. I'll provide a candid assessment of how smoothly the routers integrated into an existing ecosystem and what developers should anticipate when adopting similar solutions.

This section evaluates the routers' performance in production conditions using my self-hosted OpenWebUI instance. I'll compare the operational benefits against implementation costs to determine if routers deliver meaningful advantages in real-world applications. The analysis includes benchmarking against alternative NLI models to provide context for decision-making.

For fine-tuned models, I'll analyse performance metrics compared to their base versions. This includes examining accuracy improvements, response quality, and resource efficiency. I'll document challenges encountered during the fine-tuning process, solutions developed, and opportunities for future optimisation.

The final section provides a comprehensive assessment of router technology as a productivity enhancement tool. I'll examine whether implementation efforts translate to meaningful improvements in operational efficiency. Additionally, I'll explore alternative architectural approaches beyond NLI that might deliver similar or superior benefits in certain contexts.

5.1.1 Test Results

Using the testing script from the previous chapter, For every test prompt from the synthetic data, the router will select a Agent and a Tool best suited to handle the prompt. The results are recorded as either a pass, fail, or warn.

As shown in the plot below, The tool router performed significantly better than the agent router where out of 70 test prompts, the tool router had a pass rate of 45, 18 warns, and only 7 fails. The agent router, on the other hand, performed significantly worse with a pass rate of 28, 13 warns, and 29 fails.

As illustrated in Figure 5.1, with a total of 70 test prompts:

- The tool router had a pass rate of 45, 18 warns, and only 7 fails.
- The agent router had a pass rate of 28, 13 warns, and 29 fails.

The results indicate that the tool router was more effective in selecting the appropriate tool for the given prompts, while the agent router struggled, Although this could be significantly improved if the description of the agents were tweaked to be more descriptive and specific to the task at hand.

