

University of Trieste

Department of Mathematics, Informatics and Geosciences

Master Degree in Data Science and Scientific Computing



Master Degree's thesis

Quantized LLMs for Function Calling: Evaluating Agent-Augmented Qwen and LLama on the BFCL Non Live AST Benchmark

Candidate:

Gianmarco Alessio

Advisor:

prof. Alessio Ansuini

ACADEMIC YEAR 2024/2025

Italian Abstract

Questa tesi esplora l'efficacia dei modelli linguistici di grandi dimensioni (LLM) quantizzati (S. Li et al., 2024), in particolare Qwen (Yang et al., 2024) e LLama (Dubey et al., 2024) con 1.5B, 3B, 7B, 8B di parametri, nell'affrontare compiti di chiamata di funzione. I modelli saranno valutati sul BFCL Non Live AST Benchmark (Fanjia Yan et al., 2024), sia nella loro versione base che in una versione con agenti (Wu et al., 2023). Come parte dello studio, verrà presentato un caso pratico che mostra l'applicazione dei modelli per la creazione automatica di oggetti basata su input generici (chiamato input di 'chiamata all'oggetto'), utilizzando una struttura gerarchica con agenti (Patil et al., 2024). L'obiettivo è valutare se l'integrazione di agenti e la comunicazione inter-agente, attraverso l'apprendimento per contesto (ICL) (Q. Dong et al., 2024), possano migliorare le prestazioni dei modelli quantizzati senza ricorrere al training e/o al fine-tuning.

English Abstract

This thesis explores the effectiveness of quantized large language models (LLMs), specifically Qwen (Yang et al., 2024) and LLama (Dubey et al., 2024) with 1.5B, 3B, 7B, 8B parameters, in addressing function calling tasks. The models will be evaluated using the BFCL Non Live AST Benchmark (Fanjia Yan et al., 2024), both in their base version and with integrated agents (Wu et al., 2023). As part of this study, a practical case will be presented to demonstrate the application of these models for the automatic creation of objects based on generic inputs (referred to as 'object calling input'), using a hierarchical structure with agents (Patil et al., 2024). The objective is to assess whether the integration of agents and inter-agent communication, through In Context Learning (ICL) (Q. Dong et al., 2024), can enhance the performance of quantized models without the need for training and/or fine-tuning.

Contents

1	Background and Related Work	7
1.1	Large Language Models	7
1.1.1	LLAMA 3.1 & Tool-use Capabilities	7
1.1.2	QWEN 2.5 & Tool-use Capabilities	8
1.2	Quantization of LLMs	8
1.3	Function Calling and BFCL Benchmark	10
1.3.1	Evaluation categories	10
1.3.2	Python Evaluation	11
1.3.3	Evaluation Metrics	11
1.3.4	Common Mistakes	13
1.3.5	Cost and Latency	13
1.4	In Context Learning (ICL)	13
1.4.1	Model Training & ICL approach	14
1.4.2	Instruction Formatting	14
1.4.3	Other improvements	14
1.5	Agent-Augmented LLMs	15
1.6	Related Research	15
2	Methodology	17
2.1	Model Selection	17
2.2	Quantization Process	19
2.3	Framework for Conversational Agents	20
2.3.1	Code Structure	20
2.3.2	Highlights	22
2.3.3	Math Problem-Solving Architecture (v001)	22

2.3.4	Agent Architecture Feedback Validator	23
2.3.5	Validator	25
2.4	Context Learning Strategy	27
2.5	Evaluation Metrics	31
2.6	Object Calling Case Study	32
2.6.1	User interACTION	34
2.6.2	Autonomous ACTION	34
2.7	Hardware Support	39
2.7.1	GeForce RTX 3060 Setup	39
2.7.2	Nvidia T4 Setup	40
2.7.3	Integration with BFCL Benchmark	40
3	Results and Analysis	41
3.1	Problem Analysis of the quantize models on BFCL Benchmark	42
3.2	Agent-Augmented Qwen and LLama on the BFCL Benchmark	47
3.2.1	Experimental Setup	47
3.2.2	Results and Analysis	48
3.3	Object Calling Case Study Results	48
4	Conclusion	52
4.1	Discussion	52
4.1.1	Quantization on BFCL benchmark	52
4.1.2	Agent-Augmented Qwen and LLama on the BFCL Benchmark .	53
4.1.3	Object Calling Case Study	55
4.1.4	Challenges & Improvements	55
4.2	Implications and Future Work	56
A		59
A.1	Code reference & Github	59
A.2	Costs	59

Introduction

Motivation

Large language models have demonstrated impressive value in performing as autonomous agents when equipped with external tools and API calls. Nonetheless, effectively harnessing their potential for executing complex tasks crucially relies on enhancements in their function-calling capabilities. (Lin et al., 2024)

Problem Statement

Clearly state the research question: Can the integration of agents and In Context Learning ICL (Q. Dong et al., 2024) improve the performance of quantized LLMs (Qwen (Yang et al., 2024) and LLama (Dubey et al., 2024)) on function calling tasks, as measured by the BFCL benchmark (Fanjia Yan et al., 2024)? Through this analysis, we seek to identify methods to enhance models' stability and generalization capabilities in real-world scenarios.

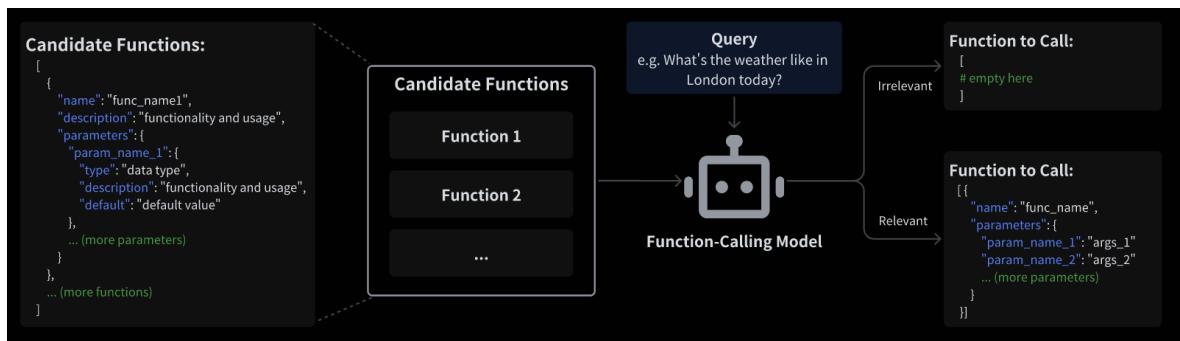


Figure 0.0.1: Demonstration of a simple function-calling process. (Lin et al., 2024)

In the development and deployment of large language models (LLMs) for function calling, a significant challenge arises from the reliance on function and parameter

names to infer functionality. Function names and parameter labels are often compact and subject to individual designers’ stylistic preferences, resulting in potential ambiguities. This can mislead the model in determining the intended use of a function, especially in complex or varying contexts. For example, a function labeled *fetch_data* might retrieve user data from a database in one setting but access data from an external API in another. Similarly, parameter names like *timeout* may represent distinct data types or formats depending on the context (e.g., seconds as an integer in one case, and “10s” as a string in another). Additionally, discrepancies in naming conventions—such as variations between **CamelCase** and **snake_case**—further challenge the model’s robustness, as it may struggle to generalize across these stylistic differences when encountered in testing environments divergent from its training data. Overcoming these issues requires enhancing the model’s ability to interpret function and parameter names within the context of their specific application, while mitigating the impact of stylistic and contextual naming variations. Addressing this problem is essential for improving the model’s accuracy and reliability in selecting appropriate functions, especially when deployed across diverse, real-world scenarios where such naming inconsistencies are prevalent.

Contributions

The main contributions of this research can be summarized as follows:

- **Agents Architecture & Framework:** This work proposes a versatile framework for developing multi-agent conversational architectures, facilitating a broad spectrum of applications based on large language models (LLMs) (Wu et al., 2023). The framework enables the creation of agents that can engage in dynamic, customizable, and collaborative conversations to solve complex tasks. Agents may utilize a variety of interaction models, including LLMs, tools, human involvement, or combinations thereof, allowing for flexible and scalable conversation patterns that incorporate humans into the loop as necessary.
- **Efficient Scaling and Cost-Effectiveness in LLM Quantization:** This research proposes a direction aimed at leveraging LLMs with model sizes under 8

billion parameters, specifically quantized models, to manage scalability and reduce the costs associated with fine-tuning and inference. By focusing on the optimization of small models ($<=8$ B parameters) that can be executed on hardware such as the NVIDIA RTX-3060, this approach seeks to advance the capabilities of quantized models (S. Li et al., 2024) for practical applications. The potential for locally hosted, quantized models enhances data privacy by enabling users to operate and fine-tune these models directly on their own devices. Furthermore, it opens up possibilities for individuals to utilize agents and in-context learning (ICL) (Q. Dong et al., 2024) to refine model outputs, without incurring the high computational expenses typically associated with larger-scale models.

Thesis Structure

This thesis begins with an overview of related work and background, establishing the foundation for this research on quantized large language models (LLMs) and multi-agent systems (Wu et al., 2023). The methodology chapter follows, detailing the design choices and implementation process aimed at enhancing function-calling tasks using agent frameworks and in-context learning (ICL) (Q. Dong et al., 2024).

Next, the results and analysis chapter presents the experimental findings and evaluates the performance of the proposed approach, comparing it with existing work. The thesis concludes with a summary of key findings, discussing their implications for future research and addressing the ongoing challenge of making scalable AI solutions accessible for startups amid high operational costs.

Chapter 1

Background and Related Work

1.1 Large Language Models

1.1.1 LLAMA 3.1 & Tool-use Capabilities

Llama 3’s tool-use capabilities extend its functionality far beyond standard conversation, allowing it to call external tools, interpret code, and conduct multi-step reasoning to solve complex tasks. Key tools include a search engine for real-time information retrieval (via Brave Search), a Python interpreter for running code and handling data operations, and the Wolfram Alpha API for advanced computational tasks (Nakano et al., 2022; Thoppilan et al., 2022; Parisi, Zhao, and Fiedel, 2022; Gao et al., 2024; Mialon et al., 2023; Qu et al., 2024). . The model can operate in both single-turn and multi-turn dialogs, as well as in single-step and multi-step contexts. In single-turn, single-step interactions, the model processes one user prompt, performs a tool call, and returns an answer. For example, if the user requests a quick data summary, Llama 3 might use a Python object to perform a computation and respond in a single interaction.

In contrast, multi-turn, multi-step interactions involve several back-and-forth exchanges with the user, requiring multiple tool calls and reasoning steps to achieve a solution. For instance, if Llama 3 receives a complex query like “analyze this dataset, summarize findings, and graph trends,” it can sequentially call a Python interpreter to read and analyze data, visualize results, and then follow up with explanations in response to additional user questions.

Python objects in Llama 3 provide functions with tool descriptions and parameters, making it straightforward for the model to generate the correct function calls. A simple example might be a Python object with methods for various mathematical operations, such as a Statistics object with a mean method. When the model receives a user prompt requiring a statistical calculation, it generates a call to Statistics.mean, executes it, and returns the result. This design supports dynamic, adaptable interaction, where Llama 3 is trained to recognize in-context tool definitions and automatically generate the right calls, even with unfamiliar tools.

1.1.2 QWEN 2.5 & Tool-use Capabilities

The QWEN models are designed as versatile agents with robust capabilities in daily tasks (Bai et al., 2023), primarily through effective tool-use and strategic task planning.

To advance QWEN’s functionality as an agent, a self-instruction approach (Y. Wang et al., 2023) was applied to supervised fine-tuning (SFT). This method leverages in-context learning, where QWEN generates increasingly precise and well-structured responses based on a few provided examples. Specifically, prompts are designed in the ReAct framework (S. Yao et al., 2023) to guide QWEN’s responses. Human annotators were then involved to refine this data by filtering out noise, resulting in a progressively larger, high-quality sample set. After multiple iterations of this process, we compiled a collection of approximately 2000 high-quality examples. These samples were integrated into QWEN’s training dataset alongside other general-purpose SFT samples, ensuring the model retains broad applicability while also gaining enhanced capabilities for agent-specific tasks.

1.2 Quantization of LLMs

Post-training quantization (PTQ) is a crucial technique for optimizing large language models (LLMs) by reducing memory requirements and computational overhead, making deployment more feasible across diverse environments. Through careful selection of quantization methods, one can strike a balance between model efficiency and performance. In the experiments, focused on `k_m` models, particularly the `q4_k_m` quantization format, which provided an optimal trade-off between perplexity and model size.

They follow a particular naming convention: “q” + the number of bits used to store the weights (precision) + a particular variant.

Below is an overview of various quantization methods, each offering a unique balance between performance and resource requirements (*Saving to GGUF / Unsloth Documentation* 2024):

- **f32**: Not recommended due to high memory usage and slow speed.
- **f16**: Offers fast conversion and accuracy retention but is memory-intensive and slow.
- **q8_0**: Fast conversion, high resource use but generally suitable for various applications.
- **q4_k_m**: Recommended. Uses Q6_K for half of attention.wv and feed_forward.w2 tensors, with the rest in Q4_K.
- **q5_k_m**: Similar to q4_k_m but with q5_k.
- **q4_k_s**: All tensors are quantized to q4_k.
- **q3_k_m, q3_k_l, q3_k_s**: Variations in tensor quantization precision, balancing resource use and accuracy.
- **q6_k**: High accuracy but requires more resources.

Table 1.1 presents the experimental results from llama.cpp (*k-quants by ikawrakow · Pull Request #1684 · gggerganov/llama.cpp* 2024), showcasing the impact of different quantization methods on perplexity and file size for the 7B and 13B models.

Model	Measure	f16	q2_k	q3_k_s	q3_k_m	q3_k_l	q4_k_s	q4_k_m	q5_k_s	q5_k_m	q6_k
7B	Perplexity	5.9066	6.7764	6.4571	6.1503	6.0869	6.0215	5.9601	5.9419	5.9208	5.9110
7B	File Size (GB)	13.0	2.67	2.75	3.06	3.35	3.56	3.80	4.33	4.45	5.15
13B	Perplexity	5.2543	5.8545	5.6033	5.4498	5.4063	5.3404	5.3002	5.2785	5.2638	5.2568
13B	File Size (GB)	25.0	5.13	5.27	5.88	6.45	6.80	7.32	8.36	8.60	9.95

Table 1.1: Quantization Results for 7B and 13B Models

In function calling experiments, q4_k_m emerged as the preferred quantization method. The model achieved a desirable balance between file size and perplexity, ensuring efficient inference without compromising accuracy. Specifically, q4_k_m allocates q6_k precision to critical tensors (attention.wv and feed_forward.w2), while

using `q4_k` for others. This balance allowed for optimized computational efficiency and model fidelity, making `q4_k_m` an ideal choice for scenarios requiring high precision at manageable resource costs.

1.3 Function Calling and BFCL Benchmark

The Berkeley Function-Calling Leaderboard (BFCL) is an initiative to assess the function-calling capabilities of large language models (LLMs) for real-world applications (Fanjia Yan et al., 2024). This leaderboard evaluates models' abilities to generate accurate function calls, which are critical for integration into enterprise workflows, APIs, and diverse domains.

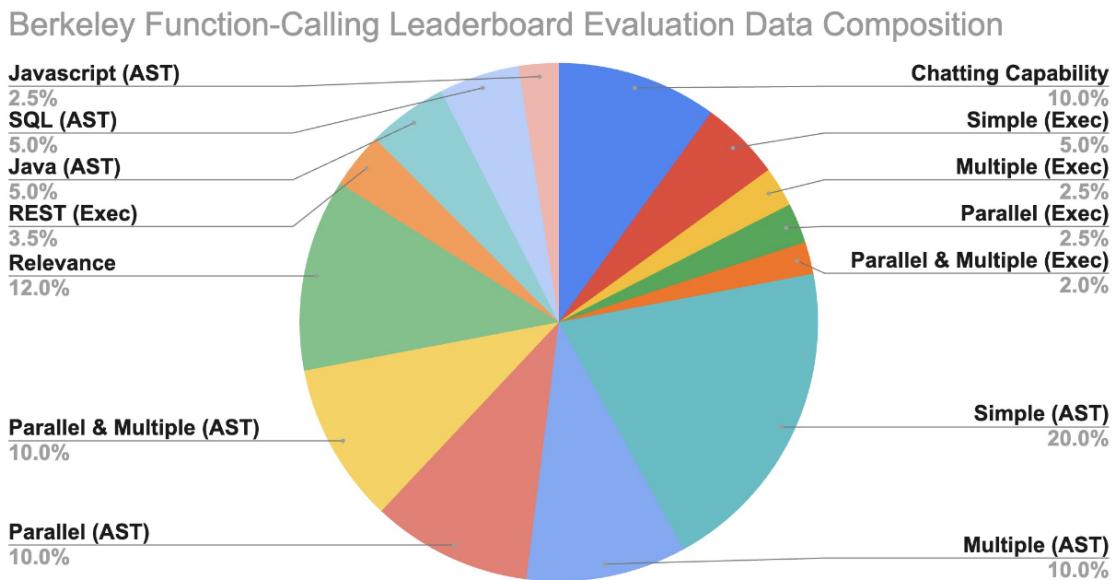


Figure 1.3.1: Berkeley Function-Calling Leaderboard (BFCL) Data Composition (Fanjia Yan et al., 2024)

1.3.1 Evaluation categories

I break down the majority of the evaluation into two categories:

- **Python:** Simple Function, Multiple Function, Parallel Function, Parallel Multiple Function
- **Non-Python:** Chatting Capability, Function Relevance Detection, REST API, SQL, Java, Javascript.

