Computer and Telecommunication Engineering 2025, 3(1), 3168.
https://doi.org/10.54517/cte3168

Article

Fine-tuned large language models into traditional back-end web architectures

Bowen Li[1], Chuan Zhang[2,*]

[1] School of Computing Technologies, Royal Melbourne Institue of Technology, Melbourne 3001, Australia
[2] Wanghaiyiyuan 2-12D, Shenzhen 518000, China
*** Corresponding author:** Chuan Zhang, chuan.z@hotmail.com

CITATION

Li B, Zhang C. Fine-tuned large language models into traditional back-end web architectures. Computer and Telecommunication Engineering. 2025; 3(1): 3168. https://doi.org/10.54517/cte3168

ARTICLE INFO

Received: 18 December 2024
Accepted: 11 March 2025
Available online: 19 March 2025

COPYRIGHT

Copyright © 2025 by author(s). *Computer and Telecommunication Engineering* is published by Asia Pacific Academy of Science Pte. Ltd. This work is licensed under the Creative Commons Attribution (CC BY) license. https://creativecommons.org/licenses/by/4.0/

Abstract: Integrating Large Language Models (LLMs) into traditional back-end systems can significantly reduce development overhead and enhance flexibility. This paper presents a novel approach using a fine-tuned LLama3 model as a modular back-end component capable of processing JSON-formatted inputs and outputs. We developed a specialized dataset through advanced prompt engineering with the Phi-3.5 model and fine-tuned LLama3 using Quantized Low-Rank Adaptation (QLoRA) on a single NVIDIA T4 GPU. An API layer was designed to facilitate seamless communication between clients and the LLM, effectively replacing conventional application logic. Our fine-tuned model achieved an average accuracy of 76.5¥% and a response time of 3.56 s across 100 test cases, demonstrating its effectiveness in handling back-end tasks. This work underscores the potential of LLMs to transform AI-driven back-end architectures, offering scalable and efficient solutions for modern web services.

Keywords: LLM; AI; back-end; fine-tuning; web service; LLama3

1. Introduction

A back-end web server operates as the hidden backbone of a website, managing logic and data processing to support user-facing interactions. It is responsible for receiving requests from clients' web browsers, processing them, interacting with databases to retrieve or store information, and delivering appropriate responses to the user's browser. The core components of a back-end server include three key parts: the server (hardware), the application, and the database. The database maintains the system state, while the application handles incoming requests, processes data, and generates responses. Both databases and applications operate on physical machines or cloud-based environments. Among these components, the application often requires extensive development resources to implement and maintain.

Recent advances in artificial intelligence (AI) have introduced Large Language Models (LLMs), which excel at processing, understanding, and generating human-like text. However, this language-centric capability can extend beyond human languages. For instance, JSON-formatted messages, a common communication format in web services, can be treated as a structured language. By leveraging this perspective, LLMs can emulate the logic and behavior of traditional back-end applications.

To tap into this potential, LLMs must be tailored to the specific needs and workflows of a given back-end. Fine-tuning—a machine learning technique that further trains a pre-trained model on a domain-specific dataset [1]—offers a

promising solution [2]. Through fine-tuning, LLMs can evolve from general-purpose language generators into specialized components that accurately process input, apply custom logic, and return responses in prescribed formats.

The contribution of this paper is to demonstrate how an LLM can be fine-tuned to function as a modular component within a traditional back-end architecture. Specifically, we show how to adapt a pre-trained LLM to produce responses in a predefined, structured format—akin to a custom back-end application. This approach highlights a practical pathway for integrating LLMs into conventional web back-end servers, potentially reducing development overhead and streamlining the implementation of specialized logic.

2. Related work

In this section, we review related work on integrating Large Language Models (LLMs) into traditional web services.

Recent advances in LLMs have spurred a surge in research exploring their integration into back-end and service-oriented architectures. For instance, RestGPT Song et al. [3] introduces a framework that enables LLMs to interact directly with RESTful APIs, allowing them to perform complex tasks by decomposing instructions and selecting appropriate API endpoints. This approach highlights how LLMs can serve as dynamic components, replacing certain traditional back-end functions. However, RESTGPT is designed to convert human language into RESTful outputs, facilitating human interaction with other systems. On the other hand, our method utilizes LLMs to directly interface with APIs, encompassing both the generation of outputs and the processing of RESTful inputs. Similarly, the Large Search Model, Wang et al. [4] proposes a unified framework that employs LLMs to handle traditionally distinct search pipeline tasks—such as query interpretation, retrieval, and ranking—by framing these operations as autoregressive text generation problems. This innovation streamlines the search process and reduces the complexity of conventional, modularized search stacks.