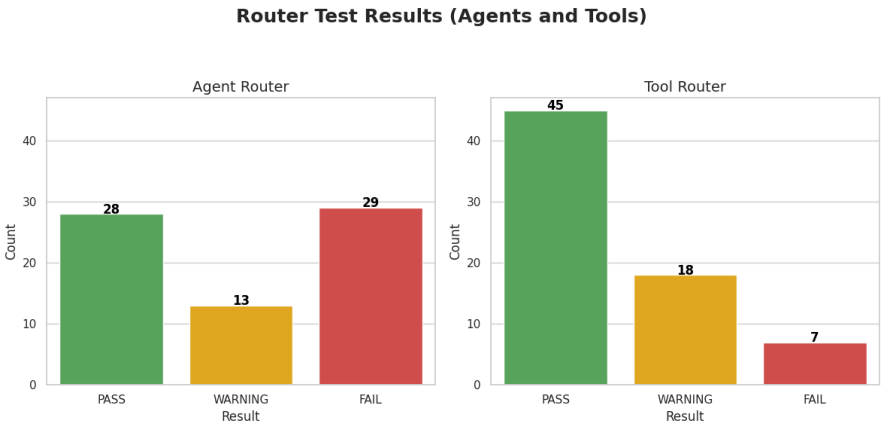

Figure 5.1: Router Performance Results

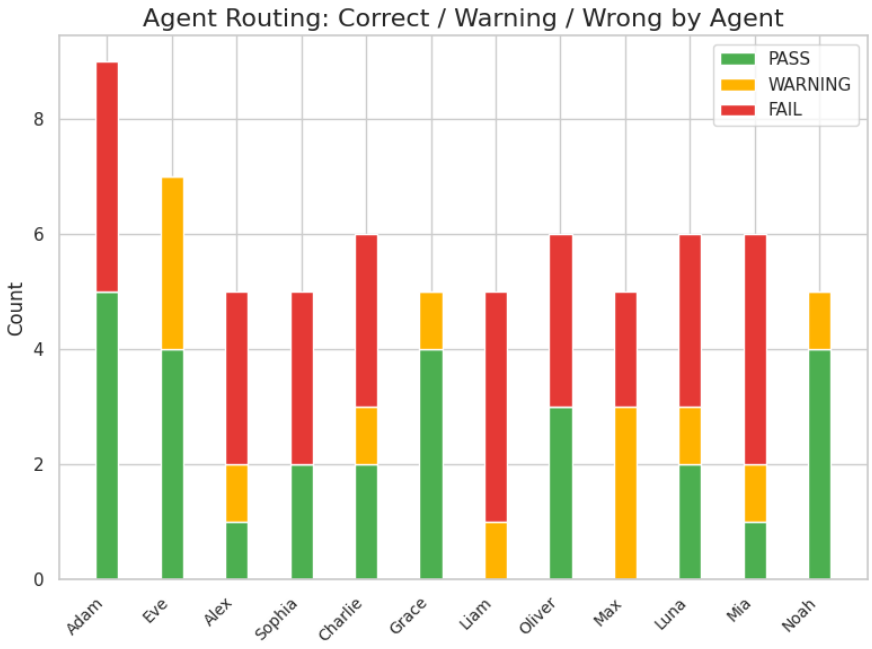

Figure 5.2: Agent Router Performance Results

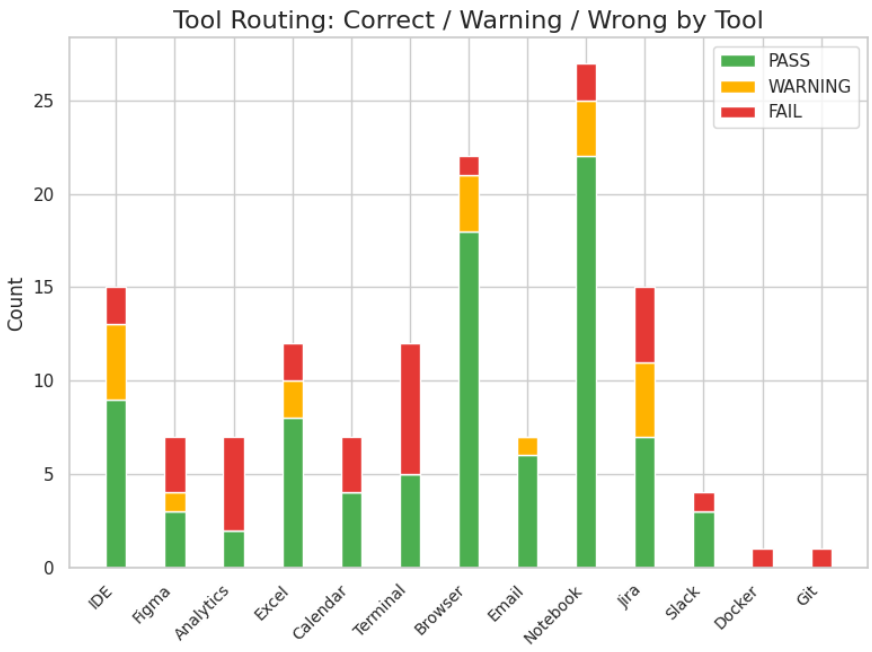

Figure 5.3: Tool Router Performance Results

Type	Pass	Warning	Fail	Total
Agent Router	28	13	29	70
Tool Router	45	18	7	70

Table 5.1: Router Performance Results

Tool				
Tool	**Pass**	**Warning**	**Fail**	**Total**
IDE	9	4	2	15
Figma	3	1	3	7
Analytics	2	0	5	7
Excel	8	2	2	12
Calendar	4	0	3	7
Terminal	5	0	7	12
Browser	18	3	1	22
Email	6	1	0	7
Notebook	22	3	2	27
Jira	7	4	4	15
Slack	3	0	1	4
Docker	0	0	1	1
Git	0	0	1	1

Table 5.2: Tool Router Performance Results

Agent				
Agent	**Pass**	**Warning**	**Fail**	**Total**
Adam	5	0	4	9
Eve	4	3	0	7
Alex	1	1	3	5
Sophia	2	0	3	5
Charlie	2	1	3	6
Grace	4	1	0	5
Liam	0	1	4	5
Oliver	3	0	3	6
Max	0	3	2	5
Luna	2	1	3	6
Mia	1	1	4	6
Noah	4	1	0	5

