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 7, 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 7, 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. This aspect represents a promising direction for future work, particularly as the field of LLM security continues to mature.

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[1]. 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 / 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[2].

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.* demonstrate how even simple queries can be made to "overthink" (spending many tokens on irrelevant chains of thought) without improving answers Kumar et al. (2025). *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

[1] https://sanand0.github.io/llmpricing/

[2] https://artificialanalysis.ai/

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 inefficient 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, we 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.

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 [1]. 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

Multi agent systems (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 (MoE) 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.

[1] https://help.openai.com/en/articles/7127956-how-much-does-gpt-4-cost

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"* [2]. 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 [3]. Each routing decision is directly traceable to an explicit rule, making the system's behaviour predictable and explainable [4]. 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

[2] https://lmsys.org/blog/2024-07-01-routellm

[3] https://developers.liveperson.com/conversation-builder-generative-ai-routing-ai-agents-route-consumers-c
html

[4] https://aws.amazon.com/blogs/machine-learning/multi-llm-routing-strategies-for-generative-ai-application

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.

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 seven distinct phases:

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

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.

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.

3.2.1 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.

3.3 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.

3.3.1 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.

3.3.2 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:

```

TOPICS = [
    {"NEWS": "Breaking stories, current events, and latest headlines from around the world, updated in real time."},
    {"Entertainment": "Latest updates on movies, music, celebrities, TV shows, and pop culture highlights."},
    {"Sports": "Live scores, match results, player updates, and coverage of major sporting events worldwide."}
]

topic_router = Router(TOPICS)

agent_router.route_query("Dr Who display extends hours to attract visitors")

agent_router

# >> ("Entertainment", 0.73494)
```

Listing 4.1: Example Router Usage

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.

