Compound AI System: Design and Implementation Plan

Project Title: Efficient Question Answering via a Difficulty-Aware Routed Compound AI System on the ARC Dataset

1. Introduction

1.1. Abstract

This document outlines the design, implementation, and evaluation strategy for a Compound AI system. The system aims to optimize resource utilization while maintaining competitive performance in question-answering tasks. It comprises a small, locally-run Large Language Model (LLM) (e.g., Llama 3B via Ollama), a larger, more powerful LLM (e.g., DeepSeek or a proprietary model API like Claude), and an LLM-based router (e.g., fine-tuned Phi-2). The router will be trained on the AI2 Reasoning Challenge (ARC) dataset, specifically using the 'Easy' and 'Challenge' subsets to classify incoming queries by difficulty. This classification will then route queries to the appropriate LLM—simpler queries to the small LLM and complex ones to the large LLM. The primary goal is to evaluate this Compound AI system against a standalone large LLM, demonstrating the potential for comparable accuracy with significantly reduced computational resource usage, and thereby highlighting the benefits and trade-offs inherent in such architectures.

1.2. Motivation

The rapid advancement of Large Language Models has led to remarkable capabilities but also to substantial computational and energy demands. This limits accessibility and increases operational costs. Compound AI systems, which strategically combine multiple specialized models, offer a pragmatic approach to mitigate these challenges. By routing queries based on their intrinsic difficulty, we can avoid invoking resource-intensive large models for tasks that a smaller, more efficient model can handle. The ARC dataset, with its distinct 'Easy' and 'Challenge' sections, provides an ideal testbed for training and evaluating such a difficulty-aware routing mechanism. This project seeks to answer: *Can a fine-tuned router effectively differentiate query difficulty from the ARC dataset to optimize LLM selection, thereby maintaining high accuracy with lower resource consumption compared to a monolithic large LLM?*

1.3. Project Goals

1. **Develop a Functional Compound AI System:** Implement the proposed architecture in Python, integrating all components.
2. **Fine-Tune the LLM-based Router:** Train a small LLM (e.g., Phi-2, DistilBERT) on

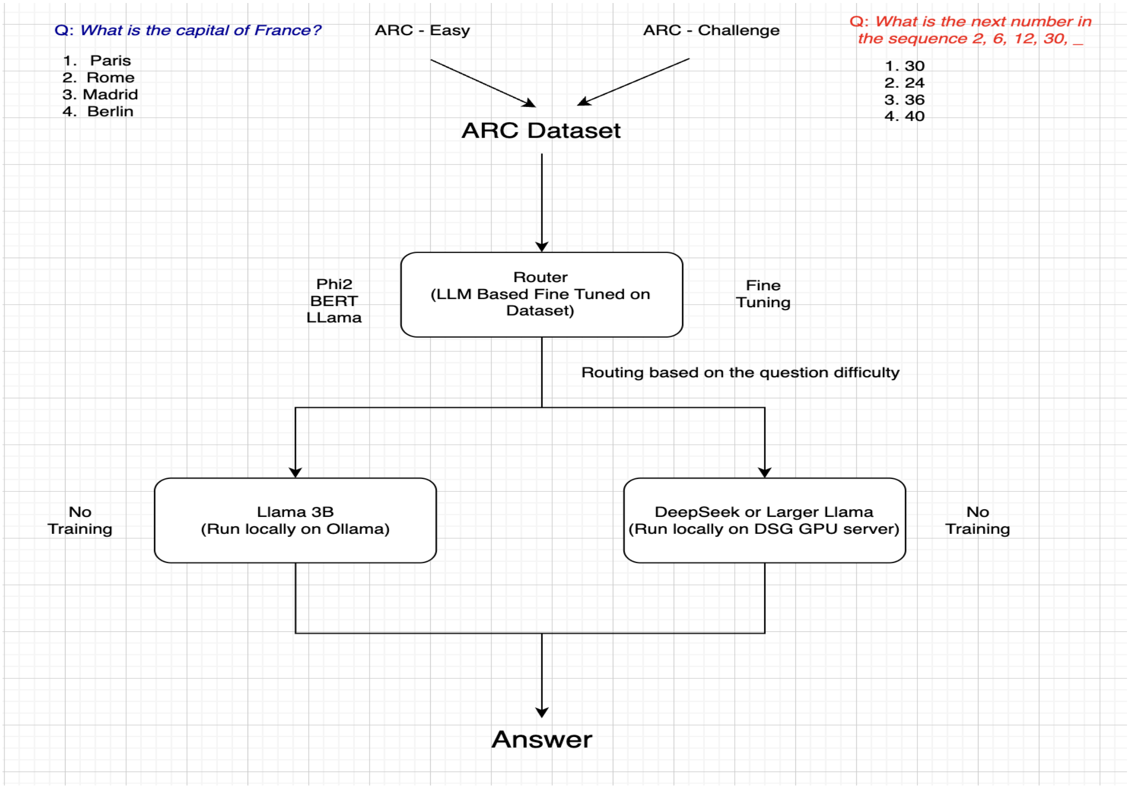
the ARC dataset to classify questions as 'easy' or 'hard'.