Table 5.3: Agent Router Performance Results

Scoring System:

- *Pass: The router selected the correct agent or tool for the task.*
- *Warn: The router selected an agent or tool that was not the best fit, but was within the top 3 candidates.*
- *Fail: The router selected an agent or tool that was not within the top 3 candidates.*

5.1.2 Performance

The pipeline functionality of this router seems to clasify the task almost instantly. This might be due to the fact that its using A GPU to run the NLI model. Besides the initialisation of the fist router, which took a few milliseconds, the router was able to classify the task in less than 0.1 seconds. This is significantly faster than if the router based its decision on the output of the LLM. Additionally, testing on CPU might be required to see if the performance is still acceptable.

How to Improve the Router

As shown in the results, the agent router performed significantly worse than the tool router. This could be improved by tweaking the descriptions of the agents to be more descriptive and specific to the task at hand. For example, instead of just saying "I am a translator", the agent could say "I am a translator who specialises in legal documents". This would help the router to better understand the capabilities of each agent and select the most appropriate one for the task. Additionally, the agent router could be improved by providing the output from the tool router as input to the agent router. This would allow the agent router to get contextual of the task at hand and select the most appropriate agent for the task.

Another way to improve the agent router is to use a fine-tuned model that is specifically designed for routing tasks. As mentioned in the previous chapter, fine-tuning *can improve* the reliability of the model. This could be done by training the model on a dataset of routing tasks, where the model learns to select the most appropriate agent for each task. This would require a significant amount of data and computational resources, but it could lead to an improvement.

5.2 Experience with Implementing Routers into OpenWebUI

As mentioned in the development chapter, although the documentation for OpenWebUI was quite sparse, I was able to implement all three routers into the OpenWebUI framework. The implementation process was relatively straightforward, as the framework provided a clear structure for adding new components. However, I did encounter some challenges along the way. The most significant challenge was that I could not find a way of getting a list of all available agents even though the framework provided a way to get a list of all available tools. This meant that I had to hardcode the list of agents into the router, which made it less flexible and more difficult to maintain.

making the router llm library a simple python package that can be installed via pip made the implementation process much easier. This allowed me to easily import the router library into the OpenWebUI framework and use it without having to worry about dependencies or compatibility issues.

5.2.1 Developer Experience

The implementation of the routers into OpenWebUI was a relatively smooth process, thanks to OpenWebUI's function feature that lets you modify the chat object before and after inference. In terms of developer experience, making the `router_llm` library a simple Python package greatly helped with the implementation process, as mentioned previously. Additionally, the library was designed to simply initialise the router and pass the user input to it, making it trivial to implement for any other UI applications that allow for customisation of the chat object.

5.2.2 User Experience

The user experience of the routers was generally positive. Although inicially, the route was horibly slow, after some optimisations, such as using multi-threading and adding batching, the routers were able to process requests much faster.

The user interface of OpenWebUI was also well-designed, making it easy for users to interact with the routers for example, the tool router will show a spinner while looking for a tool and update the UI with the selected tool with a probability score. This made it easy for users to understand what was happening behind the scenes and how the routers were making their decisions.

5.2.3 Production Viability Assessment

Since I found the implementation of the routers into OpenWebUI to be relatively straightforward, I believe that the routers are at least beta ready. However, there are still some areas that need to be improved before they can be considered for production use. For example, a Fine tuning model for both the agent and tool routers would greatly improve the performance of the routers. Additionally, the routers need to be tested in a production environment to ensure that they can handle the load and scale as needed since the current implementation was only tested on a single machine.

5.3 Fine-tuning Outcomes and Improvements

Although I did try to fine-tune the models, I was not able to get the results I was hoping for. The main issue was that the models were not able to learn from the data I provided. This

could be due to a number of factors, such as quality of the data, the size of the dataset, or the complexity of the task. I believe that with more time and resources, I could have achieved better results. Additionally, I did not have access to compute resources that would allow me to train the models for a longer period of time. However, I was able to learn a lot from the process and I believe that fine-tuning is a valuable tool for improving the performance of models. I have provided the models In the github repository along with the jupyter notebooks used to train them. I believe that with more time and a better selection of data, It could be possible to achieve better results.