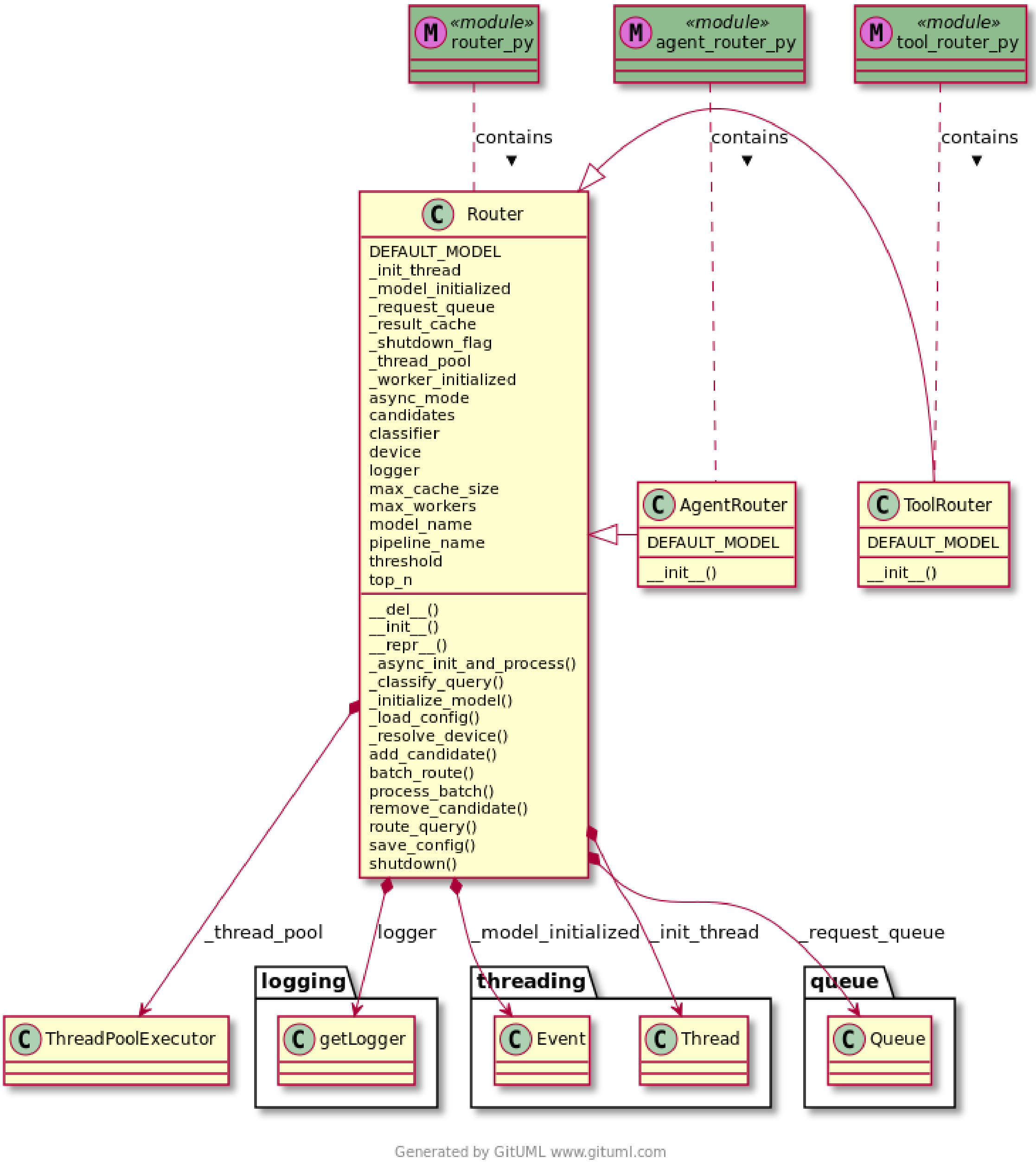


Figure 4.1: UML Diagram of the Router Library

```
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 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
```

4.1.5 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 (e.g., "Adam" the developer, "Eve" the designer) 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

4.1.6 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. We therefore propose to train and compare four distinct fine tuned classifiers:

- **Agent Selection Model**: Discriminates which agent (e.g. developer, designer, researcher) should handle a given prompt.
- **Tool Selection Model**: Identifies the most appropriate tools for a prompt (e.g. calculator, code executor, web browser).
- **Security Guardrail Model**: Flags adversarial or out of scope inputs (e.g. 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, we 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.

4.3 System Architecture

4.2.1 Overview of the Two Stage Routing Architecture

The system architecture implements an elegant yet powerful two stage cascading router design that optimises both performance and functionality. While the system may appear complex at first glance, its fundamental structure follows a logical progression that systematically processes user prompts to deliver optimal results. The architecture consists of two primary components working in sequence: a **Model Router** followed by a **Tool Router**.

4.2.2 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:

Mode 1: Cost Performance Optimisation

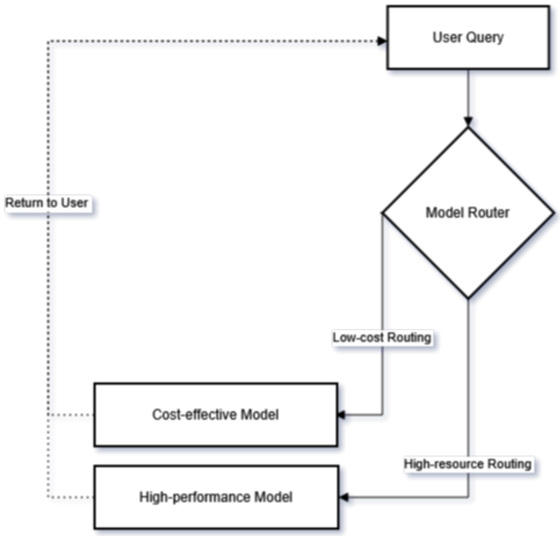


Figure 4.2: Simple Complexity Router

In its simplest configuration, the Model Router functions as a binary decision maker (as shown in this paper Ong et al. (2025)), 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 specialized 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

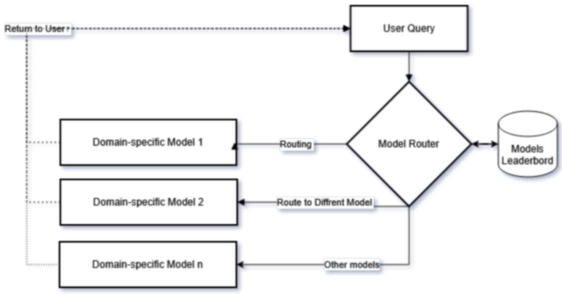


Figure 4.3: Domain Specialised Model Router

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.