Another direction, exemplified by the Sahaay system [5] and efforts in e-government services [6], demonstrates how LLMs can replace human-driven or fixed-rule components in specialized applications. In the case of Sahaay, LLMs integrate with customer service platforms, automating tasks like query resolution and response generation, thereby reducing reliance on human agents or individually tailored modules. In the e-government domain, researchers have explored Retrieval-Augmented Generation (RAG) architectures to enhance public services [7], illustrating that LLMs can improve processes traditionally reliant on rigid, manually coded logic.

While previous research has explored the use of Large Language Models (LLMs) for back-end tasks, often leveraging pre-trained models or generic prompts, our work distinguishes itself by focusing on the fine-tuning of LLMs for modular back-end integration. This targeted customization enables LLMs to generate application-specific responses, effectively bridging the gap between the flexibility of natural language processing and the stringent demands of traditional back-end systems. Unlike approaches relying on extensive, handcrafted prompting or utilizing

unmodified LLMs, our method systematically refines the model's parameters to directly implement domain logic and produce structured, application-specific outputs. This fine-tuning approach allows for a more robust and efficient integration compared to prompt engineering or relying on the general capabilities of pre-trained models.

Our key contribution is demonstrating the transformative potential of fine-tuning to convert a general-purpose LLM into a specialized, production-ready back-end service. By tailoring the model to a specific back-end function and training it to generate domain-specific, structured output formats, we offer a practical and efficient methodology for seamlessly integrating LLMs as drop-in replacements or supplementary components within existing systems. This fine-tuning approach significantly reduces the development complexity and engineering overhead associated with traditional server-side logic, thereby enabling the creation of more flexible and intelligent, AI-driven back-end architectures.

3. Methodology

This section outlines the methodology for integrating a fine-tuned Large Language Model (LLM) into a traditional back-end system, encompassing dataset creation, API layer design and implementation, deployment of the LLM as a service, and the overall system workflow.

3.1. Dataset creation

To effectively fine-tune the Large Language Model (LLM) for handling JSON-formatted inputs and outputs, we developed a specialized dataset [8] tailored to the requirements of our back-end system. The dataset creation process is pivotal, as it directly influences the model's ability to accurately parse, process, and generate structured responses.

This subsection delineates the steps undertaken to create the dataset, including:

1) Dataset Composition: Description of the data fields, format, and structure.
2) Generation Methodology: Strategies and tools used to generate and validate the dataset.
3) Computational Resources: Details of the hardware and software environments utilized for dataset creation and processing.

3.1.1. Dataset composition

The dataset [9] comprises 5,000 entries, each structured to facilitate the training of the LLM in handling diverse types of inputs and generating corresponding JSON-formatted outputs. Each entry in the dataset consists of four columns (examples shown in **Table 1**):

- Output String: The desired JSON-formatted response that the model should generate.
- Structured Input String: A JSON-formatted request that the model needs to process.
- Direct Input String: A direct, non-structured query related to the structured input.
- Conversational Input String: A conversationally phrased version of the direct

input query.

Table 1. Different format prompt for the same structured output.

Input	Output
{"A":74, "op":"*", "B":70}	{"result": "5180"}
What is the result of multiplying 74 by 70?	{"result": "5180"}
Calculate the result when you multiply 74 by 70?	{"result": "5180"}

3.1.2. Dataset generation methodology

The dataset was generated using the Phi-3.5 model [10] through meticulous prompt engineering. This approach involved designing specific prompts that instructed Phi-3.5 to produce the required structured and unstructured inputs alongside their corresponding outputs. The process ensured that the dataset encapsulated a wide variety of request types and response scenarios, enhancing the LLM's ability to generalize across different contexts.

3.1.3. Prompt engineering

Prompt engineering was employed to guide the Phi-3.5 model in generating high-quality, diverse data entries. The following Python template demonstrates how prompts were designed to convert structured inputs into various question types.

```
PROMPT_TEMPLATE = [
    {
        "role": "system",
        "content": "You are an AI assistant that converts structured prompts into {conversion_type} questions and no need to answer it"
    },
    {
        "role": "user",
        "content": (
            "Convert the following structured prompt into a {conversion_type} question:\n"
            "Structured Prompt: {structured_input}\n"
        )
    }
]
```

Code example: Prompt Template for generating customize dataset

To address the fundamental aspects of back-end API interactions, our dataset focuses on basic arithmetic operations, specifically addition (+), subtraction (-), multiplication (*), and division (/). The dataset comprises fewer than 100 entries, each representing a distinct arithmetic computation.