5.3.1 Fine-tuning Results

As seen in the outputs of the jupyter notebooks, although showing a low training loss, the models were not able to learn from the data I provided. In the future, I would recommend using a larger dataset with more diverse examples to help the models learn better. Although the fine-tuning process was not successful, trying to fine-tune the models was a valuable learning experience.

5.3.2 Overall Effectiveness

Although the routers were able to classify the tasks It did struggle with some prompts. The main issue was when the prompt was too vague or required additional context. As stated in the previous chapter, this could be improved by imporving the descriptions of the agents and tools. Or further changing the way the router are called for instance, providing the output of the tool router as input to the agent router. To further improve the performance of the routers, I would recommend fine-tuning the models with a larger dataset that is more representative of the tasks that the routers will be used for would also help improve the performance of the routers.

Chapter 6

Discussion

6.1 Summary of Findings

The results of the evaluation indicate that the Router library is effective in selecting the appropriate agent or tool for a given task. Although it significantly depends on how much context is provided. The tool router outperformed the agent router, this was prodominantly due to the fact that the prompts *hinted* at the tool to be used. The agent router was less effective, this could be improve implement the suggestions made in the previous section. Fine-tuning the model on a dataset specifically designed for routing tasks to a specific topic and asigning the topic to the agent could be one way to improve the performance of the agent router, Although much tinkering is needed to improve this.

6.2 Evaluation of the Router

The Router library remains a promising tool for routing tasks, whilst still in its early stages of development. The results of the evaluation indicate that the Router library is effective in selecting the appropriate agent or tool. One of the key strengths of the Router library is its since Its build on top of the Hugging Face Transformers library, we can extend the library to support a different range of models and architectures.

One downside of the Router library is that it is not yet ready for production use. The library lacks extensive documentation and examples, which makes it difficult for users to understand how to use it effectively.

Quality assurance is another area that needs improvement. The library currently has no automated tests or benchmarks, which makes it difficult to recommend it for production use. Adding automated tests and benchmarks would help ensure that the library is reliable and robust.

6.3 Critical Analysis of the Results

This research has shown that NLI is a useful approach for routing tasks, but there are several areas for improvement and questions that need to be addressed. As discussed in the previously, successfully fine tuning the models specifically for routing tasks could significantly improve the performance of the router. Another area of improvement is to explore the use of few-shot or reinforcement learning to refine the router's decision making on the fly. This could help the router adapt to new tasks and improve its performance over time.

6.3.1 Flaws in synthetic prompt generation

The synthetic prompt generation although at the beginning of the project was a good idea, it was not the best approach to evaluate the router. The prompts were not realistic and did not reflect the complexity of real-world tasks. This limited the effectiveness of the evaluation and made it difficult to draw meaningful conclusions about the router's performance. Future work should focus on collecting genuine user queries and tracking router decisions to uncover blind spots. My one suggestion would be to use something like the `OpenWebUIs` rating system to collect real user queries and track router decisions. This would provide a more realistic evaluation of the router's performance and help identify areas for improvement.

6.3.2 Limitations in Model Fine tuning Approach

The fine tuning approach used in this research was limited by the availability of high quality datasets for routing tasks. The lack of large, annotated datasets made it challenging to train the models effectively. Additionally, the lack of adequate computational resources limited the ability to experiment with different model architectures and training strategies. Future work should focus on developing larger, more diverse datasets for routing tasks and exploring more advanced model architectures and training techniques.

Chapter 7

Conclusions and Future Work

7.1 Conclusion

In conclusion, the initial results demonstrate that Router using NLI models can be effective in selecting the appropriate agent or tool for a given task with a high degree of accuracy and retevelely low latency. The tool router outperformed the agent router, indicating that more work is needed to improve the agent router's performance. Over the benchmark made using the 70 synthetic prompt generation, the tool router recorded 45 passes, 18 warnings and only 7 fails, whereas the agent router managed just 28 passes, 13 warnings and 29 fails. This demonstrates that the entailment approach was much more reliable for selecting the correct tool, but the agent router was less effective.

