LLMOrchestrator: A Multi-Model LLM Orchestration Framework for Reducing Bias and Iterative Reasoning

Trisanth Srinivasan [1] and **Santosh Patapati** [1]

1 Cyrion Labs

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

Summary

LLMOrchestrator is a Python package for studying Large Language Model (LLM) behavior, capabilities, and limitations, enabling researchers to explore diverse perspectives, validate outputs, and analyze model interactions. It orchestrates multiple LLMs in roles like generation, verification, and refinement, supporting iterative reasoning (e.g., chain-of-thought (Wei et al., 2023)). By integrating API-based (OpenAI (OpenAI, 2023)) and local models (Hugging Face Transformers (Wolf et al., 2020)), it enables systematic output comparison, helping uncover biases and emergent behaviors. Its framework abstracts complex workflows, offering tools for prompt management, structured experimentation, and performance monitoring for reproducibility. For AI safety, multi-agent simulations, or reliability benchmarking, LLMOrchestrator empowers researchers to investigate LLM complexities through controlled experimentation.

Statement of Need

Research on Large Language Models (LLMs) involves more than simple prompt-response; it requires exploring diverse perspectives, validating outputs, and understanding nuanced model reasoning. This demands managing multi-step processes, iterative self-correction, and orchestrating multiple models. **LLMOrchestrator** addresses these needs by providing a controlled, flexible environment for custom workflows, rigorous validation, and comparing diverse output patterns. This framework helps researchers efficiently navigate complex LLM experiments, ensuring varied insights and rigorous checks are integral to their research process.

Core capabilities

LLMOrchestrator offers a flexible and powerful framework for designing, executing, and analyzing complex interactions between Large Language Models. It enables structured orchestration of multiple models, allowing different LLMs (e.g., `OpenAIModel`, `LocalModel`) to perform roles such as generation, critique, and validation. Researchers can implement iterative refinement loops to study reasoning processes like chain-of-thought and self-correction, and integrate API-based with locally hosted models for diverse comparative studies. The framework supports custom logic through Python functions for tailored generation strategies and domain-specific validation. Prompt templating (`PromptTemplate`) facilitates systematic experimentation, while parallel execution (`execute_parallel`) and caching (`OutputCache`) improve efficiency. Additionally, LLMOrchestrator provides built-in monitoring (`ValidationMetrics`) to track key performance indicators like processing times and verification outcomes, ensuring rigorous evaluation of LLM outputs. Its modular design allows customizing workflows for various experimental needs, from multi-step reasoning to adaptive model interactions.

Core LLM Orchestration components

The framework's architecture centers on key components researchers use to build experiments.

Controller

The `Controller` is the central class in LLMOrchestrator. To set up an experiment, the user instantiates a `Controller` object, providing it with generator and verifier components. They also configure parameters governing the orchestration, such as `max_iterations` and `max_verifications` per iteration.

```python
from LLMOrchestrator.controller import Controller
from LLMOrchestrator.models.openai_model import OpenAIModel
from LLMOrchestrator.verifier import Verifier # Or a custom verifier

# Assume generator_model and verifier_logic are initialized
controller = Controller(
    generator=generator_model,
    verifier=verifier_logic,
    max_iterations=5,
    max_verifications=3,
    monitoring_enabled=True # Enable metrics collection
)
```

Models (`BaseModel`, `OpenAIModel`, `LocalModel`)

These classes represent LLMs in the orchestration, abstracting underlying API calls or local inference. Researchers can use different model instances for various roles (e.g., `OpenAIModel` for generation, a `LocalModel` for verification).