3. **Integrate LLMs:** Connect the router with a locally hosted small LLM (Llama 3B via Ollama) and a larger LLM (local DeepSeek or an API-based model like Claude).
4. **Evaluate System Performance:** Assess the Compound AI system's accuracy on the ARC test set.
5. **Comparative Analysis:** Compare the Compound AI system's performance (accuracy, latency, computational resource indicators) against a standalone large LLM.
6. **Analyze Trade-offs:** Document the benefits (e.g., resource savings) and potential drawbacks (e.g., complexity, minor accuracy trade-offs) of the compound approach.

1.4. System Architecture Overview

The system operates as follows:

1. A user query (from the ARC dataset) is received.
2. The **Router** (a fine-tuned LLM) analyzes the query and classifies its difficulty as 'easy' or 'hard'.
3. Based on this classification:
    - 'Easy' queries are routed to the **Small LLM** (Llama 3B on Ollama).
    - 'Hard' queries are routed to the **Large LLM** (DeepSeek running locally on GPU server or Claude API).
4. The selected LLM processes the query and generates an answer



2. Background and Related Work

2.1. Large Language Models (LLMs)

A brief overview of LLM architectures (e.g., Transformer), their capabilities in reasoning and generation, and the distinction between smaller, efficient models (Llama 3B, Phi-2) and larger, more powerful ones (DeepSeek, Claude, GPT-4). Discussion on their respective resource demands.

2.2. Compound AI Systems / Mixture of Experts

Definition and exploration of Compound AI systems, also related to Mixture of Experts (MoE) architectures. Benefits include efficiency, modularity, and the ability to leverage specialized models. Cite relevant research or existing systems that use multiple models or cascading approaches.

2.3. LLM-based Routers and Query Classification

Discussion of techniques for routing queries in multi-LLM systems. Using LLMs for classification tasks, specifically for query difficulty. Advantages of fine-tuning a smaller LLM as an intelligent and adaptable router.

2.4. The ARC (AI2 Reasoning Challenge) Dataset

Detailed description of the ARC dataset: its purpose (evaluating question answering and reasoning), structure (Easy and Challenge sets, multiple-choice format), and its suitability for this project due to its pre-defined difficulty levels.

3. System Design and Implementation

3.1. Overall Architecture (Detailed)

Reiterate the system flow with more detail on data pathways and decision logic. Python will be the primary implementation language, leveraging libraries such as Hugging Face Transformers, Ollama client, and specific SDKs for proprietary LLMs if used.

3.2. Class Architecture Plan

3.2.1. ARCDataManager

- **Purpose:** Manages loading, preprocessing, and splitting of the ARC dataset.
- **Key Attributes:**
    - easy_data_path: str
    - challenge_data_path: str

  - arc_easy_df: pd.DataFrame (or list of dicts)
  - arc_challenge_df: pd.DataFrame (or list of dicts)
- **Key Methods:**
  - __init__(self, easy_path: str, challenge_path: str)
  - load_data(self) -> None: Loads ARC JSON/CSV files.
  - _preprocess_question(self, question_data: Dict) -> str: Formats a question and its choices into a single string suitable for LLM input.
  - create_router_training_data(self, val_split_ratio: float = 0.1, test_split_ratio: float = 0.1) -> Tuple[List[Dict], List[Dict], List[Dict]]: Combines easy (label 0) and challenge (label 1) questions, splits them into train/validation/test sets for the router. Each item: {'text': formatted_question, 'label': 0/1, 'id': question_id}.
  - get_arc_evaluation_set(self, use_router_test_split: bool = False) -> List[Dict]: Provides a distinct set of ARC questions (potentially the router's test split or another held-out portion) for end-to-end system evaluation. Includes question, choices, correct answer, original difficulty.