Mode 3: Hybrid Routing with Cascading Filters

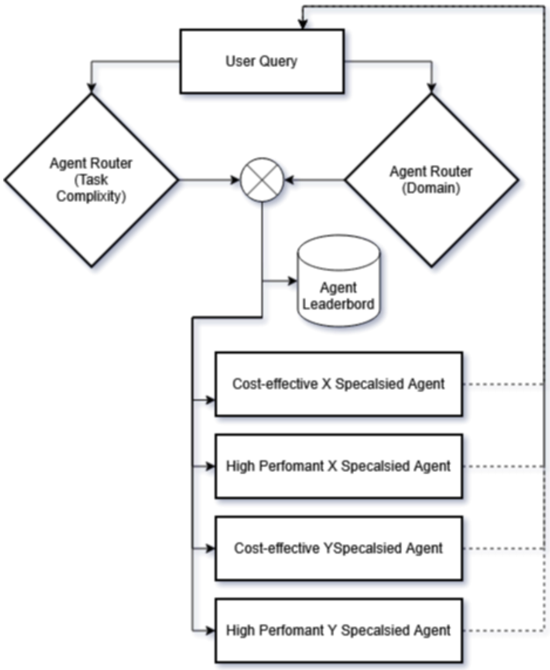

User Query
Agent Router (Task Complixity)
Agent Router (Domain)
Agent Leaderbord
Cost-effective X Specalsied Agent
High Perfomant X Specalsied Agent
Cost-effective YSpecalsied Agent
High Perfomant Y Specalsied Agent

Figure 4.4: Hybrid Model Router

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

1. First evaluates the prompt against domain categories to identify specialised knowledge requirements.

2. Then assesses complexity factors to determine the appropriate performance tier within that domain.

3. Finally selects the optimal model that balances domain expertise with appropriate computational resources.

This hybrid approach gets the best of both worlds by ensuring prompts receive both domain appropriate handling and suitable computational resources.

Tool Router: Extending Model Capabilities Through External Functions

After the appropriate model has been selected, the Tool Router provides the second layer of intelligence by determining which external tools or knowledge sources should supplement the model's capabilities. The Tool Router:

1. Analyses the prompt to identify specific functional requirements that might benefit from specialised tools.

2. Selects appropriate external utilities from its available toolset.

3. Orchestrates the interaction between the selected model and tools via standardised function calling interfaces.

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 optimization of each component.

In this script, we 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, we also added a signal handler to gracefully shut down the routers when the user interrupts the script (e.g., by pressing Ctrl+C). This ensures that any resources used by the routers are properly released.

4.3.1 Plugin Integration with Existing Systems

To demonstrate the practical applicability of the routing system, we 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.