1.3.2 Python Evaluation

- **Simple Functions:** Single function evaluation contains the simplest but most commonly seen format, where the user supplies a single JSON function document, with one and only one function call will be invoked.
- **Multiple Functions:** Multiple function category contains a user question that only invokes one function call out of 2 to 4 JSON function documentations. The model needs to be capable of selecting the best function to invoke according to user-provided context.
- **Parallel Functions:** Parallel function is defined as invoking multiple function calls in parallel with one user query. The model needs to digest how many function calls need to be made and the question to model can be a single sentence or multiple sentence.
- **Parallel Multiple Functions:** Parallel Multiple function is the combination of parallel function and multiple function. In other words, the model is provided with multiple function documentation, and each of the corresponding function calls will be invoked zero or more times.

Programming languages evaluated include Python, Java, JavaScript, REST APIs, and SQL.

Simple	Multiple	Parallel	Parallel Multiple	Irrelevance
Query What is $1 + 2$? Candidates <code>[add(int a, int b)]</code> Correct Answer <code>[add(a=1, b=2)]</code>	Query What is $1 + 2$? Candidates <code>[add(int a, int b), mult(int a, int b)]</code> Correct Answer <code>[add(a=1, b=2)]</code>	Query What is $(1 + 2)$ and (3×4) ? Candidates: <code>[add(int a, int b)]</code> Correct Answer <code>[add(a=1, b=2), add(a=3, b=4)]</code>	Query What is $(1 + 2)$ and (3×4) ? Candidates: <code>[add(int a, int b), mult(int a, int b)]</code> Correct Answer <code>[add(a=1, b=2), mult(a=3, b=4)]</code>	Query What is $1 + 2$? Candidates: <code>[mult(int a, int b)]</code> Correct Answer <code>[]</code>

Figure 1.3.2: Demonstration of different function-calling tasks. (Lin et al., 2024)

1.3.3 Evaluation Metrics

BFCL employs two primary evaluation methods:

1. **Abstract Syntax Tree (AST) Evaluation:** This approach compares the structure of model-generated function calls to expected outputs (Figure ??).

2. **Executable Evaluation:** Tests real-world executability by running generated function calls.

Additional metrics include *Function Relevance Detection*, which assesses whether the model refrains from invoking irrelevant functions. In this research, the evaluation relied exclusively on Abstract Syntax Tree (AST) analysis. This approach enabled precise structural examination of code elements, allowing for detailed assessment of function names, parameters, and overall syntax without executing the code. By focusing on AST evaluation, we ensured a robust and consistent framework for verifying function structure and parameter requirements.

AST Evaluation (Figure ??)

- **Function Matching:** The procedure first extracts and verifies the function name, replacing any dots (.) with underscores (_) to ensure compatibility with certain models (e.g., OpenAI). This substitution is applied consistently during evaluation.
- **Required Parameters Matching:** Required parameters, as defined in the function documentation, are extracted and matched exactly. Only documented parameters are allowed, and any additional outputs are flagged as errors.

The final counts of each of the categories selected in BFCL with the composition of more granular types are shown in the following table:

Abstract Syntax Tree (AST) Evaluation 🌱			
Simple Function	Multiple Functions	Parallel Functions	Parallel Multiple
Py: 400 Java: 100 JS: 50	Py: 200	Py: 200	Py: 200

Figure 1.3.3: The final counts of each of the categories selected (Fanjia Yan et al., 2024)

1.3.4 Common Mistakes

- Models underperforms in scenarios where the parameters are not immediately available in the user question but instead require some implicit conversions.
- Models function documents are difficult to format and their typings are restrictive in real-world scenarios.
- Chat Models tend to generate malformed function calls in which parameters can be extracted but not executable

1.3.5 Cost and Latency

- **Latency:** Latency was calculated based on serving the model on a locally hosted RTX 3060 GPU without batching. Since the model was evaluated on individual entries, latency is measured directly per entry as the time taken for each inference.
- **Cost:** The operational cost is detailed in the appendix A.2. The primary evaluation used an RTX 3060 GPU in a non-commercial environment, providing an estimate of resource utilization on consumer-grade hardware. Additionally, an NVIDIA T4 GPU from the Hugging Face endpoint was utilized for benchmarking, offering a comparative perspective on performance in a cloud-based setup (*Dedicated Endpoints / Inference Endpoints - Hugging Face 2024*).

1.4 In Context Learning (ICL)

In-context learning (ICL) leverages analogy-based reasoning. Figure 1.4.1 illustrates how language models make decisions using ICL. First, a small set of demonstration examples is provided to form the prompt context, typically formatted as natural language templates.

Given these demonstrations, a new query (e.g., “Review: Good meal!”) is appended to the prompt, creating an input sequence for the language model. Unlike supervised learning, where models are trained through parameter updates with gradient back-propagation, ICL operates without modifying model parameters. Instead, the model identifies patterns from the demonstrations to make accurate predictions based on the prompt context.

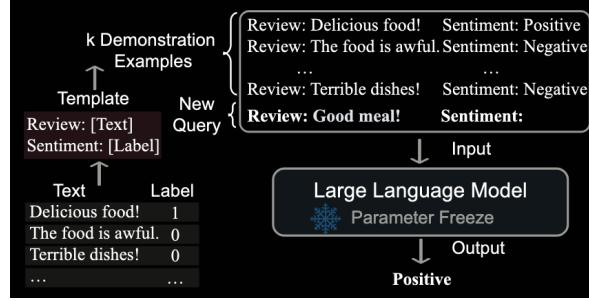


Figure 1.4.1: Illustration of in-context learning. (Q. Dong et al., 2024)

1.4.1 Model Training & ICL approach

Although LLMs have demonstrated promising ICL capability directly, many studies revealed that these ICL capabilities can be further enhanced through specialized training before inference (Gu et al., 2023; Shi et al., 2023).

1.4.2 Instruction Formatting

A common approach for formatting demonstrations involves concatenating examples $(x_1, y_1), \dots, (x_k, y_k)$ directly using a template T . However, for tasks requiring complex reasoning (e.g., math word problems or commonsense reasoning), learning the mapping from x_i to y_i may be challenging with only k demonstrations.

Considering the generation abilities of LLMs, (Zhou et al., 2023) proposed an Automatic Prompt Engineer for automatic instruction generation and selection. In its simplest form, APE takes as input a dataset (a list of inputs and a list of outputs), a prompt template, and optimizes this prompt template so that it generates the outputs given the inputs.

1.4.3 Other improvements

- In-context Vectors: Making In Context Learning More Effective and Controllable Through Latent Space Steering (S. Liu et al., 2024)
- Meta Prompting emphasizes the structure and syntax of information over traditional content-centric methods, enhancing token efficiency, and enabling more equitable problem-solving comparisons, especially against few-shot prompting methods. (Y. Zhang, Yuan, and A. C.-C. Yao, 2024)

- Reasoning traces help the model induce, track, and update action plans as well as handle exceptions, while actions allow it to interface with and gather additional information from external sources such as knowledge bases or environments (S. Yao et al., 2023)

1.5 Agent-Augmented LLMs

The agent framework in this thesis is inspired by the concept introduced in the open-source AutoGen2 framework (Wu et al., 2023), which enables developers to build applications with Large Language Models (LLMs) through multiple agents that interact to complete tasks. AutoGen agents are designed to be customizable and conversable, capable of operating in various modes that integrate LLMs, human input, and external tools. Although AutoGen provides a generic framework for building diverse applications across different complexities and LLM capabilities, the framework developed in this thesis is an attempt to optimize and improve the tool-based paradigm. This approach focuses on enhancing function-calling optimization in quantized models, offering improved performance for specific tasks. Empirical studies have shown the effectiveness of AutoGen across domains such as mathematics, coding, question answering, operations research, online decision-making, and entertainment. In this research, the framework's utility is demonstrated in optimizing function calling within the context of quantized LLM models.

1.6 Related Research

Existing studies have proposed the use of multi-agent collaborations to enhance the effectiveness of large language model (LLM)-based code generation. Benchmarks such as HumanEval (M. Chen et al., 2021) and HumanEval-ET focus on a range of programming challenges, while MBPP (Austin et al., 2021) and MBPP-ET (Y. Dong et al., 2024) offer a comprehensive collection of Python programming problems. These benchmarks evaluate a model's proficiency in Python syntax and its capacity to address various coding scenarios. Several notable frameworks illustrate this multi-agent approach:

- **MetaGPT** (Hong et al., 2024)
- **ChatDev** (Qian et al., 2024)
- **AgentVerse** (W. Chen et al., 2023)
- **AgentCoder** (Huang et al., 2024)

While these frameworks attempt to overcome the limitations of single-agent systems by distributing the workload and optimizing performance across different aspects of the code generation process, they also present certain limitations. First, many of these systems lack an effective feedback mechanism to provide LLMs with valuable insights on their generated outputs. For example, MetaGPT achieves only 80% accuracy in its generated test cases for HumanEval (Hong et al., 2024). Additionally, these frameworks often involve a substantial number of agents (e.g., MetaGPT employs five agents, and ChatDev utilizes seven agents), which leads to significant token overhead due to the communication and coordination demands among agents.

These multi-agent collaboration frameworks are designed to address these challenges, particularly in the domain of function generation, by balancing the workload among multiple specialized agents to enhance overall code generation efficiency.

Chapter 2

Methodology

2.1 Model Selection

To select appropriate models, I began by examining models in the BFCL benchmark with fewer than 8 billion parameters. Within this range, I focused on LLama and Qwen models are recognized as foundational structures in research, often serving as base architectures for further fine-tuning. This choice aimed to minimize experimental bias potentially introduced by previous studies and provided a neutral starting point for evaluation.

Attempts to locate other benchmarks with a focus on small ($<=8B$) or quantized models were unsuccessful, as existing benchmarks predominantly assess larger models or do not account for quantization (see 4.1.4). Figure 2.1.1 presents the performance of these selected models on the BFCL benchmark (Fanjia Yan et al., 2024). Given additional resources (e.g., more GPUs) and time, further testing could be extended to include a broader set of models for comprehensive analysis.

Excluding models that have been extensively explored in previous studies, we arrived at the following evaluation, focusing specifically on the Qwen and LLama models with fewer than 8 billion parameters. The histogram 2.1.2 excludes models such as ToolAce-8B (W. Liu et al., 2024), Hammer (Lin et al., 2024), xlam-7B (J. Zhang et al., 2024), Hermes (*Hermes 2 - a NousResearch Collection* 2024; Teknium, Quesnelle, and Guang, 2024) and Minicpm3 (Hu et al., 2024) to reduce potential bias from previous analyses.

To provide additional context, it is essential to distinguish between the "instruct"

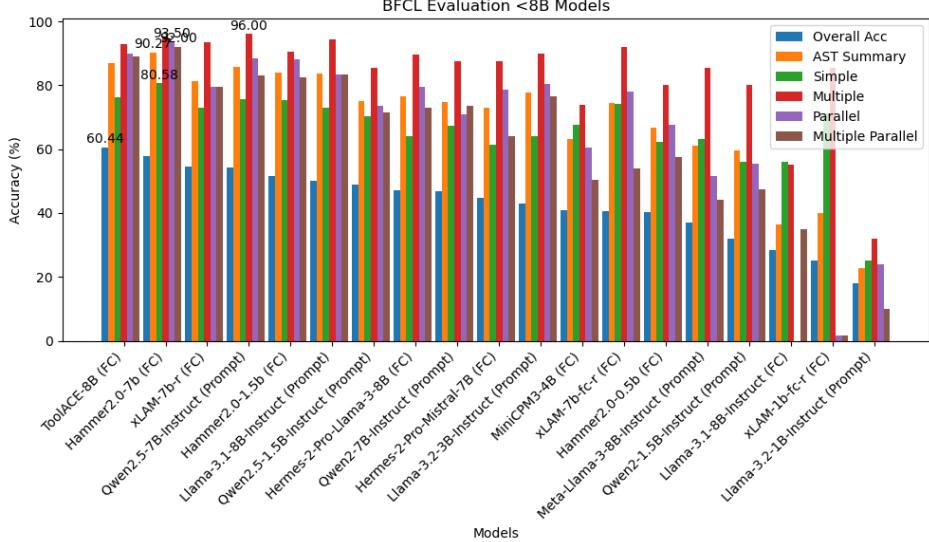


Figure 2.1.1: Performance of Small Models (<8B) on the BFCL Benchmark (Fanjia Yan et al., 2024)

and "noninstruct" categories, as well as the "FC" (Function Calling) and "prompt" modes. The "instruct" category refers to models fine-tuned to follow user instructions closely, optimizing them for tasks that require directive understanding and execution. In contrast, "non-instruct" models are typically pre-trained without this specific fine-tuning, making them less responsive to explicit instructions but often effective in generating responses based on broad contextual understanding.

The terms "FC" and "prompt" further describe the modes of interaction with the models. "FC" (Function Calling) indicates an operational mode in which models execute specific function calls as part of their output, useful for applications requiring structured action-based responses. "Prompt" mode, on the other hand, leverages traditional text-based prompting, where models generate responses based on open-ended text cues, without direct function invocation.

This differentiation highlights the unique capabilities of each model set-up, with instruct-tuned and FC-enabled models potentially offering advantages in structured response generation and direct task execution.

The results, summarized in Table 2.1, reflect the performance metrics derived from the BFCL benchmark in several categories, including the tasks AST Summary, Simple, Multiple, Parallel and Multiple Parallel.

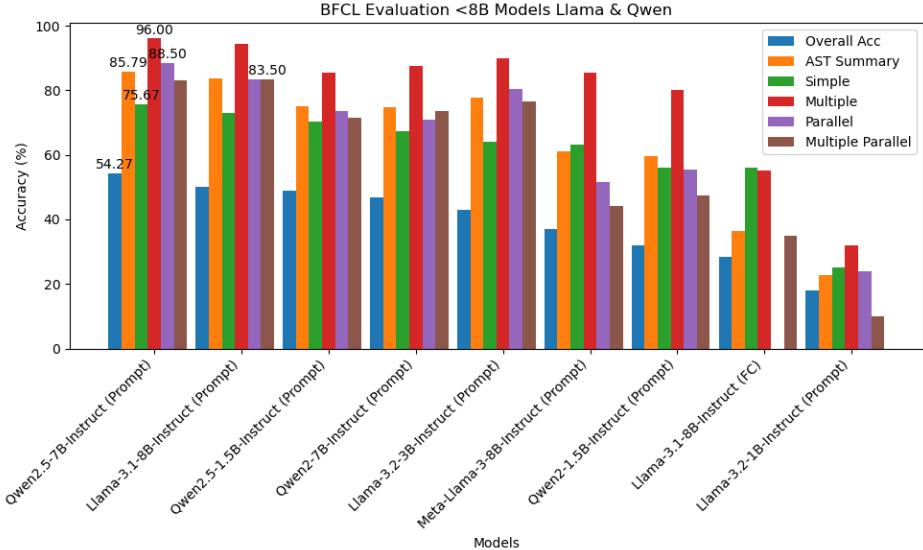


Figure 2.1.2: Performance of Qwen & Llama Small Models (<8B) on the BFCL Benchmark (Fanjia Yan et al., 2024)

Model	AST Summary	Simple	Multiple	Parallel	Multiple Parallel
Qwen2.5-7B-Instruct (Prompt)	85.79	75.67	96.0	88.5	83.0
Llama3.1-8B-Instruct (Prompt)	83.62	73.00	94.5	83.5	83.5
Qwen2.5-1.5B-Instruct (Prompt)	75.19	70.25	85.5	73.5	71.5
Qwen2-7B-Instruct (Prompt)	74.85	67.42	87.5	71.0	73.5
Llama3.2-3B-Instruct (Prompt)	77.77	64.08	90.0	80.5	76.5

Table 2.1: Results coming from the BFCL benchmark for the small models selected full precision weight matrices (Fanjia Yan et al., 2024)

2.2 Quantization Process

For the quantization procedure, I utilized the Unsloth library (*Saving to GGUF / Unsloth Documentation* 2024) and implemented it for all selected state-of-the-art models included in this analysis, as described in the previous section. These models, listed in Table 2.1, were selected based on their performance and relevance for the quantization comparison presented in the next chapter of the section 3.1.