3.2.2. LLMInterface (Abstract Base Class or Protocol)

- **Purpose:** Defines a standardized interface for interacting with any LLM.
- **Abstract Methods:**
  - generate(self, prompt: str, **kwargs) -> str: Generates a response from the LLM.
  - get_model_name(self) -> str: Returns an identifier for the LLM.

3.2.3. OllamaLLM (Implements LLMInterface)

- **Purpose:** Client for the small LLM (Llama 3B) via Ollama.
- **Key Attributes:**
  - model_name: str (e.g., "llama3:8b-instruct-q4_0" or specific 3B tag)
  - ollama_host: str (default: "http://localhost:11434")
- **Key Methods:**
  - __init__(self, model_name: str, host: str = "http://localhost:11434")
  - generate(self, prompt: str, **kwargs) -> str: Uses the ollama library or requests to interact with the Ollama API. Includes error handling.
  - get_model_name(self) -> str

3.2.4. LargeLLMClient (Implements LLMInterface)

- **Purpose:** Client for the large LLM (DeepSeek on server or proprietary API).
- **Key Attributes:**
  - model_identifier: str (e.g., "deepseek-coder", "claude-3-opus-20240229")
  - api_key: Optional[str]
  - api_endpoint: Optional[str]

    - client: Any (e.g., Anthropic client, custom requests session)
- **Key Methods:**
    - __init__(self, model_identifier: str, api_key: Optional[str] = None, api_endpoint: Optional[str] = None)
    - generate(self, prompt: str, **kwargs) -> str: Implements API call logic specific to the large LLM. Includes error handling and retry mechanisms.
    - get_model_name(self) -> str

3.2.5. QueryRouter

- **Purpose:** LLM-based router for classifying query difficulty.
- **Key Attributes:**
    - model_name_or_path: str (e.g., "microsoft/phi-2", or path to fine-tuned model)
    - tokenizer: transformers.PreTrainedTokenizer
    - model: transformers.AutoModelForSequenceClassification
    - device: str ("cuda" or "cpu")
    - label_map: Dict[str, int] (e.g., {'easy': 0, 'hard': 1})
- **Key Methods:**
    - __init__(self, model_name_or_path: str, num_labels: int = 2, device: Optional[str] = None)
    - _load_model(self) -> None: Loads tokenizer and sequence classification model.
    - fine_tune(self, train_data: List[Dict], val_data: List[Dict], output_dir: str, epochs: int = 3, batch_size: int = 8, learning_rate: float = 5e-5) -> None: Fine-tunes the model using transformers.Trainer. Involves data preparation, defining TrainingArguments, and model saving.
    - predict_difficulty(self, query_text: str) -> str: Takes raw question text, tokenizes, predicts the class, and returns 'easy' or 'hard'.
    - load_fine_tuned_model(self, model_path: str) -> None: Loads a previously fine-tuned router model.

3.2.6. CompoundAISystemOrchestrator

- **Purpose:** Integrates all components and manages the query processing workflow.
- **Key Attributes:**
    - router: QueryRouter
    - small_llm: LLMInterface
    - large_llm: LLMInterface
- **Key Methods:**
    - __init__(self, router: QueryRouter, small_llm: LLMInterface, large_llm: LLMInterface)

    - process_query(self, query_id: str, query_text: str, choices: List[str]) -> Dict[str, any]:
        1. Records start time.
        2. Gets difficulty = self.router.predict_difficulty(query_text).
        3. Records routing time.
        4. Constructs a detailed prompt for the chosen LLM, including the question and choices, possibly asking it to return only the letter/number of the correct choice.
        5. If difficulty == 'easy', calls self.small_llm.generate(). Else, calls self.large_llm.generate().
        6. Records LLM inference time.
        7. Calculates total time.
        8. Returns a dictionary with query_id, query_text, predicted_difficulty, chosen_llm_name, raw_response, routing_time_ms, llm_inference_time_ms, total_time_ms.

3.2.7. EvaluationEngine

- **Purpose:** Evaluates the Compound AI system and the baseline standalone large LLM.
- **Key Attributes:**
    - arc_evaluation_set: List[Dict] (from ARCDataManager)
    - compound_system: Optional[CompoundAISystemOrchestrator]
    - standalone_large_llm: LLMInterface
- **Key Methods:**
    - __init__(self, arc_evaluation_set: List[Dict], standalone_large_llm: LLMInterface, compound_system: Optional[CompoundAISystemOrchestrator] = None)
    - _parse_llm_output_for_arc(self, llm_output: str, choices: List[str], correct_answer_key: str) -> bool: Parses the LLM's textual output to determine the selected answer choice (e.g., 'A', 'B', '1', '2') and compares it against the ground truth. This is a critical and potentially complex step.
    - run_evaluation(self, system_to_evaluate: Union[CompoundAISystemOrchestrator, LLMInterface]) -> List[Dict]: Iterates through arc_evaluation_set. For each question:
        - If system_to_evaluate is CompoundAISystemOrchestrator, calls process_query.
        - If system_to_evaluate is LLMInterface (standalone), calls generate.
        - Calls _parse_llm_output_for_arc to check correctness.
        - Records all relevant data: question ID, correctness, predicted difficulty (if