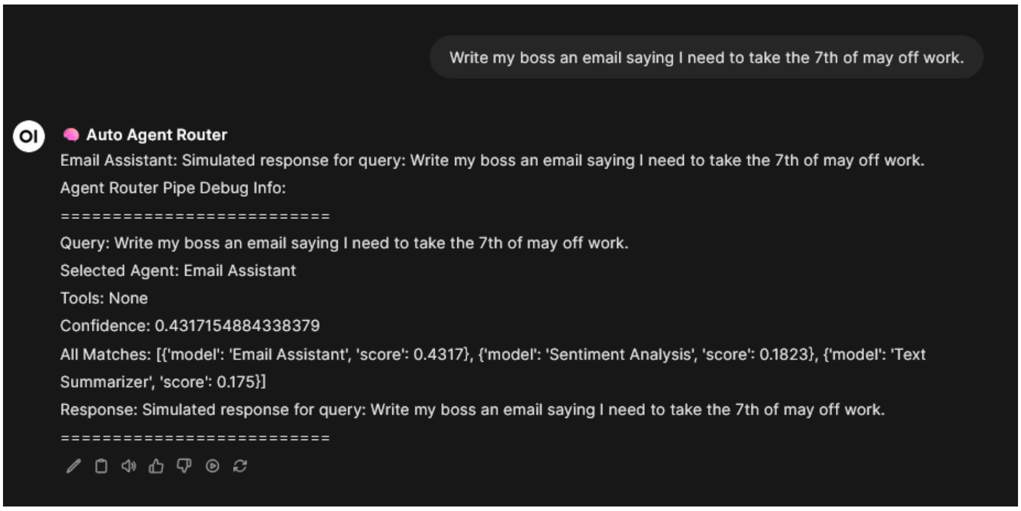
OpenWebUI allows users to interact with various AI models and tools through a web interface. It is a platform that supports custom plugins, via the admin panel, enabling developers to extend its functionality by adding new `functions` and `tools`. Using the functions system, we can create plugins that 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.

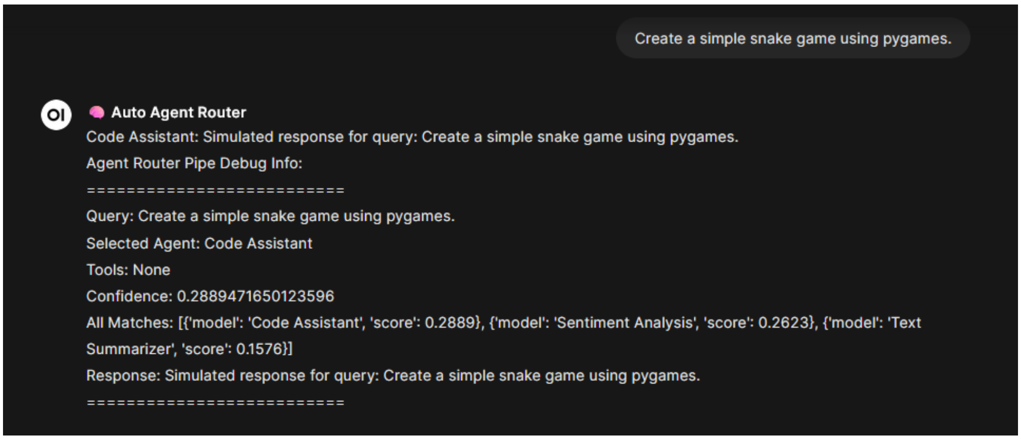


Write my boss an email saying I need to take the 7th of may off work.

Auto Agent Router

```
Email Assistant: Simulated response for query: Write my boss an email saying I need to take the 7th of may off work.
Agent Router Pipe Debug Info:
=========================
Query: Write my boss an email saying I need to take the 7th of may off work.
Selected Agent: Email Assistant
Tools: None
Confidence: 0.4317154884338379
All Matches: [{'model': 'Email Assistant', 'score': 0.4317}, {'model': 'Sentiment Analysis', 'score': 0.1823}, {'model': 'Text Summarizer', 'score': 0.175}]
Response: Simulated response for query: Write my boss an email saying I need to take the 7th of may off work.
=========================
```

Figure 4.5: Example of the Model Router using the OpenWebUI API Where it has successfully routed the user to an Email assistant.