The results also highlight the importance of providing sufficient context to the router. The tool router was able to make more accurate decisions when the prompts provided clear hints about the tool to be used.

The routers were build around facebooks BART-Large-MNLI model, which was last updated in 2023 (as of writing). More recent models such that are potentially based on more advanced architectures such as the deepseek model with more specific training data could be used to improve the performance of the router. Although the results are promising, there are several improvements and questions that need to be addressed. I will go into more detail about these in the next section.

Although my router library is not yet ready for production use, and this research is more of a proof of concept, it does demonstrate the potential of using NLI models for routing tasks, Furthermore, this project has given me valuable insights into the challenges and opportunities in this field and has made me more aware of the limitations of current routing mechanisms. The research has also highlighted the need for further investigation into the use of NLI models for routing tasks, particularly in production environments. This is an area that I am keen to explore further in future work or building apon the work done in this project.

7.2 Recommendations for Future Work

Building on these results, several promising directions for future research emerge:

- **User-Centric Evaluation:** Expanding the evaluation to include real user studies would provide deeper insights into the router's practical effectiveness. Collecting authentic user queries and systematically tracking router decisions could help identify blind spots and areas for improvement. Additionally, integrating analytics would enable continuous monitoring of router accuracy and facilitate data-driven refinements over time.

- **Model Enhancements:** Improving the router model itself offers significant potential. Possible strategies include: (a) applying few-shot or reinforcement learning to enable the router to adapt its decision-making dynamically, (b) implementing hierarchical routing, where a lightweight intent classifier initially filters queries and only escalates to the LLM-based router when necessary, and (c) developing ensemble approaches, such as combining GPT-4 with a smaller local model for faster, resource-efficient decisions. Following best practices—starting with simple solutions and iteratively evolving based on performance metrics—could yield a more robust and adaptable system.
- **Fine-Tuning for Agent Routing:** Using a fine tuned model specifically for the agent router could substantially enhance its performance. This would involve training on a dedicated dataset of routing tasks, enabling the model to learn optimal agent selection for diverse scenarios. While this approach requires considerable data and computational resources, it holds promise for significantly improving agent router accuracy and reliability.

Pursuing these avenues could lead to a more comprehensive understanding of routing strategies and further advance the state of the art in automated tool and agent selection.

Chapter 8

Reflection

8.0.1 Future improvements to the Router Library

With the completion of this project, I have gained valuable insights into the challenges and opportunities in the field of routing tasks using NLI models. Learning about the intricacies of NLI models as well as the pipelines from Hugging Face Library has been a rewarding experience. The Router library is a promising tool for routing tasks, although it is still in its early stages of development It has helped me gain a better understanding working with not only NLI models but a better understanding of LLM pipelines in general. Researching and implementing the Router library has been a valuable learning experience, and I have gained a deeper understanding of the challenges and opportunities in this field. Building the Router library has also helped me get better acquainted with deploying python libraries and setting up a proper CI/CD pipeline.

Some of the key takeaways I could improve upon in the future are:

- Improve my knowledge on multithreading and parallel processing.
- Get a better understanding of fine tuning models and how to do it effectively. Since I failed to get the model to work as well as the base model.
- Improve my knowledge of the Hugging Face Transformers library and how to use the other models available in the library.

As discussed in the previous section, there are several areas for improvement and questions that need to be addressed. Further work is needed to improve the reliability of the Router library. However, I am confident enough to use the tool router plugins generated for the OpenWebUI project in my self hosted instance of it and looking forward for a fix of the API to be able to use the agent router as well. All in all, I am pleased with the results and the progress I have made in this project.

References

artificialanalysis (n.d.), 'Independent analysis of ai models and api providers'.
URL: *https://artificialanalysis.ai/*

Bender, E. M., Gebru, T., McMillan-Major, A. and Shmitchell, S. (2021), 'On the dangers of stochastic parrots: Can language models be too big?', *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency* pp. 610–623.

