# Large Language Models as General-Purpose Servers: A New Approach

Chuan Zhang, Bowen Li, Bohan Cheng

***Abstract*—This essay explores the concept of using Large Language Models (LLMs) as general-purpose servers, drawing parallels between traditional web servers and finite-state machines. LLMs, like servers, produce outputs based on input and state, and can be adapted for non-conversational tasks through fine-tuning and structured input-output formatting. By packaging inputs and outputs in JSON format, with an API layer to handle RESTful requests, LLMs can serve various computational tasks. The primary advantage of this approach is the simplification of logic design, reducing development complexity.**

***Keywords—Large Language Models (LLM), general-purpose server, finite-state machine, fine-tuning,***

## I. Introduction

Web servers, traditionally understood as finite-state machines, operate by taking inputs, processing them based on their current state, and producing outputs accordingly. This fundamental structure can also be applied to Large Language Models (LLMs), which, though designed for language processing, can be adapted to function as general-purpose servers. By treating an LLM like a finite-state machine, developers can take advantage of its flexibility and processing power to handle a variety of tasks, while simplifying the complexity of traditional server logic.

## II. LLM as a Finite-State Machine

At their core, most servers can be modeled as finite-state machines: systems that take input, process it according to their current state, and return an appropriate output. In this sense, an LLM is no different from a traditional server. When provided with the right input, the LLM processes it based on its internal state (its training data and learned parameters) and generates an output. This behavior allows LLMs to be repurposed as servers for tasks beyond natural language conversation.

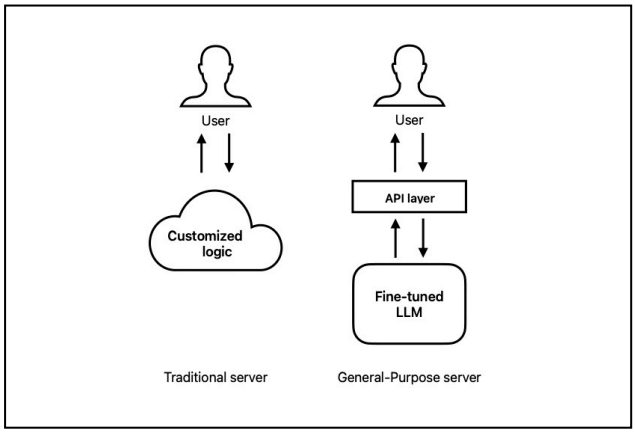
### *A. Packaging Inputs and Outputs*

To function as a general-purpose server, the input to an LLM can be structured as a string, similar to how chatbots receive input from users. The input can be packaged in JSON format, which is widely used in web applications due to its ease of use and readability. This string represents a request to the LLM, which processes it and generates a response.

The output from the LLM, typically a natural language response, can then be extracted and truncated to fit a desired format, ensuring that it is suitable for consumption by other applications. By using JSON for both input and output, the communication with the LLM remains consistent and manageable, allowing seamless integration into existing server infrastructures.

### *B. API Layer for Input/Output Management*

To streamline the communication between clients and the LLM-based server, an additional layer of API is necessary. This API layer handles the packaging of input data into the appropriate format before sending it to the LLM, as well as extracting the necessary information from the LLM's output. This layer is crucial for managing RESTful requests, ensuring that the interaction between the client and the LLM remains smooth and structured.




The API acts as an intermediary, converting client requests into a format that the LLM can understand and ensuring the outputs are ready for further use in the system. This makes the implementation of an LLM as a server more feasible, as it provides a reliable interface for communication.

### *C. Fine-Tuning for Specific Results*

While general-purpose LLMs are trained on broad datasets, specific server tasks often require more precise control over the outputs. This is where fine-tuning becomes crucial. By fine-tuning an LLM on specific data related to the desired task, the model can be optimized to produce consistent and expected outputs.

Fine-tuning involves training the LLM on a specialized dataset, often generated from existing input-output patterns. For instance, fine-tuning a model to perform basic calculations or handle specific requests can be accomplished by providing the LLM with example inputs and their corresponding outputs. This ensures that the server responds correctly to particular queries.

### *D. Automating the Dataset Generation for Fine-Tuning*

The process of fine-tuning an LLM to function as a general-purpose server requires a training dataset. Fortunately, many of the input-output patterns required for such tasks already exist in many use cases, making it possible to generate the training data automatically. For example, an LLM can be fine-tuned to perform basic calculations by providing a dataset of mathematical inputs and their respective outputs.

This ability to generate datasets from existing data sources reduces the time and effort needed for deployment, making the LLM more adaptable to different server functions.

## III. Fine-Tuning Llama 3 as a basic calculator

LLaMA 3 (Large Language Model Meta AI) is the third iteration of Meta's LLaMA series, designed to advance AI language models in terms of performance, efficiency, and scalability. Like its predecessors, LLaMA 3 is designed to be highly customizable. Developers can fine-tune the model on specific datasets to adapt it to specialized tasks, making it ideal for domain-specific applications.