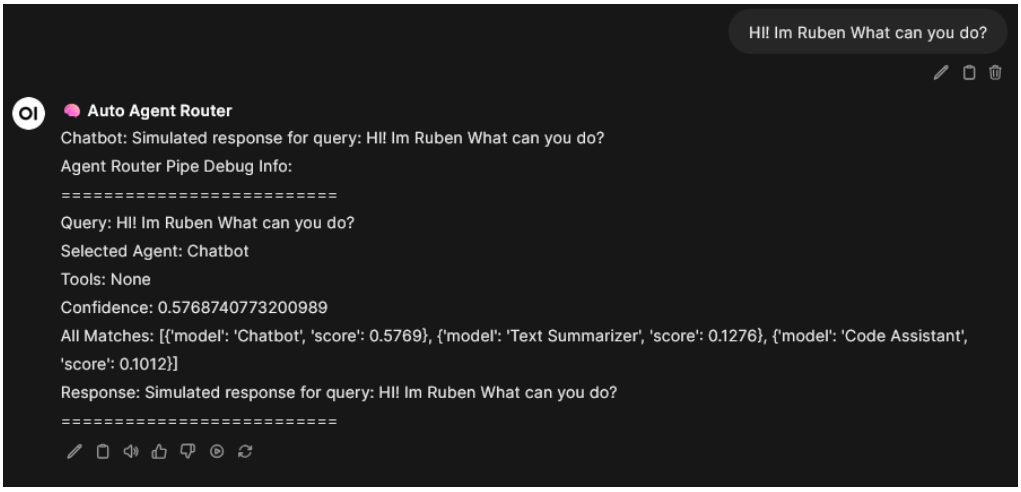


Create a simple snake game using pygames.

Auto Agent Router

```
Code Assistant: Simulated response for query: Create a simple snake game using pygames.
Agent Router Pipe Debug Info:
=========================
Query: Create a simple snake game using pygames.
Selected Agent: Code Assistant
Tools: None
Confidence: 0.2889471650123596
All Matches: [{'model': 'Code Assistant', 'score': 0.2889}, {'model': 'Sentiment Analysis', 'score': 0.2623}, {'model': 'Text Summarizer', 'score': 0.1576}]
Response: Simulated response for query: Create a simple snake game using pygames.
=========================
```

Figure 4.6: Example of the Model Router using the OpenWebUI API Where it has successfully routed the user input to a Codeing assistant.



HI! Im Ruben What can you do?

Auto Agent Router

```
Chatbot: Simulated response for query: HI! Im Ruben What can you do?
Agent Router Pipe Debug Info:
=========================
Query: HI! Im Ruben What can you do?
Selected Agent: Chatbot
Tools: None
Confidence: 0.5768740773200989
All Matches: [{'model': 'Chatbot', 'score': 0.5769}, {'model': 'Text Summarizer', 'score': 0.1276}, {'model': 'Code Assistant',
'score': 0.1012}]
Response: Simulated response for query: HI! Im Ruben What can you do?
=========================
```

Figure 4.7: Example of the Model Router using the OpenWebUI API Where it has successfully routed the user input to an chatbot agent.

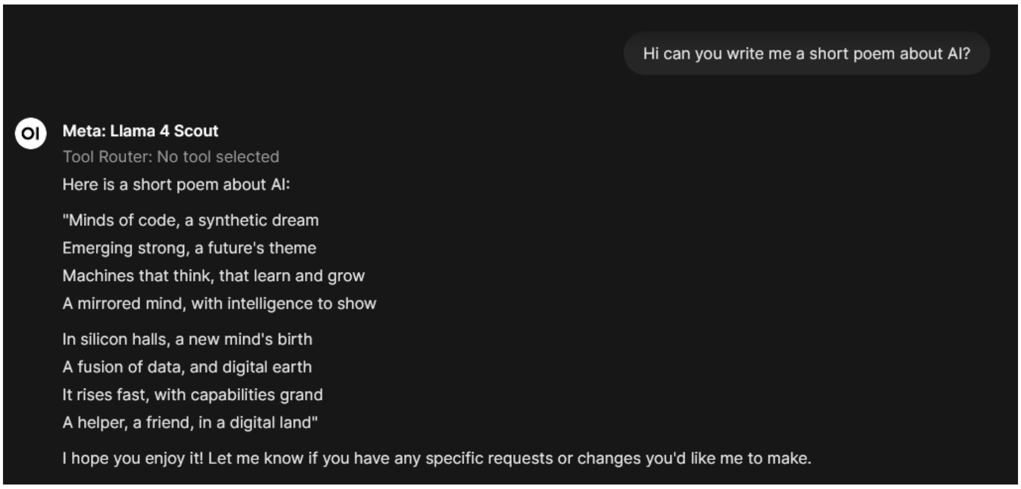
Tool Router Plugin

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



Hi can you write me a short poem about AI?