While the current dataset is limited in scope, it serves as a critical initial step towards training the LLM for more complex operations and diverse business logic implementations in future iterations.

3.1.4. Customized dataset development and reproducible workflow

Our primary contribution is the development of a specialized dataset tailored for

training Large Language Models (LLMs) to process and generate structured JSON inputs and outputs. Utilizing the Phi-3.5 model and advanced prompt engineering techniques, we created a dataset that enables LLMs to accurately interpret diverse input formats and produce corresponding JSON-formatted responses. Additionally, we established a reproducible workflow that allows researchers and practitioners to customize and generate similar datasets tailored to their specific structured output requirements. This dual contribution not only enhances the integration of LLMs into traditional back-end systems by ensuring precise data handling but also provides a scalable framework for future dataset creation efforts, fostering consistency and reliability in training LLMs for structured data tasks.

3.2. Dataset for database operations

To further evaluate the generalizability of our fine-tuning approach and its applicability to a wider range of data-driven tasks, we introduce a new dataset specifically designed to simulate database operations [11] (micost-database-op, 2024). This dataset complements our initial calculator model dataset and allows us to assess the model's ability to handle more complex data structures and operations commonly encountered in real-world applications. By simulating typical database interactions, such as querying, inserting, updating, and deleting data, we aim to provide a comprehensive evaluation of the model's performance in a more realistic and challenging scenario [12].

Dataset composition 1

The dataset comprises 5000 entries, each structured to facilitate the training of the LLM in handling diverse types of inputs and generating corresponding JSON-formatted outputs. That data structure is the same as the simple calculator dataset. Here below is the sample data.

input : {"action": "insert", "table": "sales", "data": {"order_date": "'2021-12-08'", "amount": "860.13"}}

output: INSERT INTO sales (order_date, amount) VALUES (''2021-12-08'', '860.13');

3.3. Supervised fine-tuning the large language model

We performed supervised fine-tuning of the LLaMA3 model [13] using our custom dataset, employing QLoRA (Quantized Low-Rank Adaptation) [14] to efficiently adapt the model with limited computational resources. This fine-tuning process was executed on a single NVIDIA T4 GPU, enabling the model to accurately interpret and generate structured JSON outputs based on the diverse inputs provided in the dataset.

By leveraging QLoRA, we optimized memory usage and accelerated training times without compromising the model's performance. The resulting fine-tuned LLaMA3 model is now capable of reliably handling both machine-readable and human-readable requests within our back-end system, enhancing its overall functionality and responsiveness.

3.4. API Layer design and implementation

The API Layer is responsible for managing communication between the client and the server. It formats and interprets RESTful requests and responses, handling tasks such as routing, data validation, and serialization. This layer ensures that requests from clients conform to the expected structure and that responses from the server are appropriately formatted for delivery. By isolating this functionality into a dedicated layer, we decouple the communication mechanics from the back-end's Core Logic, allowing the LLM to focus exclusively on higher-order tasks.

The API Layer also interfaces with the database. However, unlike traditional back-ends, it doesn't directly execute database operations. Instead, it forwards database requests generated by the Core Logic Unit to the database service, treating it similarly to a front-end client.

3.5. Integration of the Fine-tuned LLM as a Service

The Core Logic Unit contains the bulk of the back-end's application logic. Traditionally, this unit handles tasks such as business logic, database interactions, and processing rules specific to the application. In our design, this unit is replaced by an LLM, which is fine-tuned to emulate and execute the Core Logic of the application. The LLM processes structured inputs received from the API Layer, performs the necessary computations or decision-making processes, and generates structured outputs for the API Layer to format and deliver.