Typically, machine learning models, when not quantized, are processed using full precision weight matrices. In other words, the model parameters are stored and processed in their complete high-precision format, which maximizes accuracy, but also requires considerable computational resources. In contrast, quantization compresses these weights, such as through the `q4_k_m` quantization format, into lower bit representations, reducing the model’s memory footprint and computational demands.

In this analysis, I first compared the results of quantized models in the `q4_k_m` format with their non-quantized counterparts to assess any discrepancies in accuracy

or performance. This allowed for an evaluation of whether quantization introduces notable trade-offs in model effectiveness. Following this, the quantized models were integrated with an agent framework for further testing.

2.3 Framework for Conversational Agents

This Python file implements a framework for developing and managing AI-powered conversational agents, enabling interaction between users and AI models or between multiple AI agents. The design is modular and extensible, featuring a base `Agent` class and specialized subclasses such as `OpenSourceAgent` and `UserAgent`. These agents can process and respond to messages, maintaining an internal memory of conversations to facilitate context-aware responses. A `Mediator` class oversees interactions, allowing dynamic communication between agents.

2.3.1 Code Structure

The foundational `Agent` class defines the blueprint for any conversational agent:

```
1 class Agent:
2     def __init__(self, name: str, system_instruction: str):
3         self.id = str(uuid.uuid4())
4         self.name = name
5         self.system_instruction = system_instruction
6         self.memory: List[Dict[str, Any]] = []
```

The subclass `OpenSourceAgent` is designed to integrate with various AI models, such as Llama or GPT-based models, using local inference or remote APIs. For instance, the following snippet initializes a local Llama model:

```
1 if not huggingface_endpoint_url and ('gpt' not in self.
2     model_name):
3     self.my_model = Llama(
4         model_path=model_config["path"],
5         n_ctx=4096,
6         verbose=False
7     )
```

```
7     self.stop_token_ids = ["<| EOT |>"]
```

To handle message processing, `OpenSourceAgent` provides a `process_message` method, which interacts with the AI model and appends the response to the agent's memory:

```
1 def process_message(self, message: str) -> str:
2     self.memory.append(message)
3     messages = [
4         {"role": "system", "content": self.system_instruction},
5         message
6     ]
7     # Model interaction logic
8     ...
9     self.memory.append({
10         "role": self.name,
11         "content": response_text
12     })
13
14     return response_text
```

The `Mediator` class acts as a controller, enabling communication between agents. It supports multi-turn dialogues and dynamic message routing. The `chat` method handles iterative exchanges, as shown below:

```
1 def chat(self, sender_id: str, recipient_id: str, content: str,
2         str_condition: str = "", max_turns: int = 4):
3     sender = self.get_agent(sender_id)
4     recipient = self.get_agent(recipient_id)
5     res_recipient = content
6     counter = 0
7     while counter < max_turns:
8         res_recipient = self.send(sender_id, recipient_id,
9             res_recipient if counter == 0 else res_sender)
10        print(f'[{recipient.name}]', res_recipient)
11        res_sender = self.send(recipient_id, sender_id,
12            res_recipient)
13        print(f'[{sender.name}]', res_sender)
```

```

11     counter += 1
12     return res_recipient

```

2.3.2 Highlights

1. **Modularity:** The use of a base `Agent` class and specialized subclasses promotes code reuse and scalability.
2. **Multi-Model Support:** By integrating handlers for models like Llama and Qwen, the framework can accommodate diverse use cases.
3. **Dynamic Communication:** The `Mediator` class enables complex workflows involving multiple agents, such as collaborative problem-solving.
4. **Extensibility:** The framework's design allows additional models and functionalities to be seamlessly integrated.

This file provides a robust foundation for developing AI-based conversational systems, combining modularity, scalability, and multi-model compatibility.

2.3.3 Math Problem-Solving Architecture (v001)

The framework introduces a specialized method, `math_problem_solving_architecture`, designed to simulate collaborative problem-solving among agents. This architecture leverages a multi-agent setup, including an `action_caller`, a `syntax_expert`, and a `action_caller` with a prepare system prompt to handle expert critiques, all instantiated from the `OpenSourceAgent` class. Each agent is initialized with a unique system prompt to align with its specialized role, creating a dynamic and iterative communication flow managed by the `Mediator`. The method's core objective was to facilitate interaction between the `action_caller` and `syntax_assistant`, aiming to collaboratively address syntax issues commonly arising from quantized models (discussed in the sec: 3.1), particularly using In-Context Learning (ICL) techniques.

The implementation allows for structured communication, as shown below:

```

1 def math_problem_solving_architecture(self, assistant_id: str,
                                         assistant2_id: str, expert_id: str, user_input: str = "Write
                                         your message", str_condition: str = "", max_turns: int = -1):

```

```

2     res_sender = self.send("user", assistant_id, user_input)
3     print(f'[user]', user_input)
4     print(f'[{self.get_agent(assistant_id).name}]', res_sender)
5     res_sender2 = self.send(assistant_id, assistant2_id,
6                               res_sender)
7     print(f'[{self.get_agent(assistant2_id).name}]',
8           res_sender2)
9
10    res = self.chat(assistant2_id, expert_id, content=
11                      res_sender2, str_condition=str_condition, max_turns=
12                      max_turns)
13
14    return res

```

While this architecture served as a starting point, the results were limited, primarily due to the lack of precise control over the function call results produced by the `action_caller`. The naive approach highlighted the need for more sophisticated mechanisms to validate and refine outputs, which led to the development of the next iteration, the `Agent Architecture Feedback Validator`, discussed in the following section. Inspired by principles from the AutoAgent (Wu et al., 2023) and AgentCoder (Huang et al., 2024), this framework laid the groundwork for future improvements but demonstrated the necessity of tighter integration and control mechanisms.

2.3.4 Agent Architecture Feedback Validator

The `Agent Architecture Feedback Validator` builds upon the foundations of the `Math Problem-Solving Architecture` by introducing a more sophisticated iterative validation loop to handle function calls and their correctness. This new architecture addresses the limitations identified in the earlier approach by incorporating additional agents for validation and feedback, enabling a more structured and rigorous evaluation of function call results, inspired by the `AgentCoder` but instead of using a compiler I used a validator built looking at the quantize models mistakes (section 3.1). The architecture includes five key agents: `tester`, `caller`, `caller_feedback`, `validator`, and `validator_feedback`. These agents work collaboratively through the `Mediator` to ensure iterative improvement of the responses.

The core workflow begins with a user-provided question, processed sequentially by

Math Problem Solving

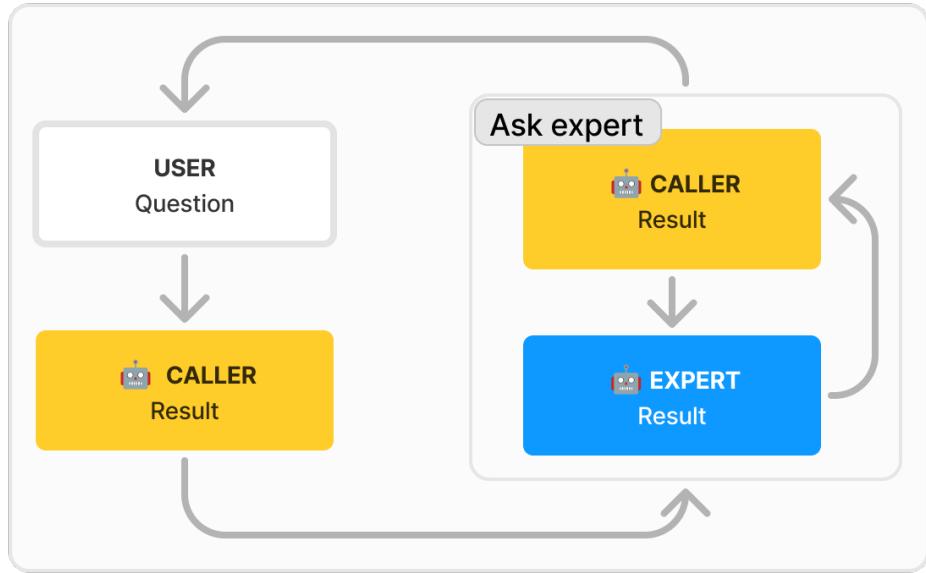


Figure 2.3.1: The Math Problem solving architecture (Wu et al., 2023)

the `tester` and `caller`. Their outputs are evaluated by the `validator`, which applies predefined criteria to assess the correctness of the function calls, as demonstrated in the implementation below:

```

1 res_tester = send_and_print(mediator, "user", tester.id,
2                               user_question, "[TESTER]")
3
2 res_caller = send_and_print(mediator, tester.id, caller.id,
3                               res_tester, "[CALLER]")
3
3 validator_result = validate_and_print(res_caller, functions, "[VALIDATOR]")

```

The results of each test iteration are stored in a `previous_tests` list, enabling feedback aggregation. A feedback loop between the `caller_feedback` and `validator_feedback` agents refines the outputs based on earlier results, creating a cycle of improvement. This iterative process continues until a predefined `VALIDATION_CONDITION` is met or a maximum number of turns is reached, as illustrated:

```

1 while counter < MAX_TURNS:
2     res_recipient = send_and_print(mediator, validator.id,
3                                     validator_feedback.id, res_sender, "[VALIDATOR FEEDBACK]")

```

```

3     validator_result = validate_and_print(res_recipient,
4                                         functions, "VALIDATION LOOP: ")
5
6     if VALIDATION_CONDITION in res_sender:
7         break
8
9     counter += 1

```

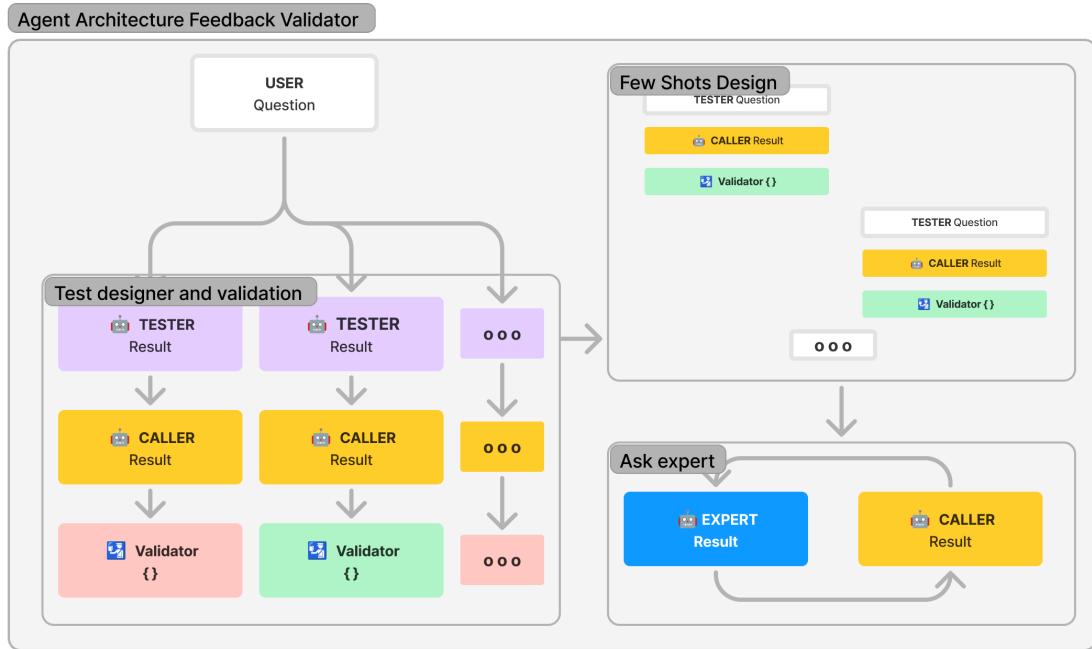


Figure 2.3.2: Structure Rappresentation of the Agent Architecture Feedback Validator

The primary goal of this architecture is to validate the adherence of function calls to parameter requirements and correctness. By iteratively refining responses through feedback, the architecture attempts to overcome the challenges of ensuring accuracy in outputs from quantized models. However, despite its structured approach, limitations such as dependency on model precision and context understanding remain. These challenges are pivotal in driving the exploration of more robust mechanisms (fine tuning, for example, which is not going to be treated in this thesis).

2.3.5 Validator

The primary goal of this component is to ensure that the output adheres to the expected format, parameter types, and functional requirements. This validator operates through a series of well-defined steps:

```

Generating results for agent-network: 56% | 148/252 [47:53<30:29, 16.33s/it]
[QUESTION] Get me two tickets for next Eminem concert in New York City.
[CALLER] Response: [concert_booking.book_ticket(artist='Kendrick Lamar', city='Los Angeles', num_tickets=2)]
[VALIDATOR] Validation NOModel: {'model_output': "[concert_booking.book_ticket(artist='Kendrick Lamar', city='Los Angeles', num_tickets=2)]", 'validation_result': 'Valid: The function call respects the parameter types and requirements.'}
[TESTER] Response: Get me two tickets for the upcoming Kendrick Lamar concert in Los Angeles.
[CALLER] Response: [concert_booking.book_ticket(artist='Kendrick Lamar', city='Los Angeles', num_tickets=2)]
[VALIDATOR] Validation NOModel: {'model_output': "[concert_booking.book_ticket(artist='Kendrick Lamar', city='Los Angeles', num_tickets=2)]", 'validation_result': 'Valid: The function call respects the parameter types and requirements.'}
CONTENT FEEDBACK: The question that you have to reply is: Get me two tickets for next Eminem concert in New York City.

Below you can find the feedback for the previous tests:

==Test 1==
==Tester Question==: Get me two tickets for the upcoming Kendrick Lamar concert in Los Angeles.
==Function Calling Response==: [concert_booking.book_ticket(artist='Kendrick Lamar', city='Los Angeles', num_tickets=2)]
==Validation Feedback Response==: Valid: The function call respects the parameter types and requirements.

==Test 2==
==Tester Question==: Get me two tickets for the upcoming Kendrick Lamar concert in Los Angeles.
==Function Calling Response==: [concert_booking.book_ticket(artist='Kendrick Lamar', city='Los Angeles', num_tickets=2)]
==Validation Feedback Response==: Valid: The function call respects the parameter types and requirements.

[CALLER FEEDBACK] Response: [concert_booking.book_ticket(artist='Eminem', city='New York City', num_tickets=2)]
[VALIDATOR FEEDBACK] Validation NOModel: {'model_output': "[concert_booking.book_ticket(artist='Eminem', city='New York City', num_tickets=2)]", 'validation_result': 'Valid: The function call respects the parameter types and requirements.'}
[VALIDATOR LOOP] Validation NOModel: {'model_output': "[concert_booking.book_ticket(artist='Eminem', city='New York City', num_tickets=2)]", 'validation_result': 'Valid: The function call respects the parameter types and requirements.'}
[TESTER] Response: ==Question==: Can you book a ticket for me to watch the next Coldplay concert in London?
==Function Calling Response==: [concert_booking.book_ticket(artist='Coldplay', city='London', num_tickets=1)]
==Validation Feedback Response==: Valid: The function call respects the parameter types and requirements.
==Content Feedback==: The question that you have to reply is: Can you book a ticket for me to watch the next Coldplay concert in London.

Below you can find the feedback for the previous tests:

==Test 1==
==Tester Question==: Get me two tickets for the upcoming Kendrick Lamar concert in Los Angeles.
==Function Calling Response==: [concert_booking.book_ticket(artist='Kendrick Lamar', city='Los Angeles', num_tickets=2)]
==Validation Feedback Response==: Valid: The function call respects the parameter types and requirements.

==Test 2==
==Tester Question==: Get me two tickets for the upcoming Kendrick Lamar concert in Los Angeles.
==Function Calling Response==: [concert_booking.book_ticket(artist='Kendrick Lamar', city='Los Angeles', num_tickets=2)]
==Validation Feedback Response==: Valid: The function call respects the parameter types and requirements.

[FINAL RESPONSE]: [concert_booking.book_ticket(artist='Eminem', city='New York City', num_tickets=2)]

```

Figure 2.3.3: The exchange of the messages between the models and validator

- Syntax Validation:** The validator first checks whether the model's output is enclosed in square brackets and follows a valid function call format. For example, the regex `r"\[(\w+\.)+)(.*\]\]"` is employed to extract the function name and its parameters.
- Parameter Parsing:** Parameters are parsed into a dictionary format, ensuring the key-value structure is correctly maintained. Empty parameters are handled gracefully to prevent parsing errors.
- Function Definition Matching:** The extracted function name is matched against a list of predefined function definitions. If no match is found, the output is flagged as invalid, with detailed feedback provided.
- Dynamic Model Creation:** Using the Pydantic library, a dynamic validation model is generated based on the function definition. This model enforces parameter type constraints and validates their presence against required fields.
- Type Conversion and Validation:** Parameter values are converted to their expected types, such as integers or strings, using the dynamically created model. Type mismatches or unknown parameters trigger precise validation errors with specific feedback.