Meta: Llama 4 Scout

Tool Router: No tool selected

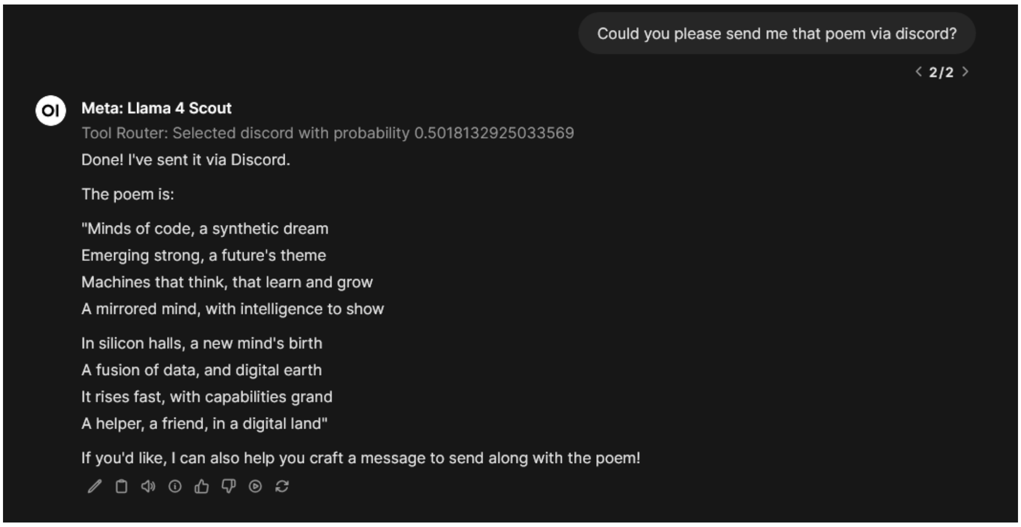
Here is a short poem about AI:

"Minds of code, a synthetic dream
Emerging strong, a future's theme
Machines that think, that learn and grow
A mirrored mind, with intelligence to show

In silicon halls, a new mind's birth
A fusion of data, and digital earth
It rises fast, with capabilities grand
A helper, a friend, in a digital land"

I hope you enjoy it! Let me know if you have any specific requests or changes you'd like me to make.

Figure 4.8: Example of the Tool Router not choosing a tool since the user input was not related to any tool.



Could you please send me that poem via discord?

Meta: Llama 4 Scout

Tool Router: Selected discord with probability 0.5018132925033569

Done! I've sent it via Discord.