- `OpenAIModel`: Interfaces with the OpenAI API. Requires an API key (typically via environment variable).
- `LocalModel`: Uses Hugging Face `transformers` for local inference. Requires appropriate libraries (`transformers`, `torch`, `accelerate`) and model weights. Supports CPU/GPU.

```python
from LLMOrchestrator.models.openai_model import OpenAIModel
from LLMOrchestrator.models.local_model import LocalModel

# API key read from environment or passed during init
generator = OpenAIModel(model_name="gpt-4o")
verifier_model = LocalModel(model_name="google/flan-t5-base", device="cuda")
# These can then be passed to the Controller
```

Generator and Verifier

These components define actions within the orchestration loop. They can be `BaseModel` instances or custom wrappers (`CustomGenerator`, `CustomVerifier`) around Python functions. The `Verifier` is crucial for research, implementing criteria (simple checks, rubrics, or LLM calls) to evaluate output and guide refinement. Verification results, with quality scores, are captured in metrics.

```python
from LLMOrchestrator.verifier import Verifier
import json

# Example: Verifier using a custom function checking for keywords
def keyword_check(text: str, prompt: str = None) -> tuple[bool, str]:
    keywords = ["research", "LLM", "framework"] # Example keywords
    found = all(kw in text.lower() for kw in keywords)
    score = 1.0 if found else 0.0
    message = json.dumps({'score': score, 'message': 'Keyword check passed.' if found el
    return found, message

keyword_verifier = Verifier(custom_verifier=keyword_check)
# controller = Controller(..., verifier=keyword_verifier)
```

Iteration and Refinement Loop

The `Controller.execute()` method manages the core loop. For a prompt, it calls the generator, then passes output to the verifier. Based on verification outcome and configured `max_iterations`/`max_verifications`, it may stop, retry verification, or proceed to the next iteration, potentially refining prompts/outputs via internal or custom logic. This controlled iteration allows systematic study of multi-step reasoning.

Experimentation and analysis

LLMOrchestrator provides features specifically aimed at supporting the research analysis phase.

Metrics Collection

When `monitoring_enabled=True`, the `Controller` automatically collects `ValidationMetrics` per execution. These metrics include processing time, approximated token counts, verification outcomes, quality scores from verifier feedback, and potential indicators via `AdaptiveLearning`. This data can be retrieved programmatically using `controller.get_validation_metrics()` or `controller.get_performance_report()` for analysis.

Parallel Processing for Experiments

Running experiments often requires multiple trials or testing across various parameter settings. The `Controller.execute_parallel()` method leverages Python's `concurrent.futures` to process a list of prompts concurrently, significantly reducing the time needed to gather data for analysis, especially when interacting with slower local models or rate-limited APIs.

```
# Example: Running multiple experimental conditions
prompts_conditions = [
    "Explain AI safety (Condition 1)",
    "Explain AI safety (Condition 2 with different template)",
    "Explain AI ethics (Condition 3)"
]
results_data = controller.execute_parallel(prompts_conditions, max_workers=4)
# results_data can be further processed along with collected metrics
```

Caching for Development and Exploration

While potentially disabled for final experimental runs, `OutputCache` is valuable during research development. It speeds up testing of orchestration logic and prompt variations by storing and retrieving results for previously seen inputs, saving resources and time.

Conclusions

LLMOrchestrator bridges the gap between LLM capabilities and structured research requirements, enabling deeper exploration of diverse perspectives, validation methods, and LLM output variability. By providing a modular framework for orchestrating multiple models, facilitating iterative reasoning, and systematically collecting performance data, it empowers researchers to study how models generate, refine, and critique information, while helping to reduce bias in reasoning. Its design supports reproducibility and comparative analysis, making it easier to investigate emergent behaviors and interaction patterns. For multi-agent simulations, AI safety studies, or improving AI system reliability, **LLMOrchestrator** offers a controlled environment for probing language model complexities and ensuring more robust, well-validated insights.

Acknowledgements

We acknowledge the developers of foundational libraries used in LLMOrchestrator, including `openai`, `transformers` (Wolf et al., 2020), `torch`, `python-dotenv`, and others listed in project dependencies.

References

OpenAI. (2023). *OpenAI API*. https://openai.com/api/.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2023). *Chain-of-thought prompting elicits reasoning in large language models*. https://arxiv.org/abs/2201.11903

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., Platen, P. von, Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., … Rush, A. M. (2020). HuggingFace's transformers: State-of-the-art natural language processing. *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 38–45. https://doi.org/10.18653/v1/2020.emnlp-demos.6