3.6. System architecture

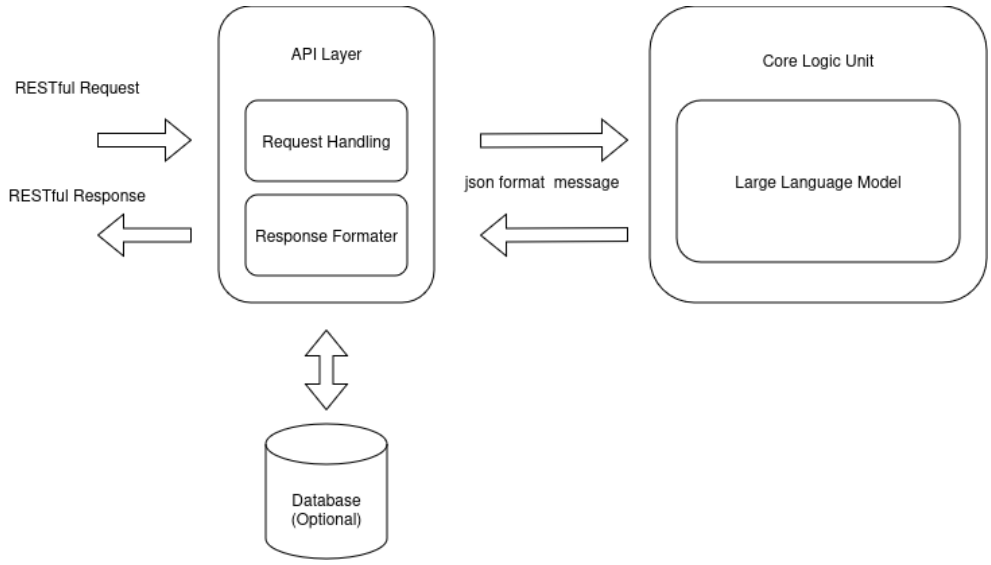

Figure 1. System architecture.

The system architecture is shown in **Figure 1**.

- Request Handling: A RESTful request from the client is received by the API Layer, which validates and formats the request into a structured format, such as JSON.
- Core Logic Execution: The structured request is passed to the LLM in the Core Logic Unit, which processes the input according to the application's logic and

generates an appropriate response.

- Response Formatting: The API Layer formats the LLM's output into a RESTful response and sends it back to the client.

This modular design ensures flexibility and scalability, allowing the API Layer to handle evolving communication protocols while the LLM adapts to increasingly complex back-end logic through iterative fine-tuning [15]. The separation of concerns also facilitates maintenance and testing, as each layer can be developed and validated independently. Our methodology demonstrates how LLMs can effectively replace traditional application logic in back-end systems, paving the way for AI-driven web back-end architectures.

4. Result

This section presents the experimental results demonstrating the performance of the fine-tuned LLama3 model integrated into the back-end system.

4.1. Performance metrics from the model fine-tuning process

We summarized the fine-tuning results using WandB. Results are shown in **Table 2**.

Table 2. Detailed evaluation and training metrics for fine-tuned model.

Metric	Value
eval/loss	0.40704
eval/runtime	26.0878
eval/samples_per_second	3.833
eval/steps_per_second	3.833
total_flos	1.27385×10^{15}
train/epoch	1
train/global_step	450
train/grad_norm	1.03545
train/learning_rate	0
train/loss	0.4144
train/runtime	670.3678
train_samples_per_second	1.343
train_steps_per_second	0.671

4.2. API layer accuracy and response time

We use the fine-tuned model as the Core Logic and test result accuracy and timing. The total number of test cases is 100. Results are shown in **Table 3**.

Table 3. Performance Metrics of the API Layer.

Average accuracy	0.765
Average response time	3.56

5. Limitations and future work

5.1. Limitations

Despite the successful integration of the fine-tuned Large Language Model (LLM) into our back-end system, several limitations must be acknowledged:

- Dataset Size and Computing Resources: The relatively small size of our dataset, coupled with limited computing resources, constrains the accuracy and generalizability of the trained model. A larger and more diverse dataset, alongside enhanced computational capabilities, could facilitate more nuanced learning and improve the model's performance across a broader range of tasks.
- Response Time: While the LLM-based service demonstrates stability across all tasks, its response time is noticeably slower compared to traditional back-end services. This latency may impact real-time applications where swift responses are critical. Although the system maintains consistent performance, optimizing the model architecture and deployment strategies is necessary to achieve response times comparable to conventional services.

5.2. Future work

To overcome current limitations and enhance the system, future work will focus on:

- Expanding the Dataset and Upgrading Computational Resources: Increasing the dataset size and diversity will improve the LLM's accuracy and adaptability. Additionally, enhancing computing infrastructure will allow for more extensive training and better overall performance.
- Optimizing Model Efficiency and Reducing Latency: Implementing optimization techniques like quantization and pruning can lower computational demands and speed up response times. Exploring specialized hardware and alternative deployment frameworks will further enhance efficiency without compromising system stability.

These improvements will address existing limitations and boost the system's effectiveness in real-world applications.