### A. Dataset for Fine tuning

Dataset for fine turning LLM is generated with a separated function. Here below are some example of the data.

| input<br>string | output<br>string |
|---|---|
| {"A":1,"op":"/","B":93} | {"result": "0.010752688172043012"} |
| {"A":99,"op":"-","B":86} | {"result": "13"} |
| {"A":47,"op":"*","B":16} | {"result": "752"} |
| {"A":18,"op":"+","B":50} | {"result": "68"} |

### B. Training arguments and trainer parameters

The following code snippet demonstrates the training parameters used for fine-tuning the LLaMA model. These parameters control various aspects of the training process, such as batch size, learning rate, and the number of epochs, which are crucial for optimizing the model's performance. In this case, the parameters are fine-tuned to adapt LLaMA for a specific task (e.g., acting as a basic calculator)..

```
training_arguments = TrainingArguments(
    output_dir=new_model,
    per_device_train_batch_size=1,
    per_device_eval_batch_size=1,
    gradient_accumulation_steps=2,
    optim="paged_adamw_32bit",
    num_train_epochs=1,
    evaluation_strategy="steps",
    eval_steps=0.2,
    logging_steps=1,
    warmup_steps=10,
    logging_strategy="steps",
    learning_rate=2e-4,
    fp16=False,
    bf16=False,
    group_by_length=True,
    report_to="wandb"
)
trainer = SFTTrainer(
    model=model,
    train_dataset=dataset["train"],
    eval_dataset=dataset["test"],
    peft_config=peft_config,
    max_seq_length=512,
    dataset_text_field="text",
    tokenizer=tokenizer,
    args=training_arguments,
    packing= False,
)
```

### C. Package and extract the result

To effectively utilize a Large Language Model (LLM) like LLaMA as a general-purpose server, it is crucial to establish a structured method for packaging inputs and extracting outputs. The following code snippet illustrates how to format input data into a JSON structure suitable for sending to the model, as well as how to process and extract the resulting output after the model has generated a response. This approach mimics typical API interactions, ensuring that communication between the client and the LLM is both efficient and reliable.

```
def extract_result(text):
    pattern = r'\{.*"result":\s*"(\d+)".*\}'
    match = re.search(pattern, text)
    if match:
        return {"result": match.group(1)}
    else:
        return None


def calculator(a, b, op, tokenizer, model):
    messages = [
        {
            "role": "user",
            "content": f"{{\"A\": \"{a}\", \"B\": \"{b}\", \"op\": \"{op}\"}}"
        }
    ]
    prompt = tokenizer.apply_chat_template(messages, tokenize=False, add_generation_prompt=True)
    inputs = tokenizer(prompt, return_tensors='pt',padding=True, truncation=True).to("cuda")
    outputs = model.generate(**inputs, max_length=150, num_return_sequences=1)
    text = tokenizer.decode(outputs[0], skip_special_tokens=True)
    result_str = extract_result(text.split("assistant")[1])
    result_value = result_str["result"]
    return result_value
```

## IV. Simplifying Logic Design: The Main Benefit

One of the key benefits of using LLMs as general-purpose servers is the elimination of complex logic design from the development process. In traditional server development, developers must design specific logic to handle different types of requests and generate corresponding responses. This can be time-consuming and prone to errors.

With an LLM-based server, developers need only focus on the input and expected output. The model itself handles the complex task of generating the response based on its training. This simplifies the development process, much like the use of general-purpose chips has simplified hardware design in some applications, compared to Application-Specific Integrated Circuits (ASICs).

## V. Limitations and Future Work

Despite the potential advantages, there are limitations to using LLMs as general-purpose servers. The most significant drawback is efficiency. LLMs are primarily designed for language processing tasks, making them resource-intensive when applied to general-purpose tasks. Fine-tuning models for specific tasks can improve performance, but using an LLM as a server remains less efficient than traditional servers optimized for specific applications.

Moreover, as finite-state machines, servers often rely on external storage systems like databases to maintain state and retrieve information. Currently, LLMs do not have the capability to directly interact with databases in real time, which limits their functionality as general-purpose servers. Developing ways for LLMs to connect with external databases and manage state information will be a critical area for future work.

## VI. Conclusion

Using LLMs as general-purpose servers is an innovative approach that leverages their processing power and flexibility. By treating LLMs as finite-state machines and managing input and output through JSON formats, developers can simplify the logic design process and focus on the desired outcomes. Fine-tuning and API layers are necessary to ensure the model functions efficiently and consistently in server environments.

Performance inefficiencies and the lack of database integration present challenges that need to be addressed. With further development and optimization, LLMs have the potential to become valuable tools in general-purpose server architecture, expanding their role beyond natural language processing.

## References

1. D. V. Lindberg and H. K. H. Lee, “Optimization under constraints by applying an asymmetric entropy measure,” *J. Comput. Graph. Statist.*, vol. 24, no. 2, pp. 379–393, Jun. 2015, doi: 10.1080/10618600.2014.901225.