Strengths of the Validator:

- **Robustness:** The multi-step validation process ensures that errors in syntax, parameter parsing, or type mismatches are detected and communicated with clear feedback.
- **Dynamic Adaptability:** By leveraging Pydantic's `create_model`, the validator dynamically adapts to varying function definitions, enabling scalable and flexible validation.
- **Detailed Feedback:** Specific error messages guide improvements in model outputs, enhancing the debugging and refinement process.

Areas for Improvement:

- **Extended Type Support:** The current implementation primarily supports integers and strings. Expanding the type conversion logic to include more complex types such as floats, booleans, or nested objects would improve versatility.
- **Error Handling:** While robust, the error messages could be standardized further to improve interpretability and reduce ambiguity for users.
- **Performance Optimization:** The dynamic model creation and type validation process may introduce overhead for large-scale operations. Optimizing these steps could enhance runtime efficiency.

Overall, the validator establishes a strong foundation for enforcing output validity in the framework, offering clear, actionable feedback to improve model-generated function calls. This structure lays the groundwork for future iterations to refine its capabilities and extend its applicability.

2.4 Context Learning Strategy

For the context learning strategy, I developed four distinct system prompts, heuristically crafted based on prompt engineering strategies. These were designed to enhance the performance of the validator feedback mechanism within the agent framework.

In particular, I adopted a few-shot approach (Brown et al., 2020) combined with self-consistency techniques (X. Wang et al., 2023), constructed over test cases generated by the model's test designer. This strategy draws inspiration from the original

methodology outlined in (Huang et al., 2024). The test cases, however, were designed as questions for function calls, which posed challenges due to the model’s capacity to abstract the meaning of the questions. To address these issues, I incorporated meta system prompts (Y. Zhang, Yuan, and A. C.-C. Yao, 2024) for the tester agent, focusing on the structure of the sentences.

Meta prompting differs from few-shot prompting in that it emphasizes a structure-oriented approach rather than a content-driven strategy (*Prompt Engineering Guide* 2024). While few-shot prompting provides examples to guide the model’s output, meta prompting structures the process, ensuring consistency and adaptability across various queries. This approach was applied not only to the tester system prompt but also to the other system prompts within the framework.

Implementation

The context learning workflow begins with a user question processed by the tester and caller agents sequentially. The results are then used to construct feedback content, enabling iterative refinement. The following snippet demonstrates this interaction:

```

1 res_tester = send_and_print(mediator, "user", tester.id,
2                             user_question, "[TESTER]")
3
4 res_caller = send_and_print(mediator, tester.id, caller.id,
5                             res_tester, "[CALLER]")
6
7 content_feedback = f"The Question that you have to reply is: {user_question}\n\nBelow you can find the feedback for the previous tests: \n\n"
8
9 for idx, test in enumerate(previous_tests):
10    content_feedback += f"""
11    **Test {idx+1}:**
12    **Tester Question**: {test['res_tester']}
13    **Function Calling Response**: {test['res_caller']}
14    **Validation Feedback Response**: {test['validator_result']}
15 """

```

Tester System Prompt

The following example illustrates the system prompt designed for the tester agent. The tester's role is to generate structurally similar variations of the input question, ensuring diversity while maintaining the original intent. This aligns with the meta prompting strategy:

```
1 You are an advanced prompt generator that creates variations of
2   QUESTION while maintaining their structure and intent.
3   Follow these steps:
4
5 1. Identify the key components of the input question, such as
6     **variables**, **numbers**, **functions**, or **specific
7     descriptions**.
8
9 2. Replace these key components with **plausible alternative
10    values**, ensuring the overall structure, tone, and intent of
11    the question remain the same.
12
13 3. Ensure no repeated inputs are generated across examples,
14    providing unique variations for each prompt.
15
16 4. DO NOT ANSWER THE QUESTION. Focus on creating one and
17    plausible variation of the input question. DO NOT include any
18    additional text in your response. JUST INCLUDE THE OUTPUT
19    QUESTION, NOT INCLUDE ANY STARTING TEXT.
20
21
22 <|eot_id|>
23 <|start_header_id|>user<|end_header_id|>
24 Given patient with id 546382, retrieve their brain MRI report
25   with the status 'concluded'.
26 <|eot_id|>
27 <|start_header_id|>assistant<|end_header_id|>
28 Given patient with id 762913, retrieve their spinal MRI report
29   with the status 'draft'.
30
31 <|eot_id|>
32 <|start_header_id|>user<|end_header_id|>
33 Help me generate the full SQL creation script with a header for
```

```

    a Firebird database view named 'EmployeeView', using a
    progress monitor 'dbMonitor' and the original source 'SELECT
    * FROM Employee WHERE status = 'active' '?  

18 <|eot_id|>  

19 <|start_header_id|>assistant<|end_header_id|>  

20 Help me generate the full SQL creation script with a header for
    a Firebird database view named 'DepartmentView', using a
    progress monitor 'dbMonitor' and the original source 'SELECT
    * FROM Department WHERE status = 'active' '?  

21  

22 <|eot_id|>  

23 <|start_header_id|>user<|end_header_id|>  

24 Find a Yamaha flute with the specifications of open hole, C
    foot, and silver headjoint available for sale.  

25 <|eot_id|>  

26 <|start_header_id|>assistant<|end_header_id|>  

27 Find a Gemeinhardt flute with the specifications of closed hole
    and B foot available for sale.

```

Key Advantages of the Approach

- **Scalability:** The meta prompting strategy ensures the system can handle diverse queries with consistent structure and intent.
- **Flexibility:** Few-shot examples provide a content-driven context, while meta prompting guarantees structural adaptability.
- **Feedback Integration:** The iterative feedback mechanism facilitates continuous improvement in function call responses.

This hybrid approach combining meta prompting and few-shot examples establishes a robust foundation for refining and validating function calls within the agent framework, laying the groundwork for more advanced strategies in subsequent iterations.

2.5 Evaluation Metrics

The primary evaluation metric for this study is the number of correct responses, determined by comparing the model outputs against a predefined ground truth using the Abstract Syntax Tree (AST) checker from the BFCL benchmark (Fanjia Yan et al., 2024). This metric provides a straightforward and reliable means of assessing performance in different scenarios (as I explained in the previous chapter 1.3.3).

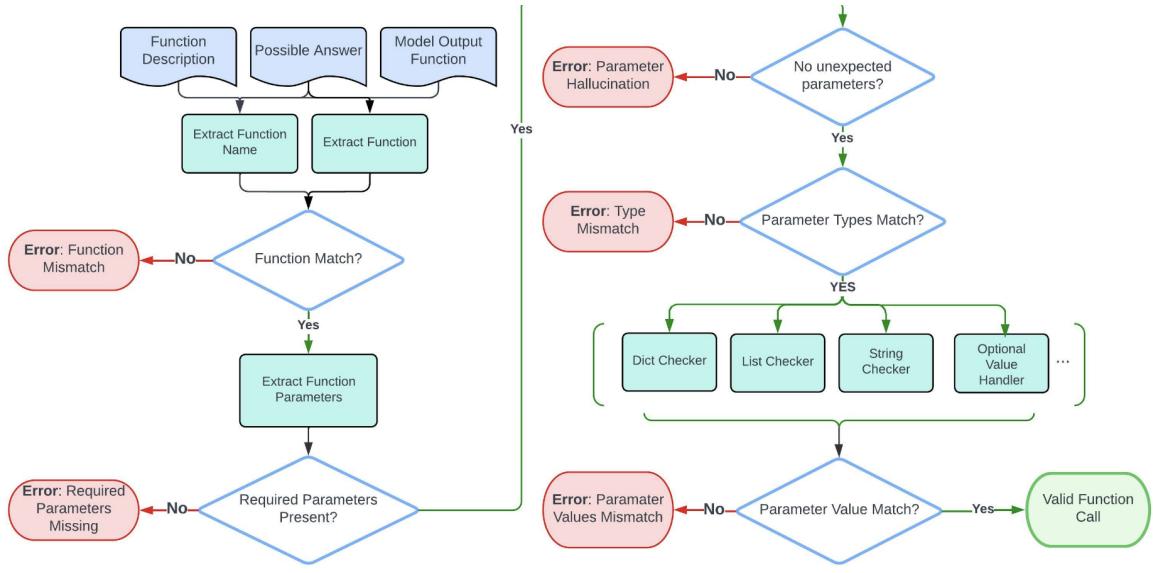


Figure 2.5.1: Evaluating function calls. The function description, and possible answer are used to evaluate the model’s output function. (Fanjia Yan et al., 2024)

The evaluation focuses on AST parameters, which include a diverse set of question categories: 100 Java questions, 50 JavaScript questions, 400 simple questions, 200 multiple questions, 200 parallel questions, and 200 multiple-parallel questions. To provide a comprehensive analysis, the results are compared across three key dimensions: quantized model performance, leaderboard results from the BFCL benchmark, and agent-based results.

The outcomes will be presented as histograms, illustrating the comparative performance across these dimensions for each question category. This visualization highlights strengths and weaknesses of each approach, offering insights into the effectiveness of the agent-based framework relative to traditional methods and quantized models.

Validation structure

- **Function Matching:** Function names are checked for consistency. Dots (.) are replaced with underscores (_) for compatibility during inference and evaluation.
- **Parameter Matching:**
 - Required parameters must exactly match as per function documentation.
 - Extra parameters are flagged as hallucinations.
- **Type and Value Matching:**
 - **Bool:** Direct match required.
 - **Int/Float:** Python allows int for float; other languages are strict.
 - **List/Tuple:** Order matters, with recursive type matching.
 - **String:** Case-insensitive, standardized before evaluation.
 - **Dict:** Keys and values are checked; ordering doesn't matter.
- **Optional Parameters:** Defaults are allowed only for truly optional parameters.
- **Multiple Functions:** Evaluated independently; all outputs must match at least one possible answer.

2.6 Object Calling Case Study

The object calling case study explores the potential of function calling in a more intricate scenario, leveraging a hierarchical agent specialization structure to assign distinct "jobs" to each model. This approach diverges from traditional function calling by focusing on object creation, inspired by the methodologies proposed in the BFCL benchmark (Fanjia Yan et al., 2024) and the theories outlined in (Patil et al., 2024).

This study targets web applications, where the goal is to construct objects rather than invoke functions. The same principles used in function calling are applied to create objects, emphasizing the importance of nesting object structures. The case study highlights the challenges models face in interpreting a request not merely as a series of function calls, but as a structured object composed of multiple keys extracted

from the request. This object-oriented approach aims to integrate seamlessly with web technologies and systems, opening up new avenues for enhancing web-based workflows.

By presenting this direction, the study underscores a shift in perspective—towards enabling models to build autonomous, structured outputs that align with real-world applications. This shift represents a step forward in designing more adaptable and autonomous large language models (LLMs) for practical, integrated environments.

The parallelism with the BFCL scenario is as follows:

- **Simple, Java, JavaScript questions** ⇒ Simple object (no nested object, and no array of objects)
- **Parallel questions** ⇒ Parallel objects (array of objects)
- **Multiple questions** ⇒ Multiple objects (nested objects)
- **Parallel and multiple questions** ⇒ Complete objects

In this discussion, we introduce a system for object calling that leverages two distinct agents: the **Query Objector Agent** (referred to as the *Objector*) and the **Coordinator Objector Agent** (referred to as the *Coordinator*). The Objector is designed to handle specific query-related tasks, such as inserting a user or an order into the database or retrieving information based on specific criteria. In contrast, the Coordinator is a higher-level agent responsible for selecting the appropriate Objector or even delegating the task to another Coordinator, creating a hierarchical structure.

To achieve optimal functionality, models with higher accuracy or more generalized knowledge are ideal candidates for the Coordinator role, while highly specialized, smaller models are better suited as Objectors.

Web Integration Communication Framework

This thesis explores two distinct approaches to building a web integration communication framework between the user and the object calling system. The focus of this section is on the **User Interaction** scenario, which involves direct user engagement with the object calling framework.

As outlined in the paper Patil et al., 2024, we propose two levels of abstraction to conceptualize the communication:

- **Abstraction: User interACTION** (Figure 2.6.1)
- **Abstraction: Autonomous ACTION** (Figure 2.6.2)

2.6.1 User interACTION

In the User interACTION abstraction, the user directly interacts with the system to perform actions such as submitting queries or managing database objects. This abstraction emphasizes user-driven processes, requiring dynamic responses from the object calling system.

Abstraction: User interACTION

```

1 User 1: I need to add a client, Mario Rossi mariorossi97@gmail.com
2 Coordinator Agent: {"client": "true", ...}
3 Objector Agent: [{"name": "Mario", "last": "Rossi", "email": "mariorossi97@gmail.com"}, ...]
4 ----- Query performance / User interaction -----
5 ----- Action -----

```

Figure 2.6.1: Example of User interACTION

2.6.2 Autonomous ACTION

The Autonomous ACTION abstraction focuses on system-driven operations where the Coordinator autonomously selects and delegates tasks to the appropriate Objectors or other Coordinators. This abstraction is particularly suited for scenarios requiring minimal user input and high levels of automation.

Abstraction: Autonomous ACTION

```

1 User 1: I need to add a client, Mario Rossi mariorossi97@gmail.com
2 Coordinator Agent: [{"client": "true", "insert": "true", "delete": "false", "search": "false"}, ...]
3 Objector Agent: [{"name": "Mario", "last": "Rossi", "email": "mariorossi97@gmail.com"}, ...]
4 ----- Action -----

```

Figure 2.6.2: Example of Autonomous ACTION

Description Schema

The strength come with the possibilities to keep the models updated through injection of a function into the attribute of the object or description to add updated information from the database, for example, or to perform mathematical computations (e.g. create a list of numbers to add).

```

1 User 1: Add and event next week at 10:00 monday
2 Coordinator Agent: {"event":"true", "client":"false",...}
3 --> Description injection: {"description_begin": "extract a
   date knowing that today is {datetime.now()}"}
4 Objector Agent: [{"begin":"29/07/2024 10:00", ...},...]
5 ----- Query performance / User interaction -----
6 ----- Action -----

```

Web Services Strategy

The associated data is stored on a remote server and transmitted to client machines through APIs, which facilitate web services for third-party users. These APIs can adopt different architectures, such as SOAP or REST, serving as guidelines for transferring data from the server to the client. REST APIs adhere to the constraints of the REST architectural style, commonly utilizing the JSON format.

- A client-server architecture consisting of clients, servers, and resources, where requests are managed through HTTP (see Fig. 2.6.3).
- Stateless client-server communication, meaning no client information is stored between requests, and each request is treated independently.
- Cacheable data to improve performance and scalability.
- Dual-server communication setup, with one server implemented in Python and the other in JavaScript.
- A dynamic and intuitive front-end to ensure user engagement with the system and maintain awareness of its internal communications. Latency must remain below one minute to provide a seamless experience.
- Implementation of a recursive Svelte JavaScript component to manage the Coordinator-Objector recursive structure (e.g., `recursiveTable`).