6. Conclusions

This study presents a specialized dataset for training Large Language Models (LLMs) to handle structured JSON inputs and outputs, addressing a significant research gap. Utilizing the Phi3.5 model and advanced prompt engineering techniques [16], we created a robust dataset that enables LLama3 to accurately interpret and generate JSON-formatted responses. Additionally, we established a reproducible workflow, allowing other researchers and practitioners to develop customized datasets for integrating LLMs into traditional back-end systems. Despite limitations in dataset size and computational resources, our approach demonstrates the potential of fine-tuned LLMs to enhance the functionality and responsiveness of back-end services. This work lays the foundation for future advancements in training methodologies and dataset expansion, paving the way for more efficient and accurate integration of LLMs in diverse operational environments.

Author contributions: Conceptualization, CZ; methodology, CZ and BL; software, BL; validation, CZ and BL; formal analysis, BL; investigation, BL; resources, CZ; data curation, BL; writing—original draft preparation, CZ; writing—review and editing, BL; visualization, CZ; supervision, BL; project administration, BL; funding acquisition, BL. All authors have read and agreed to the published version of the manuscript.

Conflict of interest: The authors declare no conflict of interest.

References

1. RoX818. How to Curate Datasets for OpenAI Fine-Tuning Success. AI Competence. Available online: https://aicompetence.org/how-to-curate-datasets-for-openai-fine-tuning/ (accessed on 22 January 2025).
2. Parthasarathy VB, Zafar A, Khan A, et al. The Ultimate Guide to Fine-Tuning LLMs from Basics to Breakthroughs: An Exhaustive Review of Technologies, Research, Best Practices, Applied Research Challenges and Opportunities. Available online: https://arxiv.org/html/2408.13296v1 (accessed on 15 October 2024).
3. Song Y, Xiong W, Zhu D, et al. RestGPT: Connecting Large Language Models with Real-World RESTful APIs. Available online: https://arxiv.org/abs/2306.06624 (accessed on 15 October 2024)
4. Wang L, Yang N, Huang X, et al. Large Search Model: Redefining Search Stack in the Era of LLMs. Available online: https://arxiv.org/abs/2310.14587 (accessed on 21 October 2024).
5. Pandya K, Holia M. Automating Customer Service Using LangChain: Building Custom Open-Source GPT Chatbot for Organizations. Available online: https://arxiv.org/abs/2310.05421 (accessed on 21 October 2024).
6. Papageorgiou G, Sarlis V, Maragoudakis M, et al. Enhancing E-Government Services Through State-of-the-Art, Modular, and Reproducible Architecture Over Large Language Models. Applied Sciences. 2024; 14(18): 8259. doi: 10.3390/app14188259.
7. Lewis P, Oguz B, Rinott R, et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. Available online: https://arxiv.org/abs/2005.11401 (accessed on 21 October 2024).
8. Aisuko. Diverse Calculation. Hugging Face. Available online: https://huggingface.co/datasets/aisuko/diverse_calculation. https://doi.org/10.57967/hf/3724 (accessed on 9 December 2024).
9. Micost. simple_calculation. Hugging Face Datasets. Available online: https://huggingface.co/datasets/micost/simple_calculation (accessed on 14 December 2024).
10. Abdin M, Jacobs SA, Awan AA, et al. Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone. Available online: https://api.semanticscholar.org/CorpusID:269293048 (accessed on 8 December 2024).
11. Micost. db_operate. Hugging Face Datasets. Available online: https://huggingface.co/datasets/micost/db_operate (accessed on 8 December 2024).
12. Micost. Fine-tuning Llama 3 with Simple Calculation. Kaggle. Available online: https://www.kaggle.com/code/micost/fine-tuning-llama-3-with-simple-calculation (accessed on 8 December 2024).
13. Llama Team, AI @ Meta. The Llama 3 Herd of Models. Available online: https://arxiv.org/abs/2407.21783 (accessed on 8 December 2024).
14. Dettmers T, Pagnoni A, Holtzman A. QLoRA: Efficient Finetuning of Quantized LLMs. Available online: https://arxiv.org/abs/2305.14314 (accessed on 8 December 2024).
15. Mathav Raj J, Kushala VM, Warrier H, et al. Fine Tuning LLM for Enterprise: Practical Guidelines and Recommendations. Available online: https://arxiv.org/abs/2404.10779 (accessed on 8 December 2024).
16. Microsoft. Phi-3.5-mini-instruct. Hugging Face. Available online: https://huggingface.co/microsoft/Phi-3.5-mini-instruct (accessed on 8 December 2024).