        compound), chosen LLM (if compound), latencies, etc.
        - Returns a list of these detailed result dictionaries.
    - calculate_metrics(self, results: List[Dict], system_name: str) -> Dict[str, float]: From the list of result dictionaries, calculates:
        - Overall accuracy.
        - Accuracy on 'easy' questions (based on true ARC labels) and 'hard' questions.
        - Average latencies (total, routing, LLM inference).
        - For compound system: percentage of queries routed to small/large LLM, accuracy of small LLM on 'easy' queries, accuracy of large LLM on 'hard' queries.
    - generate_report(self, standalone_results: List[Dict], compound_results: List[Dict]) -> None: Prints or saves a comparative summary of metrics.

3.3. Key Libraries and Tools

- **Python 3.8+**
- **Hugging Face transformers:** For router model (e.g., Phi-2, DistilBERT) loading, fine-tuning, tokenization.
- **Hugging Face datasets:** (Optional) For loading and handling the ARC dataset.
- **torch:** Backend for transformers.
- **ollama:** Python client for Ollama.
- **Proprietary LLM SDKs:** (e.g., anthropic for Claude) if applicable.
- **pandas:** For data manipulation and analysis.
- **scikit-learn:** For router evaluation metrics (accuracy, F1, confusion matrix) and potentially for data splitting if not using datasets library features.
- **tqdm:** For progress bars.
- **logging:** For detailed operational logs.
- **time:** For latency measurements.

4. Step-by-Step Development Guide

Phase 1: Setup & Foundational Components

1. **Environment & Tools:**
    - Set up a Python virtual environment. Install all necessary libraries (pip install ...).
    - Install Ollama and pull the small LLM (e.g., ollama pull llama3:8b or the specific 3B variant). Verify it's operational.
    - Ensure access to the large LLM (configure API keys/endpoints or server access for DeepSeek).

2. **ARCDataManager Implementation:**
   - Download the ARC dataset (main, easy, challenge).
   - Implement load_data and _preprocess_question. Test thoroughly.
   - Implement create_router_training_data and get_arc_evaluation_set. Verify data integrity and splits.
3. **LLMInterface and Concrete LLM Clients:**
   - Define LLMInterface.
   - Implement OllamaLLM. Test with simple prompts.
   - Implement LargeLLMClient. Test with simple prompts.

Phase 2: Router Development and Fine-Tuning

1. QueryRouter - Initial Setup:
   - Implement __init__ and _load_model in QueryRouter using a pre-trained sequence classification head (e.g., AutoModelForSequenceClassification.from_pretrained("microsoft/phi-2", num_labels=2, trust_remote_code=True) if using Phi-2, or a DistilBERT variant).
   - Ensure the label_map correctly maps labels to integers.
2. QueryRouter - Fine-Tuning:
   - Use train_data and val_data from ARCDataManager.
   - Implement the fine_tune method using transformers.Trainer and TrainingArguments.
   - Define a compute_metrics function (using sklearn.metrics.accuracy_score, f1_score) for evaluation during training.
   - Execute fine-tuning. Save the best model checkpoint.
3. Router Evaluation (Standalone):
   - Implement predict_difficulty.
   - Use the router's test set (from ARCDataManager) to evaluate its performance. Calculate accuracy, F1-score, precision, recall, and generate a confusion matrix. Analyze misclassifications.
   - Implement load_fine_tuned_model.

Phase 3: System Integration & Orchestration

1. CompoundAISystemOrchestrator Implementation:
   - Implement __init__ to accept the fine-tuned QueryRouter and instances of OllamaLLM and LargeLLMClient.
   - Implement process_query, ensuring correct prompt formatting for the chosen LLM.

    - Manually test the orchestrator with a few sample easy and hard questions.