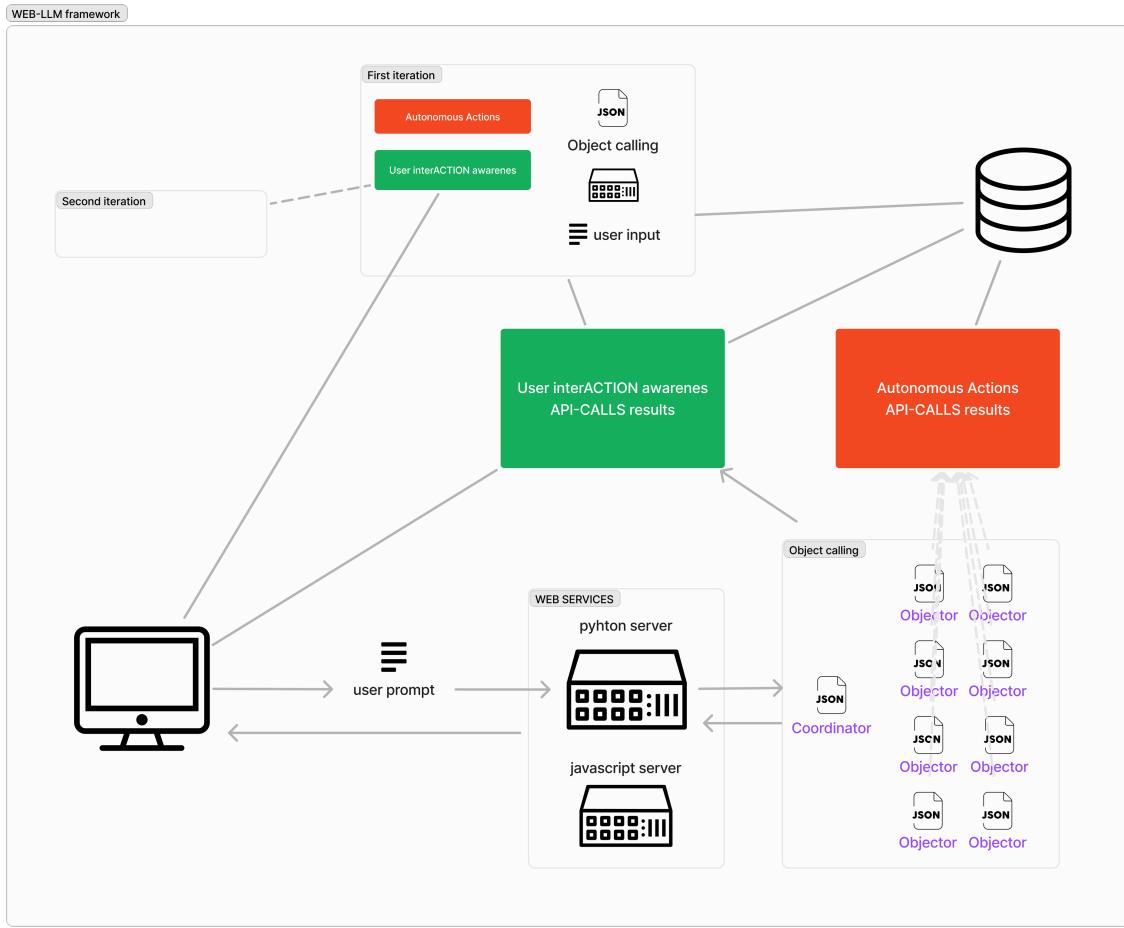


Figure 2.6.3: Web LLM Framework

Object calling framework

So the underline idea is to perform data extraction from the input users, with a fixed schema. Leveraging the best capacity of the models in the constraint of the WEB SERVICE structure that is built up on the consistency of the keys/attributes of the schema. In order to do that we need a evaluation framework.

Evaluation Metrics

Five distinct levels have been developed to define the schema structure required for setting the objector tasks:

- **LVL1:** Linear object creation
- **LVL2:** List of objects inside an object
- **LVL3:** List of objects with an object list

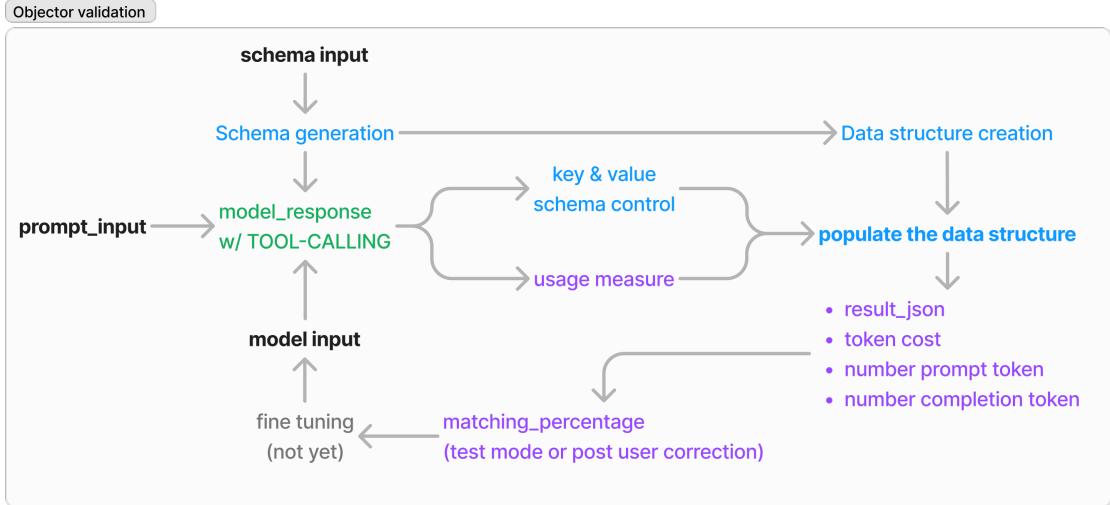


Figure 2.6.4: Object calling Framework

- **LVL4:** At least three object lists inside at least two object lists
- **LVL5:** At least three object lists inside at least three object lists of at least five object lists

Each model must meet the constraints of its respective level with an accuracy exceeding 80% to be promoted for handling the corresponding schemas.

A dataset was created with three examples for each level, and the differences between the generated results and the ground truth were measured using the Levenshtein distance (Haldar and Mukhopadhyay, 2011). The mean score over seven iterations was calculated. Below is an example of the recorded data:

```

1 id,model,level,matching_percentage,time_elapsed,
   completion_token,prompt_token,cost
2 1721663922_1,gpt
   -4,1,99.19786096256684,5.285665988922119,74,158,0.00918
3 1721664921_1,gpt-4o-mini
   ,1,52.16178521617852,1.6742339134216309,81,242,0.000157
4 1721741472_1,gorilla-openfunctions-v2
   ,1,97.25848563968668,10.007141828536987,96,731,0.0
5 ...

```

The model selected are:

- gemini-1.0-pro

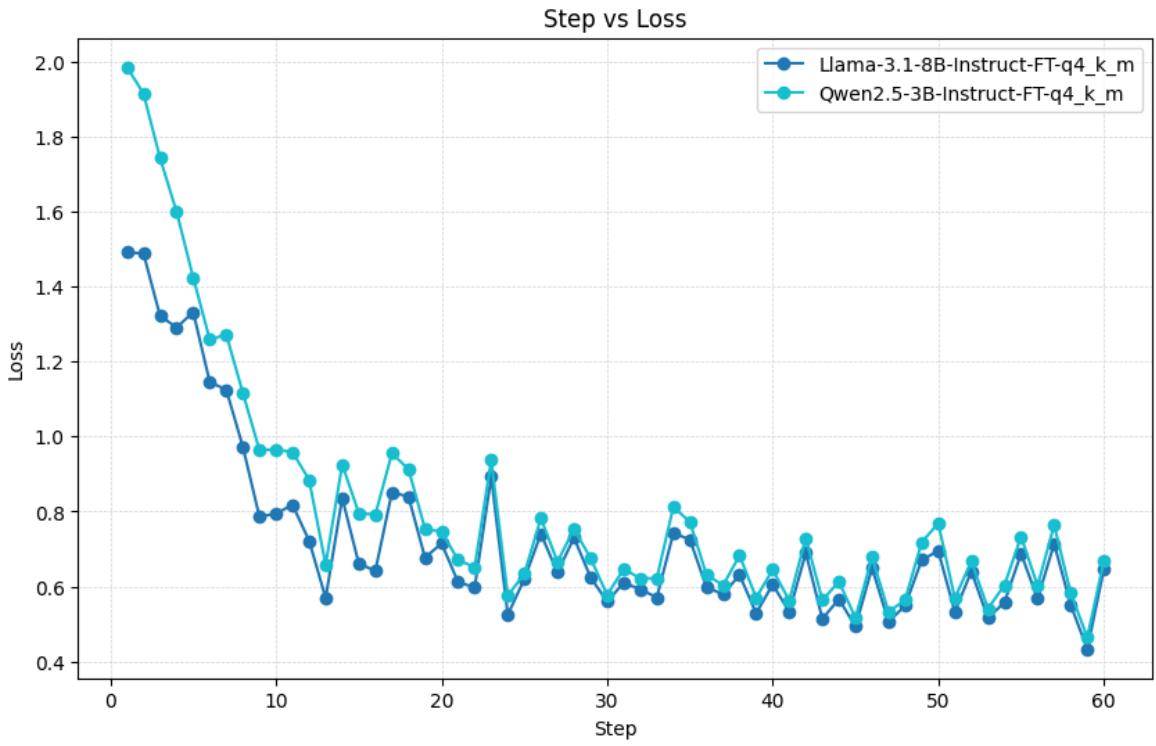


Figure 2.6.5: Loss functions for the two models over 60 steps

- gemini-1.5-flash-latest
- gemini-1.5-pro-latest
- gorilla-openfunctions-v2
- gpt-3.5-turbo-0125
- gpt-4
- gpt-4o
- gpt-4o-mini
- Llama-3.1-8B-Instruct-FT-q4_k_m: Fine tuned over the Alpaca dataset (*Saxo/alpaca_function_calling_dataset · Datasets at Hugging Face 2024*) for function calling and q4_k_m quantized
- Qwen2.5-3B-Instruct-FT-q4_k_m: Fine tuned over the Alpaca dataset (*Saxo/alpaca_function_calling_dataset · Datasets at Hugging Face 2024*) for function calling and q4_k_m quantized

The fine-tuning process (show in the Fig:2.6.5) utilized the `unsloth` library due to its ease of managing the stop token in Alpaca format, which is compatible with both the LLaMA and Qwen handlers. This approach simplified the process. For further optimization, a better strategy can be developed by leveraging the stop tokens specific to the two handlers.

2.7 Hardware Support

To conduct the testing of the five selected models Qwen2.5-1.5B-Instruct, Qwen2.5-3B-Instruc, Qwen2.5-7B-Instruc, LLama-3.2-3B-Instruc and LLama-3.1-8B-Instruc. I utilized two hardware setups to optimize performance and ensure scalability for different model sizes. These setups were chosen to address the computational demands of models ranging from 1.5 billion to 8 billion parameters.

2.7.1 GeForce RTX 3060 Setup

Initial testing was performed on a GeForce RTX 3060 GPU, which features the following specifications:

- **CUDA Cores:** 3584
- **VRAM:** 12 GB GDDR6
- **Memory Bandwidth:** 360 GB/s
- **Base/Boost Clock Speeds:** 1320 MHz / 1777 MHz
- **Architecture:** Ampere
- **Power Consumption:** 170 W

The RTX 3060's ample 12 GB of VRAM provided sufficient capacity to load smaller models, such as the 1.5B and 3B parameter variants, allowing for single-GPU inference. However, the higher memory and compute demands of the 7B and 8B models necessitated more efficient scaling or reduced batch sizes, which slightly impacted the throughput during testing. While effective for initial experiments, the RTX 3060's limited compute power presented a bottleneck for larger models in high-throughput scenarios.

2.7.2 Nvidia T4 Setup

To accelerate testing and streamline model evaluations, I uploaded the quantized models to the Hugging Face repository and deployed them via dedicated endpoints. For inference, I utilized Nvidia T4 GPUs, which offered the following specifications:

- **CUDA Cores:** 2560
- **VRAM:** 16 GB GDDR6
- **Memory Bandwidth:** 320 GB/s
- **Base/Boost Clock Speeds:** 585 MHz / 1590 MHz
- **Architecture:** Turing
- **Power Consumption:** 70 W

Compared to the RTX 3060, the T4 GPU features a slightly lower number of CUDA cores but compensates with 16 GB of VRAM, making it better suited for handling larger models like the 7B and 8B parameter variants. The T4's reduced power consumption (70 W compared to the 3060's 170 W) and its optimization for data center deployments made it ideal for running high-throughput inference workloads via Hugging Face endpoints. These endpoints enabled seamless scaling, facilitating faster validation cycles and improved resource utilization.

2.7.3 Integration with BFCL Benchmark

Both setups were fully integrated with the BFCL benchmark functionality to extend its capabilities for testing quantized models and evaluating Hugging Face inference endpoints. This integration allowed the AST-checker validator of the BFCL to function seamlessly with the deployed models, ensuring consistent and reliable results across various model sizes and configurations. By leveraging both local and cloud-based solutions, the testing pipeline achieved a balance between flexibility and performance, enabling comprehensive evaluation of the proposed methodologies.

Chapter 3

Results and Analysis

This chapter presents the outcomes of the study, highlighting the performance of the selected quantized large language models (LLMs) on various tasks, their integration into an agent-augmented framework, and the effectiveness of the proposed methodologies. The analysis aims to address the central objectives of the research: to evaluate the feasibility of quantized models in complex scenarios, to demonstrate the capabilities of the agent-based framework, and to explore the role of context learning in enhancing model performance.

The chapter is structured as follows:

- **Section 3.1: Problem Analysis of the quantize models on BFCL Benchmark** - This section explores the most common errors encountered in the selected quantized models and outlines the strategies employed within the validator agent protocol to address them effectively.
- **Section 3.2: Agent-Augmented Qwen and LLama on the BFCL Benchmark** - This section evaluates the performance of Agent-Augmented Qwen and LLama on the BFCL Benchmark. The results are analyzed across various task categories, including simple, multiple, parallel, and multiple-parallel questions. Comparisons are made between the baseline models and their agent-augmented counterparts to assess the contributions of the framework.
- **Section 3.3: Object Calling Case Study Results** - This section discusses the findings from the object calling case study, showcasing the ability of the agent-augmented architecture to facilitate automatic object creation. The anal-

ysis includes accuracy and efficiency metrics, demonstrating the challenges and potential of hierarchical object-oriented requests.

The results provide a comprehensive view of the strengths and limitations of the proposed framework and methodologies, laying the groundwork for future research in scalable and adaptive large language model architectures.

3.1 Problem Analysis of the quantize models on BFCL Benchmark

To implement the agent-augmented structure, the weight matrices of the LLaMA and Qwen models were quantized from FP32 to int4, specifically using the `q4_k_m` format as described in Section 1.2, leveraging the `Unsloth` library (*Saving to GGUF / Unsloth Documentation* 2024). Subsequently, the BFCL-AST-Test was conducted on the data composition described in Figure 1.3.1, comprising a total of 1,150 questions.

The results for each model are presented in Fig. 3.1.1. Notably, the leaderboard result for the Qwen2.5-3B-Instruct model is unavailable, as it is not included in the BFCL benchmark. A detailed discussion of these results is provided in the conclusions. It is observed that quantization impacts distinct categories differently across models, with no clear overarching trend emerging from the analysis.

Table 3.1 highlights the number of mistakes made by each quantized model, categorized according to the dataset composition described in Section 1.3.3. For clarity, the table emphasizes the highest error per test category in **bold**.

For simplicity in presenting the results, the "Instruct" label is omitted, though the models remain identical across tests.