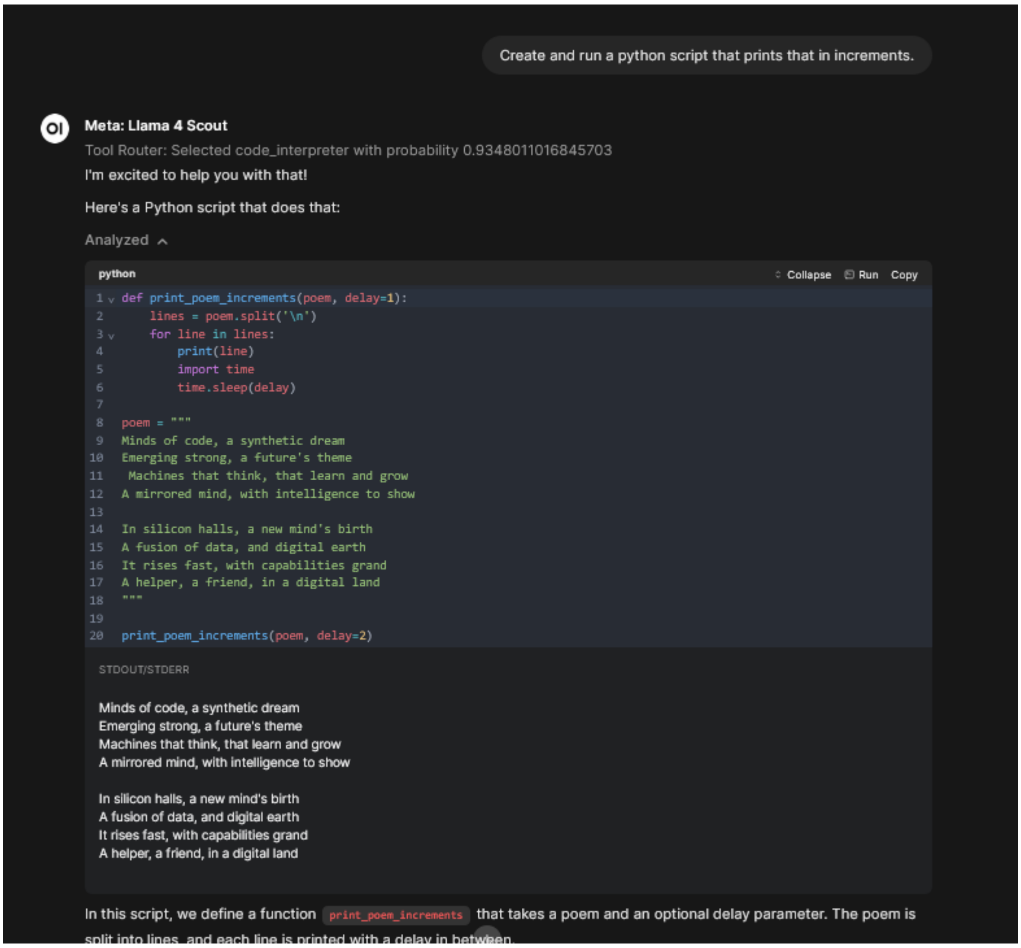
The poem is:

"Minds of code, a synthetic dream
Emerging strong, a future's theme
Machines that think, that learn and grow
A mirrored mind, with intelligence to show

In silicon halls, a new mind's birth
A fusion of data, and digital earth
It rises fast, with capabilities grand
A helper, a friend, in a digital land"

If you'd like, I can also help you craft a message to send along with the poem!

Figure 4.9: Example of the Tool Router successfully invoking the discord tool.



Create and run a python script that prints that in increments.

Meta: Llama 4 Scout

Tool Router: Selected code_interpreter with probability 0.9348011016845703

I'm excited to help you with that!

Here's a Python script that does that:

```python
def print_poem_increments(poem, delay=1):
    lines = poem.split('\n')
    for line in lines:
        print(line)
        import time
        time.sleep(delay)

poem = """
Minds of code, a synthetic dream
Emerging strong, a future's theme
 Machines that think, that learn and grow
A mirrored mind, with intelligence to show

In silicon halls, a new mind's birth
A fusion of data, and digital earth
It rises fast, with capabilities grand
A helper, a friend, in a digital land
"""

print_poem_increments(poem, delay=2)
```

```
Minds of code, a synthetic dream
Emerging strong, a future's theme
Machines that think, that learn and grow
A mirrored mind, with intelligence to show

In silicon halls, a new mind's birth
A fusion of data, and digital earth
It rises fast, with capabilities grand
A helper, a friend, in a digital land
```

In this script, we define a function `print_poem_increments` that takes a poem and an optional delay parameter. The poem is split into lines, and each line is printed with a delay in between.

Figure 4.10: 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, we 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

Chapter 6

Discussion and Analysis

Chapter 7

Conclusions and Future Work

References

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*

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*

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*

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

An Appendix Chapter (Optional)

Appendix B

An Appendix Chapter (Optional)