Phase 4: Evaluation

1. EvaluationEngine Implementation:
    - Implement __init__.
    - Crucially, implement _parse_llm_output_for_arc. This might require iterative refinement. Consider prompting LLMs to output answers in a very specific format (e.g., "The correct option is: (X)").
    - Implement run_evaluation.
    - Implement calculate_metrics.
2. Execute Full Evaluation:
    - Run run_evaluation for the standalone large LLM.
    - Run run_evaluation for the CompoundAISystemOrchestrator.
    - Store all raw results (e.g., as JSON or CSV files) for detailed analysis.
3. Resource Usage Measurement:
    - Latency: Already captured.
    - Routing Distribution: Calculate from compound system results.
    - API Costs (if applicable): If using a proprietary API, log token counts (input/output) for each call to the large LLM. Sum these to get total token usage for both standalone and compound system scenarios.
    - Qualitative Hardware Notes: Document the hardware used for local models.

Phase 5: Analysis, Documentation, and Paper Writing

1. Analyze Results:
    - Use calculate_metrics and generate_report from EvaluationEngine.
    - Compare overall accuracy, ARC-Easy accuracy, ARC-Challenge accuracy.
    - Compare resource usage metrics.
    - Discuss the impact of router accuracy on overall system performance.
    - Analyze trade-offs: e.g., plot accuracy vs. latency/cost.
2. Write the Paper:
    - Structure your paper using sections similar to this document.
    - Include detailed descriptions of your implementation, fine-tuning parameters, router performance, and comparative evaluation results.
    - Use tables and charts to present findings clearly.
3. Future Work:

- Suggest potential improvements: more sophisticated routing logic (e.g., confidence-based), dynamic adjustment of routing, expanding the pool of LLMs.

5. Evaluation Methodology

5.1. Baseline System

- **Description:** Standalone Large LLM (DeepSeek or Claude).
- **Process:** All questions from the arc_evaluation_set (test split) are processed directly by this LLM.
- **Metrics:** Accuracy (overall, Easy, Challenge), average latency per query, total token consumption (if applicable).

5.2. Compound AI System

- **Description:** The implemented system with router, small LLM, and large LLM.
- **Process:** Questions from arc_evaluation_set are processed via CompoundAISystemOrchestrator.
- **Metrics:** Accuracy (overall, Easy, Challenge), average latencies (total, routing, LLM inference), distribution of queries to small/large LLM, token consumption for the large LLM component.

5.3. Performance Metrics for QA Task

- **Accuracy:** (Number of correctly answered questions) / (Total number of questions).
- **Answer Parsing:** A robust _parse_llm_output_for_arc method is critical. It must reliably extract the chosen answer (e.g., A, B, C, D, or 1, 2, 3, 4) from the LLM's textual output and compare it to the ground truth from the ARC dataset.

5.4. Resource Usage Metrics

- **Latency:** End-to-end query processing time, router decision time, LLM inference time (measured in milliseconds).
- **Routing Distribution:** Percentage of queries handled by the small LLM versus the large LLM.
- **Computational Cost Proxy / API Cost Reduction:** For API-based large LLMs, track the number of tokens processed or API calls made by the compound system versus the standalone system. For local models, the routing distribution itself is a key indicator of resource shifting.

6. Expected Results and Analysis

6.1. Router Performance

- Expect the fine-tuned router to achieve high accuracy (e.g., >85-90%) in distinguishing ARC Easy from Challenge questions.
- Analyze the router's confusion matrix: identify if it's more prone to misclassifying hard questions as easy, or vice-versa, and the implications.

6.2. Task Performance (Accuracy)

- **Hypothesis:** The Compound AI system's overall accuracy will be close to, but potentially slightly lower than, the standalone large LLM. The magnitude of this difference will depend heavily on router accuracy and the small LLM's capability on genuinely easy questions.
- Performance on 'easy' questions by the compound system should be high if routed correctly.
- Performance on 'hard' questions will depend on correct routing to the large LLM. Misrouting 'hard' questions to the small LLM is a likely source of errors.

6.3. Resource Usage

- **Hypothesis:** Significant reduction in average latency for the compound system, driven by the faster processing of 'easy' queries by the local small LLM.
- Substantial reduction in computational load on the large LLM (or API calls/tokens if proprietary), proportional to the percentage of queries handled by the small LLM.

6.4. Trade-offs

- A key part of the analysis will be to quantify the trade-off: e.g., a Y% reduction in resource usage (or cost/latency) for an X% change (hopefully small or negligible) in accuracy.
- Discuss scenarios where this trade-off is most beneficial.

This detailed plan should serve as a strong foundation for your project and subsequent paper. Remember to maintain modular, well-commented code and use version control (e.g., Git) throughout the development process.