Table 3.1: Mistakes count for each model

File	Linguaggio	Qwen2.5-1.5B	Qwen2.5-3B	Qwen2.5-7B	Llama-3.2-3B	Llama-3.1-8B
Simple	Python	47	70	24	34	38
Multiple	Python	28	41	11	24	17
Parallel	Python	53	80	23	59	26
Parallel Multiple	Python	58	84	36	71	36
Java	Java	44	42	36	40	42
Javascript	JavaScript	16	19	14	15	16
Totale		246	336	144	243	175

This analysis is further expanded in the following section to address issues related

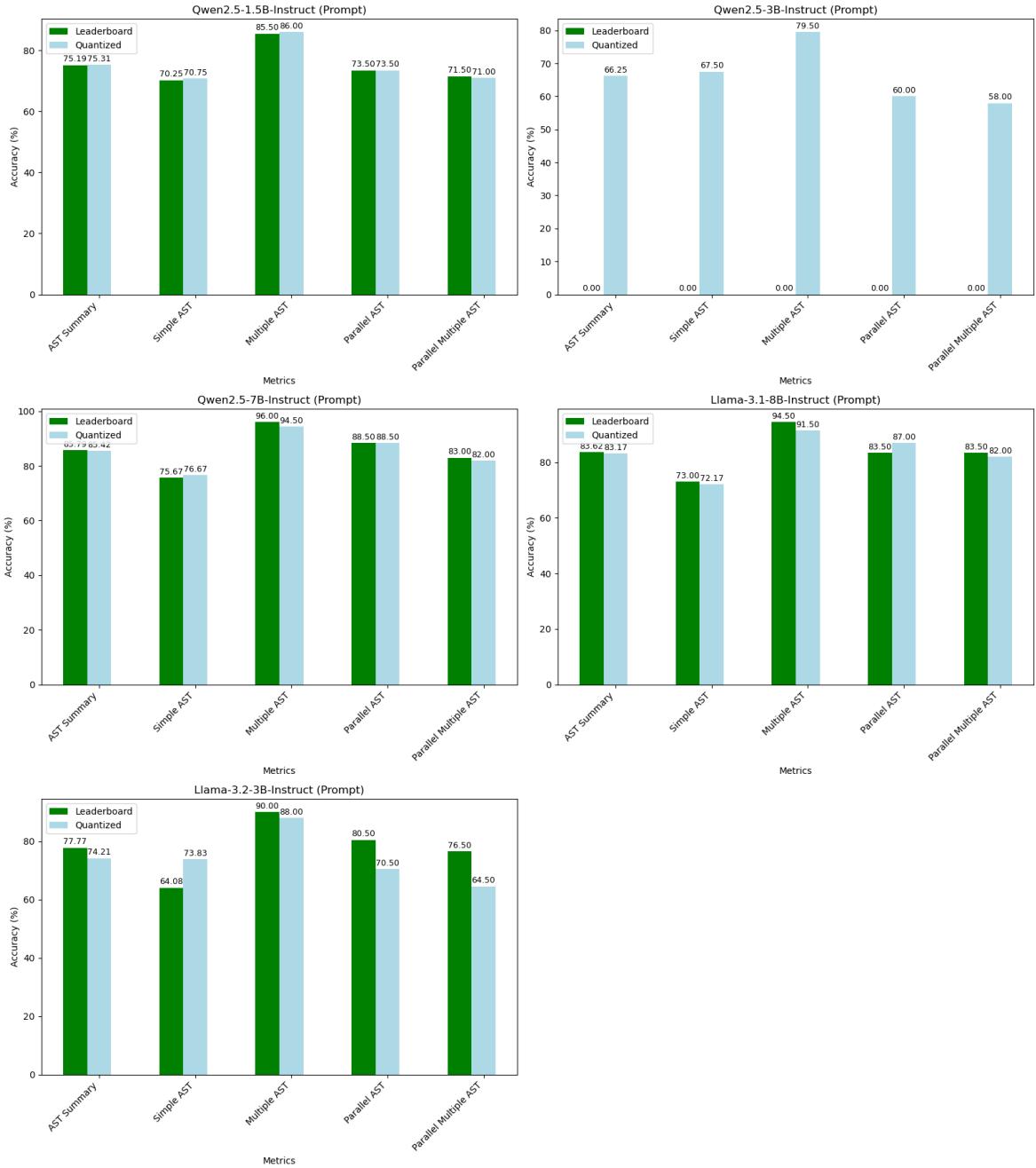


Figure 3.1.1: Performance comparison of quantized models versus leaderboard benchmarks.

to the validation loop in the Agent Architecture Feedback Validator (Section 2.3.4).

Building the Validator

In this section, I analyze all errors generated by the AST-checker during the BFCL benchmark evaluation, which is based on the considerations detailed in Section 2.5. The analysis involved inspecting the error strings from the AST-checker to identify substrings that correspond to validation categories. As a result, I identified 15 specific error categories to address the validation issues:

- The value provided for the parameter is invalid because it is not one of the expected values.
- The function provided is not found in the model output.
- A required parameter is missing in the provided function.
- The parameter provided is marked as optional and is not in the question.
- The parameter provided is not expected in the function.
- The parameter provided is expected to be an array but is a string.
- The parameter provided is expected to be a string but is an integer.
- The parameter provided is expected to be a float but is a string.
- The parameter provided is expected to be a dictionary but is a string.
- The parameter provided is expected to be an ArrayList but is a string.
- The parameter provided is not closed with a] character.
- Nested type checking failed for the parameter. Expected outer type array with inner type float.
- Invalid syntax: Failed to decode AST - Error parsing Java source code.
- Invalid syntax: Failed to decode AST - Error parsing JavaScript source code.
- The number of functions is not correct.

- Other errors.

The error categories were identified and analyzed using the code provided in the file `v002-validator-analysis.ipynb`. As an example, I present the distribution of mistakes per error category for the model Qwen2.5-1.5B in Table 3.1.

Error category	Counts	Percentage
The value provided for the parameter is invalid because it is not one of the expected values.	71	28.86%
The function provided is not found in the model output.	59	23.98%
In the function provided, a required parameter is missing.	6	2.44%
The parameter provided is marked as optional and is not in the question.	3	1.22%
The parameter provided is not expected in the function.	3	1.22%
The parameter provided is expected to be an array but is a string.	7	2.85%
The parameter provided is expected to be a string but is an integer.	0	0.00%
The parameter provided is expected to be a float but is a string.	0	0.00%
The parameter provided is expected to be a dictionary but is a string.	0	0.00%
The parameter provided is expected to be an ArrayList but is a string.	4	1.63%
The parameter provided is not closed with a ']' character.	0	0.00%
Nested type checking failed for parameter. Expected outer type array with inner type float.	5	2.03%
Invalid syntax. Failed to decode AST. - Error parsing java the source code	9	3.66%
Invalid syntax. Failed to decode AST. - Error js parsing the source code.	2	0.81%
The number of functions is not correct	35	14.23%
Other Errors	42	17.07%

Table 3.2: Distribution of Error categories for Model Qwen2.5-1.5B

To facilitate better comparison of results across different models, I assigned a concise 1-4 word key to each error typology, as shown in Table 3.1. These error categories were selected based on the interpretability of the AST-checker responses. However, a more refined approach to achieve better results involves labeling all errors systematically and creating illustrative examples of solutions. This strategy enables the model to learn effectively from context (Kossen, Gal, and Rainforth, 2024).

Furthermore, this method can be leveraged to structure agent communication for managing future complexities, even as model accuracy improves.

Using the key error categories, Table 3.1 summarizes the results obtained from the five models. The percentages are calculated relative to the total number of mistakes for each model, which explains why the same count can correspond to different percentages across models. For clarity, the model most influenced by a specific mistake is highlighted in **bold**.

Using these insights, I developed the validator, located in the file `utils.py` within the `berkeley-function-call-leaderboard` folder, which is the main directory of the GitHub repository for the BFCL benchmark. This benchmark was modified to ensure compatibility with the agent-augmented framework.

Key	Error category
InvalidValue	The value provided for the parameter is invalid because it is not one of the expected values.
NoFunction	The function provided is not found in the model output.
MissingParam	In the function provided, a required parameter is missing.
OptionalParam	The parameter provided is marked as optional and is not in the question.
UnexpectedParam	The parameter provided is not expected in the function.
ArrayToStr	The parameter provided is expected to be an array but is a string.
StrToInt	The parameter provided is expected to be a string but is an integer.
FloatToStr	The parameter provided is expected to be a float but is a string.
DictToStr	The parameter provided is expected to be a dictionary but is a string.
ArrayListToStr	The parameter provided is expected to be an ArrayList but is a string.
UnclosedBracket	The parameter provided is not closed with a ']' character.
NestedTypeFail	Nested type checking failed for parameter. Expected outer type array with inner type float.
JavaSyntaxErr	Invalid syntax. Failed to decode AST. - Error parsing java the source code.
JsSyntaxErr	Invalid syntax. Failed to decode AST. - Error js parsing the source code.
FuncCountErr	The number of functions is not correct.
OtherErrors	Other Errors.

Table 3.3: Mapping of Keys to Error categories

Key	Qwen2.5-1.5B	Qwen2.5-3B	Qwen2.5-7B	Llama-3.2-3B	Llama-3.1-8B
InvalidValue	71 (28.86%)	53 (15.77%)	47 (32.64%)	46 (18.93%)	52 (29.71%)
NoFunction	59 (23.98%)	78 (23.21%)	42 (29.17%)	43 (17.70%)	41 (23.43%)
MissingParam	6 (2.44%)	9 (2.68%)	1 (0.69%)	4 (1.65%)	1 (0.57%)
OptionalParam	3 (1.22%)	12 (3.57%)	1 (0.69%)	6 (2.47%)	0 (0.00%)
UnexpectedParam	3 (1.22%)	3 (0.89%)	2 (1.39%)	2 (0.82%)	1 (0.57%)
ArrayToStr	7 (2.85%)	16 (4.76%)	1 (0.69%)	4 (1.65%)	2 (1.14%)
StrToInt	0 (0.00%)	1 (0.30%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
FloatToStr	0 (0.00%)	1 (0.30%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
DictToStr	0 (0.00%)	1 (0.30%)	1 (0.69%)	1 (0.41%)	0 (0.00%)
ArrayListToStr	4 (1.63%)	3 (0.89%)	5 (3.47%)	2 (0.82%)	3 (1.71%)
UnclosedBracket	0 (0.00%)	3 (0.89%)	0 (0.00%)	0 (0.00%)	0 (0.00%)
NestedTypeFail	5 (2.03%)	2 (0.60%)	4 (2.78%)	3 (1.23%)	4 (2.29%)
JavaSyntaxErr	9 (3.66%)	11 (3.27%)	1 (0.69%)	11 (4.53%)	18 (10.29%)
JsSyntaxErr	2 (0.81%)	4 (1.19%)	0 (0.00%)	4 (1.65%)	1 (0.57%)
FuncCountErr	35 (14.23%)	29 (8.63%)	15 (10.42%)	26 (10.70%)	27 (15.43%)
OtherErrors	42 (17.07%)	110 (32.74%)	24 (16.67%)	91 (37.45%)	25 (14.29%)

Table 3.4: Counts and Percentages of the Error Categories over the Models

The following observations can be made:

- **InvalidValue:** This category represents a lower threshold since it is challenging to assess the ground truth for potential answers. The model's fine-tuning capacity is a key factor in reducing this percentage. In Section 3.3, I propose an original approach to address this issue. Notably, **LLaMA-3.2-3B-Instruct** outperforms others in this category.
- **NoFunction:** This category is particularly difficult to tackle, as models struggle to identify function names with extensive use of camel case and snake case styles. These naming conventions make the models prone to errors. As noted, "Chat

models tend to generate malformed function calls in which parameters can be extracted but are not executable" (Fanjia Yan et al., 2024).

I then proceeded to perform the analysis using the Agent Architecture Feedback Validator, discussed in Section 2.3.4 of the previous chapter.

3.2 Agent-Augmented Qwen and LLama on the BFCL Benchmark

This section provides a comprehensive comparison of the base models and their agent-augmented counterparts on the BFCL benchmark. The analysis evaluates performance metrics such as accuracy and task success rate to illustrate the impact of agent integration on model capabilities. Additionally, histograms are utilized to visualize the results, highlighting trends and variations across task categories, including simple, multiple, parallel, and multiple-parallel questions.

3.2.1 Experimental Setup

The experiments were conducted using the following constants, defined in the `v002_generator_bfcl` script:

- **Agent Network:** Two configurations were tested:
 - `agent-network-gpt`: This configuration uses the OpenAI handler for message formatting.
 - `agent-network`: This configuration alternates between the Qwen Handler and the LLama Handler. The results presented in this section are based on the `agent-network` setup.
- **Hugging Face Endpoint URL:** This parameter selects the GPU for inference. For the reported results, the NVIDIA T4 GPU setup was used via the Hugging Face endpoint.
- **Temperature:** The temperature parameter was set to 0.1 for deterministic responses during model evaluation. However, for the Agent Test Designer, a higher temperature of 0.9 was used to enable more creative scenarios.

- **MAX_TURNS**: This constant defines the maximum number of turns between the expert and the caller in the Agent Architecture Feedback Validator (see Figure 2.3.2). If no satisfactory solution is found within 4 turns, the process terminates.
- **N_TESTS**: This parameter specifies the number of tests created by the designer for the Agent Caller, which was set to 2.
- **With Agent**: A boolean variable `with_agent=True` indicates whether the agent architecture is implemented or the model is used directly.

To generate test results, the following commands were executed:

```
1 bfcl generate --model agent-network --test-category {simple,
parallel,multiple,parallel_multiple,java,javascript} --num-
threads 1
```

This command creates a folder named `result`, containing subfolders with the `agent-network` results. To compute the evaluation metrics, the following command was used:

```
1 bfcl evaluate
```

3.2.2 Results and Analysis

The results of the experiments are visualized in Figure 3.2.1. These histograms illustrate the performance differences across task categories, highlighting the impact of agent augmentation on accuracy and task success rates.

To reproduce these results, refer to the Appendix (Section A.1) for the relevant code and configuration files.

3.3 Object Calling Case Study Results

This section focuses on the outcomes of the object calling case study, which extends the function-calling paradigm to automatic object creation. The study evaluates the ability of the agent-augmented architecture to handle complex requests, producing structured,

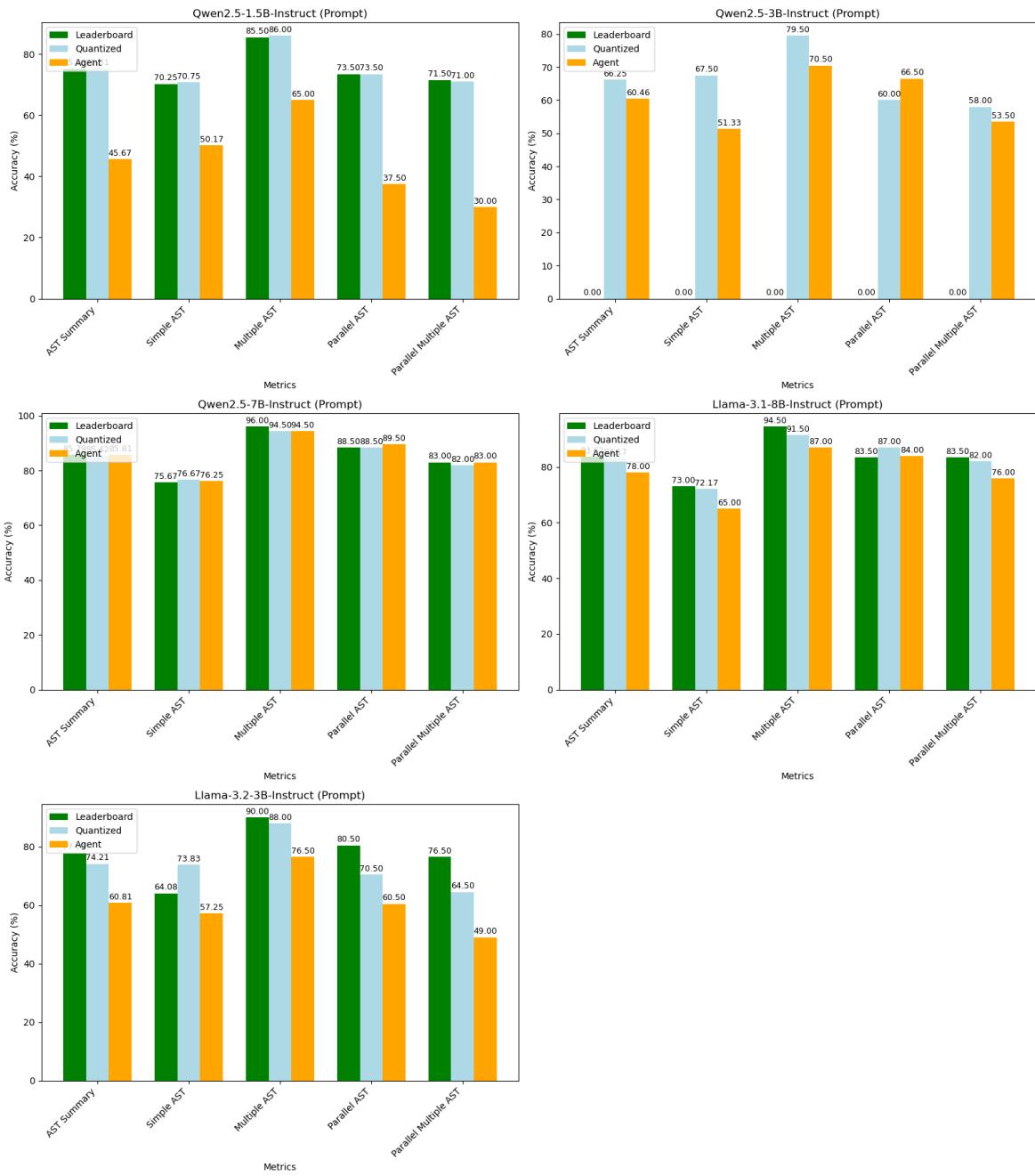


Figure 3.2.1: Result of Agent-Augmented Qwen and LLama on the BFCL Benchmark

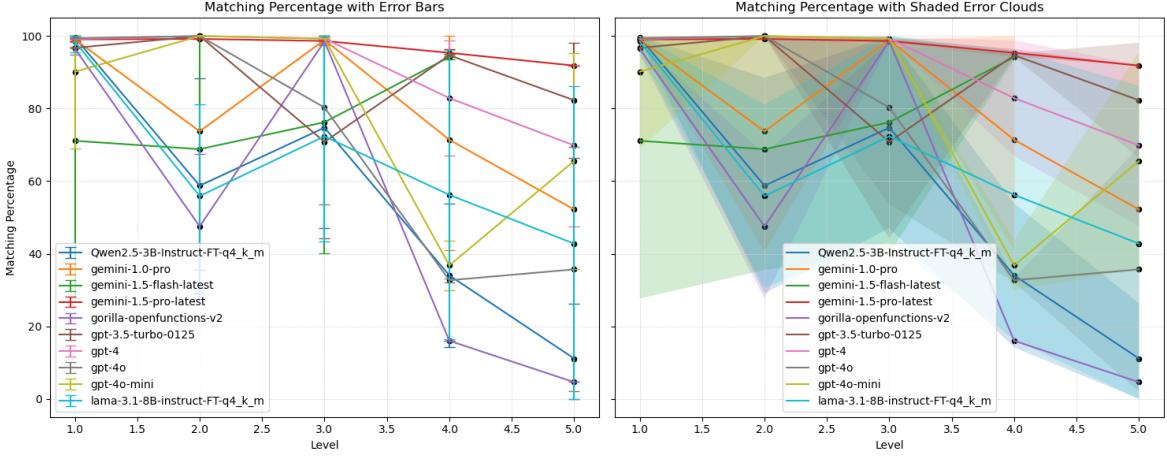


Figure 3.3.1: Matching Percentage using the Levenshtein distance (Haldar and Mukhopadhyay, 2011)

hierarchical objects seamlessly integrated into web applications. Key performance metrics, such as accuracy and efficiency, are analyzed to assess the effectiveness of the proposed approach. Challenges encountered during the process, such as nested object creation and model limitations, are also discussed, providing insights into the potential of object-oriented LLM outputs.