Hu, Q. J., Bieker, J., Li, X., Jiang, N., Keigwin, B., Ranganath, G., Keutzer, K. and Upadhyay, S. K. (2024), 'Routerbench: A benchmark for multi-llm routing system'.
URL: *https://arxiv.org/abs/2403.12031*

Jiang, Z., Xu, F. F., Gao, L., Sun, Z., Liu, Q., Dwivedi-Yu, J., Yang, Y., Callan, J. and Neubig, G. (2023), 'Active retrieval augmented generation'.
URL: *https://arxiv.org/abs/2305.06983*

Kumar, A., Roh, J., Naseh, A., Karpinska, M., Iyyer, M., Houmansadr, A. and Bagdasarian, E. (2025), 'Overthink: Slowdown attacks on reasoning llms'.
URL: *https://arxiv.org/abs/2502.02542*

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V. and Zettlemoyer, L. (2019), 'Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension'.
URL: *https://arxiv.org/abs/1910.13461*

liveperson (n.d.), 'Routing ai agents - route consumers conversationally'.
URL: *https://developers.liveperson.com/conversation-builder-generative-ai-routing-ai-agents-route-consumers-conversationally.html*

llmpricing (n.d.), 'Llm pricing: Which llms are the best value for money?'.
URL: *https://sanand0.github.io/llmpricing/*

lmsys.org (n.d.), 'Routellm: An open-source framework for cost-effective llm routing'.
URL: *https://lmsys.org/blog/2024-07-01-routellm/*

Ong, I., Almahairi, A., Wu, V., Chiang, W.-L., Wu, T., Gonzalez, J. E., Kadous, M. W. and Stoica, I. (2025), 'Routellm: Learning to route llms with preference data'.
URL: *https://arxiv.org/abs/2406.18665*

openai (n.d.), 'How much does gpt-4 cost?'.
URL: *https://help.openai.com/en/articles/7127956-how-much-does-gpt-4-cost*

Ramos, J. (n.d.), 'Deepseek model does not censor tiananmen square'.
URL: *https://dev.to/jeramos/deepseek-model-does-not-censor-tiananmen-square-2kcb*

Sankaran, V. (n.d.), 'China's new deepseek ai refuses to answer these questions, experts warn'.
URL: *https://www.independent.co.uk/tech/deepseek-china-questions-refuse-beijing-b2687605.html*

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N. and Scialom, T. (2023), 'Toolformer: Language models can teach themselves to use tools'.
URL: *https://arxiv.org/abs/2302.04761*

Strubell, E., Ganesh, A. and McCallum, A. (2019), 'Energy and policy considerations for deep learning in nlp'.
URL: *https://arxiv.org/abs/1906.02243*

Varangot-Reille, C., Bouvard, C., Gourru, A., Ciancone, M., Schaeffer, M. and Jacquenet, F. (2025), 'Doing more with less – implementing routing strategies in large language model-based systems: An extended survey'.
URL: *https://arxiv.org/abs/2502.00409*

Wang, Y., Liu, Q., Xu, J., Liang, T., Chen, X., He, Z., Song, L., Yu, D., Li, J., Zhang, Z., Wang, R., Tu, Z., Mi, H. and Yu, D. (2025), 'Thoughts are all over the place: On the underthinking of o1-like llms'.
URL: *https://arxiv.org/abs/2501.18585*

Appendix A

Appendix

A.1 Jupyter Notebook for Fine-Tuning

The Jupyter notebook for fine-tuning is in `figures/training-jpytr-nb.pdf` and the original notebook is at `src/train/router-llm_train.ipynb`.

A.2 Fine-Tuned Models

Models are on Hugging Face: https://huggingface.co/ru4en

- `ru4en/bart-large-mnli-tool-router-new`
- `ru4en/bart-large-mnli-tool-router`
- `ru4en/bart-large-mnli-agent-router`

A.3 Router Evaluation

Test outputs in the repo:

- Plots: `plots`
- Script: `src/test/test.py`
- Results: `src/test/output`

A.4 Demonstration of the Router

Demo: `src/test/demo.py`

A.5 Router Library

GitHub: https://github.com/ru4en/llm_routers

Install with:

```
pip install git+https://github.com/ru4en/llm_routers
```

Docs: see the repo README or https://fyp.rubenlopes.uk