The Journal of Open Source Software

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

**Trisanth Srinivasan** [1]

**1** Cyrion Labs

**DOI:** 10.xxxxxx/draft

**Software**

- Review
- Repository
- Archive

**Editor:** Chris Vernon

**Reviewers:**

- @Manvi-Agrawal
- @pebabion

**Submitted:** 30 March 2025
**Published:** unpublished

## Summary

LLMOrchestrator provides a python package for studying and utilizing the behavior, capabilities, and limitations of Large Language Models (LLMs), enabling researchers to explore diverse perspectives, validate outputs, and analyze how different models interact. It facilitates the orchestration of multiple LLMs, allowing them to take on distinct roles such as generation, verification, and refinement, while supporting iterative reasoning processes like chain-of-thought (Wei et al., 2023). By integrating API-based and local models (e.g., OpenAI (OpenAI, 2023), Hugging Face Transformers (Wolf et al., 2020)), LLMOrchestrator enables systematic comparison of model outputs, helping to uncover biases and emergent behaviors. Its framework abstracts complex workflows, providing tools for prompt management, structured experimentation, and performance monitoring to ensure reproducibility and insightful analysis. Whether used for AI safety, multi-agent simulations, or benchmarking model reliability, LLMOrchestrator empowers researchers to investigate the complexities of language models through controlled and transparent experimentation.

## Statement of Need

Research on Large Language Models (LLMs) involves more than simple prompt-response interactions, it requires exploring diverse perspectives, validating varied outputs, and understanding the nuanced reasoning behind model behaviors. Achieving this demands careful management of multi-step processes, iterative self-correction, and the orchestration of multiple models to capture the full spectrum of responses. **LLMOrchestrator** addresses these needs by providing a controlled, flexible environment that supports custom workflows, rigorous validation, and the comparison of diverse output patterns. This framework helps researchers efficiently navigate the complexity of LLM experiments, ensuring that varied insights and rigorous checks are integral to the research process.

## Core capabilities

LLMOrchestrator provides a comprehensive set of tools for designing, executing, and analyzing complex interactions with and between Large Language Models. Its core capabilities enable researchers to:

**LLMOrchestrator** provides researchers with 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, while seamlessly integrating API-based and locally hosted models for diverse comparative studies. The framework supports custom logic through Python functions, allowing tailored generation strategies and domain-specific validation methods. Prompt templating (`PromptTemplate`) facilitates systematic experimentation with prompt variations, while performance optimization features like parallel execution (`execute_parallel`) and caching (`OutputCache`) improve efficiency. Additionally, **LLMOrchestrator** provides built-in monitoring (`ValidationMetrics`) to track key performance indicators such as processing times, verification outcomes, and quality scores, ensuring rigorous and reproducible evaluation of LLM outputs across diverse perspectives. Its modular design allows researchers to customize workflows for a wide range of experimental needs, from multi-step reasoning studies to adaptive model interactions.

# Core LLM Orchestration components

The framework's architecture is centered around several key components that researchers interact with to build their experiments.

## Controller

The central class in LLMOrchestrator is the `Controller`. To set up an experiment, the user instantiates a `Controller` object, providing it with generator and verifier components, and configuring parameters that govern the orchestration process, such as the maximum number of iterations (`max_iterations`) and verification attempts (`max_verifications`) per iteration.

```
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 the LLMs participating in the orchestration. They abstract the underlying API calls or local inference logic. Researchers can use different model instances for different roles (e.g., one `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.

```
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 the actions performed within each step of the orchestration loop. They can be instances of BaseModel or custom wrappers (CustomGenerator, CustomVerifier) around Python functions. The Verifier plays a crucial role in research, as it implements the criteria (which can be simple checks, complex rubrics, or even another LLM call) used to evaluate the generated output and potentially guide the refinement process. Verification results, including quality scores, are captured in the 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 given prompt, it calls the generator, then passes the output to the verifier. Based on the verification outcome and the configured max_iterations and max_verifications, it may stop, retry verification, or proceed to the next iteration (potentially refining the prompt or output based on internal logic or custom implementations). This controlled iteration allows for systematic study of multi-step reasoning processes.

## Experimentation and analysis

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

### Metrics Collection

When monitoring_enabled=True, the Controller automatically collects ValidationMetrics for each execution. These metrics include processing time, token counts (approximated), verification outcomes, quality scores derived from the verifier's feedback, and other potential indicators defined via AdaptiveLearning. This data can be retrieved programmatically using controller.get_validation_metrics() or controller.get_performance_report() for analysis across different experimental conditions.

### 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.

```python
# 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 to avoid interference, the `OutputCache` is valuable during the development phase of research projects. It speeds up testing of orchestration logic and prompt variations by storing and retrieving results for previously seen inputs, saving computational resources and time.

# Conclusions

**LLMOrchestrator** bridges the gap between the raw capabilities of Large Language Models and the structured requirements of rigorous research, enabling deeper exploration of diverse perspectives, validation methods, and the variability of LLM outputs. By providing a modular framework for orchestrating multiple models, facilitating iterative reasoning, and systematically collecting performance data, it empowers researchers to study how different models generate, refine, and critique information while also reducing the impact of bias in one segment of the reasoning progress. Its design supports reproducibility and comparative analysis, making it easier to investigate emergent behaviors, reasoning diversity, and interaction patterns across models. Whether used for multi-agent simulations, AI safety studies, or improving reliability in AI systems, **LLMOrchestrator** offers a controlled environment for probing the complexities of language models and ensuring more robust, well-validated insights.

# Acknowledgements

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