The results of the object-calling experiment are presented in the graph 3.3.1 and the time-lapsed result in the graph 3.3.2. This investigation utilizes a minimal dataset, comprising 10 carefully constructed questions, to evaluate the preliminary accuracy of the models in analyzing and interpreting the structural properties of JSON objects.

The compact dataset amplifies the sensitivity to structural nuances, making it an effective tool for identifying weaknesses and inconsistencies in model performance. This focused exploration allows for a more granular evaluation of the models' capacity to generalize and interpret complex JSON structures, providing valuable insights for further refinement and scalability testing.

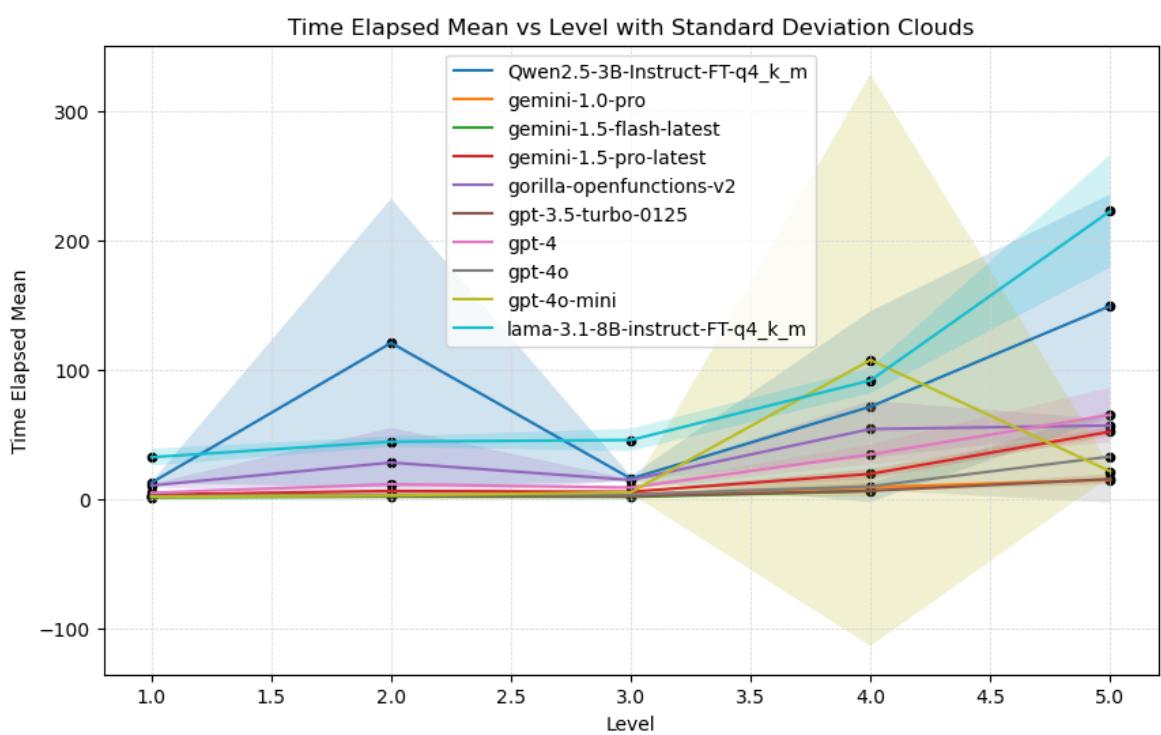


Figure 3.3.2: Time Elapsed of the models to generate the response

Chapter 4

Conclusion

4.1 Discussion

4.1.1 Quantization on BFCL benchmark

The results obtained from the evaluation of quantized models, as shown in Figure 4.1.1, indicate that the best performing model is **Qwen2.5-7B-Instruct**, followed closely by **Llama3.1-8B-Instruct**.

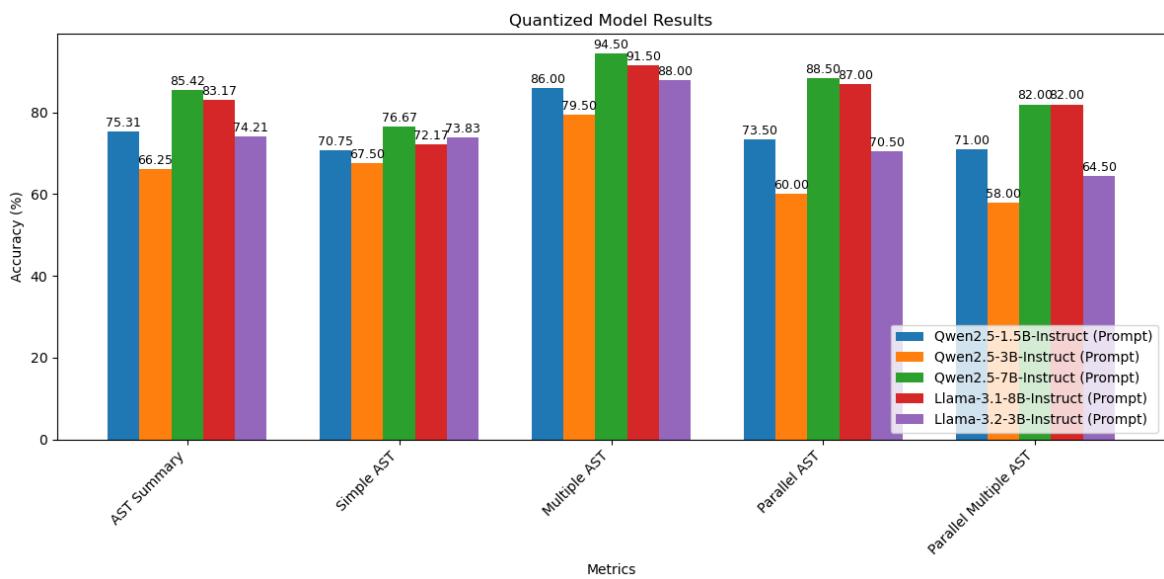


Figure 4.1.1: Result comparison between Quantized models on the BFCL Benchmark

- **Qwen2.5-1.5B-Instruct:** Both *Leaderboard* and *Quantized* show reasonable values, but *Leaderboard* outperforms *Quantized* slightly in all metrics.

- **Qwen2.5-3B-Instruct:** *Leaderboard* not values available.
- **Qwen2.5-7B-Instruct:** A significant disparity is evident between *Leaderboard* and *Quantized*, though some metrics (e.g., *AST Summary*) show smaller gaps.
- **Llama-3.1-8B-Instruct:** Strong *Leaderboard* performance across metrics, with *Quantized* slightly lagging but present for all metrics.
- **Llama-3.2-3B-Instruct:** The results are relatively closer between *Leaderboard* and *Quantized*, showing better alignment than other models.

Quantization negatively impacts performance in most models, with the extent of the impact varying depending on the model and the specific metric. However, certain metrics and models, such as *Llama-3.2-3B-Instruct*, demonstrate a closer performance between the *Leaderboard* and *Quantized* versions, suggesting the possibility for further optimizations.

In certain tests, such as the *Parallel AST* category, the quantized version of Llama-3.1-8B-Instruct demonstrates notably better performance, achieving 87% compared to the leaderboard result of 83.50%.

4.1.2 Agent-Augmented Qwen and LLama on the BFCL Benchmark

From the results, better performance can be observed (Fig. 4.1.2) with the Qwen2.5-7B-Instruct model. It achieves a slight improvement, with a percentage score of 85.81% compared to the quantized performance of 85.42% and a leaderboard score of 85.79%. This supports the direction of this thesis to pursue advancements in larger models while also maintaining a focus on improving smaller ones.

As shown in the graph (Fig. 4.1.3), the performance trends align with the previously mentioned results. The best performance is achieved by Llama-3.1-8B-Instruct and Qwen2.5-7B-Instruct, followed closely by Llama-3.2-3B-Instruct and Qwen2.5-3B-Instruct, which score 60.81% and 60.46%, respectively. The lowest performance is observed in the *AST Summary* category, with a score of 45.67%.

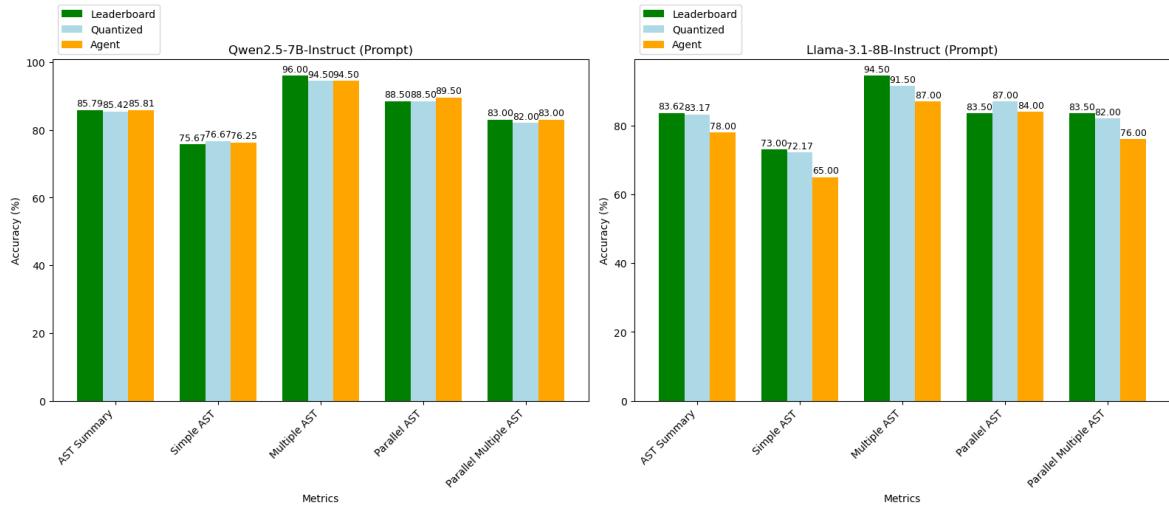


Figure 4.1.2: Focusing on the Bigger Models

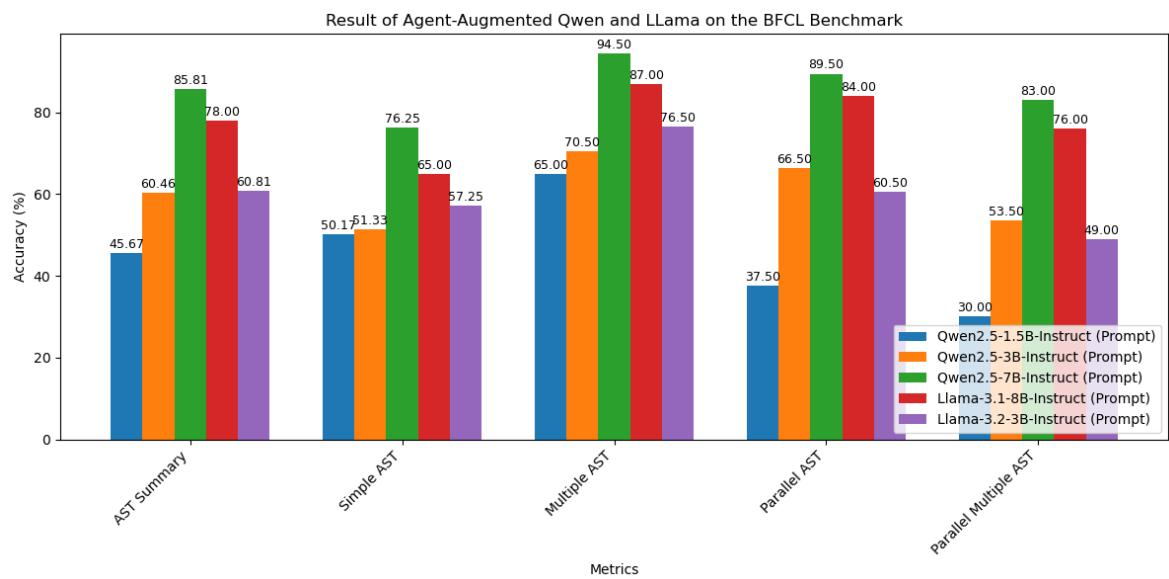


Figure 4.1.3: Result comparison between Agent-Augmented Qwen and LLama on the BFCL Benchmark

4.1.3 Object Calling Case Study

The primary limitation encountered in this case study arises from the lack of available data for evaluating object calling scenarios. This research direction remains largely unexplored in existing literature, primarily due to the rapid evolution of large language models (LLMs) and the multitude of emerging application domains. The aim of this study is to propose a novel approach to assess and improve the performance of LLMs in scenarios involving nested structures, which simulate complex query performance.

To address this, a strategy was developed to enable the model to generate nested structures, thereby creating a testing environment that simulates increasingly complex queries. The results indicate that as the complexity level rises, the performance of the tested models shows a consistent decline. Specifically, the following trends were observed across all models:

- A consistent decrease in performance metrics with increasing levels of complexity.
- A proportional increase in time elapsed as the complexity level rises.

These findings highlight the current limitations of LLMs in handling more intricate scenarios, emphasizing the need for further advancements.

Future improvements should focus on the following:

- Expanding the dataset to include a broader range of examples that reflect the targeted use cases.
- Applying domain-specific fine-tuning to prepare the models for these novel and challenging scenarios.

This study represents a step toward advancing the field from function calling to object calling paradigms, opening new possibilities for evaluating and enhancing LLMs in scenarios where performance currently degrades.

4.1.4 Challenges & Improvements

- Overcoming strong prior biases and ensuring the model gives equal weight to all contextual information remain challenges. (Kossen, Gal, and Rainforth, 2024)

- Testing with other less popular benchmark can help mitigate bias (M. Li et al., 2023)
- Larger models make increasingly efficient use of in-context information (Fig. 1.2 (Brown et al., 2020)). But smaller problems have smaller place for improvements.
- The tester for smaller models (3B, 1.5B) tend to reply to the questions instead of respecting the system prompt a possible solution can be in improving the few shot examples, insert a validator that if the first test goes wrong it stops and exit from the testing generation loop

4.2 Implications and Future Work

This thesis approaches the frontier of the agent-augmented framework, wherein Large Language Models (LLMs) are utilized in significantly smaller configurations compared to their largest representations (e.g., the largest LLaMA model, with 405 billion parameters). Recent advancements demonstrated by the Qwen2.5-7B-instruct model highlight the potential for optimizing performance even at smaller scales, thereby presenting promising opportunities in this direction. Furthermore, quantized models have proven to be both effective and manageable, even in contexts requiring In-Context-Learning (Q. Dong et al., 2024). These findings pave the way for future research, which could explore several promising directions, including the optimization of agent-augmented systems, enhanced quantization techniques, and improved contextual learning methodologies.

- Investigating different types of quantized models to enhance performance and efficiency.
- Developing grid optimization techniques to identify the best initial parameters for the proposed Agent-Augmented Framework.
- Applying Automatic Prompt Engineering (APE) (Zhou et al., 2023) to refine the system prompts of the agent caller.
- Expanding the dataset for object-calling tasks to create benchmarks that assess model capabilities across novel and challenging scenarios.

- Testing models of varying sizes and architectures, leveraging fine-tuning to optimize performance based on specific use cases.
- Extending the validator function to improve robustness and adaptability for diverse tasks.
- Broadening the framework to include all categories defined in the BFCL benchmark, such as hallucinations, irrelevance, multi-turn interactions, external API integration, and live scenarios.

The overarching aim of this thesis is to demonstrate that seamless integration of LLMs with the web is achievable by leveraging advancements in function calling (commonly referred to as "tools"). Since the web is a cornerstone of modern society, this work lays a foundation for future researchers and developers to handle these models effectively while maintaining cost efficiency (see Appendix A.2). By focusing on open models, cost reduction, and enhanced connectivity, this thesis opens new avenues for practical and impactful applications of LLMs.

All references to the code and implementations used in this thesis are available in Appendix A.1.

Ringraziamenti

A conclusione di questo elaborato, desidero menzionare tutte le persone, senza le quali questo lavoro di tesi non esisterebbe nemmeno.

Appendix A

A.1 Code reference & Github

The GitHub repositories used in this study are as follows:

- **Quantized LLMs for Function Calling:** Evaluating Agent-Augmented Qwen and LLama on the BFCL Non-Live AST Benchmark. Repository link: <https://github.com/gianmarcoalessio/thesis>
- **Object Calling Case Study:** Repository link: <https://github.com/gianmarcoalessio/croswil>

A.2 Costs

For the BFCL benchmark and object-calling performance evaluation, the total costs were calculated using the Hugging Face endpoint on an NVIDIA T4 instance, priced at €0.50 per hour (for hardware specifications, see Section 2.7). The cost analysis is summarized as follows:

Table A.1: Cost and Request Analysis of Different Models

Model	Cost (€)	Requests	Latency
Qwen2.5-1.5B-Instruct	7.98	11,768	Not available
Qwen2.5-3B-Instruct	3.56	10,097	Not available
Qwen2.5-7B-Instruct	5.30	9,424	Not available
Llama-3.2-3B-Instruct	7.96	10,431	Not available
Llama-3.1-8B-Instruct	11.49	10,052	9.85 seconds

For object-calling performance, the cost analysis is illustrated in Figure A.2.1, which shows that GPT-4 incurs a cost of 0.10 € per call over a test of 200 calls.

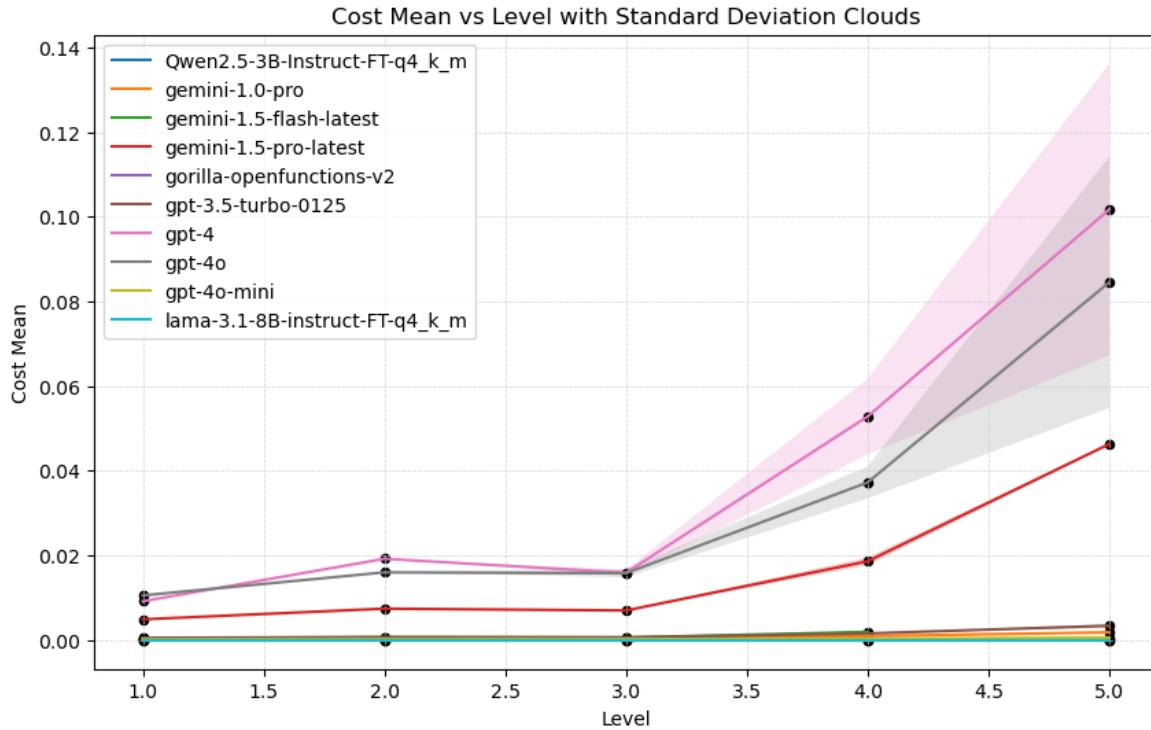


Figure A.2.1: Cost Mean per Level per Call

These results provide insights into the cost-performance trade-offs of various models, highlighting potential avenues for optimizing both inference costs and latency in practical deployments.

Bibliography

- Austin, Jacob et al. (Aug. 2021). *Program Synthesis with Large Language Models*. arXiv:2108.07732. DOI: 10.48550/arXiv.2108.07732. URL: <http://arxiv.org/abs/2108.07732> (visited on 11/13/2024).
- Bai, Jinze et al. (Sept. 2023). *Qwen Technical Report*. arXiv:2309.16609. URL: <http://arxiv.org/abs/2309.16609> (visited on 11/10/2024).
- Brown, Tom B. et al. (July 2020). *Language Models are Few-Shot Learners*. arXiv:2005.14165. DOI: 10.48550/arXiv.2005.14165. URL: <http://arxiv.org/abs/2005.14165> (visited on 11/23/2024).
- Chen, Mark et al. (July 2021). *Evaluating Large Language Models Trained on Code*. arXiv:2107.03374. DOI: 10.48550/arXiv.2107.03374. URL: <http://arxiv.org/abs/2107.03374> (visited on 11/13/2024).
- Chen, Weize et al. (Oct. 2023). *AgentVerse: Facilitating Multi-Agent Collaboration and Exploring Emergent Behaviors*. arXiv:2308.10848. DOI: 10.48550/arXiv.2308.10848. URL: <http://arxiv.org/abs/2308.10848> (visited on 11/13/2024).
- Dong, Qingxiu et al. (Oct. 2024). *Dedicated Endpoints / Inference Endpoints - Hugging Face* (2024). URL: <https://ui.endpoints.huggingface.co/croswil/endpoints/dedicated> (visited on 11/29/2024).
- Dong, Qingxiu et al. (Oct. 2024). *A Survey on In-context Learning*. arXiv:2301.00234. DOI: 10.48550/arXiv.2301.00234. URL: <http://arxiv.org/abs/2301.00234> (visited on 11/09/2024).
- Dong, Yihong et al. (Sept. 2024). *CodeScore: Evaluating Code Generation by Learning Code Execution*. arXiv:2301.09043. DOI: 10.48550/arXiv.2301.09043. URL: <http://arxiv.org/abs/2301.09043> (visited on 11/13/2024).

- Dubey, Abhimanyu et al. (Aug. 2024). *The Llama 3 Herd of Models*. arXiv:2407.21783. DOI: 10.48550/arXiv.2407.21783. URL: <http://arxiv.org/abs/2407.21783> (visited on 11/09/2024).
- Fanjia Yan et al. (2024). *Berkeley Function Calling Leaderboard*. URL: https://gorilla.cs.berkeley.edu/blogs/8_berkeley_function_calling_leaderboard.html#bfcl (visited on 11/09/2024).
- Gao, Yunfan et al. (Mar. 2024). *Retrieval-Augmented Generation for Large Language Models: A Survey*. arXiv:2312.10997. DOI: 10.48550/arXiv.2312.10997. URL: <http://arxiv.org/abs/2312.10997> (visited on 11/10/2024).
- Gu, Yuxian et al. (July 2023). “Pre-Training to Learn in Context”. In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki. Toronto, Canada: Association for Computational Linguistics, pp. 4849–4870. DOI: 10.18653/v1/2023.acl-long.267. URL: <https://aclanthology.org/2023.acl-long.267> (visited on 11/11/2024).
- Haldar, Rishin and Debajyoti Mukhopadhyay (Jan. 2011). *Levenshtein Distance Technique in Dictionary Lookup Methods: An Improved Approach*. arXiv:1101.1232. DOI: 10.48550/arXiv.1101.1232. URL: <http://arxiv.org/abs/1101.1232> (visited on 11/26/2024).
- Hermes 2 - a NousResearch Collection* (Aug. 2024). URL: <https://huggingface.co/collections/NousResearch/hermes-2-650a66656fb511ba9ea86ff1> (visited on 11/12/2024).
- Hong, Sirui et al. (Nov. 2024). *MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework*. arXiv:2308.00352. DOI: 10.48550/arXiv.2308.00352. URL: <http://arxiv.org/abs/2308.00352> (visited on 11/13/2024).
- Hu, Shengding et al. (June 2024). *MiniCPM: Unveiling the Potential of Small Language Models with Scalable Training Strategies*. arXiv:2404.06395. DOI: 10.48550/arXiv.2404.06395. URL: <http://arxiv.org/abs/2404.06395> (visited on 11/12/2024).
- Huang, Dong et al. (May 2024). *AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation*. arXiv:2312.13010. DOI: 10.48550/arXiv.2312.13010. URL: <http://arxiv.org/abs/2312.13010> (visited on 11/11/2024).

- k-quants by ikawrakow* · Pull Request #1684 · ggerganov/llama.cpp (2024). en. URL: <https://github.com/ggerganov/llama.cpp/pull/1684> (visited on 11/10/2024).
- Kossen, Jannik, Yarin Gal, and Tom Rainforth (Mar. 2024). *In-Context Learning Learns Label Relationships but Is Not Conventional Learning*. arXiv:2307.12375. DOI: 10.48550/arXiv.2307.12375. URL: <http://arxiv.org/abs/2307.12375> (visited on 11/11/2024).
- Li, Minghao et al. (Oct. 2023). *API-Bank: A Comprehensive Benchmark for Tool-Augmented LLMs*. arXiv:2304.08244. DOI: 10.48550/arXiv.2304.08244. URL: <http://arxiv.org/abs/2304.08244> (visited on 11/12/2024).
- Li, Shiyao et al. (June 2024). *Evaluating Quantized Large Language Models*. arXiv:2402.18158. DOI: 10.48550/arXiv.2402.18158. URL: <http://arxiv.org/abs/2402.18158> (visited on 11/09/2024).
- Lin, Qiqiang et al. (Oct. 2024). *Hammer: Robust Function-Calling for On-Device Language Models via Function Masking*. arXiv:2410.04587. DOI: 10.48550/arXiv.2410.04587. URL: <http://arxiv.org/abs/2410.04587> (visited on 11/10/2024).
- Liu, Sheng et al. (Feb. 2024). *In-context Vectors: Making In Context Learning More Effective and Controllable Through Latent Space Steering*. arXiv:2311.06668. DOI: 10.48550/arXiv.2311.06668. URL: <http://arxiv.org/abs/2311.06668> (visited on 11/11/2024).
- Liu, Weiwen et al. (Sept. 2024). *ToolACE: Winning the Points of LLM Function Calling*. arXiv:2409.00920. DOI: 10.48550/arXiv.2409.00920. URL: <http://arxiv.org/abs/2409.00920> (visited on 11/12/2024).
- Mialon, Grégoire et al. (Feb. 2023). *Augmented Language Models: a Survey*. arXiv:2302.07842. DOI: 10.48550/arXiv.2302.07842. URL: <http://arxiv.org/abs/2302.07842> (visited on 11/10/2024).
- Nakano, Reiichiro et al. (June 2022). *WebGPT: Browser-assisted question-answering with human feedback*. arXiv:2112.09332. DOI: 10.48550/arXiv.2112.09332. URL: <http://arxiv.org/abs/2112.09332> (visited on 11/10/2024).
- Parisi, Aaron, Yao Zhao, and Noah Fiedel (May 2022). *TALM: Tool Augmented Language Models*. arXiv:2205.12255. DOI: 10.48550/arXiv.2205.12255. URL: <http://arxiv.org/abs/2205.12255> (visited on 11/10/2024).

- Patil, Shishir G. et al. (Apr. 2024). *GoEX: Perspectives and Designs Towards a Runtime for Autonomous LLM Applications*. arXiv:2404.06921. DOI: 10.48550/arXiv.2404.06921. URL: <http://arxiv.org/abs/2404.06921> (visited on 11/10/2024).
- Prompt Engineering Guide* (Sept. 2024). en. URL: <https://www.promptingguide.ai/techniques/meta-prompting> (visited on 11/23/2024).
- Qian, Chen et al. (June 2024). *ChatDev: Communicative Agents for Software Development*. arXiv:2307.07924. DOI: 10.48550/arXiv.2307.07924. URL: <http://arxiv.org/abs/2307.07924> (visited on 11/13/2024).
- Qu, Changle et al. (Nov. 2024). *Tool Learning with Large Language Models: A Survey*. arXiv:2405.17935. URL: <http://arxiv.org/abs/2405.17935> (visited on 11/10/2024).
- Saving to GGUF / Unsloth Documentation* (July 2024). en. URL: <https://docs.unsloth.ai/basics/saving-and-using-models/saving-to-gguf> (visited on 11/10/2024).
- Saxo/alpaca_function_calling_dataset · Datasets at Hugging Face* (2024). URL: https://huggingface.co/datasets/Saxo/alpaca_function_calling_dataset (visited on 11/29/2024).
- Shi, Weijia et al. (Oct. 2023). “In-Context Pretraining: Language Modeling Beyond Document Boundaries”. en. In: URL: <https://openreview.net/forum?id=LXVswInH0o> (visited on 11/11/2024).
- Teknium, Ryan, Jeffrey Quesnelle, and Chen Guang (Aug. 2024). *Hermes 3 Technical Report*. arXiv:2408.11857. DOI: 10.48550/arXiv.2408.11857. URL: <http://arxiv.org/abs/2408.11857> (visited on 11/12/2024).
- Thoppilan, Romal et al. (Feb. 2022). *LaMDA: Language Models for Dialog Applications*. arXiv:2201.08239. DOI: 10.48550/arXiv.2201.08239. URL: <http://arxiv.org/abs/2201.08239> (visited on 11/10/2024).
- Wang, Xuezhi et al. (Mar. 2023). *Self-Consistency Improves Chain of Thought Reasoning in Language Models*. arXiv:2203.11171. DOI: 10.48550/arXiv.2203.11171. URL: <http://arxiv.org/abs/2203.11171> (visited on 11/23/2024).
- Wang, Yizhong et al. (July 2023). “Self-Instruct: Aligning Language Models with Self-Generated Instructions”. In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Anna Rogers,

- Jordan Boyd-Graber, and Naoaki Okazaki. Toronto, Canada: Association for Computational Linguistics, pp. 13484–13508. DOI: 10.18653/v1/2023.acl-long.754. URL: <https://aclanthology.org/2023.acl-long.754> (visited on 11/10/2024).
- Wu, Qingyun et al. (Oct. 2023). *AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation*. arXiv:2308.08155. DOI: 10.48550/arXiv.2308.08155. URL: <http://arxiv.org/abs/2308.08155> (visited on 11/09/2024).
- Yang, An et al. (Sept. 2024). *Qwen2 Technical Report*. arXiv:2407.10671. DOI: 10.48550/arXiv.2407.10671. URL: <http://arxiv.org/abs/2407.10671> (visited on 11/09/2024).
- Yao, Shunyu et al. (Mar. 2023). *ReAct: Synergizing Reasoning and Acting in Language Models*. arXiv:2210.03629. DOI: 10.48550/arXiv.2210.03629. URL: <http://arxiv.org/abs/2210.03629> (visited on 11/10/2024).
- Zhang, Jianguo et al. (Sept. 2024). *xLAM: A Family of Large Action Models to Empower AI Agent Systems*. arXiv:2409.03215. DOI: 10.48550/arXiv.2409.03215. URL: <http://arxiv.org/abs/2409.03215> (visited on 11/12/2024).
- Zhang, Yifan, Yang Yuan, and Andrew Chi-Chih Yao (June 2024). *Meta Prompting for AI Systems*. arXiv:2311.11482. DOI: 10.48550/arXiv.2311.11482. URL: <http://arxiv.org/abs/2311.11482> (visited on 11/23/2024).
- Zhou, Yongchao et al. (Mar. 2023). *Large Language Models Are Human-Level Prompt Engineers*. arXiv:2211.01910. DOI: 10.48550/arXiv.2211.01910. URL: <http://arxiv.org/abs/2211.01910> (visited on 11/11/2024).